



Swift 3

Разработка приложений
в среде Xcode для iPhone и iPad
с использованием iOS SDK

3-е издание

Молли Макри и др.



www.dialektika.com

Apress®

www.apress.com

Swift 3

Разработка приложений
в среде Xcode для iPhone и iPad
с использованием iOS SDK

Beginning iPhone Development with Swift 3

Exploring the iOS SDK

Third Edition

Molly Maskrey
Kim Topley
David Mark
Fredrik Olsson
Jeff Lamarche

Apress®

Swift 3

Разработка приложений
в среде Xcode для iPhone и iPad
с использованием iOS SDK

Третье издание

Молли Маскри
Ким Топли
Дэвид Марк
Фредрик Олссон
Джефф Ламарш



Москва • Санкт-Петербург • Киев
2017

ББК 32.973.26-018.2.75

М31

УДК 681.3.07

Компьютерное издательство "Диалектика"

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция докт. физ.-мат. наук Д.А. Клюшина

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:

info@dialektika.com, <http://www.dialektika.com>

Маскри, Молли, Топли, Ким, Марк, Дэвид, и др.

М31 Swift 3: разработка приложений в среде Xcode для iPhone и iPad с использованием iOS SDK.
3-е изд. : Пер. с англ. — Спб. : ООО "Альфа-книга", 2017. — 896 с. : ил. — Парал. тит. англ.

ISBN 978-5-9908910-2-9 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства APress, Berkeley, CA.

Authorized translation from the English language edition published by APress, Copyright © 2016 Molly Maskrey, Kim Topley, David Mark, Fredrik Olsson and Jeff Lamarche.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Russian language edition is published by Dialektika Computer Books Publishing according to the Agreement with R&I Enterprises International. Copyright © 2017.

Научно-популярное издание

Молли Маскри, Ким Топли, Дэвид Марк, и др.

Swift 3: разработка приложений в среде Xcode для iPhone и iPad с использованием iOS SDK 3-е издание

Литературный редактор *Л.Н. Красножон*

Верстка *Л.В. Чернокозинская*

Художественный редактор *В.Г. Павлютин*

Корректор *Л.А. Гордиенко*

Подписано в печать 22.03.2017. Формат 70x100/16.

Гарнитура Times.

Усл. печ. л. 72.24. Уч.-изд. л. 50.6.

Тираж 500 экз. Заказ № 2095.

Отпечатано в АО «Первая Образцовая типография»

Филиал «Чеховский Печатный Двор»

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(499)270-73-59

ООО "Альфа-книга". 195027. Санкт-Петербург. Магнитогорская ул., д. 30

ISBN 978-5-9908910-2-9 (рус.)

© Компьютерное издательство "Диалектика", 2017.
перевод, оформление, макетирование

ISBN 978-1-4842-2222-5 (англ.)

© 2017 Molly Maskrey, Kim Topley, David Mark,
Fredrik Olsson and Jeff Lamarche

Оглавление

Об авторе	17
Благодарности	19
Глава 1. Знакомство с системой iOS	21
Глава 2. Первое приложение	37
Глава 3. Основы взаимодействия	77
Глава 4. Новые упражнения с интерфейсом	117
Глава 5. Вращение устройства	167
Глава 6. Приложения с несколькими представлениями	215
Глава 7. Панели вкладок и селекторы	247
Глава 8. Введение в табличные представления	293
Глава 9. Контроллеры навигации и табличные представления	347
Глава 10. Представление коллекции	387
Глава 11. Разделенные представления и всплывающие меню	401
Глава 12. Настройки приложений и пользовательские настройки по умолчанию	431
Глава 13. Основы долговременного хранения данных	475
Глава 14. Документы и служба iCloud	529
Глава 15. Многопотоковое программирование с помощью технологии Grand Central Dispatch	567
Глава 16. Графика и рисование	607
Глава 17. Введение в каркас Sprite Kit	635
Глава 18. Нажатия, касания и жесты	681
Глава 19. Определение местоположения	713
Глава 20. Распознавание ориентации и перемещения устройства	739
Глава 21. Камера и фотоархив	767
Глава 22. Локализация приложений	783
Приложение. Введение в язык Swift	815
Предметный указатель	885

Содержание

Об авторе	17
О техническом редакторе	18
Благодарности	19
Глава 1. Знакомство с системой iOS	21
О книге	22
Что вам требуется	22
Возможности разработчика	25
Что необходимо знать	27
Отличительные особенности программирования для системы iOS	27
Только одно активное приложение (как правило)	28
Только одно окно	28
Ограниченный доступ с целью обеспечения безопасности	29
Приложение должно быстро реагировать на действия пользователя	29
Ограниченный размер экрана	29
Ограниченные ресурсы системы	31
Уникальные возможности устройств iOS	32
Ввод и вывод информации	32
Содержание книги	32
Что нового в данном издании	35
Версии языка Swift и каркаса Xcode	35
Начнем	35
Глава 2. Первое приложение	37
Проект Hello World	38
Окно проекта в среде Xcode	41
Приступим к проекту Hello World	53
Введение в программу Interface Builder	54
Форматы файлов	55
Раскладовка	56
Вспомогательная область	58
Добавление метки на представление	60
Изменение атрибутов	63
Завершающие штрихи	65
Экран запуска приложения	69
Запуск приложения на устройстве	72
Резюме	76
Глава 3. Основы взаимодействия	77
Парадигма “модель–контроллер–представление”	77
Создание приложения ButtonFun	79

Создание контроллера представления	80
Выходы и действия	81
Выходы	82
Действия	83
Разработка контроллера представления	84
Разработка пользовательского интерфейса	85
Добавление кнопок и метода действия	86
Добавление метки и выхода	91
Создание метода действия	94
Тестирование приложения Button Fun	95
Решение проблем с помощью механизма Auto Layout	96
Предварительный просмотр макета	108
Изменение стиля текста	110
Использование делегата приложения	111
Резюме	115
Глава 4. Новые упражнения с интерфейсом	117
Активные, статические и пассивные элементы управления	120
Создание приложения Control Fun	122
Реализация графического представления и полей редактирования	123
Добавление графического представления	123
Изменение размеров графического представления	125
Настройка атрибутов представления	127
Добавление полей редактирования	132
Добавление ограничений	138
Создание и присоединение выходов	140
Закрытие клавиатуры	142
Закрытие клавиатуры при нажатии кнопки Done	143
Закрытие клавиатуры прикосновением к фону	144
Добавление ползунка и метки	146
Создание и связывание действий и выходов	148
Реализация метода действия	148
Реализация переключателей, кнопки сегментированного элемента управления	150
Добавление переключателей с метками	151
Связывание и создание выходов переключателя и действий	152
Реализация действий переключателя	152
Добавление изображения на кнопку	154
Растягивающиеся изображения	155
Состояния элемента управления	156
Связывание выходов и действий кнопки	157
Реализация действия сегментированного элемента управления	158
Реализация списка действий и сигнала	159
Демонстрация списка действий	160
Вывод предупреждения	163
Резюме	165
Глава 5. Вращение устройства	167
Механизм автоматического поворота	168
Точки, пиксели и дисплей Retina	169
Способы реализации автоматического вращения	170

8 СОДЕРЖАНИЕ

Выбор ориентации представления	170
Поддержка ориентации на уровне приложения	171
Настройка поддержки поворота	173
Создание макета проекта	175
Переопределение ограничений, заданных по умолчанию	180
Кнопки, занимающие всю ширину представления	181
Создание адаптивных макетов	184
Создание приложения Restructure	184
Настройка конфигурации iPhone в альбомной ориентации (wC hC)	193
Настройка конфигурации iPad (iPhone Plus в альбомной ориентации (wR hR))	202
Резюме	213
Глава 6. Приложения с несколькими представлениями	215
Основные типы приложений с несколькими представлениями	215
Архитектура приложения с несколькими представлениями	221
Корневой контроллер	223
Устройство представления содержимого	223
Создание переключателя представлений	224
Переименование контроллера представления	224
Добавление контроллеров представления содержимого	226
Модификация файла SwitchingViewController.swift	227
Создание представления с панелью инструментов	228
Связывание кнопки панели инструментов с контроллером представления	231
Создание корневого контроллера представления	232
Реализация представлений содержимого	238
Анимация перехода	242
Резюме	245
Глава 7. Панели вкладок и селекторы	247
Приложение Pickers	248
Делегаты и источники данных	251
Создание приложения Pickers	252
Создание контроллеров представлений	252
Создание контроллера представления панели вкладок	254
Первичный тест на симуляторе	258
Реализация селектора даты	260
Реализация однокомпонентного селектора	264
Создание представления	264
Реализация контроллера как источника данных и делегата	268
Реализация многокомпонентного селектора	271
Создание представления	272
Реализация контроллера	272
Реализация зависимых компонентов	275
Создание простой игры с пользовательским селектором	283
Подготовка контроллера представления	283
Создание представления	283
Реализация контроллера	284
Последние штрихи	288
Резюме	292

Глава 8. Введение в табличные представления	293
Основы табличных представлений	293
Табличные представления и ячейки табличного представления	293
Сгруппированные и простые таблицы	295
Реализация простой таблицы	296
Проектирование представления	297
Реализация контроллера	299
Добавление изображения	303
Использование стилей ячеек табличных представлений	305
Настройка уровня отступа	308
Обработка выбора строки	310
Изменение размера шрифта и высоты строки	312
Настройка ячеек табличного представления	314
Добавление дочерних представлений к ячейкам табличного представления	314
Реализация приложения с пользовательскими табличными представлениями	314
Создание подкласса UITableViewCell	315
Загрузка объекта класса UITableViewCell из XIB-файла	321
Проектирование ячейки табличного представления с помощью программы Interface Builder	322
Использование новой ячейки табличного представления	328
Группированные и индексированные разделы	328
Создание представления	328
Импортирование данных	329
Реализация контроллера	331
Добавление индекса	334
Реализация строки поиска	334
Отладка представления	343
Резюме	346
Глава 9. Контроллеры навигации и табличные представления	347
Основы контроллеров навигации	348
Стеки	348
Стек контроллеров	349
Font — простой браузер шрифтов	350
Подконтроллеры приложения Fonts	352
Основа приложения для работы с шрифтами	354
Создание контроллера корневого представления	359
Начальная настройка раскладовки	363
Первый подконтроллер: представление списка шрифтов	364
Раскладовка списка шрифтов	367
Создание контроллера представления размеров шрифтов	370
Создание раскладовки контроллера представления размеров шрифтов	372
Подготовка контроллера представления списка шрифтов к переходам	372
Создание контроллера представления информации о шрифте	373
Раскладовка контроллера представления информации о шрифте	375
Адаптация контроллера представления списка шрифтов для нескольких переходов	379
Предпочитаемые шрифты	380
Тонкости табличного представления	380
Реализация жеста удаления	381
Реализация переупорядочения перетаскиванием	383
Резюме	385

10 СОДЕРЖАНИЕ

Глава 10. Представление коллекции	387
Создание проекта DialogViewer	388
Определение пользовательских ячеек	389
Настройка контроллера представления	393
Предоставление содержимого ячеек	394
Создание потока	395
Представления заголовка	397
Резюме	399
Глава 11. Разделенные представления и всплывающие меню	401
Построение приложения master-detail с помощью класса UISplitViewController	403
Определение структуры с помощью раскладовки	406
Определение функциональной возможности в коде	408
Контроллер главного представления	410
Как работает приложение master-detail	412
Добавление данных о президентах	415
Создание пользовательского всплывающего меню	421
Резюме	429
Глава 12. Настройки приложений и пользовательские настройки по умолчанию	431
Знакомство с пакетом настроек	431
Приложение Bridge Control	433
Создание проекта Bridge Control	436
Подготовка пакета настроек	437
Добавление пиктограмм в пакет настроек	454
Чтение настроек из приложения	458
Изменение пользовательских настроек по умолчанию	
непосредственно из приложения	463
Регистрация значений по умолчанию	467
Суровая действительность	468
Переключение в приложение Settings	471
Резюме	473
Глава 13. Основы долговременного хранения данных	475
“Песочница” приложения	476
Определение местоположения каталогов Documents и Library	479
Определение местоположения каталога tmp	481
Стратегии сохранения файлов	481
Долговременное хранение одного файла	482
Долговременное хранение нескольких файлов	482
Использование списков свойств	483
Сериализация списка свойств	483
Первая версия приложения Persistence	485
Архивирование объектов моделей	491
Поддержка протокола NSCoding	491
Реализация протокола NSCopying	493
Архивирование и разархивирование объектов данных	494
Приложение Archiving	494
Использование встроенной в iOS базы данных SQLite3	498
Создание или открытие базы данных	499
Использование связанных переменных	500

Приложение SQLite3	502
Использование каркаса Core Data	508
Приложение Core Data	514
Модификация файла AppDelegate.swift	519
Резюме	527
Глава 14. Документы и служба iCloud	529
Управление хранилищем документов с помощью класса UIDocument	530
Создание приложения TinyPix	531
Создание класса TinyPixDocument	532
Основной код	535
Начальная раскладовка	543
Создание класса TinyPixView	546
Раскладовка детализированного представления	551
Добавление поддержки службы iCloud	555
Создание профиля ресурсов	556
Как послать запрос	559
Местоположение файлов	562
Сохранение настроек в службе iCloud	562
Резюме	566
Глава 15. Многопотоковое программирование с помощью технологии Grand Central Dispatch	567
Создание приложения SLOWWORKER	568
Основы многопотоковой работы	572
Единицы работы	573
Организация очередей на низком уровне средствами GCD	574
Усовершенствование приложения SlowWorker	575
Фоновая работа	581
Жизненный цикл приложения	582
Уведомления о смене состояния	584
Создание приложения State Lab	586
Исследование состояний выполнения	587
Практическое применение смены состояний выполнения	589
Обработка неактивного состояния	591
Обработка фонового состояния	596
Резюме	605
Глава 16. Графика и рисование	607
Библиотека QUARTZ 2D	607
Подход к рисованию в библиотеке Quartz 2D	608
Графические контексты Quartz 2D	608
Система координат	610
Задание цветов	612
Рисование изображений в графическом контексте	614
Рисование форм: прямоугольников, прямых и кривых линий	615
Образцы инструментальных средств Quartz 2D: узоры, градиенты и пунктиры	615
Приложение QuartzFun	616
Создание приложения QuartzFun	616
Ввод кода рисования из библиотеки Quartz 2D	625
Оптимизация приложения QuartzFun	631
Резюме	634

12 СОДЕРЖАНИЕ

Глава 17. Введение в каркас Sprite Kit	635
Создание приложения Textshooter	636
Первоначальная настройка сцены	641
Движение игрока	645
Создание противников	651
Размещение противников на сцене	652
Начало стрельбы	653
Атака на противников с соблюдением законов физики	659
Завершение игры на разных уровнях	660
Настройка столкновений	662
Придание игре остроты с помощью частиц	666
Размещение частиц на сцене	669
Завершение игры	672
Создание начальной сцены	673
Добавление звуковых эффектов	676
Усложнение игры с помощью силовых полей	677
Резюме	680
Глава 18. Нажатия, касания и жесты	681
Мультисенсорная терминология	682
Цепочка реагирующих элементов	683
Реакция на события	683
Передача события по цепочке реагирующих элементов, поддерживаемой в активном состоянии	685
Мультисенсорная архитектура	686
Четыре метода уведомления о касаниях	687
Создание приложения TouchExplorer	688
Создание приложения Swipes	693
Обнаружение скольжения с помощью событий касания	694
Автоматическое распознавание жестов	697
Распознавание скольжения несколькими пальцами по экрану	699
Распознавание многократных нажатий экрана	701
Распознавание щипков и вращения	707
Резюме	710
Глава 19. Определение местоположения	713
Диспетчер местоположения	714
Задание требуемой точности	714
Установка фильтра расстояния	715
Получение разрешения на пользование службами определения местоположения	716
Запуск диспетчера местоположения	716
Благородное использование диспетчера местоположения	716
Делегат диспетчера местоположения	717
Обновление координат	717
Определение широты и долготы средствами класса CCLocation	717
Уведомления об ошибках	720
Создание приложения WhereAmI	720
Использование новых координат в диспетчере местоположения	727
Отображение движения на карте	730
Изменение разрешений на пользование службами определения местоположения	734
Резюме	736

Глава 20. Распознавание ориентации и перемещения устройства	739
Физические основы работы акселерометра	739
Распознавание вращения с помощью гироскопа	741
Каркас Core Motion и диспетчер движения	741
Создание приложения MotionMonitor	742
Упреждающий доступ к данным движения	748
Результаты гироскопических и пространственных измерений	750
Результаты акселерометрических измерений	751
Распознавание сотрясений	752
Встроенные средства обнаружения сотрясений	754
Сотрясение на грани поломки	754
Акселерометр в качестве контроллера направления	757
Катание шаров	758
Построение представления для шарика	760
Расчет движения шарика	763
Резюме	766
Глава 21. Камера и фотоархив	767
Применение селектора изображений и класса UIIMAGEICKERCONTROLLER	768
Применение контроллера селектора изображений	768
Реализация делегата для контроллера селектора изображений	770
Разработка пользовательского интерфейса приложения	773
Функции защиты конфиденциальности	775
Реализация контроллера представления камеры	776
Резюме	781
Глава 22. Локализация приложений	783
Архитектура локализации	784
Файлы символьных строк	785
Содержимое файла символьных строк	786
Функция для локализации символьных строк	787
Реализация приложения LocalizeMe	788
Локализация проекта	795
Локализация раскадровки	798
Локализация отображаемого названия приложения	809
Добавление еще одной локализации	812
Резюме	813
Резюме книги	813
Приложение. Введение в язык Swift	815
Основы языка Swift	815
Игровые площадки, комментарии, переменные и константы	816
Предопределенные типы, операции и управляющие операторы	821
Массивы, диапазоны и словари	834
Необязательные типы данных	841
Управляющие операторы	848
Функции и замыкания	854
Обработка ошибок	861
Классы и структуры	867
Структуры	868
Классы	870

14 СОДЕРЖАНИЕ

Свойства	872
Методы	874
Связывание необязательных типов в цепочку	875
Создание производных классов и наследование	877
Протоколы	881
Расширения	883
Резюме	884
Предметный указатель	885

Я невероятно счастлива получить возможность написать новый вариант этой книги. Именно ее первое издание вдохновило меня приступить к программированию для системы iOS. В связи с этим я хотела бы выразить благодарность людям, сыгравшим особую роль в моей жизни и во многом придавшим ей смысл.

Я благодарю Чану, которая была рядом со мной 95% времени и постоянно вдохновляла меня, особенно тогда, когда мне казалось, что я уже больше не могу. Из обычной коллеги по работе она превратилась в мою ближайшую подругу. Каждый день, проведенный вместе с ней, я считаю большим подарком. Я всегда готова прийти к ней на помощь, где бы она ни была.

Последние два года моей лучшей подругой была Джессика (Голди). Она стояла рядом со мной как подружка невесты, и с тех пор мы неразлучны. Мы пили вместе, танцевали вместе, вместе являлись на вечеринки без приглашения и вместе плакали... Она не делает мне поблажек, желая мне добра. Во время ее семинедельного путешествия по Южной Америке мы общались с ней практически каждый день. И несмотря на то что нас разделял континент, эти разговоры сближали нас.

Эшли, я обещала тебе упомянуть и я это делаю. Эта связана со мной совсем не так, как остальные друзья. Мы обе — технари и часто сидим у нее на кухне, попивая вино и рассуждая о модернизации компьютеров. Она умна, красива, добра и одна из самых веселых людей, которых я знаю.

Наконец я посвящаю книгу своей спутнице жизни Дженифер. Она — моя бизнес-партнер, босс, иногда мы танцуем вместе. Она всегда рядом, когда я засыпаю и когда просыпаюсь. Она прочла и одобрила каждое слово в этой книге.

Мои друзья поддерживали меня на протяжении последних трудных месяцев. Работа над книгой — дело уединенное, и поддержка таких красивых, восхитительных женщин стала единственным залогом моего успеха. Один старый друг сказал мне несколько месяцев назад, что иногда дружба длится не больше одного года. Я молюсь, чтобы эти женщины были моими подругами на всю жизнь.

ММ, Сентябрь 2016

Об авторе



Молли Маскри (Molly Maskrey) работала инженером-электриком в разных аэрокосмических компаниях, в том числе в IBM Federal Systems, TRW (в настоящее время — Northrup Grumman), Loral Systems, Lockheed Martin и Boeing. Свободно ориентируясь в первом буме доткомов, она поняла, что пузырь скоро лопнет, и на несколько лет изменила вид деятельности — переехала на остров Мауи и стала учиться виндсерфингу на прекрасном пляже Kanaha Beach Park. Однако Молли никогда долго не сидит на одном месте,

поэтому вскоре она переехала в Денвер, штат Колорадо, и открыла несколько компаний вместе со своей подругой и партнером Дженифер, в частности компанию Global Tek Labs, которая занимается разработкой программ и проектированием вспомогательных служб для системы iOS, которая стала одной из самых успешных консалтинговых компаний, ориентирующихся на новых разработчиков, желающих создавать хорошие программы для устройств компании Apple.

В том же году Молли опубликовала первую книгу в издательстве Apress. В ней описывалось, как создать утилиты для операционной системы iOS; до сих пор она остается одной из лучших книг на эту тему.

В 2014 году Молли и Дженифер основали компанию Quantitative Bioanalytics Laboratories, дочернюю компанию Global Tek, чтобы внедрить технологию мобильных сенсоров высокого разрешения в медицину для разработки приборов, контролирующих равновесие и предотвращающих падение людей преклонного возраста, устройств, оценивающих спортивные упражнения, и приборов для инструментального анализа походки (*instrumented gait analysis — IGA*).

Недавно Молли организовала программу Galvanize Data Science Immersive в Денвере, в рамках которой проводит обучение языкам Python, SQL и R, учит создавать веб-агрегаторы данных, проводить анализ больших объемов данных, байесовский анализ и создавать многие другие полезные вещи, в частности — масштабируемую аналитическую платформу реального времени для Интернета вещей (*Internet of Things*).

Молли живет в городе Паркер, штат Колорадо, вместе с Дженифер и двумя лабрадорами.

О техническом редакторе

Брюс Уэйд (Bruce Wade) — разработчик программного обеспечения из Британской Колумбии, Канада. Он разрабатывает программное обеспечение с 16 лет, когда создал первый веб-сайт. Он изучал компьютерные информационные системы в Технологическом институте Деври (DeVry Institute of Technology) в Калгари, а затем совершенствовал свои знания в области программирования компьютерных игр и визуального программирования в Институте искусств в Ванкувере (The Art Institute of Vancouver). Он работал как в крупных корпорациях, так и в начинающих компаниях. Приобретая опыт разработки программного обеспечения, Брюс освоил разные языки, включая C/C++, Python, Objective-C, Swift, Postgres и JavaScript. В 2012 году он основал компанию Warply Designed, занимающуюся разработкой мобильных приложений 2D/3D для систем OS X. Когда Брюс не генерирует новые идеи, он любит кататься на велосипеде со своей собакой-боксером Rasco, заниматься спортом и путешествовать.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@dalektika.com
WWW: <http://www.dalektika.com>

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116
в Украине: 03150, Киев, а/я 152

Благодарности

Прежде всего я хотела бы выразить благодарность всем моим друзьям, окававшим мне поддержку в ходе работы над книгой. Спасибо, Сэм, Бриттани, Аманда, Кристи, Пайпер, Петер, Майкс, Келли и вся команда Galvanize-Platte в Денвере, которую я знаю и люблю!

Спасибо Детскому госпиталю в Колорадо (Children's Hospital Colorado) и Центру анализа равновесия и движений (Center for Gait and Movement Analysis), которые были столь любезны, что позволили мне участвовать в изучении детского церебрального паралича и других нарушений равновесия: эти знания позволили мне сосредоточить свои усилия на оказании помощи тем, кто в ней действительно нуждается.

Благодарю клиентов и друзей компании Global Tek Labs, разрешивших мне использовать некоторые их проекты для иллюстрации. Спасибо сотням людей, посетившим мои лекции за последние годы и поделившимся своими идеями, таким как Джон Хейли (John Haley), рассказавший мне обо всех сложностях, с которыми он столкнулся, изучая механизм Auto Layout в среде Xcode. Их реальный опыт помог мне уточнить материал книги.

В заключение я хочу выразить благодарность всем авторам предыдущих книг, ставших основой моей скромной работы и позволивших расширить мой кругозор.

ГЛАВА 1



Знакомство с системой iOS

Программирование мобильных устройств Apple не только приносит разработчикам моральное и материальное удовлетворение от того, что их приложения изменяют жизнь людей (рис. 1.1), но и позволяет найти замечательных единомышленников. Несмотря на определенные трудности, с которыми вы можете столкнуться при изучении языка, инструментов и процессов, знакомство с этими людьми поможет вам освоиться в новом мире и проявить свои лучшие качества.

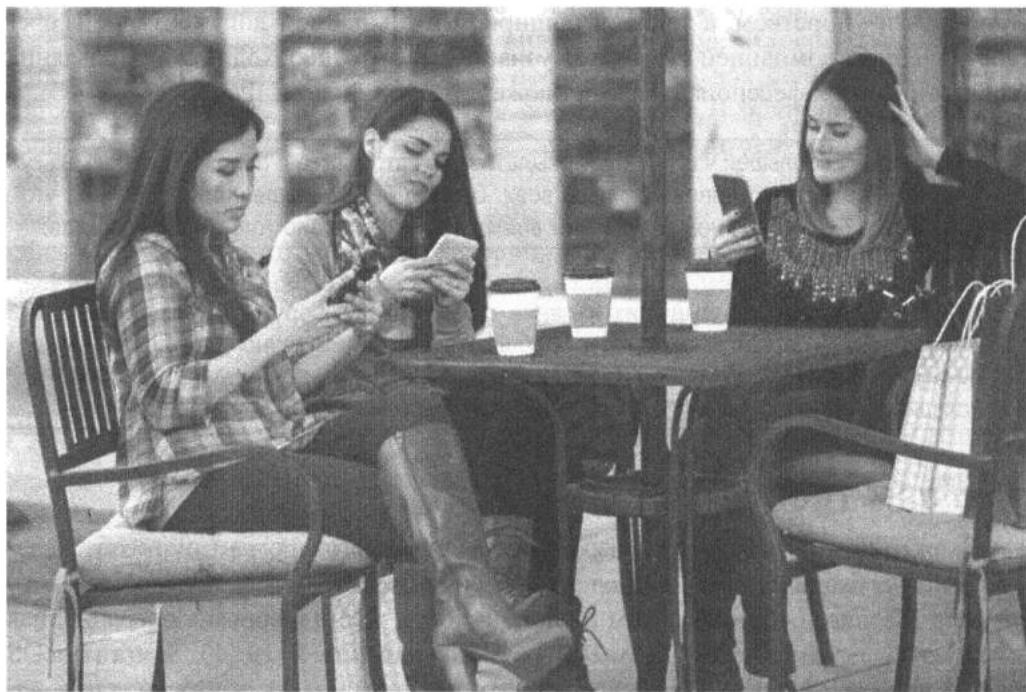


Рис. 1.1. Как разработчик приложения для системы iOS вы сможете познакомиться с людьми, использующими продукт вашего творчества на практике

Считайте меня своим другом, который будет сопровождать вас на пути освоения системы iOS. Я горжусь возможностью ввести вас в мир разработки приложения для устройств iOS — iPhone, iPod Touch или iPad. Система iOS представляет собой перспективную платформу, демонстрирующую бурный рост, начиная с ее появления в 2007 года. Распространение мобильных устройств означает, что люди могут использовать программное обеспечение везде, где они захотят, с помощью как телефонов, так и других аксессуаров, например Apple Watch. С появлением системы iOS 10, среды Xcode 8, языка Swift 3 и последней версии комплекта для разработки программного обеспечения (SDK) возможности стали еще более захватывающими и простыми.

О книге

Эта книга представляет собой руководство, призванное помочь вам приступить к разработке собственных приложений для системы iOS. Ее цель — помочь преодолеть первичные трудности и объяснить, как работают приложения для системы iOS и как они создаются.

Прорабатывая материал книги, вы создадите множество небольших приложений, которые иллюстрируют конкретные возможности системы iOS и демонстрируют, как можно управлять этими возможностями или взаимодействовать с ними. Объединив теоретические знания, полученные в этой книге, со своими талантом и упорством, а также с обширной и подробной документацией, предоставленной компанией Apple, вы узнаете все, что необходимо для создания собственных профессиональных приложений для iPhone и iPad.

ЗАМЕЧАНИЕ. По большей части я буду ссылаться на устройства iPhone и iPad, поскольку именно они распространены больше всех. Однако это совершенно не означает, что устройство iPod Touch не заслуживает внимания; это сделано только для удобства.

ПОДСКАЗКА. Авторы предыдущих изданий организовали форум, посвященный этой книге. Это прекрасная возможность пообщаться со своими единомышленниками, получить ответы на свои вопросы и даже ответить на вопросы других участников. Форум находится по адресу <http://forum.learncoocao.org>. Добро пожаловать!

Что вам требуется

Прежде чем приступить к разработке собственного программного обеспечения, вам необходимо иметь несколько инструментов. Новичкам понадобится компьютер Macintosh с процессором Intel, на котором инсталлирована операционная система Yosemite (версия OS X 10.10), El Capitan (OS X 10.11), Sierra (macOS 10.12) или еще новая. Все новые компьютеры — как ноутбуки, так и настольные — должны работать без проблем. Кроме компьютеров, вам понадобится программное обеспечение. Имея идентификатор Apple ID, вы сможете научиться

разрабатывать приложения для системы iOS и получить необходимое программное обеспечение. Если вы являетесь владельцем устройства iPhone, iPad или iPod, то у вас почти наверняка уже есть идентификатор Apple ID. Если же его у вас нет, то просто посетите сайт <https://appleid.apple.com> и создайте учетную запись. После этого перейдите на страницу <https://developer.apple.com/develop> — и увидите страницу, показанную на рис. 1.2.



Рис. 1.2. Веб-сайт Development Center компании Apple

Щелкните на кнопке **Downloads**, расположенной на верхней панели, чтобы перейти на страницу основных ресурсов (рис. 1.3), на которой размещены текущие версии программ и текущая бета-версия системы iOS. Здесь вы увидите ссылки на множество документов, видеоматериалов, исходных кодов и других справочных материалов. Все это предназначено для более качественного обучения разработке приложений для системы iOS. Выполнив вертикальную прокрутку экрана, вы найдете разделы **Documentation** и **Videos**. Кроме того, вы найдете ссылку на форумы Apple Developer Forums, содержащие обсуждения самых разных тем, касающихся платформы iOS, а также macOS, watchOS и tvOS. Для того чтобы стать участником форума, вам необходимо зарегистрироваться как разработчику приложений для устройств компании Apple.

ЗАМЕЧАНИЕ. На конференции разработчиков WWDC 2016 компания Apple вернула название системы "OS X" к старому варианту "macOS", чтобы обеспечить ее согласованность с именами четырех основных платформ.

Наиболее важным инструментом, который вы будете использовать, является интегрированная среда разработки программ Xcode компании Apple. Она включает в себя инструменты, предназначенные для создания и отладки исходного кода, компиляции и настройки производительности написанных вами приложений.

Загрузить текущую версию среды Xcode можно с помощью ссылки **Xcode**, расположенной на странице **Download** (см. рис. 1.3). Если вы предпочитаете использовать новейшую версию, то зайдите на сайт Mac App Store, используя меню **Apple** на компьютере Mac.



Рис. 1.3. На странице Downloads можно загрузить любые программные продукты и бета-версии инструментов для их разработки. Для этого необходимо зарегистрироваться с помощью своего идентификатора Apple ID

Версии комплекта SDK и исходный код для примеров

По мере развития версий комплекта SDK и среды Xcode механизм их загрузки будет изменяться. В настоящее время компания Apple выкладывает текущую "стабильную" версию среды XCode и пакета iOS SDK на веб-сайте Mac App Store, часто одновременно предоставляя разработчикам возможность загружать предварительные версии выпусков со своего сайта для разработчиков. Если вы хотите загрузить самую свежую (не бета) версию среды Xcode и пакета iOS SDK, используйте веб-сайт Mac App Store.

Данная книга ориентирована на последнюю версию комплекта SDK. Иногда мы использовали новые функциональные возможности, которые стали доступными в версии iOS 10 и могут быть не совместимыми с более ранними версиями комплекта SDK.

Не забудьте загрузить самые свежие и самые лучшие архивы исходного кода с веб-сайта, посвященного нашей книге (www.apress.com). По мере появления новых версий комплекта SDK мы будем обновлять наши примеры, поэтому рекомендуем периодически посещать наш сайт.

Возможности разработчика

Бесплатно загружаемый комплект SDK содержит симулятор, позволяющий создавать и выполнять приложения для устройств iPhone и iPad на компьютерах Mac. Это прекрасная возможность для обучения программированию для системы iOS. Однако симулятор не поддерживает функциональные возможности аппаратного обеспечения, например работу акселерометра и видеокамеры. Для того чтобы проверить эти возможности, необходимо иметь физические устройства iPhone, iPod Touch или iPad. Несмотря на то что большую часть вашего кода можно тестировать с помощью симулятора, для всех программ это невозможно. Даже если ваша программа работает на симуляторе, перед публикацией ее необходимо тщательно протестировать на реальном устройстве.

Предыдущие версии среды Xcode требовали, чтобы разработчики регистрировались как участники программы Apple Developer Program (за деньги). Только в этом случае они могли загружать свои приложения на реальный iPhone или другое устройство. К счастью, ситуация изменилась. Среда Xcode 7 позволяет пользователям тестировать приложения на реальных устройствах (при некоторых ограничениях), не требуя, чтобы они регистрировались как участники программы Apple Developer Program. Это значит, что большую часть примеров, приведенных в книге, можно бесплатно запускать на устройствах iPhone и iPad. Однако это не позволяет вам распространять свои приложения через магазин App Store. Для того чтобы иметь эту возможность, необходимо подписаться на одну из нижеприведенных программ, заплатив определенную сумму.

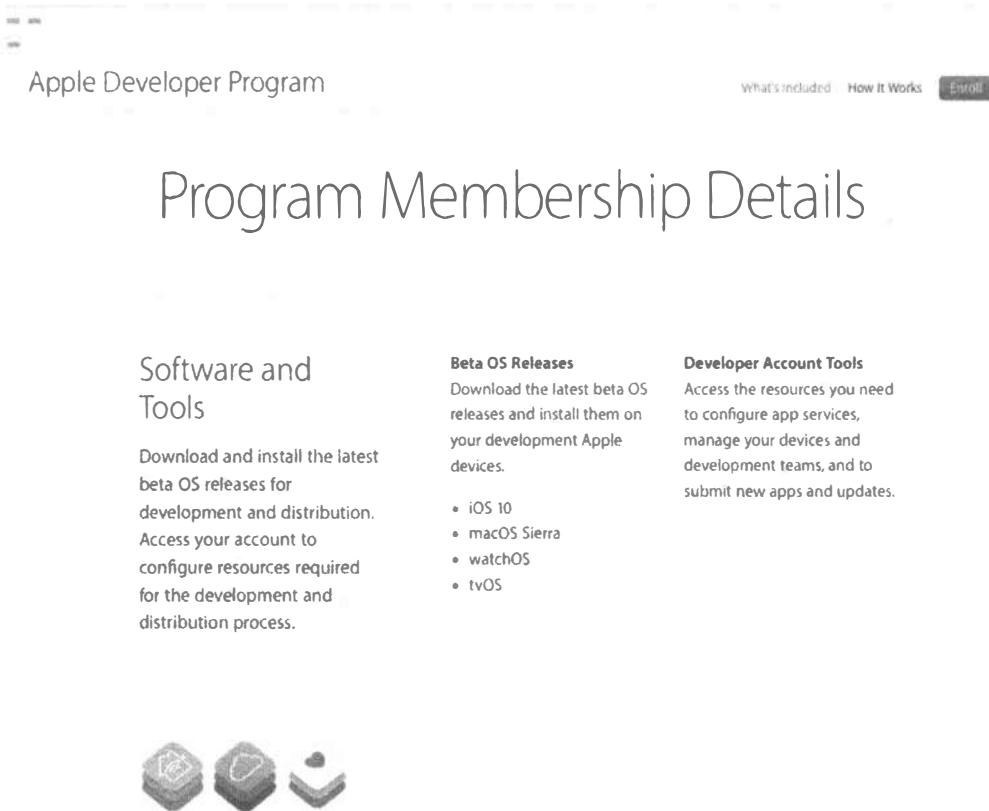
Программа Standard стоит 99 долларов в год. Она предоставляет множество инструментов для разработки, ресурсы, техническую поддержку и распространение ваших разработок через магазин Mac App Store, а также дает возможность разрабатывать и распространять программы для систем iOS, watchOS, tvOS и macOS.

- **Программа Enterprise стоит 299 долларов в год.** Она предназначена для компаний, разрабатывающих лицензионное корпоративное программное обеспечение для системы iOS.

Для того чтобы узнать больше об этих программах, посетите веб-страницу <http://developer.apple.com/programs> (рис. 1.4). Если вы — независимый разработчик, то вам целесообразно стать участником программы Standard. Впрочем, вы не обязаны это делать, если не собираетесь запускать приложения, использующие такие функциональные возможности, как iCloud, которые требуют наличия платного членства, или не хотите становиться участником форумов Apple Developer Forums, или не планируете продавать свои программы через магазин App Store.

Поскольку операционная система iOS поддерживает работу постоянно подключенного мобильного устройства, использующего беспроводную инфраструктуру

других компаний, компания Apple была вынуждена установить более строгие ограничения для разработчиков программного обеспечения для системы iOS по сравнению с ограничениями, наложенными на разработчиков программ для компьютеров Mac (которые могут, по крайней мере могли, в момент написания книги, писать и распространять программы без ведома или одобрения компании Apple). Несмотря на то что устройства iPod touch и версии iPad, поддерживающие только подключение Wi-Fi, не используют никакой другой инфраструктуры, на них распространяются те же самые ограничения.



The screenshot shows the 'Program Membership Details' page of the Apple Developer Program. At the top, there's a navigation bar with links for 'What's Included', 'How It Works', and a prominent 'Enroll' button. Below the navigation, the title 'Program Membership Details' is displayed. The page is divided into several sections: 'Software and Tools' (with a sub-section for 'Beta OS Releases' showing download links for iOS 10, macOS Sierra, watchOS, and tvOS), 'Developer Account Tools' (describing access to resources for app services, devices, and teams), and three small icons representing different developer tools or services.

Рис. 1.4. Подписка на платное членство предоставляет доступ к бета-версиям и основным версиям системных инструментов

Компания Apple наложила эти ограничения не от скучности, а для того, чтобы минимизировать вероятность появления злонамеренных или плохо написанных программ, распространение которых может снизить производительность совместно эксплуатируемой сети. Может показаться, что разработка программ для операционной системы iOS напоминает бег с препятствиями, но компания Apple приложила много усилий для того, чтобы этот процесс был как можно более безболезненным. Кроме того, подписка стоимостью 99 долларов намного

меньше, чем цена покупки, например, интегрированной среды разработки Visual Studio, разработанной компанией Microsoft.

Что необходимо знать

Предполагается, что вы уже умеете программировать и понимаете принципы объектно-ориентированного программирования (т.е. знаете, что такое, например, классы, объекты, циклы и переменные). Кроме того, предполагается, что вы знаете язык Swift. Приложение в конце книги содержит введение в язык Swift и описание функциональной возможности Playground в среде Xcode. Если после прочтения приложения вы захотите углубить свои знания о языке Swift, лучше всего немедленно приступите к программированию и прочитайте справочник компании Apple *The Swift Programming Language*, который можно приобрести в магазине iBooks или на сайте для разработчиков приложения для системы iOS по адресу https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/index.html.

Вы должны также знать систему iOS как пользователь. Поскольку вы собираетесь писать приложения для любой платформы, освойте нюансы и особенности устройства iPhone, iPad или iPod. Посвятите время изучению интерфейса системы iOS, а также внешнему виду приложений для устройств iPhone и/или iPad.

Поскольку разные термины могут ввести вас к заблуждению, в табл. 1.1 приведены связи между платформами, интегрированными средами разработки, интерфейсами прикладного программирования и языком.

Таблица 1.1. Соответствие между платформами, инструментами и языками

Платформа	Среда	Интерфейс API	Язык
macOS	Xcode	Cocoa	Objective-C, Swift
iOS	Xcode	Cocoa Touch	Objective-C, Swift

Отличительные особенности программирования для системы iOS

Если вы никогда не использовали систему Соса на компьютерах Mac, то, возможно, каркас Соса Touch, предназначенный для разработки приложений на платформе iOS, покажется вам немного недружелюбным. Он существенно отличается от остальных обычных сред разработки приложений, используемых, например, для разработки приложений для платформы .NET или на языке Java. Не стоит беспокоиться о том, что на первых порах вам будет несколько неудобно. Просто поработайте над упражнениями, и все встанет на свои места.

ЗАМЕЧАНИЕ. В книге часто встречается термин “каркас” (framework). Несмотря на то что его смысл довольно туманный и зависит от контекста, мы будем считать, что это коллекция инструментальных средств, к которым могут относиться одна или несколько библиотек, сценарии, элементы пользовательского интерфейса и многое другое. Элементами каркаса часто являются специальные функции, такие как службы локации в каркасе CoreLocation.

Если вы уже писали программы с помощью каркаса Cocoa, то большая часть комплекта iOS SDK будет вам знакома. Очень многие классы, содержащиеся в этом пакете, остались неизменными еще со времен версий для разработки программ под управлением операционной системы macOS. Однако даже те классы, которые подверглись изменениям, подчиняются тем же фундаментальным принципам, которые были установлены в предыдущей версии. Тем не менее между каркасами Cocoa и Cocoa Touch существует несколько различий.

Независимо от уровня подготовки вы должны помнить о некоторых ключевых различиях между разработкой программ для системы iOS и для настольных компьютеров.

Только одно активное приложение (как правило)

В системе iOS в каждый момент времени быть активным и выводить информацию на экран может только одно приложение. Начиная с версии iOS 4 приложения иногда продолжают выполняться в фоновом режиме после того, как пользователь щелкнет на кнопке Home, но даже это возможно лишь в редких ситуациях, которые необходимо предусматривать специально (как именно, мы расскажем в главе 15). В системе iOS 9 компания Apple добавила возможность выполнять два приложения в фоновом режиме с одним экраном, но для этого пользователь должен иметь одну из новейших версий устройства iPad. Эта функциональная возможность, которую компания Apple называет многозадачностью, описывается в главе 11.

Когда приложение не является активным или выполняется в фоновом режиме, центральный процессор не обращает на него никакого внимания, что позволяет этому приложению оставлять открытые сетевые соединения и занимать другие ресурсы. Система iOS разрешает фоновый режим, но, для того чтобы ваше приложение в этой ситуации работало правильно, необходимо приложить определенные усилия.

Только одно окно

Операционные системы ноутбуков и настольных систем допускают параллельное выполнение программ, каждая из которых может открывать несколько окон и управлять ими. Однако система iOS позволяет приложениям работать только с одним окном. Все взаимодействия с пользователем происходят только в этом окне, причем его размер фиксирован и совпадает с размером экрана, если пользователь не включил режим многозадачности, допускающий использование одного экрана несколькими приложениями.

Ограниченный доступ с целью обеспечения безопасности

Программы на компьютерах имеют доступ ко всем ресурсам, открытым для пользователя. Однако система iOS существенно ограничивает возможности приложений.

Вы можете читать и записывать файлы только в той части файловой системы iOS, которая была создана для вашего приложения. Эта область называется **песочницей** (*sandbox*). Здесь ваше приложение может хранить документы, настройки и другие данные, которые необходимо запомнить.

Ваше приложение ограничено не только этим. Например, вы не имеете доступа к сетевым портам iOS с малыми номерами или не можете делать то, для чего требуются права администратора на настольном компьютере.

Приложение должно быстро реагировать на действия пользователя

Вследствие особенностей своего использования система iOS должна быть придирчивой, и того же самого она ожидает от вашего приложения. После запуска программы вы должны открыть приложение, загрузить настройки и данные, а также главное представление (*main view*) на экране всего за несколько секунд.

ЗАМЕЧАНИЕ. Под задержкой (*latency*) мы не имеем в виду среднюю скорость. Скорость и задержка часто используются как синонимы, но это ошибка. Задержка означает время между действием и реакцией на него. Если пользователь щелкнул на кнопке Home, система iOS откроет домашний экран, а вы должны быстро сохранить все данные и выйти. Если вы замешкаетесь дольше пяти секунд, ваше приложение будет уничтожено независимо от того, завершили вы сохранение своих данных или нет. Существует интерфейс прикладного программирования, который предоставляет вашему приложению дополнительное время для корректного завершения работы, но для его использования сначала необходимо научиться это делать. Итак, необходимо обеспечить быстрое реагирование, возможно, за счет отбрасывания неважной информации.

Ограниченный размер экрана

Экран устройства iPhone действительно хорош. В момент своего появления он имел максимальное разрешение среди всех существовавших устройств. Однако на данный момент этот экран невелик, и в результате в вашем распоряжении есть существенно меньше места, чем на экранах современных компьютеров. Экраны первых поколений iPhone имели разрешение всего 320×480 пикселей, а экраны следующего поколения обеспечивали разрешение 640×960 (iPhone 4). Экраны новейших устройств iPhone (iPhone 6/6s Plus) имеют разрешение 1080×1920 пикселей. Однако следует иметь в виду, что дисплеи с высоким разрешением (которые компания Apple называет **Retina**) втиснуты в тот же самый

формфактор, что довольно сильно ограничивает возможности приложений. Размеры экранов всех доступных в настоящий момент устройств компании Apple, поддерживающих операционную систему iOS 10, перечислены в табл. 1.2.

Таблица 1.2. Размеры в пикселях экрана устройств iOS

Устройство	Физический размер	Программный размер	Масштаб
iPhone 5 и 5s	640×1136	320×568	2×
iPhone 6/6s	750×1334	375×667	2×
iPhone 6/6s Plus	1080×1920	414×736	3×
iPhone SE	640×1136	320×568	2×
iPad 2 и iPad mini	768×1024	768×1024	1×
iPad Air, iPad Air 2, iPad Retina и iPad mini Retina	1536×2048	768×1024	2×
iPad Pro	2732×2048	1336×1024	2×

Физический размер представляет собой реальный размер устройства, измеренный в пикселях. Однако, когда вы пишете программу, ориентируйтесь на программный размер. Легко видеть, что во многих случаях программный размер составляет только половину физического. Эта ситуация сохраняется с того момента, когда компания Apple разработала экран Retina, который имел в два раза больше пикселей в обоих направлениях по сравнению со своими предшественниками. Если бы компания Apple не предприняла специальных мер, то все существующие приложения занимали бы только половину экрана Retina и стали бы практически бесполезными. По этой причине компания Apple решила выполнять автоматическое масштабирование приложения с коэффициентом 2, чтобы они заполняли весь экран на устройствах новых поколений. Коэффициент масштабирования, равный двум, применяется ко всем устройствам с экраном Retina за исключением устройства iPhone 6/6s Plus, которое имеет экран с еще более высоким разрешением, требующим коэффициента масштабирования, равного трем. Впрочем, в большинстве случаев приложение масштабируется автоматически, так что вам просто следует учитывать программный размер, а система iOS все остальное сделает сама.

Единственным исключением из этого правила являются растровые изображения. Поскольку они по своей природе имеют фиксированный размер, на любых экранах, как Retina, так и остальных, лучше всего использовать одно и то же изображение. В этом случае вы увидите, что система iOS масштабирует ваше изображение на устройствах с экраном Retina, в результате чего происходит потеря четкости. Для того чтобы устранить этот эффект, необходимо сделать специальные копии для каждого изображения с учетом коэффициентов масштабирования 2× и 3× для экранов Retina.

ЗАМЕЧАНИЕ. Вернувшись к табл. 1.1, вы увидите, что коэффициент масштабирования в четвертом столбце равен отношению физического размера к программному. Например, на устройстве iPhone 6/6s физический размер равен 750, а программный — 375, т.е. отношение равно 2:1. Однако если присмотреться внимательнее, то можно увидеть, что ситуация с устройством iPhone 6/6s имеет свои особенности. Отношение физической ширины к программной составляет 1080/414, т.е. 2,608:1. То же самое относится к высоте. Следовательно, в терминах физического размера устройство iPhone 6/6s на самом деле не имеет трехкратного разрешения экрана Retina. Однако с точки зрения программного размера масштаб увеличивается втрое, т.е. приложение, написанное для экрана 414×736, логически отображается на виртуальном экране 1242×2208, а затем масштабируется на реальный экран 1080×1920. К счастью, для этого не требуется прикладывать никаких усилий, потому что система iOS делает это автоматически.

Ограниченные ресурсы системы

Программистам, имеющим большой опыт работы, возможно, показалось, что компьютер, имеющий 512 Мбайт оперативной памяти и 16 Гбайт памяти на жестком диске, ограничен в ресурсах, и это правда. Тем не менее разработку приложения для системы iOS, вероятно, нельзя сравнить с попыткой написать сложное приложение для работы с электронными таблицами на компьютере с объемом оперативной памяти, равным 48 Кбайт. Однако, учитывая графическую природу системы iOS и все то, что она умеет, исчерпать память очень и очень легко.

Все устройства с системой iOS, доступные в настоящее время, имеют либо 512 Мбайт (iPhone 4S, оригинальный iPad mini, последняя версия iPod touch), либо 1024 Мбайт физической оперативной памяти (iPhone 5c, iPhone 6/6s, iPhone 6/6s Plus, iPad Air, iPad Air 2, iPad mini Retina), и этот показатель со временем будет увеличиваться. Одна часть этой памяти используется для буфера экрана, а другая — для системных процессов. Обычно для приложений пользователей остается не более половины оперативной памяти, а то и меньше.

Может показаться, что половина памяти — это не так уж и мало для такого небольшого компьютера, но существует еще один фактор, влияющий на объем памяти. Современные операционные системы компьютеров, такие как macOS, создают блоки памяти, которые нельзя использовать, и записывают их на диск в виде **файла подкачки** (swap file). Файл подкачки позволяет приложениям продолжать выполнение, даже если они требуют больше памяти, чем доступно на компьютере. Однако система iOS не записывает содержимое кратковременной памяти, например данные, в файл подкачки. В результате объем памяти, доступный для вашего приложения, ограничен объемом свободной физической памяти на устройстве iOS.

Каркас Cocoa Touch имеет встроенный механизм, позволяющий приложению узнавать, когда объем доступной памяти становится слишком малым. Когда это происходит, ваше приложение должно освободить ненужную память, или оно рискует быть уничтоженным.

Уникальные возможности устройств iOS

Как отмечалось выше, каркас Сосоа Touch не имеет некоторых функциональных возможностей, которыми обладает каркас Сосоа. Справедливо ради отметим, что комплект iOS SDK имеет функциональные возможности, которых нет в каркасе Сосоа, или, по крайней мере, на каждом компьютере Mac.

- Комплект iOS SDK позволяет вашему приложению определять текущие географические координаты устройства iOS, используя механизм Core Location.
- Большинство устройств iOS имеют встроенные видеокамеры и библиотеки фотографий, и комплект SDK предоставляет механизм, позволяющий вашему приложению пользоваться этими возможностями.
- Устройства iOS имеют встроенный акселерометр, позволяющий определить, как вы держите и перемещаете свое устройство.

Ввод и вывод информации

Устройства, работающие под управлением системы iOS, не оснащены клавиатурой и мышью. Это значит, что вам предоставлен совершенно новый способ взаимодействия с пользователем по сравнению с программированием обычных компьютеров. К счастью, большинство таких взаимодействий адаптировано для вас. Например, если вы добавляете поле редактирования в свое приложение, система iOS сама знает, что нужно вывести на экран клавиатуру, когда пользователь щелкает на этом поле, и вам не нужно писать для этого специальный код.

ЗАМЕЧАНИЕ. Современные устройства позволяют подключать внешнюю клавиатуру через механизм Bluetooth. Несмотря на то что это позволяет использовать традиционные навыки работы с клавиатурой и экономить место на экране, такая возможность используется довольно редко. В настоящий момент в системе iOS соединение с мышью вообще не предусмотрено.

Содержание книги

Когда я начинала разрабатывать приложения для системы iOS, которая в то время называлась “iPhone OS”, я прочитала первое издание этой книги, основанное на языке Objective-C. Со временем мне удалось разработать мощные и производительные приложения, иногда даже получая за это деньги. По этой причине я хотела бы сделать все, чтобы написать самую современную и самую лучшую версию книги, сохранив ее доступность. Я планирую охватить в книге следующие темы.

- Из главы 2 вы узнаете, как использовать Interface Builder — неотъемлемую часть среды Xcode — для создания простого интерфейса, который выводит некоторый текст на экран.

- ❸ В главе 3 мы рассмотрим вопросы взаимодействия с пользователем, создав простое приложение, динамически обновляющее текст, который выводится на экран в ходе выполнения программы в зависимости от того, какую кнопку нажал пользователь.
- ❹ Глава 4 основана на главе 3. В ней рассматриваются некоторые стандартные элементы управления интерфейсом системы iOS. Мы также увидим, как вывести на экран предупреждение и меню, чтобы предложить пользователю принять решение или сообщить ему о том, что случилось нечто необычное.
- ❺ В главе 5 рассматриваются атрибуты механизмов автоматического поворота и изменения размеров и Auto Layout, позволяющих использовать приложения для системы iOS как в книжной, так и в альбомной ориентации.
- ❻ В главе 6 мы перейдем к более сложным пользовательским интерфейсам и обсудим создание приложений, поддерживающих несколько представлений. Мы покажем, как выбрать представление в ходе выполнения программы, что значительно повысит потенциал ваших приложений.
- ❼ Панели вкладок и палитры цветов являются частью стандартного пользовательского интерфейса системы iOS. В главе 7 мы покажем, как использовать эти элементы интерфейса.
- ❽ В главе 8 описаны табличные представления, основной способ создания списков данных для пользователя, а также принципы создания приложений, использующих иерархическую навигацию. Кроме того, будет показано, как организовать поиск данных, принадлежащих приложению.
- ❾ Одним из основных элементов интерфейса приложения в системе iOS является иерархический список, позволяющий пользователю углубиться в данные или выяснить детали. В главе 9 мы научим вас создавать этот стандартный тип интерфейса.
- ❿ С самого начала приложения iOS использовали табличные представления для обеспечения динамичной вертикальной прокрутки списков. Несколько лет назад компания Apple создала новый класс с именем UICollectionView, еще более углубляющий эту концепцию, представляя разработчикам более гибкие возможности для изображения визуальных компонентов. Коллекция представлений описывается в главе 10.
- ➌ В главе 11 показано, как создать приложения с главным и детализированным представлениями и представить список (например, список сообщений электронной почты), позволяющий пользователю видеть детали каждого пункта. Кроме того, в ней описаны элементы управления iOS, поддерживающие эту концепцию, которые изначально были разработаны для устройства iPad, но сейчас доступны и для устройств iPhone.

- В главе 12 рассматриваются вопросы настройки приложений, которые представляют собой механизм системы iOS, позволяющий пользователям задавать свои параметры на уровне приложения.
- Глава 13 посвящена управлению данными в системе iOS. В ней обсуждается создание объектов для хранения данных приложения и показано, как организовать постоянное хранение данных в файловой системе iOS. Здесь также излагаются основы использования инструмента Core Data, позволяющего легко сохранять и извлекать данные. Однако для углубленного исследования этого инструмента вам стоит прочитать книгу Майкла Прайвата (Michael Privat) и Роберта Уорнера (Robert Warner) *Pro iOS Persistence Using Core Data* (Apress, 2014).
- Новая функциональная возможность в системе iOS 5 — служба iCloud — позволяет пользователям хранить свои данные онлайн, а также синхронизировать разные экземпляры и приложения. Основы службы iCloud описываются в главе 14.
- Разработчики iOS имеют возможность использовать новый подход к многопоточной разработке с помощью технологии Grand Central Dispatch, а также в некоторых ситуациях запускать свои приложения в фоновом режиме. Из главы 15 вы узнаете, как это сделать. Кроме того, мы покажем, как использовать функции системы iOS, позволяющие в некоторых ситуациях запускать приложение в фоновом режиме.
- Все любят рисовать, поэтому в главе 16 речь пойдет о том, как создавать рисунки с помощью системы Core Graphics.
- В версию iOS 7 компания Apple включила каркас Sprite Kit для создания двумерных игр. Он включает в себя механизмы имитации физических законов и анимации, позволяя создавать игры в том числе для macOS. В главе 17 мы продемонстрируем разработку простой игры с помощью механизма Sprite Kit.
- Сенсорный экран (multitouch screen), характерный для всех устройств, работающих под управлением системы iOS, может распознавать много разных жестов пользователя. В главе 18 мы покажем, как распознать основные жесты, например щипок и скольжение.
- Благодаря технологии Core Location система iOS способна определять широту и долготу. В главе 19 мы создадим программу, которая с помощью технологии Core Location будет определять, в какой точке земного шара вы находитесь.
- В главе 20 мы изучим проблемы взаимодействия с акселерометром и гироскопом,строенными в систему iOS. Эти инструменты позволяют определить, как вы держите устройство, а также с какой скоростью и в каком направлении оно перемещается.

- Практически каждое устройство с системой iOS имеет видеокамеру и библиотеку рисунков, которые будут доступны для вашего приложения. В главе 21 рассказывается о том, как получить доступ к этим ресурсам.
- В настоящее время устройства с системой iOS доступны более чем в 90 странах. Из главы 22 вы узнаете, как написать приложение, чтобы все его части легко переводились на иностранные языки. Это позволит расширить потенциальную аудиторию для вашего приложения.
- В приложении описываются текущая версия языка программирования Swift и все функции, необходимые для понимания примеров, приведенных в книге.

Что нового в данном издании

С тех пор как первое издание этой книги поступило в книжные магазины, сообщество разработчиков приложений для системы iOS значительно увеличилось. Комплект SDK постоянно развивается, а компания Apple постоянно выпускает его новые версии. Система iOS 10 и каркас Xcode 8 содержат множество новинок. Я приложила много усилий, чтобы описать новые технологии, необходимые для того, чтобы приступить к разработке приложения для iOS.

Версии языка Swift и каркаса Xcode

Язык Swift появился всего два года назад; он все еще развивается, и, вероятно, этот процесс еще будет продолжаться некоторое время. Интересно, что компания Apple обещала, что скомпилированный двоичный код ранее написанных приложений будет работать в более поздних версиях iOS, но *не гарантировала*, что исходный код одного и того же приложения будет по-прежнему компилироваться. В результате возможна ситуация, в которой код примера, который был скомпилирован и выполнен в версии среды Xcode, которая была текущей, в будущем может перестать компилироваться. Среда Xcode 6.0 поставляется с версией Swift 1, Xcode 6.3 — с версией Swift 1.2, а Xcode 7 — с версией Swift 2. Код, представленный в книге, был написан и протестирован с помощью бета-версии Xcode 8 и Swift 3.

Если вы обнаружите, что исходный код примера больше не компилируется в вашей версии Xcode, пожалуйста, посетите страницу, посвященную книге, на сайте www.apress.com и загрузите ее последнюю версию. Если и после этого у вас будут проблемы, то сообщите об этом в разделе *Erratum* на веб-сайте Apress.

Начнем

Система iOS — невероятная вычислительная платформа и перспективная современная среда для разработки приложений. Программирование для системы iOS представляет собой совершенно новый опыт, отличающийся от любой

другой платформы. Все, что казалось знакомым, на поверку оказывается несколько необычным, но по мере освоения этой книги все понятия постепенно проясняются и образуют новую стройную конструкцию.

Имейте в виду, что упражнения в книге представляют собой не просто перечень заданий. Если вы их выполните, то волшебным образом станете экспертом по разработке программ для системы iOS. Прежде чем переходить к новому проекту, убедитесь, что понимаете, что и для чего делаете. Не бойтесь вносить изменения в программы. Наблюдение за результатами экспериментов — один из лучших способов преодолеть сложности кодирования в средах, подобных Cocoa Touch.

Итак, если вы уже загрузили и инсталлировали среду Xcode, то переверните страницу и сделайте следующий шаг на пути к своей цели — стать настоящим разработчиком приложений для системы iOS.

ГЛАВА 2



Первое приложение

Для того чтобы получить представление о программировании для системы iOS и стимул к дальнейшему развитию профессиональных навыков, лучше всего сразу приступить к разработке приложения для вашего iPhone (рис. 2.1). В этой главе мы используем интегрированную среду разработки Xcode для создания небольшого приложения для системы iOS, которое выводит на экран текст “Hello, World!” Мы увидим, какие механизмы вовлечены в создание проекта приложения для системы iOS в среде Xcode, рассмотрим специфику применения программы Interface Builder при разработке пользовательского интерфейса, а затем выполним наше приложение с помощью симулятора iOS. Кроме того, снабдим наше приложение пиктограммой, чтобы оно стало еще более похожим на реальное приложение для системы iOS.



Рис. 2.1. Результаты работы приложения, разработанного в этой главе, могут показаться примитивными, но ведь мы находимся лишь в начале пути

Проект Hello World

К данному моменту на вашем компьютере Mac должны быть установлены среда Xcode 8 и комплект инструментов iOS SDK. Желательно также загрузить архив исходных кодов, приведенных в книге, с веб-сайта Apress (www.apress.com). Кстати, на этом сайте есть форум читателей книги <http://forum.learncoocao.org>. Это прекрасное место для обсуждения вопросов, связанных с разработкой программ для системы iOS, обмена вопросами и ответами и для встреч единомышленников.

ЗАМЕЧАНИЕ. Несмотря на то что в вашем распоряжении имеется полный набор проектных файлов из книги, мы полагаем, что еще больше пользы вы сможете извлечь, самостоятельно разрабатывая собственный проект, а не просто копируя готовые образцы. Создавая свой проект, вы приобретете опыт работы с разнообразными инструментами.

Проект, который мы собираемся продемонстрировать в этой главе, находится в папке 02 – Hello World в архиве исходных кодов.

Сначала необходимо запустить среду Xcode. Это инструмент, которым мы будем пользоваться на протяжении практически всей книги. После его загрузки с веб-сайта Mac App Store он будет автоматически инсталлирован в папке /Applications, как большинство приложений для системы Mac. Мы будем много работать с этой программой, поэтому стоит перетащить ее пиктограмму на панель Dock, чтобы она всегда была под рукой.

Если вы впервые работаете со средой Xcode, не беспокойтесь — мы объясним весь процесс создания нового приложения. Если вы опытный разработчик, но еще не работали с программой Xcode 4, то сами убедитесь, что в ней многое изменилось (в основном, к лучшему).

Запустив впервые среду Xcode, вы увидите окно приглашения, похожее на то, которое показано на рис. 2.2. В этом окне можете выбрать команду для создания нового проекта, соединиться с системой проверки версий, чтобы найти существующий проект, или выбрать проект из списка недавно открывавшихся проектов. Окно приглашения представляет собой удобную отправную точку, предоставляя пользователю доступ к большинству функциональных возможностей, которые могут понадобиться после запуска программы Xcode. Все эти возможности доступны также из меню программы Xcode, поэтому закройте окно и продолжайте работу. Если не хотите снова увидеть это окно в дальнейшем, просто сбросьте флагок Show this window when Xcode Launches, прежде чем закрыть окно.

Создайте новый проект, выбрав команду **New Project...** в меню **File** или нажав комбинацию клавиш **<Shift+Command+N>**, открывающую окно новых проектов (рис. 2.3). На этом листе можно выбрать шаблон проекта, который станет основой нашего приложения. Лист разделен на пять разделов: iOS, watchOS, tvOS, macOS и Cross-platform. Поскольку мы собираемся создать приложение для системы iOS, нажмите на кнопку iOS, чтобы открыть шаблоны соответствующих приложений.

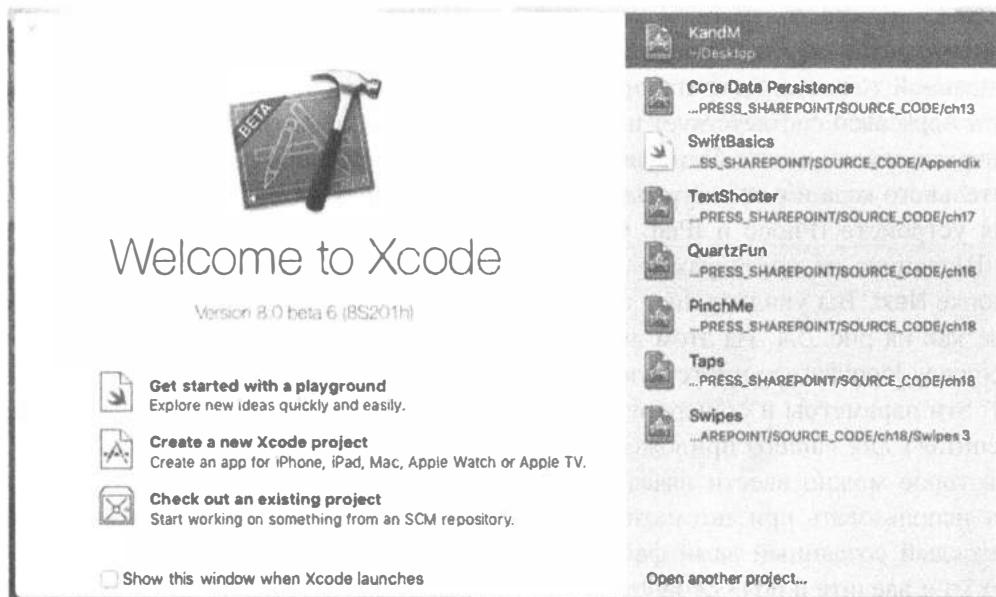


Рис. 2.2. Окно приглашения программы Xcode

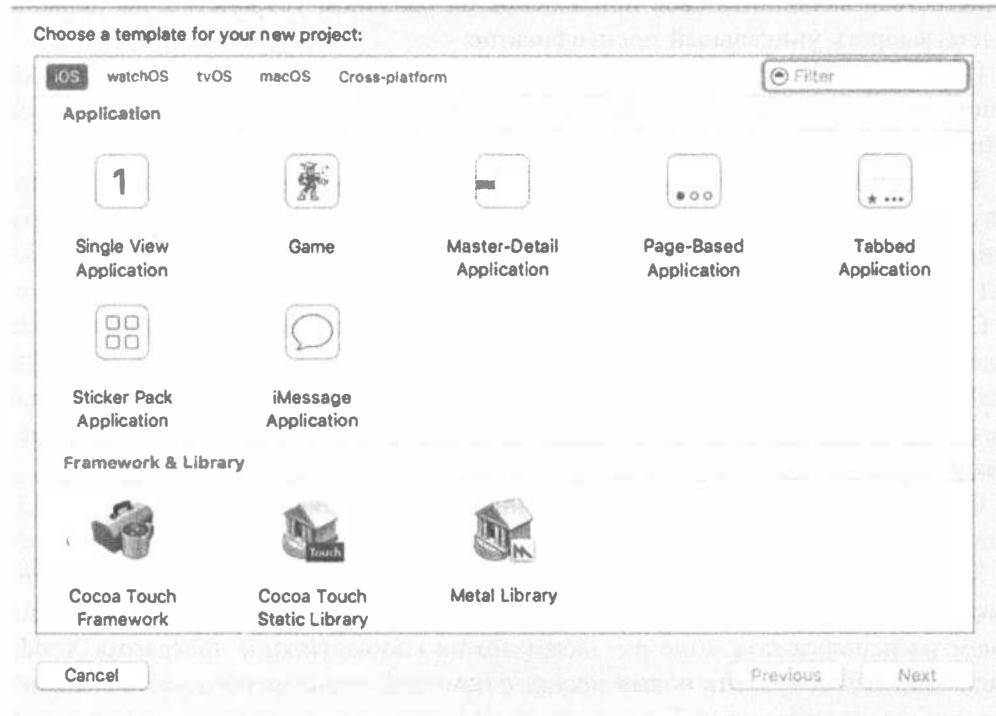


Рис. 2.3. Окно, позволяющее выбрать разные шаблоны файлов для создания нового проекта

Каждая из пиктограмм, показанных в правом верхнем углу рис. 2.3, соответствует отдельному шаблону проекта, который можно использовать в качестве отправной точки для вашего приложения для системы iOS. Пиктограмма **Single View Application** соответствует простейшему шаблону, который будет использован в первых главах книги. Остальные шаблоны обеспечивают использование дополнительного кода и/или ресурсов для создания обычных интерфейсов приложений для устройств iPhone и iPad. Они будут рассмотрены в последующих главах.

Щелкните на пиктограмме **Single View Application** (см. рис. 2.3), а затем на кнопке **Next**. Вы увидите лист параметров проекта, который выглядит примерно так, как на рис. 2.4. На этом листе необходимо заполнить поля **Product Name** и **Company Identifier**, соответствующие вашему проекту. Программа Xcode объединит эти параметры и сгенерирует уникальный идентификатор комплекта (*bundle identifier*) для вашего приложения. Вы также увидите поле **Organization Name**, в которое можно ввести название организации, которое программа Xcode будет использовать при автоматической вставке уведомления об авторском праве в каждый созданный вами файл исходного кода. Назовите наш проект **Hello World** и введите в поля **Organization Name** и **Organization Identifier** название вашей организации и идентификатор комплекта, как показано на рис. 2.4. Не вводите имя и идентификатор, продемонстрированные на рис. 2.4, потому что, когда вы попытаетесь выполнить свое приложение на реальном устройстве, вы должны будете выбрать уникальный идентификатор.

Поле **Language** предоставляет вам возможность выбрать язык программирования, на котором вы хотите работать, — Objective-C или Swift. Поскольку все примеры в книге приведены на языке Swift, выбор очевиден.

Также нужно выбрать тип устройства в списке **Devices**. Иначе говоря, программа Xcode должна знать, для какого типа устройств мы собираемся создать приложение: для iPhone, iPod Touch, iPad или для всех устройств, работающих под управлением операционной системы iOS. Выберите в раскрывающемся списке **Devices** пункт **iPhone**, если он не был выбран ранее. Теперь программа Xcode будет знать, что мы будем разрабатывать приложение для устройств iPhone и iPod, имеющих примерно одинаковый размер экрана и формфактор. В первой части книги мы будем использовать только семейство устройств iPhone, но беспокоиться не следует — устройство iPad тоже не будет забыто.

Оставьте флагок **Core Data** сброшенным — мы вернемся к нему в главе 13. Флажки **Include Unit Tests** и **Include UI Tests** также оставьте сброшенными. В среде Xcode существует много очень хороших инструментов для тестирования приложений, но они выходят за рамки нашей книги и в наших проектах мы не будем их использовать. Еще раз щелкните на кнопке **Next**, и программа Xcode предложит вам сохранить новый проект с помощью стандартного листа сохранения, показанного на рис. 2.5. Если вы еще не сохраняли проект, нажмите кнопку **New Folder** и создайте новый главный каталог для всех проектов, рассматриваемых в нашей книге, а затем вернитесь в среду Xcode и перейдите в этот каталог.

Перед тем как щелкать на кнопке **Create**, убедитесь, что флагок **Source Control** сброшен. Мы не будем обсуждать репозиторий Git в нашей книге, но программа Xcode обеспечивает определенную поддержку этого и других инструментов контроля исходного кода (*source control management — SCM*). Если вы уже знаете, как работать с репозиторием Git, и хотите использовать его, оставьте этот флагок установленным, в противном случае можете его сбросить.

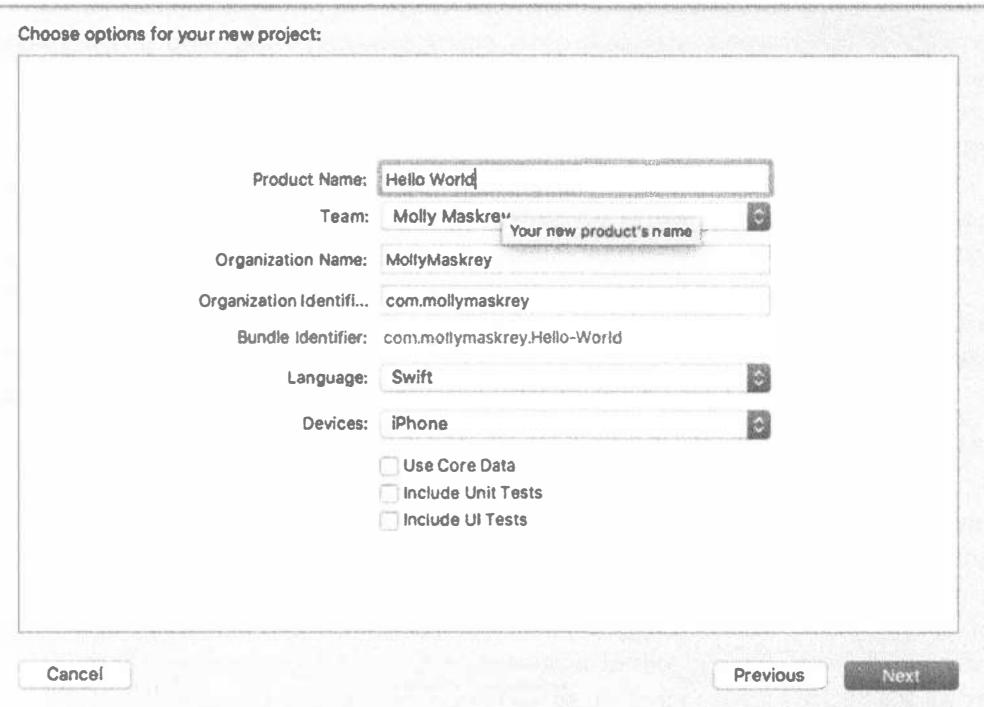


Рис. 2.4. Выбор имени и идентификатора компании для проекта

Окно проекта в среде Xcode

После того как вы щелкнете на кнопке **Save**, программа Xcode создаст и откроет ваш проект. Вы увидите новое **окно проекта** (project window) (рис. 2.6), в котором содержится много информации. Именно в этом окне вы проведете большую часть времени, разрабатывая проект.

Панель инструментов

Верхняя часть окна проекта Xcode называется **панелью инструментов** (toolbar) (рис. 2.7). Слева на ней расположены элементы управления запуском и остановкой процесса выполнения вашего проекта, и выпадающее меню для выбора схемы, которую вы собираетесь выполнить. **Схема** (scheme) — это сочетание настроек целевого устройства и сборки, причем выпадающее меню позволяет задать конкретный параметр легко и быстро.

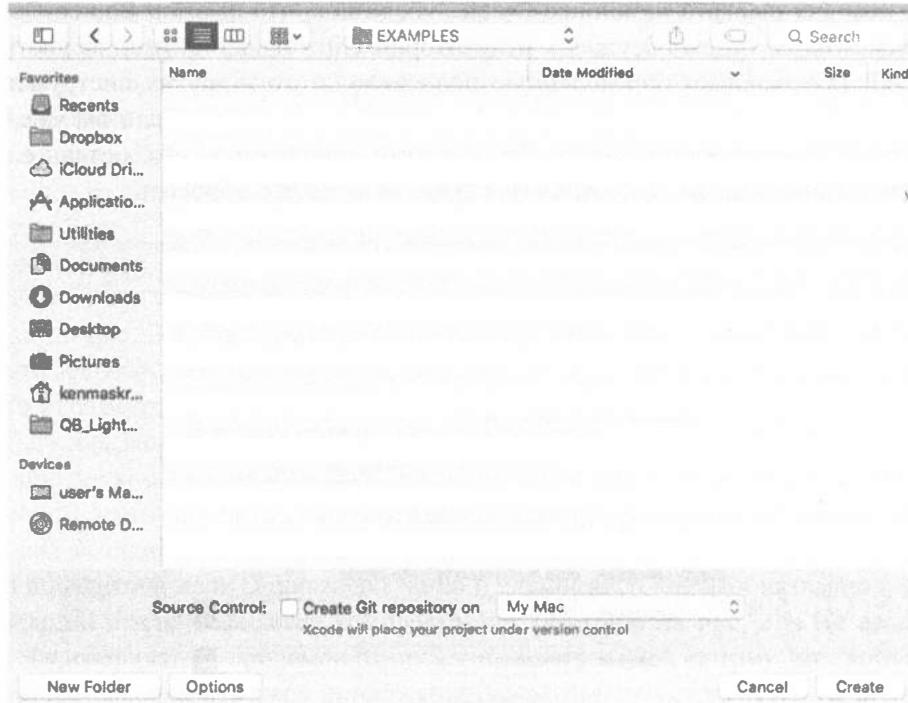


Рис. 2.5. Сохранение проекта в папке проектов на жестком диске



Рис. 2.6. Проект Hello World в среде Xcode

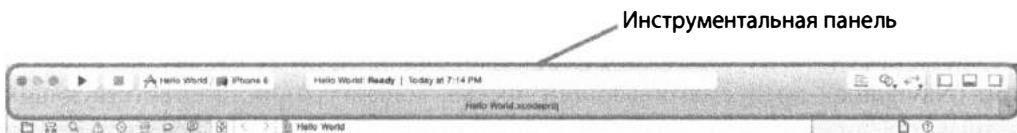


Рис. 2.7. Панель инструментов среды Xcode

Большое поле в середине панели инструментов называется **представлением действий** (action view). Как следует из этого названия, представление действий отображает любые действия или процессы, которые протекают в данный момент. Например, при запуске проекта представление действий содержит комментарии, описывающие разные этапы сборки приложения. Кроме того, здесь выводятся все сообщения об ошибках и предупреждения. Если щелкнуть на предупреждении или сообщении об ошибке, то можно немедленно оказаться в **навигаторе проблем**, который выдаст более подробную информацию о предупреждении или ошибке, как будет показано в следующем разделе.

Справа на панели инструментов содержатся два набора кнопок. Первый набор кнопок позволяет переключаться между тремя разными конфигурациями редактора.

- ☒ Кнопка **Editor Area** открывает окно, предназначенное для редактирования файла или параметров конфигурации проекта.
- ☒ Кнопка **Assistant Editor** разделяет область редактирования на несколько частей: левую, правую, верхнюю и нижнюю. Правое окно обычно используется для отображения файла, связанного с файлом, отображаемым в левом окне, или для отображения файла, который может понадобиться при редактировании файла, отображаемого в левом окне. Пользователь может самостоятельно задать, что именно должно отображаться в каждом из этих окон, или предоставить программе Xcode самой принять оптимальное решение. Например, если вы редактируете пользовательский интерфейс в левом окне, то программа Xcode автоматически покажет в правом окне код, с которым может взаимодействовать этот интерфейс. Этот режим редактора мы будем использовать на протяжении всей книги.

Кнопка **Version Editor** переводит окно редактора в режим работы, напоминающий утилиту Time Machine, которая работает с системами управления исходным кодом, такими как Git. Вы можете сравнивать текущую версию исходного файла с предыдущей версией или сравнивать любые две предшествующие версии одну с другой.

Набор кнопок редактора, расположенных справа, предназначен для того, чтобы отображать и скрывать панель навигатора, а также область отладки в левом и правом окнах области редактора.

Навигатор

Ниже панели инструментов в левой части окна проекта находится **навигатор** (Navigator), предусматривающий восемь конфигураций, соответствующих разным представлениям вашего проекта. Для того чтобы переключаться между разными конфигурациями, необходимо щелкнуть на одной из перечисленных ниже пиктограмм.

- **Навигатор проекта** (Project Navigator). Это представление содержит список файлов, используемых вашим проектом (рис. 2.8). Вы можете хранить ссылки на что угодно — от файлов исходного кода до рисунков, моделей данных, файлов списков свойств (.plist), которые рассматриваются в этой главе ниже, и даже файлов других проектов. Хранение нескольких проектов в одной и той же рабочей области облегчает совместное использование ресурсов. Если щелкнуть на каком-нибудь файле в представлении навигатора, то этот файл отобразится в окне редактора. Вы можете не только просматривать содержимое этого файла, но и редактировать его, если программа Xcode знает, как это сделать.

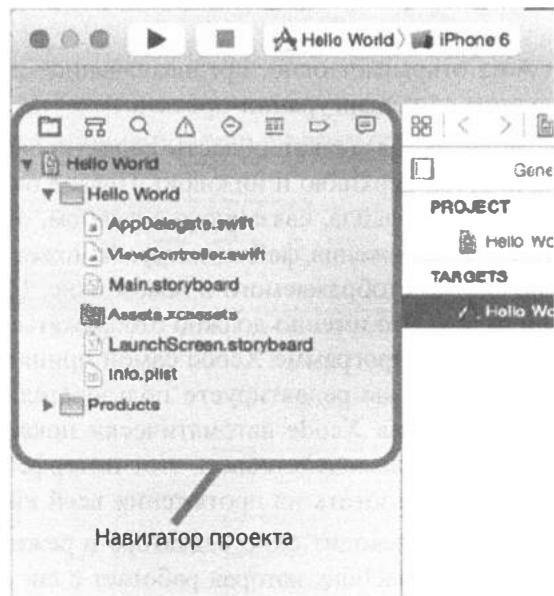


Рис. 2.8. Навигатор проекта Xcode. Для того чтобы перейти в другой навигатор, необходимо щелкнуть на одной из восьми кнопок в верхней части представления

- **Навигатор символов** (Symbol Navigator). Как следует из названия этого навигатора, он следит за **символами**, определенными в рабочей области (рис. 2.9). Символы — это объекты, которые распознаются компилятором, например классы, перечисления, структуры и глобальные переменные.

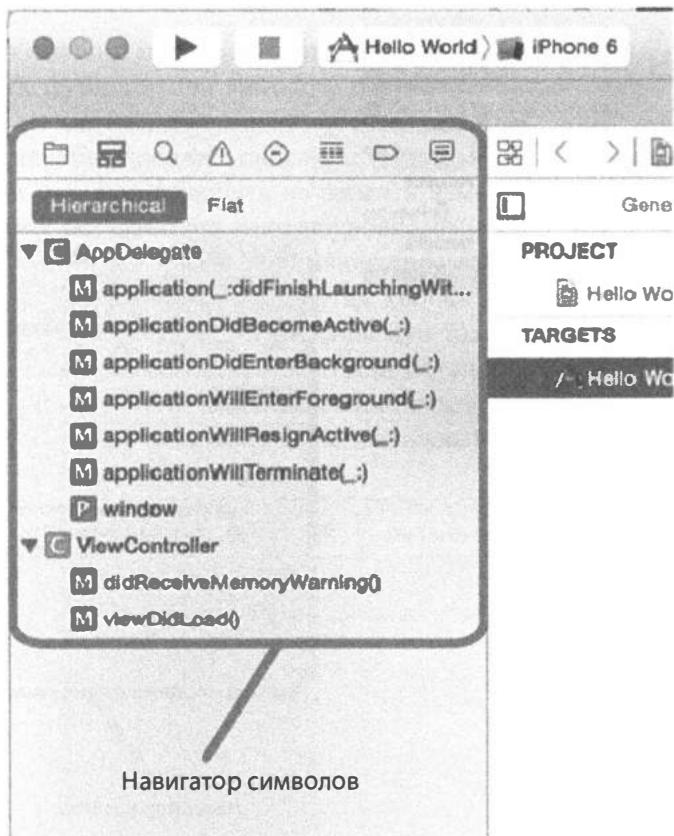


Рис. 2.9. Навигатор символов Xcode. Для того чтобы исследовать файлы и символы, определенные в каждой группе, необходимо щелкнуть на треугольнике раскрытия

- ❖ **Навигатор поиска (Search Navigator).** Используется для поиска всех файлов в рабочей области (рис. 2.10). В верхней части этого окна расположен многоуровневый элемент управления, позволяющий выбрать команду Replace вместо Find и применить критерии поиска к введенному тексту. Под полем редактирования находятся другие элементы управления, позволяющие выполнять поиск по всему проекту или по его части, а также указывать, следует ли при поиске учитывать регистр.
- ❖ **Навигатор проблем (Issue Navigator).** В этом навигаторе отображаются все сообщения об ошибках и предупреждения, возникающие при сборке проекта, а сообщения, детализирующие ошибки, выводятся в представлении действий, расположенным в верхней части экрана (рис. 2.11). Щелкнув на ошибке в окне навигатора проблем, вы перейдете в соответствующую строку кода в области редактирования.

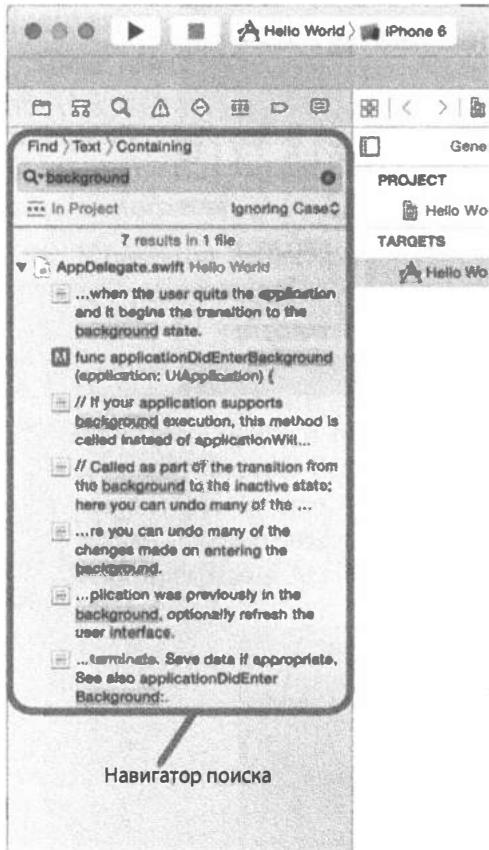


Рис. 2.10. Навигатор поиска Xcode. Для того чтобы выполнить поиск, установите флагки рядом с командами всплывающих меню, скрытых под словом “Find” и кнопками, расположенными ниже поля поиска

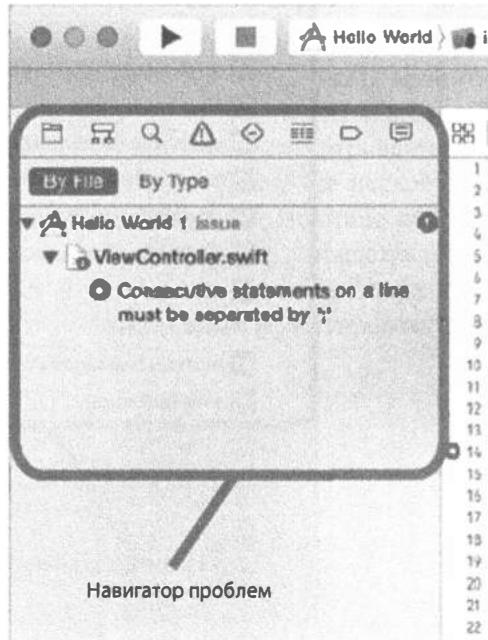


Рис. 2.11. Навигатор проблем Xcode. Здесь выводятся сообщения об ошибках и предупреждения

- **Навигатор тестирования** (Test Navigator). Если вы используете средства отладки, интегрированные в среду Xcode (эта тема в книге не рассматривается), то в этом окне появятся результаты модульного тестирования. Поскольку мы не включили модульное тестирование в свой проект, это окно остается пустым (рис. 2.12).
- **Навигатор отладки** (Debug Navigator). Этот навигатор представляет собой основной инструмент отладки вашей программы (рис. 2.13). Если вы впервые сталкиваетесь с процессом отладки, рекомендуем обратиться к документации Xcode Overview, расположенной по адресу https://developer.apple.com/library/prerelease/content/documentation/ToolsLanguages/Conceptual/Xcode_Overview/

UsingtheDebugger.html. Навигатор отладки содержит фрейм стека для каждого активного потока. **Фрейм стека** (stack frame) — это список ранее вызванных функций или методов, перечисленных в порядке их вызова. Щелкнув на методе, вы увидите в окне редактора связанный с ним код. Окно редактора содержит панель, с помощью которой можно управлять процессом отладки, выводить на экран и изменять значения переменных, а также получать доступ к низкоуровневому отладчику. Кнопка, расположенная в нижней части окна навигатора отладки, позволяет управлять визуализацией стека фрейма. Другая кнопка позволяет выбрать, следует ли показывать все потоки управления или только потоки, выполнение которых было аварийно прекращено или остановлено в точке прерывания. Для того чтобы понять предназначение каждой из этих кнопок, наведите на них курсор мыши и оставьте неподвижным на некоторое время.

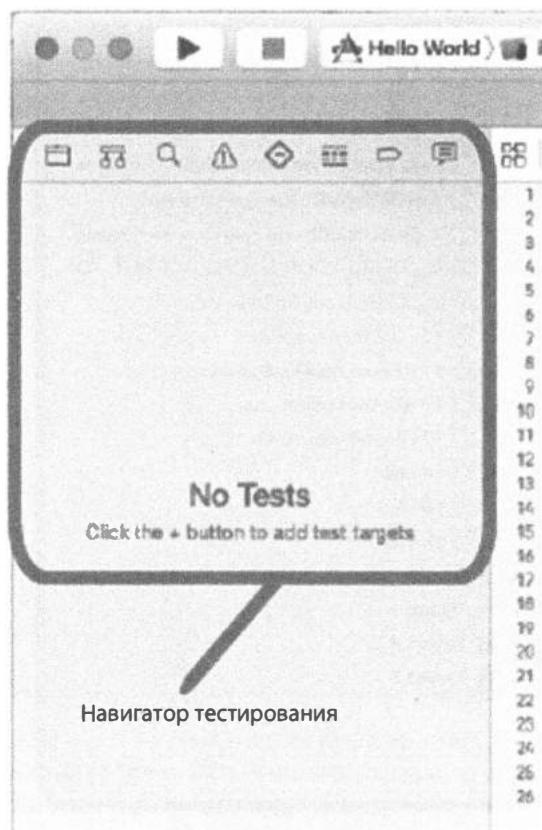


Рис. 2.12. Навигатор тестирования Xcode. Здесь выводятся результаты модульного тестирования

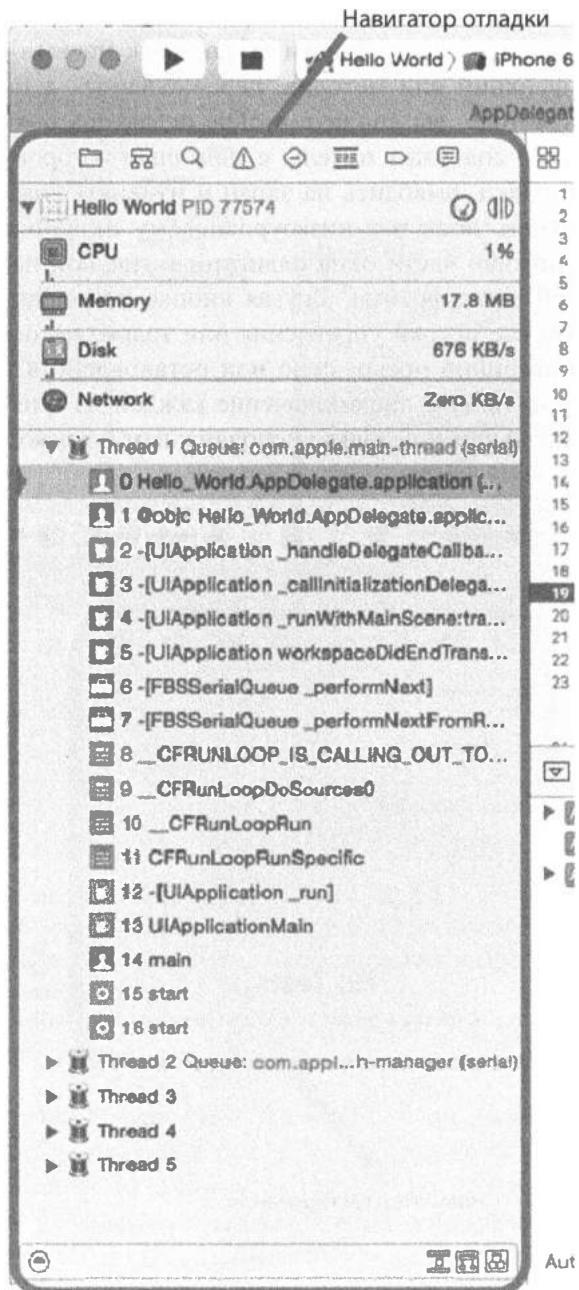


Рис. 2.13. Навигатор отладки Xcode. Элементы управления в нижней части окна позволяют определить уровень детализации информации об отладке, которую вы хотите видеть

- **Навигатор точек прерывания** (Breakpoint Navigator) позволяет увидеть все установленные вами точки прерывания (рис. 2.14). Как следует из названия, точками прерывания называются точки, в которых выполнение кода останавливается (**прерывается**), чтобы вы могли увидеть значения переменных и выполнить другие действия, необходимые для отладки программы. Список точек прерывания в навигаторе связан с содержимым исходного файла. Щелкните на точке прерывания в списке, и в окне редактора будет выделена соответствующая строка кода. Не забудьте щелкнуть на кнопке + в левом нижнем углу окна проекта при работе с навигатором прерываний. Эта кнопка позволяет добавлять исключения четырех типов, включая символьное, которое используется чаще всего.

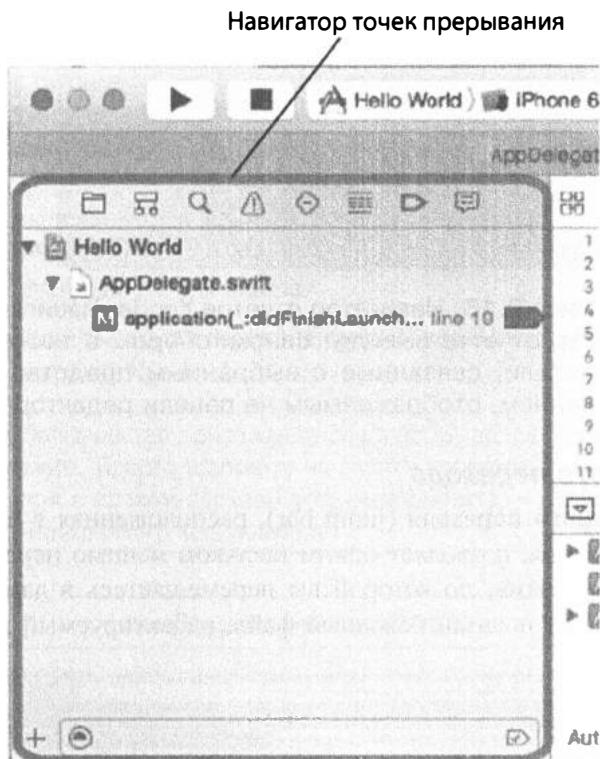


Рис. 2.14. Навигатор прерываний Xcode. Список точек прерывания связан с содержимым исходного файла

- **Навигатор отчетов** (Report Navigator). Сохраняет историю предыдущих сборок и попыток выполнения приложения (рис. 2.15). Щелкните на конкретном журнале, и на панели редактирования отобразятся команда сборки и вся связанная с этим информация.

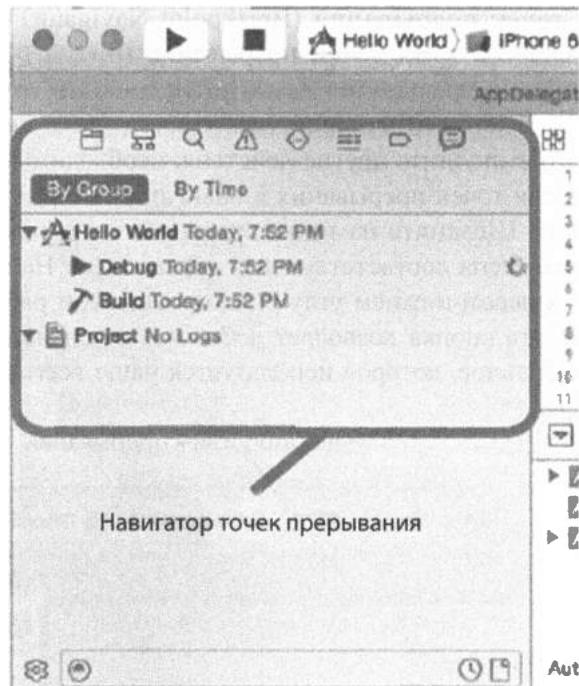


Рис. 2.15. Навигатор отчетов Xcode. Навигатор отчетов выводит списки сборок, а также детали, связанные с выбранным представлением, отображаемым на панели редактора

Панель быстрого перехода

Панель быстрого перехода (jump bar), расположенная в верхней части области редактирования, позволяет одним щелчком мышью перейти к конкретному элементу в иерархии, по которой вы перемещаетесь в данный момент. Например, на рис. 2.16 показан исходный файл, редактируемый в окне редактора.

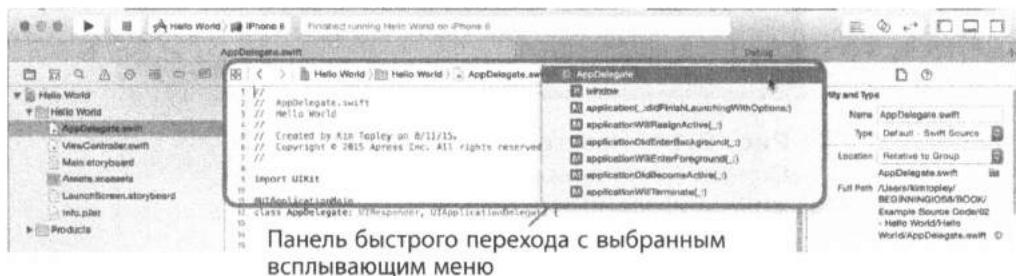


Рис. 2.16. Окно редактора Xcode с панелью быстрого перехода и выбранным файлом исходного кода. В подменю показан список методов в выбранном файле

Панель быстрого перехода появляется сразу над исходным кодом. Покажем, как она работает.

- ❖ Причудливая пиктограмма в левом конце панели быстрого перехода на самом деле является всплывающим меню, открывающим подменю, в котором перечислены недавно открывавшиеся файлы, эквиваленты, суперклассы и подклассы, категории, включения и т.д. Подменю позволяет открыть любой код, связанный с редактируемым кодом.
- ❖ Справа от необычного меню расположены стрелки, направленные влево и вправо, которые позволяют переходить к предыдущему и следующему файлам соответственно.
- ❖ Панель быстрого перехода содержит сегментированное всплывающее меню, отображающее иерархический путь к выбранному файлу в проекте. Щелкнув на любом имени группы или файла, можно увидеть другие файлы и группы, расположенные на том же уровне иерархии. Последний сегмент демонстрирует список элементов, содержащихся в выбранном файле. В конце панели быстрого перехода (рис. 2.16) расположено всплывающее меню, в котором перечисляются методы и другие символы, содержащиеся в файле, выделенном в данный момент. На панели быстрого перехода показан файл AppDelegate.swift с подменю, содержащем список символов, определенных в этом файле.

С помощью панели быстрого перехода можно перемещаться по разным элементам интерфейса среды Xcode.

ЗАМЕЧАНИЕ. Как и большинство приложений для macOS, среда Xcode поддерживает полноэкранный режим. Просто щелкните на кнопке перехода в полноэкранный режим, расположенной в правом верхнем углу окна проекта, и попробуйте испытать преимущества полноэкранного кодирования!

Комбинации клавиш XCODE

Если вы предпочитаете перемещаться по элементам интерфейса не с помощью мыши, а используя комбинации клавиш, то среда Xcode вам понравится. Большинство действий, которые вы можете регулярно выполнять в среде Xcode, имеют назначенные им стандартные комбинации клавиш, например комбинация клавиш <⌘+B> выполняет сборку приложения, а комбинация клавиш <⌘+N> создает новый файл.

Вы можете изменить назначенные комбинации клавиш и задать собственные, которых нет среди стандартных комбинаций, перечисленных на вкладке *Key Binding*. Действительно удобной является комбинация <Shift+⌘+O>, выполняющая команды быстрого открытия файла. Нажав ее, наберите имя файла, настройки или символа, и среда Xcode откроет список параметров. Когда вы сузите список до файла, который вам нужен, нажмите клавишу <Return> — и откроется панель редактирования. Таким образом, вы сможете открыть файл всего несколькими нажатиями клавиш.

Вспомогательная область

Как указывалось ранее, предпоследняя кнопка в правой части панели инструментов программы Xcode открывает и закрывает область утилит. Подобно инспектору, вспомогательная область является контекстно-чувствительной и зависит от содержимого панели редактора. Примеры ее использования мы еще встретим в книге.

Программа *Interface Builder*

Предыдущие версии программы Xcode включали инструмент для разработки интерфейса под названием “**Interface Builder**” (IB), позволявший разрабатывать и настраивать собственный пользовательский интерфейс. Одно из основных изменений в последних версиях программы Xcode заключается в интеграции компонента Interface Builder с самой рабочей областью. Программа Interface Builder больше не является самостоятельным приложением, т.е. пользователю теперь не приходится метаться между Xcode и Interface Builder при разработке своего кода и интерфейса.

Мы будем широко использовать функциональную возможность разработки интерфейса, которой обладает среда Xcode, заглядывая во все ее потаенные места. Фактически первая программа, которую мы напишем немного позднее в этой главе, будет заключаться именно в создании пользовательского интерфейса.

Интегрированный компилятор и отладчик

Одни из наиболее важных изменений в программе Xcode 4 скрыты от посторонних глаз: это принципиально новый компилятор и низкоуровневый отладчик.

Многие годы компания Apple использовала в качестве основного компилятора GCC (GNU Compiler Collection). Однако в последние несколько лет компания перешла на совершенно новый компилятор LLVM (Low Level Virtual Machine), который генерирует код, работающий быстрее, чем код, генерируемый компилятором GCC. Компилятор LLVM не только создает более быстрый код, но и обладает более полной информацией о нем и поэтому может генерировать более содержательные сообщения об ошибках и предупреждения.

Среда Xcode тесно интегрирована с компилятором LLVM. Это обстоятельство открывает широкие возможности. В частности, среда Xcode также обеспечивает более точное автодополнение кода и может делать обоснованные предположения о реальном предназначении того или иного фрагмента кода при генерировании предупреждений, открывая при этом всплывающее меню, содержащее средства для исправления ошибок. Благодаря этому обнаруживать и исправлять такие ошибки, как опечатки в именах символов, нарушение баланса скобок и пропущенные точки с запятыми, стало очень просто.

Компилятор LLVM содержит сложный **статический анализатор** (static analyzer), способный сканировать код в поисках различных потенциальных проблем, включая проблемы, связанные с управлением памятью. Фактически компилятор LLVM настолько совершенен, что может самостоятельно решать

большинство задач управления памятью на основе простых правил, если при написании кода вы будете соблюдать несколько простых правил. Новый механизм **автоматического подсчета ссылок** (Automatic Reference Counting — ARC) будет рассмотрен в следующей главе.

Приступим к проекту Hello World

Итак, изучив окно рабочей области Xcode, взглянем на файлы, образующие проект Hello World. Переключимся в навигатор проекта, щелкнув на крайней слева пиктограмме среди восьми пиктограмм навигаторов, расположенных в левой части рабочей области (см. раздел “Представление навигаторов”), или нажав комбинацию клавиш `<⌘+1>`.

ЗАМЕЧАНИЕ. Восемь конфигураций навигаторов можно открыть с помощью комбинаций клавиш `<⌘+1>`-`<⌘+8>`. Числа соответствуют номерам пиктограмм, считая слева направо, так что комбинация `<⌘+1>` соответствует пиктограмме навигатора проекта, а `<⌘+8>` — навигатору отчетов.

Первый элемент в списке навигатора проекта должен иметь такое же имя, как и проект, в данном случае — Hello World. Этот элемент представляет собой весь проект, а также место для настройки его конфигурации. Щелкнув на нем один раз, вы сможете редактировать многочисленные параметры конфигурации проекта с помощью редактора программы Xcode. Однако конкретные параметры конфигурации нас пока не будут интересовать. Нас вполне устроят параметры, установленные по умолчанию.

Взгляните на рис. 2.8 и обратите внимание на то, что треугольник раскрытия слева от папки Hello World направлен вниз и у этой папки есть несколько вложенных папок, которые в среде Xcode называются **группой** (group).

- **Папка Hello World** — это первая группа, которая всегда носит название проекта. С ней вы будете работать много времени. Именно здесь будет храниться большая часть написанного вами кода в виде файлов, образующих пользовательский интерфейс приложения. Вы можете создать вложенные папки в папке Hello World, чтобы лучше организовать свою программу, и даже использовать другие группы, если предпочитаете другой способ организации проекта. Хотя мы не планируем касаться этих файлов, пока не перейдем к следующей главе, один из них мы все же опишем в следующем разделе, когда будем говорить о программе Interface Builder. Этот файл называется **Main.storyboard** и содержит элементы пользовательского интерфейса, относящиеся к контроллеру главного представления вашего проекта. Группа Hello World также содержит файлы и ресурсы, не являющиеся исходным кодом на языке Swift, но необходимые для проекта. В частности, таким файлом является файл **Info.plist**, содержащий важную информацию о приложении, например его название, требования к

устройствам и т.д. В более ранних версиях среды Xcode эти файлы размещались в отдельной группе под названием *Supporting Files*.

- ❶ **Папка Hello WorldTests.** Эта группа предназначена для модульного тестирования проекта (напоминаем, что в нашем проекте модульное тестирование не предусмотрено). Она содержит файлы, необходимые для создания модульных тестов для вашего приложения. В этой книге модульное тестирование не рассматривается, но это не значит, что оно бесполезно. Это превосходный инструмент, который необходим для любого проекта. Как и в папке Hello world, в папке Hello Wordltests содержится свой файл Info.plist.
- ❷ **Папка Products.** Эта группа содержит приложение, которое создается проектом во время компиляции. Если вы откроете файл Products, то увидите элемент Hello World.app. Это приложение, которое создал данный проект. Если проект был создан при включенном модульном тестировании, то папка Products будет также содержать файл Hello WorldTests.xctest, представляющий тестовый код. Оба эти файла называются **целями сборки (build target)**. В данный момент имена этих файлов выделены красным цветом, потому что мы их еще не компилировали. Выделяя имя файла красным цветом, программа Xcode сообщает нам, что она не может найти указанный файл.

ЗАМЕЧАНИЕ. Папки в области навигатора не обязательно соответствуют папкам файловой системы вашего компьютера Mac. Они представляют собой логические группы в среде Xcode, помогающие организовать файлы, а также ускорить и облегчить их поиск при работе над приложением. Довольно часто элементы, содержащиеся в двух папках проекта, хранятся прямо в корневом каталоге, хотя вы можете хранить их где угодно, даже за пределами папки проекта. Иерархия в среде Xcode совершенно не зависит от иерархии файловой системы. Например, перемещая файл из группы Hello World в среде Xcode, вы не изменяете его физического расположения на жестком диске.

Введение в программу Interface Builder

Зайдите в навигатор проекта вашей рабочей области, раскройте группу Hello World, если она еще не открыта, а затем выберите файл Main.storyboard. В результате в окне редактора откроется файл (рис. 2.17). В центре вы увидите область, напоминающую контур устройства iOS, которая очень удобна для редактирования интерфейса. Это и есть компонент среды Xcode под названием Interface Builder, с помощью которого разрабатываются пользовательские интерфейсы приложений.

Программа Interface Builder имеет долгую историю. Она появилась в 1988 году и использовалась для разработки приложений для операционных систем NeXTSTEP, OpenSTEP и macOS, а теперь и для устройств, работающих под управлением системы iOS, таких как iPhone, iPad, AppleTV и Apple Watch.

Форматы файлов

Программа Interface Builder поддерживает два типа файлов: старый формат, использующий расширение .nib, и новый XML-формат, который использует расширение .xib. Оба эти формата содержат одну и ту же информацию, но версия .xib является текстовой, что дает ей множество преимуществ, особенно при использовании программного обеспечения для контроля исходного кода. На протяжении более двадцати лет существования программы Interface Builder все ее интерфейсные файлы имели расширение .nib, и в результате большинство разработчиков по привычке называют их **nib-файлами** независимо от их реального расширения — .xib или .nib. Сама компания Apple в своей документации использует термины **nib** и **nib-файл**.

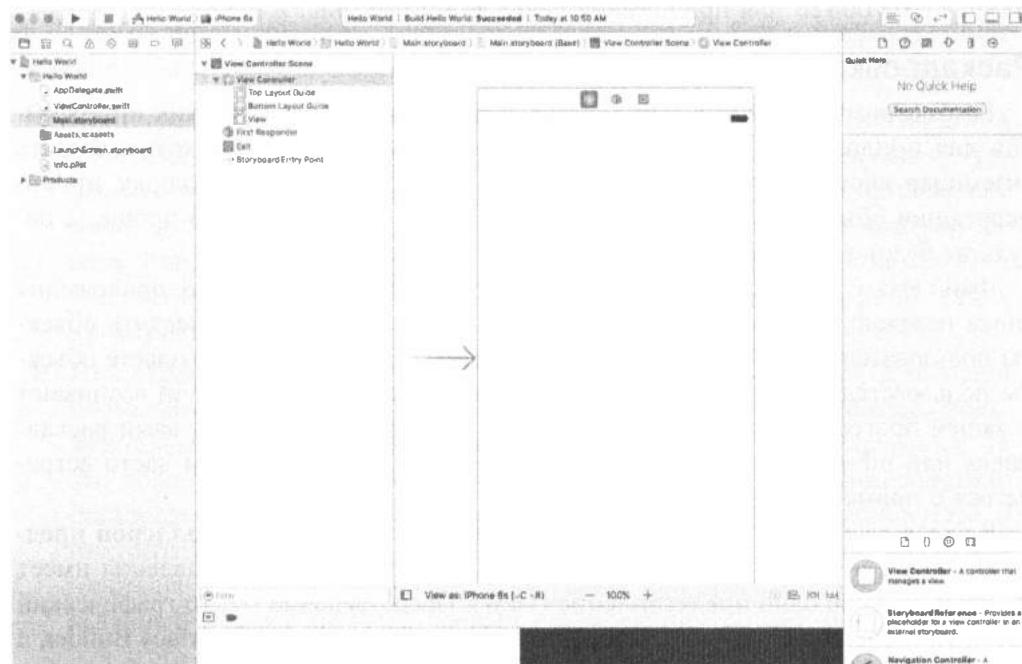


Рис. 2.17. Выбрав файл Main.storyboard в навигаторе проекта, можно открыть файл в программе Interface Builder

Каждый nib-файл может содержать любое количество объектов, но при работе с проектами для системы iOS, он обычно содержит только одно представление (часто полноэкранное) и контроллеры или другие объекты, связанные с представлением. Это позволяет разделить приложения на части, загружая nib-файл только для того представления, которое требуется для вывода на экран. В результате мы экономим память при выполнении приложения на устройстве iOS с ограниченной памятью. Вновь созданный проект для системы iOS содержит nib-файл с именем LaunchScreen.xib, содержащий макет экрана, который

будет демонстрироваться по умолчанию при запуске приложения. Об этом мы подробнее поговорим в конце главы.

Второй формат файлов, который программа Interface Builder поддерживает уже несколько лет, — **раскладровка** (storyboard). Раскладровку можно интерпретировать как расширенный nib-файл, потому что она содержит несколько контроллеров представления, а также информацию о том, как каждое из них связано с другими в ходе выполнения приложения. В отличие от nib-файла, содержимое которого загружается целиком и сразу, раскладровка не может содержать изолированных представлений и никогда не загружает все их содержимое одновременно. Все шаблонные проекты в среде Xcode 8 используют раскладровки, поэтому все примеры в книге начинаются именно с них. Раскладровки можно использовать по одной или сразу несколько. Вернемся к программе Interface Builder и файлу Main.storyboard для приложения Hello World (см. рис. 2.17).

Раскладровка

Рассмотрим основной инструмент для создания пользовательских интерфейсов для приложений в системе iOS. Теперь допустим, что мы хотим создать экземпляр кнопки. Его можно создать, написав код, но создать кнопку, просто перетащив объект из библиотеки и указав ее атрибуты, намного проще, а результат будет тот же самый.

Файл Main.storyboard загружается автоматически при запуске приложения (пока неважно, как именно), поэтому именно в него следует поместить объекты пользовательского интерфейса вашего приложения. Когда вы создаете объекты пользовательского интерфейса в программе Interface Builder, они возникают в вашей программе в тот момент, когда загружается добавленная вами раскладровка или nib-файл. На протяжении нашего изложения мы будем часто встречаться с примерами этого процесса.

Каждая раскладровка состоит из одного или нескольких **контроллеров представления** (view controllers), причем каждый контроллер представления имеет по крайней мере одно **представление** (view). Представление — это графический элемент, который можно видеть и редактировать в программе Interface Builder, а контроллер — это код приложения, который вы должны написать, чтобы приложение реагировало на действия пользователя. Именно контроллеры выполняют реальные действия вашего приложения.

В программе Interface Builder представление часто изображается в виде прямоугольника, имитирующего экран устройства iOS (на самом деле прямоугольник соответствует контроллеру представления, концепцию которого мы опишем в следующей главе, но в данном конкретном случае контроллер представления охватывает весь экран устройства, поэтому на такие тонкости можно не обращать внимания). В нижней части окна Interface Builder расположен раскрывающийся элемент управления **View as:**, в котором указан тип устройства по умолчанию. Он позволяет выбрать устройство, для которого вы собираетесь разработать макет экрана, как показано на рис. 2.18.

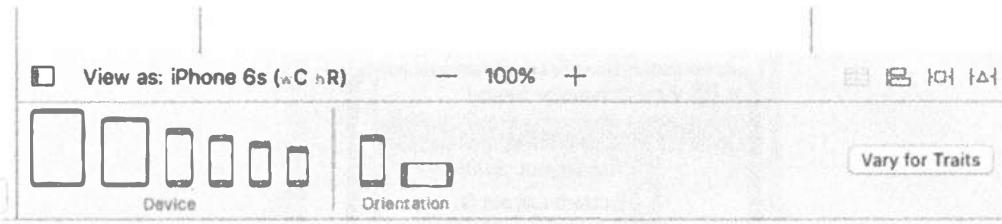


Рис. 2.18. В программе Interface Builder среды Xcode 8 можно выбрать тип устройства и его ориентацию

Вернувшись в разметку, щелкните в любом месте макета — и вы увидите в верхней части ряд из трех пиктограмм (см. рис. 2.17). Наведите на них курсор мыши — и появятся подсказки с их именами: *View Controller*, *First Responder* и *Exit*. Пока не будем обращать внимание на пиктограмму *Exit* и сосредоточимся на пиктограммах *View Controller* и *First Responder*.

- ❖ Пиктограмма *View Controller* представляет собой объект, загружающийся из файла, связанного с представлением. Задача этой пиктограммы — управлять тем, что пользователь видит на экране. Типичное приложение имеет несколько контроллеров представления, по одному для каждого экрана. Разумеется, можно написать приложение с одним экраном, а значит, с одним контроллером представления, и мы увидим в книге много таких примеров.
- ❖ Пиктограмма *First Responder* — это, в общих чертах, объект, с которым пользователь взаимодействует в данный момент. Если, например, пользователь в текущий момент вводит данные в поле редактирования, то это поле является первым реагирующим объектом. При взаимодействии пользователя с интерфейсом первый реагирующий объект изменяется и пиктограмма *First Responder* предоставляет удобную возможность для обмена информацией с элементом управления или представлением, которое в данный момент является первым реагирующим объектом и не создает для этого код.

Мы обсудим эти объекты в следующей главе, поэтому пока не стоит беспокоиться о том, что их предназначение вам не совсем понятно.

Кроме этих пиктограмм, область редактирования содержит пространство, на котором можно размещать графические объекты. Однако, прежде чем мы перейдем к нему, отметим еще одно важное обстоятельство, касающееся области редактирования, — иерархическое представление, или, точнее окно *Document outline*, которое показано на рис. 2.19.

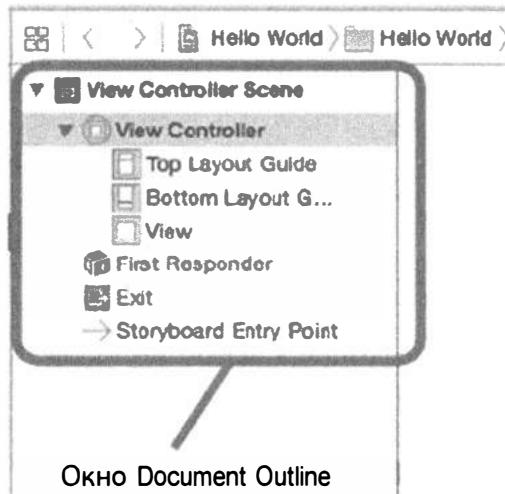


Рис. 2.19. Окно Document Outline содержит полезное иерархическое представление раскладовки

Если окно Document Outline скрыто, щелкните на маленькой кнопке в левом нижнем углу области редактирования. В открывшемся окне вы увидите содержимое раскладовки, разделенное на **сцены** (scenes), содержащие фрагменты связанного контента. В данном случае вы увидите всего одну сцену с именем **View Controller Scene**. Она содержит элемент с именем **View Controller**, который, в свою очередь, содержит элемент с именем **View** (а также другие элементы, которые мы опишем позже). Это позволяет получить полный обзор элементов, содержащихся в области редактирования.

Пиктограмма **View** является объектом класса **UIView**. Объект класса **UIView** — это область, которую видит и с которой взаимодействует пользователь. Позднее мы будем разрабатывать более сложные приложения, имеющие несколько представлений, а пока просто представьте себе, что это то, что видит пользователь, когда работает с вашим приложением.

Если щелкнуть на пиктограмме **View**, то откроется окно, размер которого соответствует размеру экрана устройства iPhone (если это окно не было открыто ранее). В этом окне вы можете графически разрабатывать пользовательский интерфейс.

Вспомогательная область

Вспомогательная область занимает правую часть рабочей области. Если она еще не открыта, то щелкните на крайней справа кнопке **View** инструментальной панели, выберите команду **View**⇒**Utilities**⇒**Show Utilities** или нажмите комбинацию клавиш **<Option+⌘+0>** (option-command-zero). Нижняя часть вспомогательной области называется **панелью библиотеки** (library pane) или просто **библиотекой** (library) (рис. 2.20).

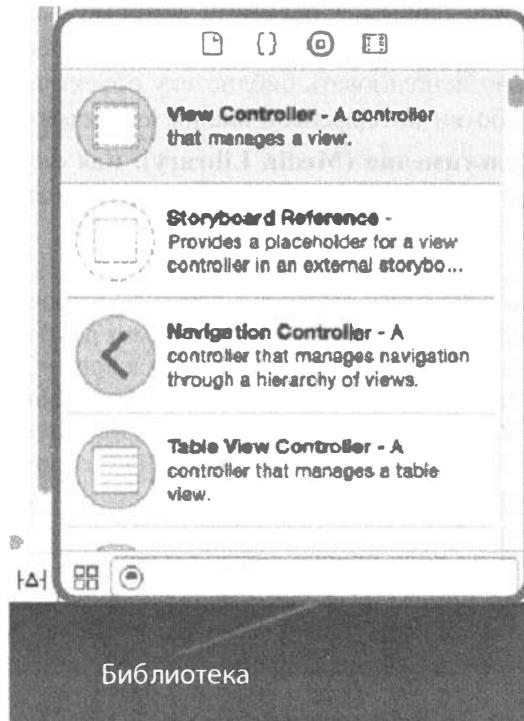


Рис. 2.20. Библиотека — это место хранения объектов из каркаса UIKit, доступного для использования в программе Interface Builder. Все, что расположено выше библиотеки, но ниже инструментальной панели, называется инспектором

Библиотека — это коллекция повторно используемых элементов, которые можно использовать в программе. Четыре пиктограммы на панели, расположенной вверху панели библиотеки, соответствуют четырем разделам библиотеки.

- ❖ **Библиотека файловых шаблонов (File Template Library).** Этот раздел содержит коллекцию файловых шаблонов, которые можно использовать, когда необходимо добавить в проект новый файл. Например, если вы хотите добавить в проект новый класс из языка Swift, перетащите в него файл класса, который находится в библиотеке файловых шаблонов.
- ❖ **Библиотека синипетов кода (Code Snippet Library).** В этом разделе хранится коллекция синипетов кода, которые можно перетаскивать в свой исходный файл. Если вы не можете вспомнить синтаксис быстрого перечисления в языке Swift, то просто перетащите нужный синипет из библиотеки.
- ❖ **Библиотека объектов (Object Library).** Этот раздел заполнен повторно используемыми объектами, например полями редактирования, ползунками,

кнопками и просто любыми объектами, которые вы хотите использовать при разработке своего пользовательского интерфейса для системы iOS. Мы будем интенсивно использовать библиотеку объектов на протяжении всей книги для разработки интерфейсов наших иллюстративных программ.

- **Библиотека мультимедиа (Media Library).** Как следует из этого названия, здесь хранятся все медиафайлы, включая изображения, звуковые файлы и фильмы.

ЗАМЕЧАНИЕ. Элементы библиотеки объектов в основном извлекаются из каркаса iOS, представляющего собой каркас объектов, используемых для создания пользовательского интерфейса приложений. Каркас UIKit играет в среде Cocoa Touch такую же роль, как каркас AppKit в среде Cocoa. Эти два каркаса принципиально похожи, но из-за различий между платформами между ними существует большое различие. С другой стороны, классы каркаса Foundation, такие как `NSString` и `NSArray`, являются общими для каркасов Cocoa и Cocoa Touch.

Обратите внимание на поле поиска, расположенное в нижней части библиотеки. Вы хотите найти кнопку? Наберите в поле поиска слово `button`, и текущая библиотека покажет вам только объекты, в именах которых есть слово `button`. Когда закончите поиск, не забудьте очистить поле поиска.

Добавление метки на представление

Испытаем программу Interface Builder. Щелкните на пиктограмме библиотеки объектов (она выглядит как круг с вписанным квадратом) в верхней части панели библиотеки (рис. 2.20). Теперь просмотрите библиотеку в поисках объекта `Table View`. Для этого можете прокрутить содержимое окна, но есть более эффективное средство — наберите слова `Table View` в поле поиска.

СОВЕТ. Для перехода в поле поиска есть удобная комбинация клавиш — `<^+Option+⌘+3>`. После этого достаточно ввести в поле поиска имя искомого объекта.

Найдите в библиотеке объект `Label`. Он должен находиться в верхней части списка. Перетащите метку на представление, которое вы открыли ранее. (Если вы не видите это представление на панели редактирования, щелкните на пиктограмме `View` в окне Document Outline в программе Interface Builder.) Как только курсор появится на представлении, отобразится стандартный зеленый знак “плюс”, который вам должен быть известен по работе с окном Finder и который означает копирование. Перетащите кнопку в центр представления. Когда будете центрировать метку, на экране появятся голубые вертикальная и горизонтальная линии. То, что метка центрируется, не важно, но важно знать, что эти линии существуют. На рис. 2.21 показана рабочая область непосредственно после того, как вы отпустили кнопку мыши.

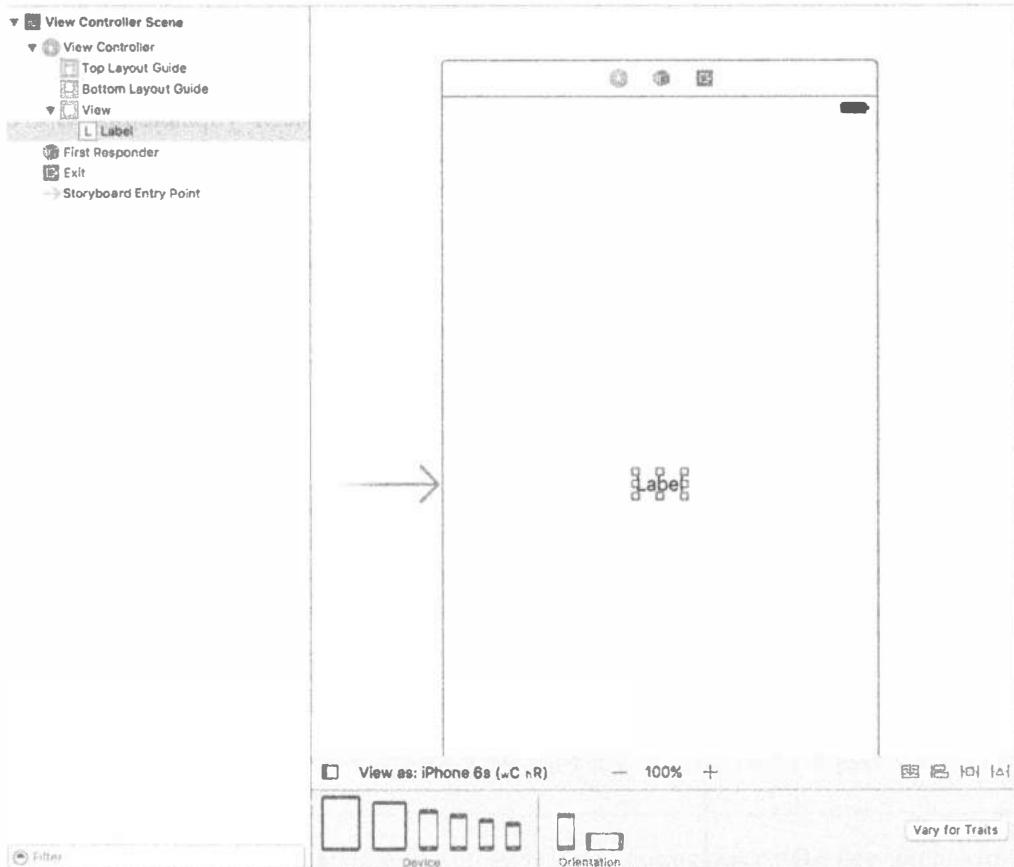


Рис. 2.21. Мы нашли метку в библиотеке и перетащили ее на представление

Объекты пользовательского интерфейса являются иерархическими. Большинство представлений могут содержать **дочерние представления** (subview), хотя некоторые объекты, такие как кнопки и большинство других элементов управления, не имеют дочерних представлений. Программа Interface Builder организована разумно. Если объект не имеет дочерних представлений, то на него нельзя перетаскивать другие объекты.

Добавим нашу метку на главное представление (которое называется *View*). В результате перетаскивания метки из библиотеки на представление *View* в главное представление добавляется новый объект класса *UILabel*, являющийся дочерним представлением.

Давайте отредактируем метку, чтобы она сообщала что-то полезное. Дважды щелкните на только что созданной метке и наберите текст *Hello, World!* Затем щелкните на экране за пределами метки, выберите ее снова и перетащите в центр или в ту часть экрана, где хотите ее видеть.

Как только вы сохранили файл, работа будет завершена. Выберите команду *File*⇒*Save* или нажмите комбинацию клавиш <⌘+S>. Затем щелкните на всплывающем меню, расположенном в левом верхнем углу окна проекта Xcode. По существу, это многосегментный всплывающий элемент управления. В его левой части можно выбрать одну из многих целей компиляции и еще несколько важных аспектов, но нас в данный момент больше интересует его правая часть, позволяющая выбирать устройство, для которого предназначено приложение. Щелкните на правой части меню — и увидите список доступных устройств. Если к вашему компьютеру подключено какое-нибудь устройство iOS, то вы увидите его в списке. В противном случае вы увидите обобщенный пункт *iOS Simulator*. Под ним расположен раздел с заглавием *iOS Simulator*, в котором перечислены все виды устройств, предусмотренных в симуляторе iOS. Выберите в последнем разделе пункт *iPhone 6/6s*, чтобы наше приложение было настроено на устройство *iPhone 6/6s*.

Существует несколько способов запустить приложение. Например, можно выбрать команду *Product*⇒*Run* или нажать комбинацию клавиш <⌘+R> или щелкнуть на кнопке *Run*, расположенной слева от всплывающего меню симулятора. Среда Xcode скомпилирует наше приложение и запустит его в симуляторе устройства *iPhone*, как показано на рис. 2.22.

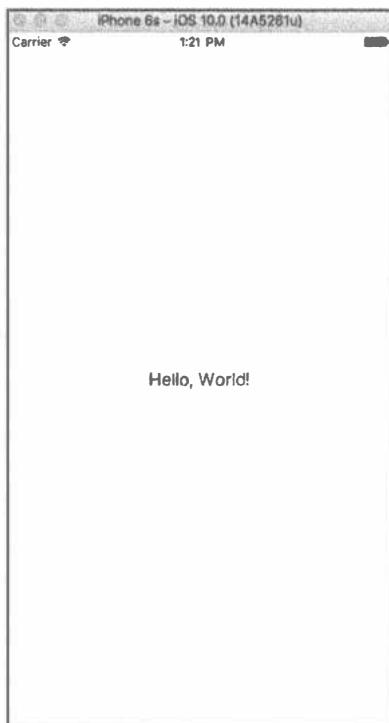


Рис. 2.22. Программа Hello World на устройстве iPhone

ЗАМЕЧАНИЕ. До появления среды Xcode 8 текст не центрировался автоматически, и приходилось использовать инструмент Auto Layout, чтобы обеспечить выполнение указанных требований на всех устройствах.

Это почти все, что необходимо знать для создания первого приложения. Обратите внимание на то, что мы до сих пор не написали ни одной строчки на языке Swift.

Изменение атрибутов

Вернитесь к программе Xcode и один раз щелкните на метке Hello World, чтобы выделить ее. Теперь обратите внимание на область, расположенную выше панели библиотеки. Эта часть вспомогательной панели называется **инспектором** (inspector). Как и библиотека, окно инспектора заполнено рядами пиктограмм, каждая из которых изменяет представление инспектора в зависимости от конкретного типа данных. Для того чтобы изменить атрибуты метки, нам нужна четвертая пиктограмма слева, которая открывает окно инспектора атрибутов, как показано на рис. 2.23.

СОВЕТ. Инспектор, как и навигатор проекта, имеет свою комбинацию клавиш для каждой из пиктограмм. Комбинации клавиш для пиктограмм на панели инспектора начинаются с `<Option+⌘+1>` для крайней слева, `<Option+⌘+2>` — для следующей и т.д. В отличие от окна навигатора проекта, количество пиктограмм в окне инспектора является контекстно-зависимым и изменяется в зависимости от того, какой объект выбран в навигаторе и/или в редакторе.

Попробуйте изменить внешний вид метки по своему вкусу. В окне инспектора можно, например, изменить шрифт, размер и цвет текста. Обратите внимание на то, что если вы изменяете размер текста, то вам, возможно, придется изменить размер метки, чтобы она могла вместить более крупный текст. Для этого выберите метку, а затем выберите команду `Editor⇒Size to Fit Content` в меню среды Xcode (рис. 2.24). Закончив эксперименты, сохраните файл и выберите команду Run снова. Внесенные вами изменения отразятся на приложении, причем и на этот раз вам не придется писать ни одной строчки кода.

ЗАМЕЧАНИЕ. Не слишком беспокойтесь о назначении всех без исключения атрибутов объектов. Читая книгу, вы еще много узнаете об окне инспектора атрибутов и о значении полей.

Позволяя вам разрабатывать свой интерфейс с помощью графических средств, программа Interface Builder освобождает вас от утомительной необходимости писать код для реализации пользовательского интерфейса, предоставляя возможность сосредоточиться только на специфике приложения.

Большинство современных сред для разработки приложений имеют инструмент графического проектирования интерфейса. Одно из отличий программы

Interface Builder от многих из этих сред разработки заключается в том, что она не генерирует никакого кода, который требовал бы сопровождения. Вместо этого программа Interface Builder создает объекты пользовательского интерфейса так, как будто вы их запрограммировали, а затем сериализует их в раскадровке или nib-файле, чтобы загрузить в память во время выполнения программы. Это позволяет избежать многих проблем, связанных с генерированием кода, и, безусловно, является более мощным методом разработки.

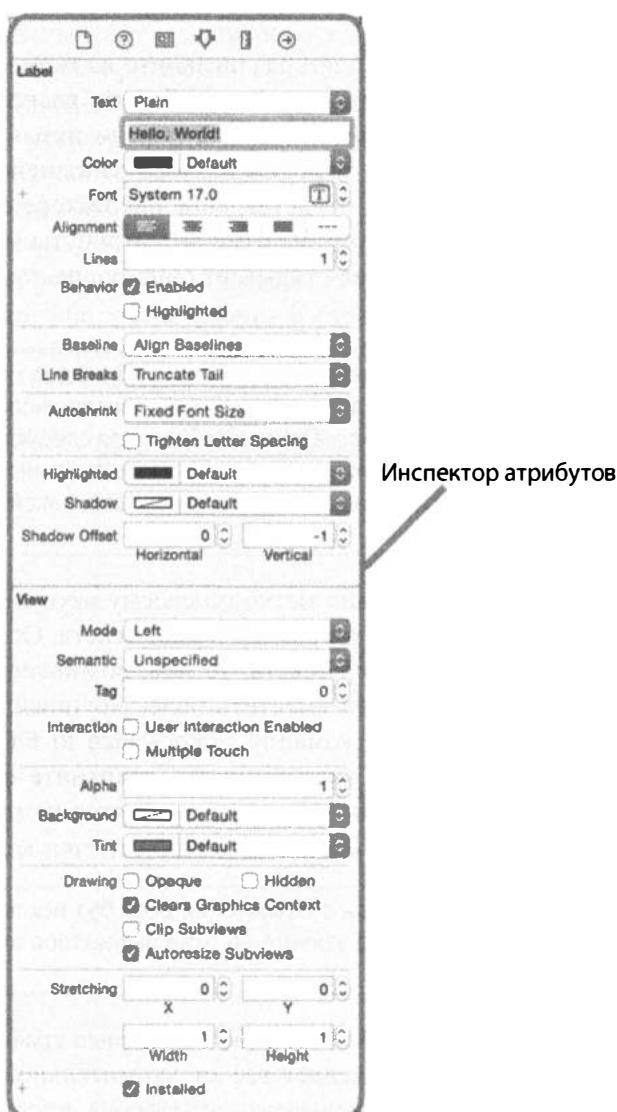


Рис. 2.23. Окно инспектора, демонстрирующее атрибуты метки

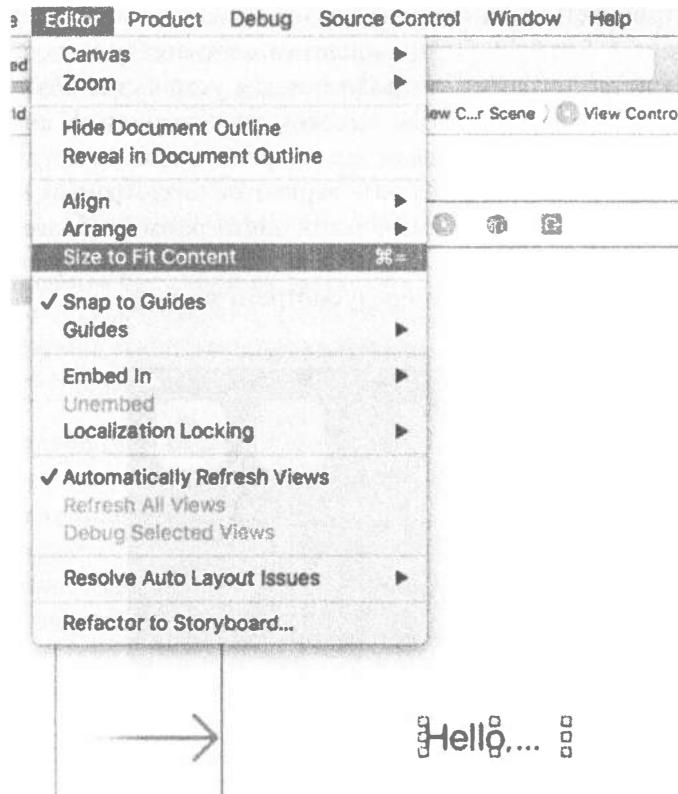


Рис. 2.24. Изменение размера шрифта на больший вынудило нас изменить ограничения макета, выбрав команду *Size to Fit Content* в выпадающем меню *Editor*

Завершающие штрихи

Прежде чем закончить главу, приадим нашему приложению немного блеска, чтобы сделать его похожим на аутентичное приложение для системы iOS. Запустите свой проект. Когда на экране появится окно симулятора, щелкните на кнопке Home симулятора устройства iPhone (это черная кнопка с белым квадратом, расположенная в самом низу окна). Это вернет вас к домашнему экрану устройства iPhone (рис. 2.25). Обратите внимание на то, что пиктограмма приложения выглядит, как обычное изображение, заданное по умолчанию.

Взгляните на пиктограмму Hello World в верхней части экрана. Для того чтобы заменить ее стандартную скучную пиктограмму новой, необходимо создать новую пиктограмму и сохранить ее в файле, имеющем формат Portable Network Graphic (.png). Лучше всего создать пять пиктограмм: размером 180×180, 120×120, 87×87, 80×80 и 58×58 пикселей. Если вы планируете запускать свое приложение на iPad, понадобятся еще четыре пиктограммы. Для iPadPro дополнительно потребуется изображение 187×187 пикселей. Дело в том, что эти пиктограммы

будут демонстрироваться на главном экране, в настройках *Settings* и в списке результатов поиска *Spotlight*. Этим вариантам соответствуют первые три пиктограммы. Еще три пиктограммы потребуются для устройства iPhone 6/6s, которое имеет более крупный экран с более высоким разрешением. К счастью, размеры одной из этих пиктограмм совпадают с размерами одной из предыдущих, поэтому на самом деле нам понадобятся пять вариантов пиктограммы приложения для iPhone. Если пропустить одну из маленьких пиктограмм, то более крупные будут автоматически масштабированы. Однако для достижения наилучших результатов это масштабирование желательно предусмотреть заранее.

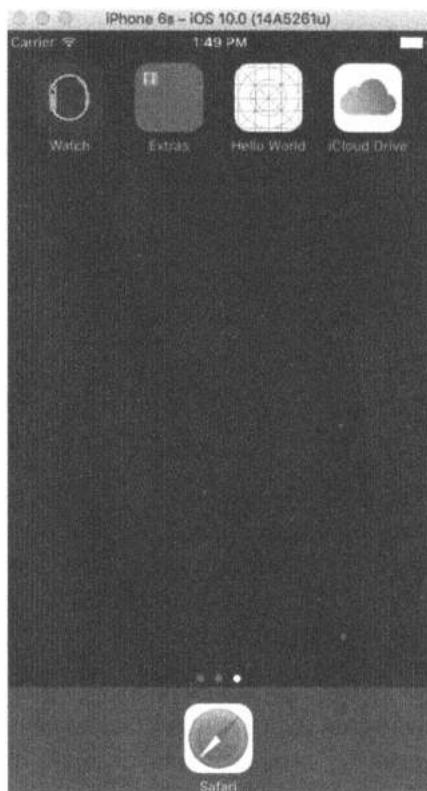


Рис. 2.25. Приложение Hello World на главном экране

ЗАМЕЧАНИЕ. На самом деле проблема размеров пиктограмм еще сложнее. До появления системы iOS 7 размеры пиктограмм для всех современных устройств iPhone были равны 114x114 пикселей. Если же вы захотите обеспечить поддержку более старых моделей, не имеющих экрана Retina, то вам будут нужны пиктограммы вдвое меньшего размера — 57x57. Кроме того, существует устройство iPad, имеющее совсем другие размеры пиктограмм как для экранов Retina, так и для других экранов, как в системе iOS 10, так и в более ранних версиях iOS.

Не пытайтесь имитировать стиль кнопок, которые уже есть на экране; ваше устройство iPhone или iPad автоматически округлит углы. Создайте простые, квадратные изображения. Вы найдете набор подходящих пиктограмм в папке пиктограмм проекта 02 – Hello World.

ЗАМЕЧАНИЕ. Для создания пиктограмм в своем приложении используйте файлы с расширением .png. Система Xcode автоматически оптимизирует изображения .png во время сборки приложения. Благодаря этому использование изображений в формате .png является самым быстрым и эффективным способом работы с пиктограммами. Несмотря на то что изображения в других распространенных форматах также отображаются правильно, основным является формат .png, а для использования других форматов должны существовать весомые причины.

Нажмите комбинацию клавиш **<⌘+1>**, чтобы открыть навигатор проекта, а затем откройте группу Hello World и найдите папку Assets.xcassets. Иногда ее называют **каталогом ресурсов** (asset catalog). По умолчанию каждый новый проект Xcode создается с каталогом ресурсов, содержащим пиктограммы и другие вспомогательные файлы. Выберите папку Assets.xcassets и переключитесь на область редактирования.

В левой части окна редактирования вы увидите столбец, содержащий элемент с именем AppIcon. Выберите его — и справа от него вы увидите область с текстом AppIcon в левом верхнем углу и пунктирными квадратиками для пиктограмм, о которых мы говорили выше (рис. 2.26). Именно сюда вы будете перетаскивать свои новые пиктограммы.

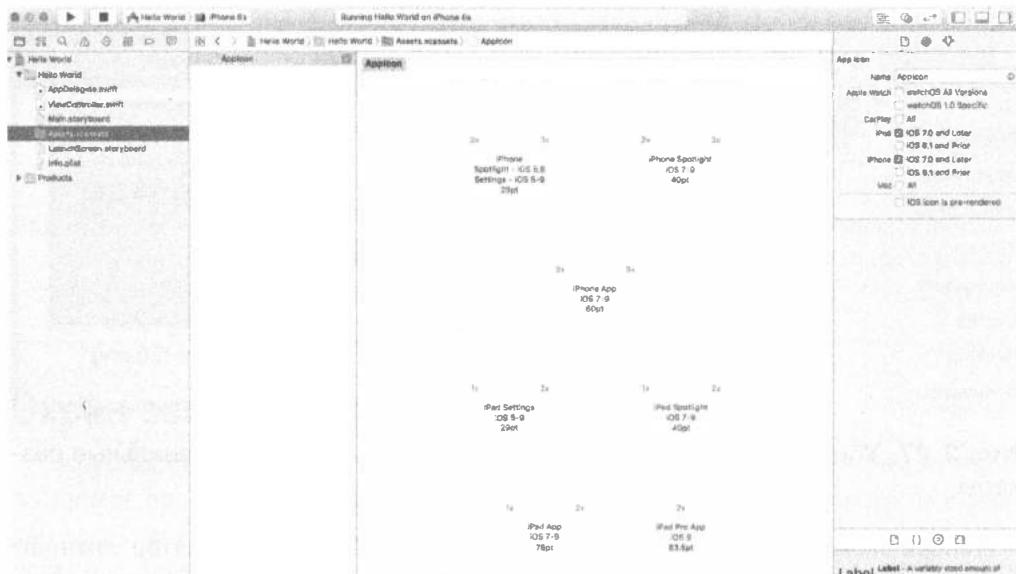


Рис. 2.26. Пиктограммы AppIcon в каталоге ресурсов проекта. Здесь можно настроить пиктограммы приложения

Откройте папку пиктограмм проекта 02 – Hello World, выберите все файлы и перетащите их в программу Interface Builder. Большинство пиктограмм будут автоматически вставлены с соответствующими именами. Возможно, на экране останется несколько пустых квадратов. Их придется заполнить по-отдельности. Для этого необходимо сравнить размер файла с размерами квадрата. Если под квадратом написано 2x или 3x, то вам понадобятся двойной и тройной размеры файлов. Например, на рис. 2.27 квадрат для iPhone Spotlight iOS 7–9 остался пустым и имеет размер 3x. Это значит, что вам нужно найти файл, соответствующий 120 пикселям, т.е. 3×40 пикселям.



Рис. 2.27. Убедитесь, что файлы с расширением .png имеют правильные размеры

Теперь скомпилируйте и выполните приложение. Когда симулятор закончит процесс запуска, выполните команду <Shift+⌘+⇟>, вернитесь к домашнему экрану и посмотрите на пиктограмму, показанную на рис. 2.28. Для того чтобы посмотреть, как выглядят более мелкие пиктограммы, проведите пальцем

сверху вниз (swipe down) по главному экрану, чтобы на нем появилось окно Spotlight, и начните вводить слово Hello. Вы сразу же увидите новую пиктограмму приложения.

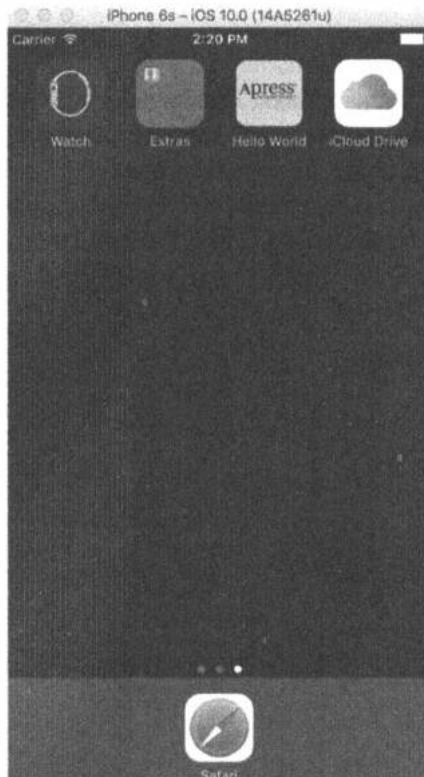


Рис. 2.28. Теперь ваше приложение имеет эффектную пиктограмму!

ЗАМЕЧАНИЕ. В ходе работы над книгой наш главный экран симулятора оказался захламленным пиктограммами выполняемых нами примеров. Если вы хотите стереть старые пиктограммы с главного экрана, выберите команду **iOS Simulator**⇒**Reset Content and Settings...** в меню симулятора приложений для системы iOS.

Экран запуска приложения

Запуская свое приложение, вы много раз могли видеть белый экран, который возникает при загрузке приложения. Приложения iOS всегда должны иметь экран запуска. Поскольку процесс загрузки приложения в память занимает определенное время (и чем крупнее приложение, тем больше времени необходимо для его загрузки), экран запуска должен как можно быстрее продемонстрировать пользователю, что происходят некие действия. До появления версии iOS 8

на экран можно было вывести изображение (фактически несколько изображений разных размеров), которое играло роль заставки. Система iOS могла загрузить правильное изображение и немедленно вывести его на экран до загрузки остальной части приложения. Начиная с версии iOS 8 компания Apple настоятельно рекомендует использовать не заставку, а файл запуска, а заставку применять для более ранних версий.

Файл запуска — это раскладовка, содержащая пользовательский интерфейс для экрана запуска. На устройствах, работающих под управлением системы iOS 8 и более поздних версий, приоритет отдается файлу запуска, а не заставке. Посмотрите в навигатор проекта — и вы увидите, что в вашем проекте уже есть файл запуска с именем `LaunchScreen.storyboard`. Если открыть его в программе `Builder`, то появится пустое представление, показанное на рис. 2.29.

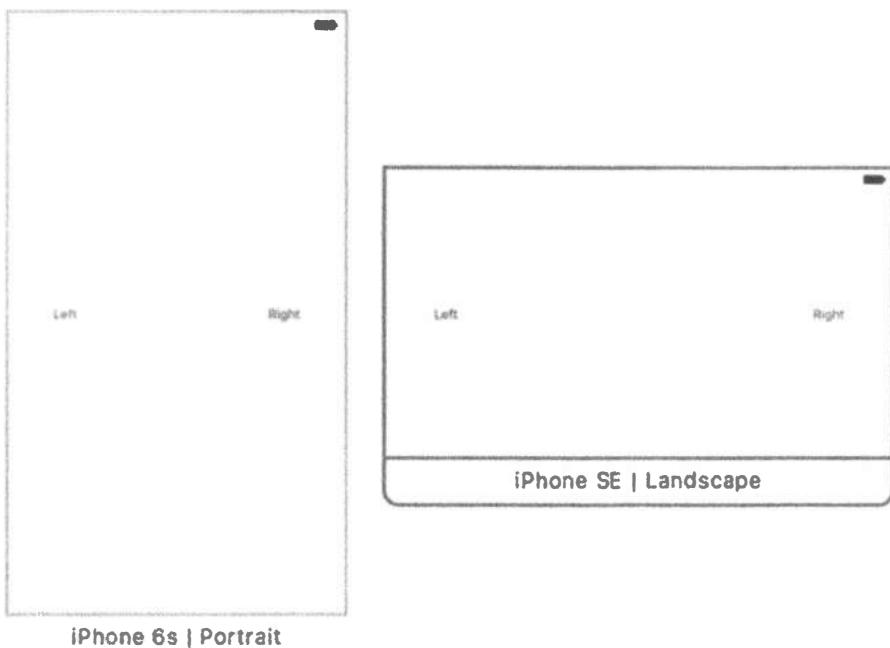


Рис. 2.29. Файл запуска приложения, предусмотренный по умолчанию

Компания Apple предполагает, что разработчики будут создавать заставку с помощью программы `Interface Builder` точно так же, как любую другую часть пользовательского интерфейса. Компания Apple рекомендует не пытаться создавать сложные или визуально пышные заставки, поэтому мы последуем их советам. Мы просто добавим метку в раскладовку и изменим цвет фона главного представления, чтобы можно было отличить заставку от самого приложения. Как и раньше, перетащите метку на раскладовку, измените ее текст на `Hello`

World, а затем с помощью инспектора атрибутов (см. рис. 2.23) измените его шрифт на System Bold 32. Выберите метку и выберите команду Editor⇒Size to Fit Content в меню Xcode. Затем отцентруйте метку в представлении и выберите команду Editor⇒Resolve Auto Layout Issues⇒Add Missing Constraints, чтобы добавить необходимые ограничения макета. Далее выберите главное представление, щелкнув на раскладовке или в окне Document Outline, и с помощью инспектора атрибутов измените цвет его фона. Для этого найдите элемент управления с меткой Background и выберите цвет, который вам нравится (мы рекомендуем наш любимый желтый цвет). Теперь просто запустите приложение еще раз. Сначала появится заставка, которая постепенно будет гаснуть по мере того, как будет появляться само приложение (рис. 2.30).



Рис. 2.30. Экран заставки приложения Hello World

Больше информации о файле запуска, заставках и пиктограммах приложений можно найти в документе компании Apple iOS Human Interface Guidelines, расположенном на веб-странице <https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/LaunchImages.html>.

Запуск приложения на устройстве

Прежде чем завершить главу, мы должны сделать еще кое-что. Загрузите приложение и запустите его на реальном устройстве. Сначала соедините устройство iOS с компьютером зарядным кабелем. После этого среда Xcode должна распознать устройство и некоторое время считывать с него информацию. Вы можете увидеть сообщение систем безопасности на компьютере Mac и на своем устройстве, спрашивающее вас, доверяете ли вы друг другу. Подождите, пока среда Xcode закончит обработку символьных файлов, полученных от устройства (чтобы видеть это, откройте представление *Activity View*), и откройте селектор устройств на панели инструментов. Вы увидите список устройств, показанный на рис. 2.31.

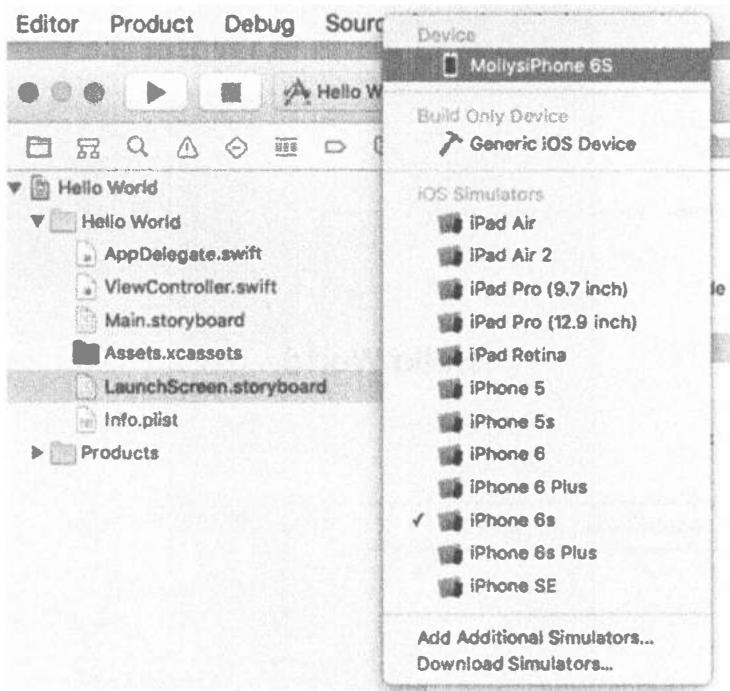


Рис. 2.31. Список устройств и симуляторов, в который входит iPhone 6/6s, принадлежащий одному из авторов

Выберите устройство и щелкните на кнопке Run, расположенной на панели инструментов, чтобы начать процесс инсталляции и запуска приложения на нем. Среда Xcode заново соберет приложение и выполнит его на вашем устройстве. Поскольку в работе над книгой использовалась бета-версия среды Xcode 8, на экране может появиться окно, показанное на рис. 2.32.



Рис. 2.32. Если предварительные условия не выполняются, на экране может появиться предупреждение

ЗАМЕЧАНИЕ. Компания Apple внесла улучшения в систему контроля использования версии Xcode 8, и многие из проблем были решены, так что процесс загрузки стал более гладким и простым. Мы привели этот пример только для демонстрации важности правильной подготовки приложения к запуску на реальном устройстве.

Перед тем как инсталлировать приложение на устройстве iOS, необходимо создать его профиль ресурсов и подпись. Подпись приложения позволяет устройству идентифицировать автора и проверять, не был ли бинарный код искачен после его создания. Профиль ресурсов содержит информацию для системы iOS о том, какие функциональные возможности нужны приложению, например доступ к службе iCloud, и на каких устройствах оно может работать. Для того чтобы подписать приложение, среда Xcode должна получить сертификат и закрытый ключ.

СОВЕТ. Больше информации о создании подписи кода, профилей ресурсов и закрытых ключей содержится в документе *Distribution Workflows*, входящем в справочник *App Distribution Guide*, расположенный по адресу <https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide>.

Раньше вы должны были подписаться на программу разработки и вручную создать профиль и подпись, а затем зарегистрировать тестовые устройства, на которых вы хотите инсталлировать приложение. Это был сложный и утомительный процесс. Среда Xcode 7 была настолько усовершенствована, что делала это автоматически, а версия Xcode 8 еще больше улучшила ситуацию, так что программисту достаточно всего лишь загрузить приложение на устройство для тестирования. В некоторых случаях, когда для разных пользователей

предназначаются разные сборки, процесс создания профиля можно настраивать, но для обучения достаточно использовать механизм, предусмотренный по умолчанию.

Многое может пойти неправильно. Например, увидев сообщение о том, что ваш идентификатор App ID недоступен, вы должны выбрать другой. Идентификатор App ID состоит из названия проекта и идентификатора организации, который вы выбираете при создании проекта (см. рис. 2.4). Вы увидите это сообщение, если использовали идентификатор com.beginningiphone или другой идентификатор, который кто-то уже зарегистрировал. Для того чтобы исправить ситуацию, откройте навигатор проекта и выберите узел Hello World, расположенный на вершине дерева проекта. Затем щелкните на узле Hello World в разделе TARGETS в окне Document Outline. В заключение щелкните на кнопке General в верхней части области редактирования (рис. 2.33).



Рис. 2.33. Изменение идентификатора комплекта приложения

Идентификатор App ID, который среда Xcode использует для подписи приложений, образуется из содержания поля *Bundle Identifier* в редакторе. Это поле содержит идентификатор организации, выбранный вами при создании проекта (эта часть поля на рис. 2.33 выделена подсветкой). Выберите другое значение и попытайтесь снова. В конце концов вы должны найти идентификатор, который еще никем не использовался. Когда вам это удастся, отметьте его и заполните поле *Organization Identifier* для нового проекта. После того как вы сделаете это правильно, среда Xcode запомнит идентификатор, и вам больше не придется повторять этот процесс.

Другая возможная проблема показана на рис. 2.34.

Вы увидите это сообщение, только если вы не присоединились к программе разработчиков. Это значит, что ваше устройство iOS не доверяет вам запуск приложений, подписанных вашим идентификатором Apple ID. Для того чтобы исправить этот недостаток, откройте на устройстве приложение *Settings* и выполните команду *General⇒Profile*. Вы увидите страницу с таблицей, содержащей ваш идентификатор Apple ID. Коснитесь строки таблицы, чтобы открыть другую страницу, показанную на рис. 2.35.

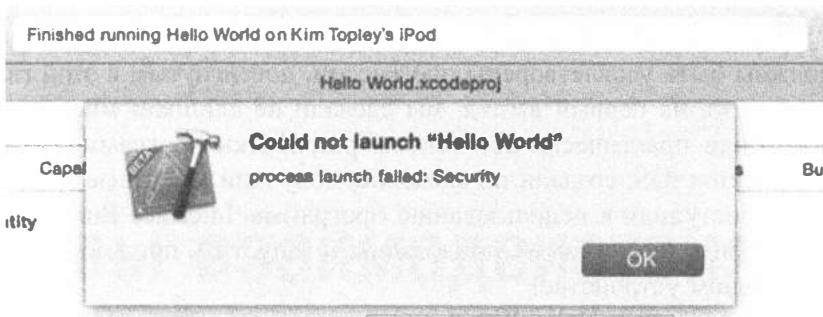


Рис. 2.34. Невозможность запустить приложение в системе iOS 9 или 10

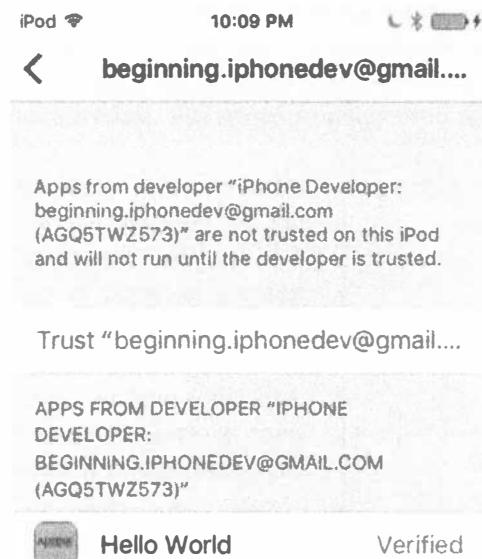


Рис. 2.35. В системе iOS 9 и более поздних версиях разработчики, не подписавшиеся на программу разработки, по умолчанию не имеют доверия

Резюме

Мы должны быть удовлетворены прогрессом, достигнутым в этой главе. Несмотря на то что, на первый взгляд, мы сделали не слишком много, на самом деле мы освоили практически все основы разработки программ. Вы изучили шаблоны проектов iOS, создали приложение, получили ключевые знания о сре-де Xcode 8, приступили к использованию программы Interface Builder, узнали о том, как настроить пиктограмму приложения и запустить приложение на симуляторе и реальном устройстве.

И все же, программа Hello World — слишком одностороннее приложение. Мы показываем пользователям какую-то информацию, но не получаем данных от них. В следующей главе мы покажем, как обеспечивать ввод данных от пользователя на устройство iOS и выполнять действия, основываясь на этих данных.

ГЛАВА 3



Основы взаимодействия

Наше приложение Hello World было хорошим введением в разработку приложений для системы iOS с помощью интегрированной среды Xcode и каркаса Cocoa Touch, но в нем не было очень важной функциональной возможности: взаимодействия с пользователем. Без этого приложение имеет очень ограниченное применение.

В этой главе мы напишем немного более сложное приложение, в котором будут две кнопки и метка, как показано на рис. 3.1. Когда пользователь нажмет одну из кнопок, текст метки изменится. Этот пример может показаться слишком упрощенным, но он демонстрирует ключевые концепции, связанные с реализацией взаимодействия пользователя с приложениями для системы iOS.

Парадигма “модель–контроллер–представление”

Концепция “модель–контроллер–представление” (Model-View-Controller — MVC) представляет собой очень логичный способ разделения кода, лежащего в основе приложений с графическим пользовательским интерфейсом. В настоящее время практически все объектно-ориентированные среды разработки в той или иной степени используют концепцию MVC, но лишь некоторые из них действительно полностью воплощают парадигму MVC, как это делает среда Cocos Touch.

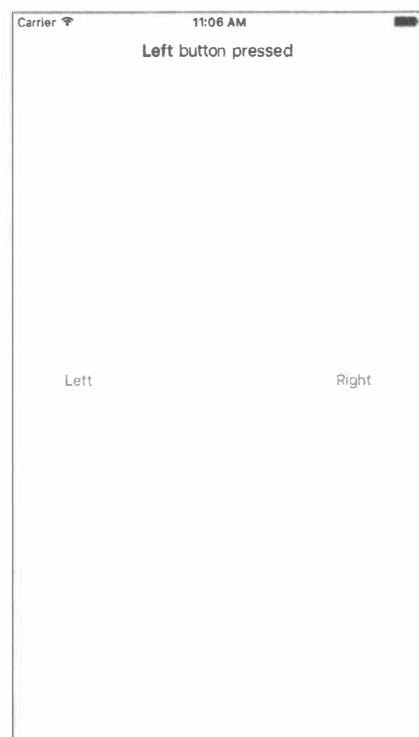


Рис. 3.1. Простое приложение с двумя кнопками, которое мы разработаем в этой главе

Шаблон MVC разделяется по функциональным возможностям на три категории.

- **Модель.** Состоит из классов, в которых хранятся данные приложения.
- **Представление.** Создает окна, элементы управления и другие элементы, которые пользователь видит и с которыми взаимодействует.
- **Контроллер.** Связывает модель и представление, реализует логику приложения, в соответствии с которой оно обрабатывает данные, введенные пользователем.

Назначение концепции MVC — создать как можно более независимые один от другого объекты, реализующие эти три типа кода. Любой объект, создаваемый вами, должен четко идентифицироваться как объект, принадлежащий одной из перечисленных выше категорий. При этом он должен вообще не иметь или иметь как можно меньше функциональных возможностей, которые можно было бы отнести к остальным двум категориям. Например, объект, реализующий кнопку, не должен содержать код для обработки данных в момент ее нажатия, а реализация банковского счета не должна содержать код для рисования таблицы для демонстрации транзакций.

Концепция MVC обеспечивает максимальное повторное использование кода. Класс, реализующий обобщенную кнопку, можно использовать в любом приложении. Класс, реализующий кнопку, выполняющую конкретные вычисления при ее нажатии, можно использовать только в том приложении, для которого он был написан изначально.

Когда вы пишете приложения в среде Cocoa Touch, вы в основном создаете компоненты представления, используя визуальный редактор Interface Builder, хотя иногда вы также модифицируете свой интерфейс с помощью кода или создаете подклассы для существующих видимых деталей и элементов управления.

Ваша модель будет создана на основе классов языка Swift, разработанных для хранения данных приложения. Мы не собираемся создавать объекты модели в этой главе, поскольку не планируем хранить или собирать данные, и описываем их для того, чтобы использовать впоследствии при разработке более сложных приложений.

Ваш контроллер будет состоять из классов, создаваемых вами, а также относящихся к вашему приложению. Контроллер может полностью состоять из обычных классов, но чаще они являются подклассами одного из существующих обобщенных классов контроллера из библиотеки UIKit, например класса `UIViewController`, с которым мы встретимся в следующем разделе. Создавая подклассы одного из существующих классов, вы получаете в свое распоряжение множество функциональных возможностей и экономите время за счет того, что не изобретаете велосипед.

По мере углубления в среду Cocoa Touch вы быстро убедитесь, что классы каркаса UIKit следуют принципам MVC. Если вы будете последовательно придерживаться этой концепции в процессе разработки, то в результате создадите более ясный и легко эксплуатируемый код.

Создание приложения ButtonFun

Настало время создать следующий проект Xcode. Мы собираемся использовать тот же шаблон, который исследовали в предыдущей главе: Single View Application. Отталкиваясь от этого простого шаблона, нам будет легче увидеть, как взаимодействуют объекты представления и шаблона в рамках приложения для системы iOS. В следующих главах мы будем использовать другие шаблоны.

Запустите программу Xcode и выберите команду **File⇒New⇒New Project...** или нажмите комбинацию клавиш **<⌘+N>**. Выберите шаблон **Single View Application** и щелкните на кнопке **Next**.

Вы увидите тот же самый лист настроек, который видели в предыдущей главе. В поле **Product Name** введите название нового приложения — **ButtonFun**. Поля **Organization Name**, **Company Identifier** и **Language** идентификатора пакета должно содержать те же значения, которые мы использовали в предыдущей главе, поэтому трогать его не следует. Мы планируем использовать механизм Auto Layout, чтобы наше приложение работало на любых устройствах iOS, поэтому в поле **Devices** выберите значение **Universal**. Полный список настроек представлен на рис. 3.2.

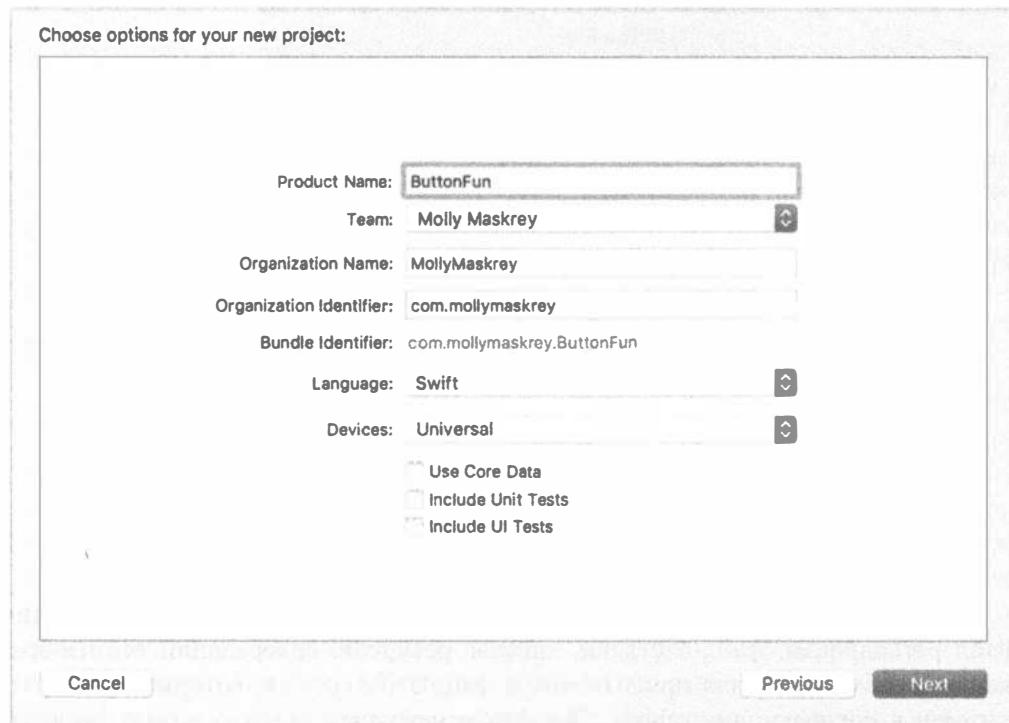


Рис. 3.2. Выбор имени проекта и его параметров

Щелкните на кнопке **Next** и выберите место хранения своего проекта. Оставьте флашок *Create Git repository* сброшенным или установленным на свое усмотрение. Сохраните проект среди остальных проектов нашей книги.

Создание контроллера представления

Немного позднее мы разработаем представление (т.е. пользовательский интерфейс) для нашего приложения, используя программу Interface Builder, как это было сделано в предыдущей главе. А пока мы собираемся внести изменения в файлы исходного кода, созданного для нас шаблоном проекта. Да, мы действительно собираемся написать часть кода в этой главе. Но прежде чем вносить какие-либо изменения, взглянем на файлы, сгенерированные шаблоном. Для этого следует раскрыть навигатор проекта, в котором уже раскрыта группа *Button Fun*. Если это не так, ее следует открыть, щелкнув на треугольнике раскрытия, расположенному рядом с ее именем (рис. 3.3).

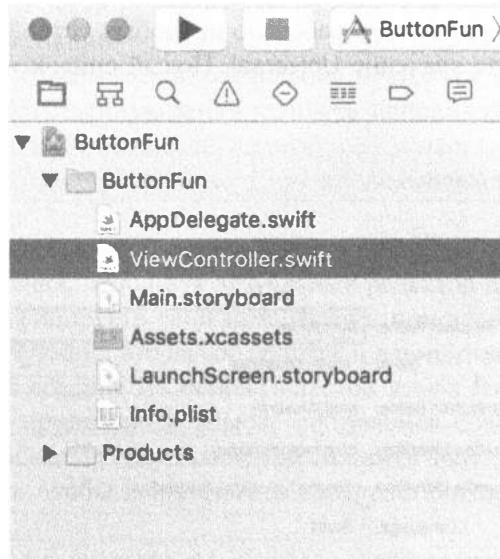


Рис. 3.3. Навигатор проекта, демонстрирующий файлы классов, созданные шаблоном проекта. Обратите внимание на то, что наш префикс класса автоматически инкорпорирован в имена класса

Группа *Button Fun* должна содержать два файла с исходным кодом, один файл раскадровки, файл заставки, каталог ресурсов, содержащий все изображения, необходимые для приложения, и файл *Info.plist*, который будет рассмотрен в последующих главах. Два файла исходных кодов содержат реализации классов, необходимых для приложения: делегат приложения и контроллер представления приложения, имеющего только одно представление. Делегат

приложения будет рассмотрен позднее в этой главе, а пока исследуем контроллер создаваемого нами представления.

Класс контроллера, ответственный за управление этим представлением, называется `ViewController`. Это имя означает, что класс является контроллером представления. Щелкните на файле `View.Controller.swift` в окне навигатора проекта, и вы увидите на экране содержимое этого файла (листинг 3.1).

Листинг 3.1. Код класса `ViewController`, сгенерированного по шаблону

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Дополнительные настройки после загрузки представления,
        // как правило, из nib-файла.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Освобождаем любые восстанавливаемые ресурсы
    }
}
```

Поскольку этот файл сгенерирован по шаблону, он имеет не очень богатое содержание. `ViewController` — это подкласс класса `UIViewController`, одного из обобщенных классов контроллера, которые мы упоминали выше. Он является частью библиотеки `UIKit` и предоставляет в наше распоряжение множество функциональных возможностей. Среда Xcode не знает, какие именно функциональные возможности должно иметь наше приложение, но ей известно, что мы собираемся сделать нечто, поэтому она создала этот класс, в котором будут реализованы требуемые функциональные возможности.

Выходы и действия

В главе 2 для разработки пользовательского интерфейса мы использовали программу `Interface Builder`, а в листинге 3.1 мы видели оболочку класса контроллера. Рассмотрим способ, с помощью которого наш код в классе контроллера представления будет правильно взаимодействовать с объектами (кнопками, метками и пр.), содержащимися в раскладовке. Класс контроллера может ссылаться на объекты в раскладовке, используя особый вид свойства под названием `выход` (`outlet`). Выход можно интерпретировать как указатель, ссылающийся на объект в пользовательском интерфейсе. Например, допустим, что вы создали текстовую метку с помощью программы `Interface Builder` (как это было сделано в главе 2) и хотите изменить ее текст, модифицируя исходный код. Объявив выход и связав его с объектом метки, вы можете использовать эту переменную в своем коде для изменения текста, изображенного на метке. Мы покажем, как это сделать, ниже в этой главе.

Перейдем на противоположную сторону. Объекты интерфейса в нашей раскладовке или nib-файле могут быть связаны со специальными методами в классе контроллера. Эти специальные методы называются действиями (actions). Вы даже можете сообщить программе Interface Builder, что, когда пользователь касается кнопки, она должна вызвать один метод, а когда пользователь отнимает палец от кнопки, должен быть вызван другой метод действия.

Среда Xcode поддерживает разные способы создания выходов и действий. Например, можно указать их в заголовочном файле контроллера представления и лишь после этого открывать программу Interface Builder и приступать к связыванию выходов и действий, однако представление помощника (assistant view) в программе Xcode позволяет быстрее и проще создавать и связывать выходы и действия одновременно. Этот процесс мы также вскоре рассмотрим. Но прежде чем приступить к установке связей между выходами и действиями, поговорим о них немного подробнее. Выходы и действия — два основных элемента, используемых при создании приложений для операционной системы iOS, поэтому нам важно понимать, что они собой представляют и как работают.

Выходы

Выход (outlet) — это обычное свойство в языке Swift, объявляемое с помощью атрибута `@IBOutlet`. Объявление выхода в заголовочном файле контроллера должно выглядеть примерно следующим образом:

```
@IBOutlet weak var myButton: UIButton!
```

Этот пример демонстрирует выход с именем `myButton`, который можно связать с любой кнопкой пользовательского интерфейса.

Видя атрибут `@IBOutlet`, компилятор Swift не делает ничего особенного. Единственное назначение — подсказать программе Interface Builder, что это свойство, которое будет связано с объектом в раскладовке или nib-файле. Любому свойству, которое вы создадите и захотите связать с объектом в nib-файле, должен предшествовать атрибут `@IBOutlet`. К счастью, программа Xcode теперь создает выходы автоматически при перетаскивании объекта на свойство, с которым вы хотите его связать, и даже при перетаскивании его на класс, в котором вы хотите создать новый выход.

У читателей может возникнуть вопрос, почему объявление свойства `myButton` завершается восклицательным знаком. По правилам языка Swift все свойства должны быть полностью инициализированы до завершения работы всех инициализаторов, если они не были объявлены как свойства необязательного типа. Когда контроллер загружает раскладовку, значения свойств выходов определяются информацией, содержащейся в раскладовке, но это происходит *после* запуска инициализатора контроллера представления. В результате, если явным образом не определить фиктивные значения (что нежелательно), то свойства выхода необходимо объявить как свойства необязательного типа. Таким образом, существует

два способа объявления свойств выходов — с помощью символов ! и ?, как показано в листинге 3.2.

Листинг 3.2. Два способа объявления переменных необязательного типа

```
@IBOutlet weak var myButton1: UIButton?
@IBOutlet weak var myButton2: UIButton!
```

Второй способ проще первого, потому что не требует явной распаковки переменной необязательного типа, когда она впоследствии понадобится в коде контроллера представления (листинг 3.3).

Листинг 3.3. Исключение необходимости явно распаковывать переменные необязательного типа

```
let button1 = myButton1! // Переменную необязательного типа
                        // необходимо явно распаковать
let button2 = myButton2 // Переменная myButton2 распакована неявно
```

ЗАМЕЧАНИЕ. Спецификатор weak перед объявлением свойства выхода означает, что данное свойство не обязано создавать сильную ссылку на кнопку. Объекты автоматически удаляются из памяти, если на них больше нет сильных ссылок. В данном случае нет никакого риска, что кнопка будет удалена из памяти, ведь на нее всегда будет указывать сильная ссылка, поскольку кнопка является частью пользовательского интерфейса. Объявление слабой ссылки позволяет удалять представление из памяти, если оно больше не нужно, одновременно исключая его из пользовательского интерфейса. При этом ссылка устанавливается равной nil.

Действия

Действия (actions) — это методы, возвращающие объекты с атрибутом @IBAction, которые сообщают программе Interface Builder, что данный метод может быть активизирован элементом управления в раскладовке или nib-файле. Как правило, объявление действия выглядит примерно так:

```
@IBAction func doSomething(sender: UIButton) {}
```

Или так:

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

Реальное имя метода может быть любым. Обычно действие либо не имеет аргументов, либо получает один аргумент, который, как правило, имеет имя sender. При вызове метода действия аргумент sender содержит ссылку на вызвавший его объект. Таким образом, например, если действие было вызвано в результате нажатия кнопки, аргумент sender содержит ссылку на конкретную кнопку, на которой произошло нажатие. Благодаря аргументу sender существует возможность отвечать нескольким элементам управления, используя один

и тот же метод действия. Он позволяет идентифицировать элемент управления, вызвавший метод действия.

СОВЕТ. На самом деле существует третье, редко используемое объявление метода действия, которое выглядит так:

```
@IBAction func doSomething(sender: UIButton,  
forEvent event: UIEvent){};
```

Этот вид объявления целесообразно использовать, если вам нужна дополнительная информация о событии, порожденном вызванным методом. Управляющие события рассматриваются в следующей главе.

Нет ничего плохого в том, что мы объявили метод действия с аргументом `sender`, а потом проигнорировали его. Методы действия в каркасе Сосоа и системе NeXTSTEP должны получать аргумент `sender` независимо от того, используют они его или нет, поэтому многие программы для системы iOS написаны именно так.

Разобравшись в том, что такое действия и выходы, перейдем к их применению при разработке пользовательского интерфейса. Однако, прежде чем начать, необходимо уделить немного времени вопросам, связанным с наведением порядка.

Разработка контроллера представления

Щелкните на файле `ViewController.swift` в навигаторе проекта, чтобы открыть файл реализации. Как видим, выбранный нами шаблон проекта сгенерировал совсем немного шаблонного кода в виде методов `viewDidLoad()` и `didReceiveMemoryWarning()`. Эти методы обычно используются в подклассах класса `UIViewController`, поэтому программа Xcode предоставила нам их шаблонную реализацию и мы должны просто добавить сюда свой код. Однако большая часть этих шаблонных реализаций для нашего проекта не нужна, так что они лишь увеличивают размер файла и затрудняют чтение. Для того чтобы привести реализацию в порядок, удалим лишние фрагменты. В результате содержание файла должно стать таким, как показано в листинге 3.4.

Листинг 3.4. Упрощенный файл `ViewController.swift`

```
import UIKit  
  
class ViewController: UIViewController {  
}
```

Разработка пользовательского интерфейса

Сохраните внесенные изменения, а затем щелкните на пункте Main.storyboard, чтобы открыть представление вашего приложения в окне Interface Builder (рис. 3.4). Как было указано в предыдущей главе, белое окно, открывшееся в области редактирования, является единственным представлением нашего приложения. Вернувшись к рис. 3.1, легко убедиться, что мы должны добавить в это представление две кнопки и одну метку.

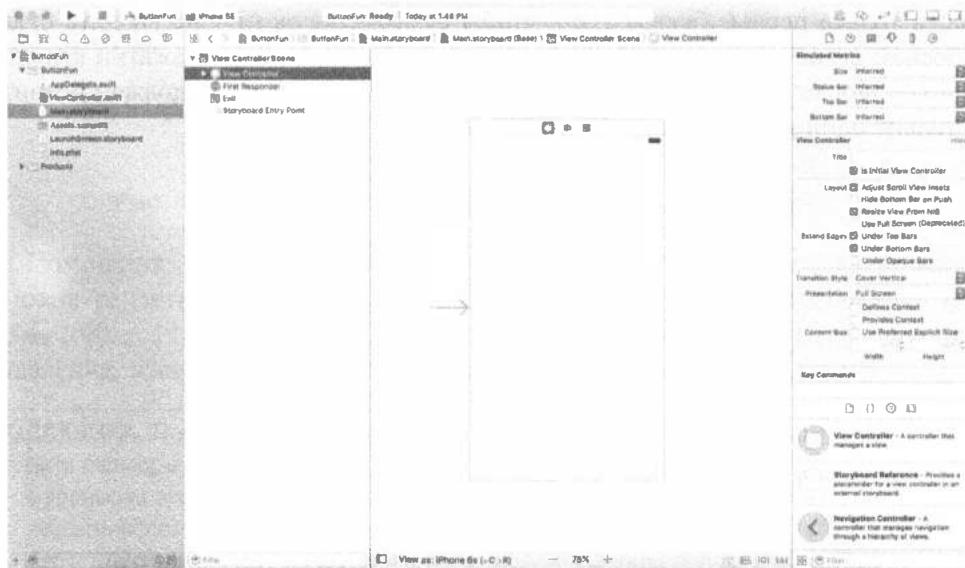


Рис. 3.4. Открытие файла Main.storyboard в программе Interface Builder среды Xcode

Подумаем секунду о нашем приложении. Мы собираемся добавить в интерфейс две кнопки и одну метку. Этот процесс очень похож на то, что мы делали в предыдущей главе. Однако теперь мы хотим использовать выходы и действия, чтобы наше приложение стало интерактивным.

Кнопки должны запускать методы действия нашего контроллера. Мы могли бы сделать так, чтобы вызывались разные методы действия, но, поскольку они будут выполнять, по существу, одно и то же задание (обновлять текст метки), нам необходимо вызывать один и тот же метод. Мы будем различать кнопки с помощью аргумента sender, который рассмотрели выше. Кроме метода действия, нам также нужен выход, связанный с меткой, чтобы мы могли изменять текст, отображаемый на метке.

Сначала добавим кнопки, а затем разместим метку. Разработав интерфейс, добавим к нему соответствующие действия и выходы. Кроме того, мы могли бы вручную объявить действия и выходы, а затем соединить их с элементами интерфейса, но среда Xcode сделает это автоматически.

Добавление кнопок и метода действия

Сначала добавим в интерфейс две кнопки. Затем среда Xcode создаст пустой шаблонный метод действия и мы свяжем с ним обе кнопки. После того как пользователь щелкнет на кнопке, будет вызван метод действия и выполнен любой код, который будет в нем записан.

Выберите команду *View⇒Utilities⇒Show Object Library* или нажмите комбинацию клавиш *<Control+Option+⌘+3>*, чтобы открыть библиотеку объектов. Введите в поле поиска библиотеки строку *UIButton*. На самом деле достаточно ввести только первые буквы *uibu*, чтобы сузить список (лучше вводить буквы в нижнем регистре, чтобы избежать путаницы при случайном нажатии клавиши *<Shift>*). После ввода этой строки в окне библиотеки объектов должен появиться только один объект: *Button* (рис. 3.5).



Рис. 3.5. В окне библиотеки объектов появляется элемент *Button*

Перетащите объект *Button* из библиотеки и оставьте его в белом окне области редактирования, чтобы добавить кнопку в представление нашего приложения. Разместите эту кнопку возле левого края представления на достаточном расстоянии, используя вертикальную голубую линию разметки. Для того чтобы выровнять кнопку по высоте, разместив ее посередине представления, используйте горизонтальную голубую линию разметки. Если это вам поможет, ориентируйтесь на рис. 3.1.

ЗАМЕЧАНИЕ. Голубые линии разметки, которые появляются при перемещении объектов в окне программы Interface Builder, помогут вам освоить принципы руководства iOS Human Interface Guidelines (HIG). Компания Apple разработала руководство HIG для людей, проектирующих приложения для устройств iPhone и iPad. Руководство HIG регламентирует, как следует (и не следует) проектировать пользовательский интерфейс. Вам необходимо прочитать его, потому что оно содержит ценную информацию, которую должен знать каждый разработчик приложений для устройства iPhone. Это руководство можно найти на веб-странице <https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/>.

Дважды щелкните на добавленной кнопке. Это даст возможность отредактировать ее название. Назовем эту кнопку *Left*.

Выберите команду *View⇒Assistant Editor⇒Show Assistant Editor* или нажмите комбинацию клавиш *<Option+⌘+Return>*. Вы можете показывать и скрывать помощник редактора, щелкнув средней кнопкой в группе кнопок *Editor*, входящей в коллекцию из семи кнопок, расположенных в правом верхнем углу окна проектирования (рис. 3.6).

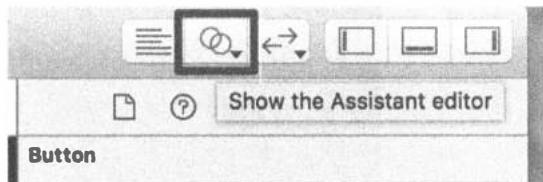


Рис. 3.6. Кнопка Show the Assistant Editor (две окружности)

Помощник редактора появляется в правой части окна редактирования. В окне редактирования помощник редактора автоматически открывает файл *ViewController.swift*, являющийся файлом реализации контроллера представления, владеющего представлением, которое вы видите на экране.

ПОДСКАЗКА. После открытия помощника редактора вам, возможно, понадобится изменить размеры окна, чтобы было достаточно места для работы. Если хотите работать с маленьким экраном, как, например, на компьютере MacBook Air, возможно, придется закрыть вспомогательное представление и/или навигатор проекта, чтобы эффективно работать с помощником редактора (рис. 3.8). Это легко сделать с помощью трех кнопок, расположенных в правом верхнем углу окна проекта (см. рис. 3.6).

Среда Xcode знает, что наш контроллер представления отвечает за демонстрацию представления в раскладовке, поэтому помощник редактора знает, что должен показать нам реализацию класса контроллера представления, в котором, скорее всего, будет происходить связывание выходов и действий. Однако, если все же вы не видите файла, который вам нужен, можете перейти на панель быстрого перехода в верхней части окна помощника редактора и устранить проблему. Сначала найдите сегмент панели быстрого перехода *Automatic* и щелкните на нем. На экране появится всплывающее меню, в котором необходимо выбрать команду *Manual⇒Button Fun⇒ViewController.swift*. Теперь на экране должен появиться правильный файл.

Попросим программу Xcode автоматически создать новый метод действия для нас и связать его с только что созданной кнопкой. Мы добавим эти определения в расширение класса контроллера представления. Для этого щелкните на кнопке, добавленной в раскладовку, чтобы она оказалась выбранной. Нажмите и удерживайте клавишу *<Control>*, а затем щелкните мышью и перетащите курсор с кнопки на окно помощника редактора. Вы увидите голубую линию,

которая будет следовать за курсором (рис. 3.8). Эта линия связывает объекты с кодом или другими объектами. Когда вы будете перемещать курсор в пределах определения класса, как показано на рис.3.8, на экране появится всплывающее меню, которое будет информировать вас, что отпускание кнопки мыши приведет к вставке выхода, действия или коллекции выходов.



Рис. 3.7. Для того чтобы видеть окна редактирования на небольших дисплеях, возможно, придется закрыть другие представления

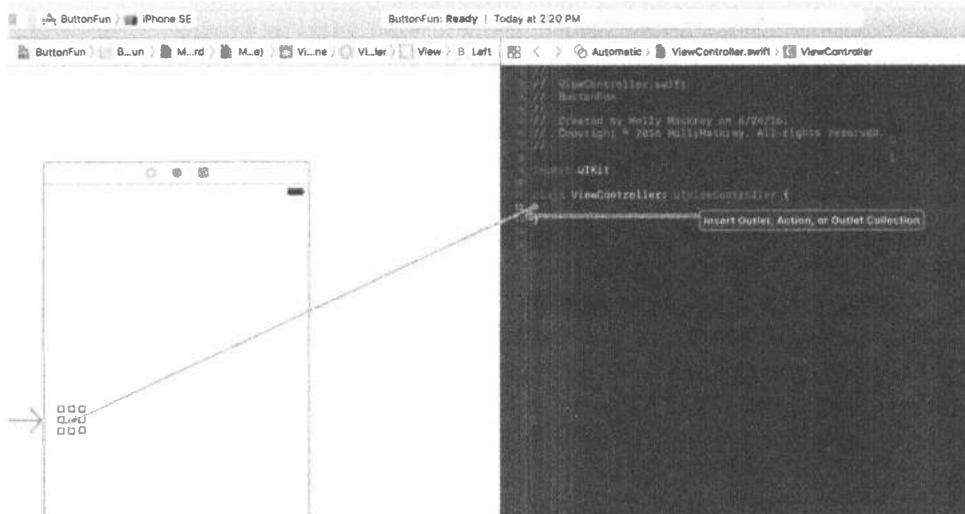


Рис. 3.8. Перетаскивание курсора в исходный код при нажатой клавише <Control> дает возможность создать выход, действие или коллекцию выходов

ЗАМЕЧАНИЕ. В настоящей книге мы используем действия и выходы, но не коллекции выходов. Коллекции выходов позволяют соединять несколько однородных объектов с одним и тем же свойством NSArray, а не создавать отдельное свойство для каждого объекта.

Для того чтобы закончить соединение, отпустите кнопку мыши, и на экране появится всплывающее меню (рис. 3.9).

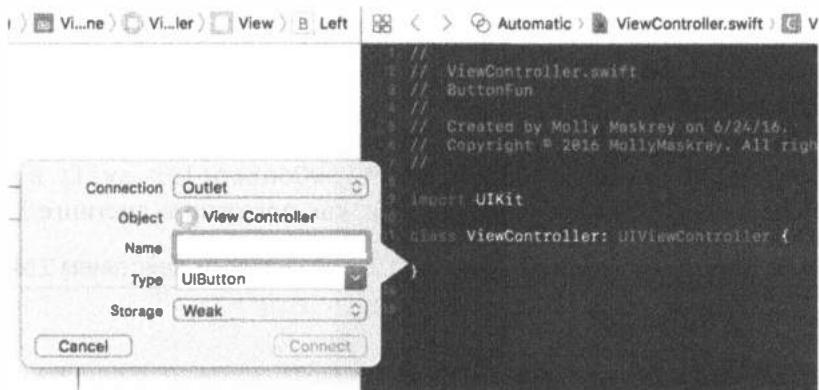


Рис. 3.9. Всплывающее меню, которое появляется после перетаскивания курсора в исходный код при нажатой клавише <Control>

Это окно позволяет настроить новое действие. Щелкните на меню Connection и измените выбор команды с Outlet на Action. Тем самым вы сообщите программе Xcode, что хотите создать действие, а не выход. В результате окно примет вид, показанный на рис. 3.10. Введите в поле Name строку buttonPressed. Когда закончите ввод, *не* нажмите клавишу <Return>. Нажатие клавиши <Return> приведет к завершению процесса создания выхода, а мы пока не собираемся этого делать. Вместо этого нажмите клавишу <Tab>, перейдите в поле Type и введите в нем строку UIButton, заменив значение AnyObject, заданное по умолчанию.



Рис. 3.10. Изменение типа соединения с Outlet на Action

Ниже поля Type расположены два поля, которые мы оставим заполненными значениями, заданными по умолчанию. Поле Event позволяет указать, когда будет вызван метод. Значение по умолчанию Touch Up Inside означает событие, когда пользователь отнимает палец от экрана над кнопкой. Это стандартное событие для кнопок. Оно дает пользователю возможность передумать. Если, перед тем как поднять палец, пользователь переместит его за пределы кнопки, метод не будет вызван.

Поле Arguments позволяет выбрать одну из трех разных сигнатур, используемых для методов действия. Мы выбрали аргумент sender, так что можем указать, какая кнопка вызвала метод. Это значение задается по умолчанию, поэтому оставим его неизменным.

Нажмите клавишу <Return> или щелкните на кнопке Connect, и программа Xcode вставит метод действия. Для файла ViewController.swift в окне помощника редактора это будет выглядеть так, как показано в листинге 3.5.

Листинг 3.5. Файл ViewController.swift с добавленным действием IBAction

```
import UIKit

class ViewController: UIViewController {

    @IBAction func buttonPressed(_ sender: UIButton) {
    }
}
```

Программа Xcode не только добавила объявление метода в код, но и связала кнопку с этим методом, сохранив эту информацию в раскладовке. Это значит, что нам не надо делать что-либо еще, чтобы кнопка вызвала этот метод при выполнении приложения.

Вернитесь к файлу Main.storyboard и перетащите на его окно другую кнопку, на этот раз поместив кнопку в правой части экрана. На экране появятся голубые линии, помогающие сориентировать кнопку по отношению к правому краю и другой кнопке. Поместив ее в требуемое место, дважды щелкните на ней и измените ее имя на Right.

ЗАМЕЧАНИЕ. Вместо перетаскивания нового объекта из библиотеки можно нажать клавишу <Option> и перетащить на представление оригиналный объект (в данном примере это кнопка Left). Удерживание клавиши <Option> заставляет программу Interface Builder скопировать перетаскиваемый объект.

Пока мы не хотим создавать новый метод действия. Вместо этого свяжем эту кнопку с существующим методом, созданным программой Xcode. Изменив имя кнопки, нажмите клавишу <Control>, щелкните на новой кнопке и снова перетащите ее на заголовочный файл. На этот раз, когда курсор достигнет объявления метода buttonPressed(), этот метод будет подсвечен

и на экране появится всплывающее окно Connect Action (рис. 3.11). Если вы не увидите это окно сразу, перетаскивайте указатель мыши по кругу до тех пор, пока оно не появится. Когда увидите это окно, отпустите кнопку мыши, и программа Xcode соединит эту кнопку с существующим методом действия. В результате при нажатии кнопки будет вызван тот же метод, что и для другой кнопки.

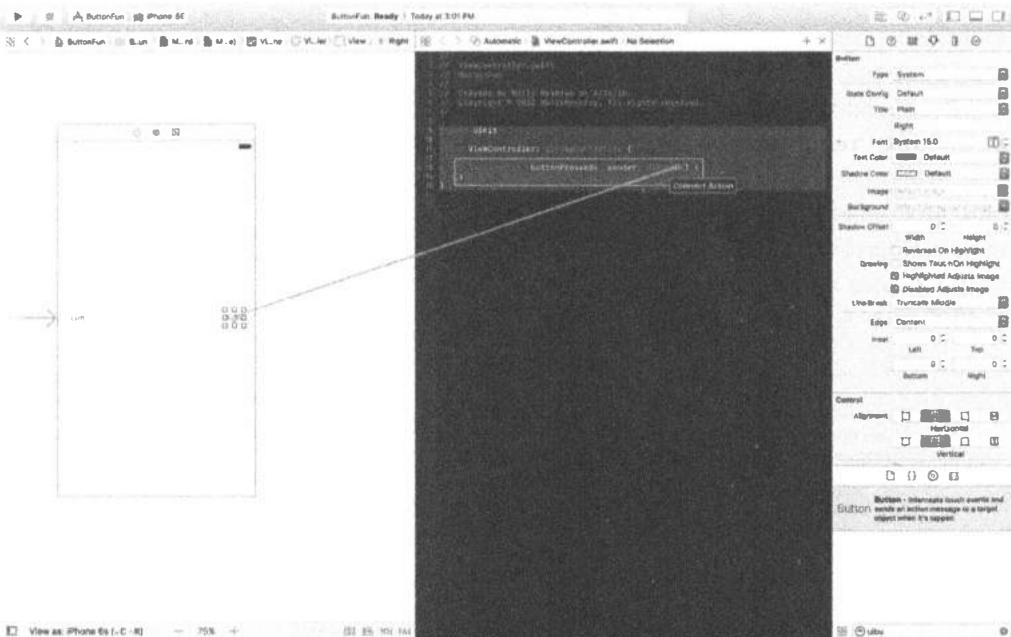


Рис. 3.11. Перетаскивая существующее действие, можно установить связь между ним и кнопкой

Добавление метки и выхода

Находясь в библиотеке объектов, введите в поле поиска слово `lab`, чтобы найти элемент интерфейса `Label` (рис. 3.12). Перетащите метку на свой пользовательский интерфейс и поместите где-нибудь между двумя кнопками. Затем, используя маркеры масштабирования, растяните метку от левого края (отмеченного голубой линией) до правого. Это обеспечит достаточно места для текста, который будет выведен для пользователя.

По умолчанию метки выравниваются по левому краю, но нашу метку мы хотим отцентровать. Выберите команду `View`⇒`Utilities`⇒`Show Attributes Inspector` (или нажмите комбинацию клавиш `<Option+⌘+4>`), чтобы открыть окно инспектора атрибутов (рис. 3.12). Выберите метку, а затем найдите в инспекторе атрибутов кнопку `Alignment`. Выберите среднюю кнопку `Alignment`, чтобы отцентровать текст метки.

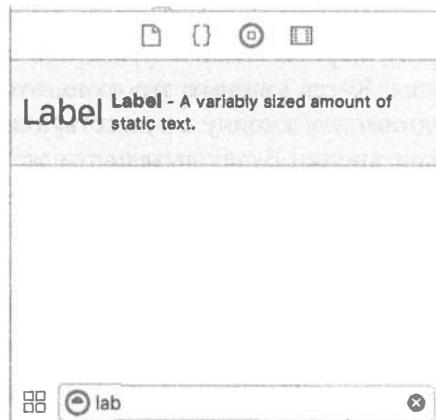


Рис. 3.12. Метка в библиотеке объектов

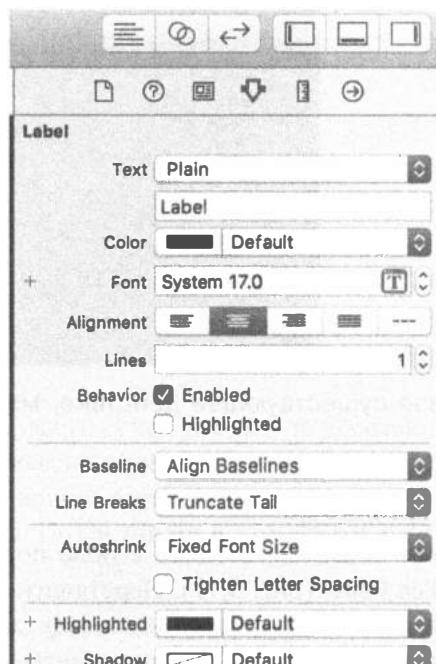


Рис. 3.13. Использование инспектора атрибутов метки для центрирования текста метки

Мы не хотим, чтобы на кнопке было что-нибудь написано, пока пользователь ее нажмет, поэтому дважды щелкните на метке (чтобы выбрать текст) и нажмите клавишу <Delete>. В результате текст, приписанный к кнопке в данный момент, будет удален. Нажмите клавишу <Return>, чтобы подтвердить исправления.

Даже если вы не видите метку, когда она не выбрана, не беспокойтесь — она на месте.

СОВЕТ. Если интерфейс содержит невидимые элементы, например пустые метки, и вы хотите их увидеть, выберите команду *Canvas* из меню *Editor*, а затем во всплывшем подменю установите флагок *Show Bounds Rectangles*. Если вы просто хотите выделить невидимый элемент, щелкните на пиктограмме в окне *Document Outline*.

Осталось только создать выход для метки. Эта процедура ничем не отличается от предыдущей. Откройте помощник редактора и файл *ViewController.swift*. Если возникнет необходимость переключать файлы, воспользуйтесь всплывающим меню на панели быстрого перехода, расположенным над помощником редактора.

Затем выберите метку в программе *Interface Builder* и, нажав клавишу *<Control>*, перетащите курсор от метки к заголовочному файлу. Установите его точно на требуемом методе действия. Увидев окна, показанные на рис. 3.14, отпустите кнопку мыши, и вы снова увидите всплывающее окно (см. рис. 3.9).

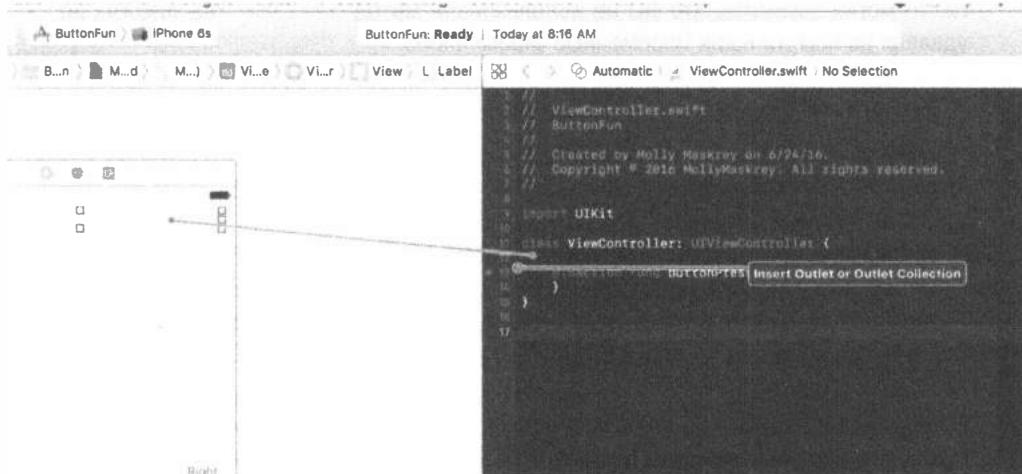


Рис. 3.14. Связывание выхода UILabel

Мы хотим создать выход, поэтому оставим тип *Connection* для объекта *Outlet*, заданный по умолчанию. Для того чтобы выбрать информативное имя для выхода, вспомним его назначение. Введите слово *statusLabel* в поле *Name*. Оставьте в поле *Type* значение *UILabel*. В последнем поле *Storage* значение можно оставить по умолчанию.

Нажмите клавишу *<Return>*, чтобы подтвердить изменения, и программа Xcode вставит свойство выхода в ваш код. Заголовочный файл контроллера будет содержать код, представленный в листинге 3.6.

Листинг 3.6. Добавление выхода метки в класс ViewController

```
import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var statusLabel: UILabel!
    @IBAction func buttonPressed(_ sender: UIButton) {
    }
}
```

Теперь у нас есть выход, и программа Xcode должна автоматически соединить с ним нашу метку. Это значит, что если мы изменим значение выхода `statusLabel` в коде, то это отобразится на метке в пользовательском интерфейсе. Если мы изменим свойство `text` для выхода `statusLabel`, то на экране изменится текст на метке.

АВТОМАТИЧЕСКИЙ ПОДСЧЕТ ССЫЛОК

Если вы знаете язык Objective-C или читали предыдущие издания настоящей книги, то могли заметить, что мы не использовали метод `dealloc`. Мы никогда не удаляли из памяти наши переменные экземпляров.

Компилятор LLVM, который компания Apple встроила в программу Xcode, настолько разумен, что освобождает объекты самостоятельно, используя новую функциональную возможность под названием Automatic Reference Counting (ARC).

Механизм ARC применяется только к объектам языка Swift и структурам, но не к объектам каркаса Core Foundation или объектам, размещенным в памяти с помощью функции `malloc()` или подобных ей функций. Есть еще несколько тонкостей и ловушек, но в целом управление памятью вручную осталось в прошлом.

Более полную информацию о механизме ARC можно найти по адресу <http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RNTransitioningToARC/>

Механизм ARC хорош, но не всемогущ. Необходимо хорошо понимать основные правила управления памятью в языке Objective-C, чтобы избежать неприятностей. Правила управления памятью в языке Objective-C можно найти в документе Memory Management Programming Guide, который компания Apple поместила на веб-странице <https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html>.

Создание метода действия

Итак, мы разработали пользовательский интерфейс и связали между собой его выходы и действия. Осталось только применить эти действия и выходы для того, чтобы изменить текст на кнопке при ее нажатии. Щелкните мышью в окне навигатора проекта на файле `ViewController.swift`, чтобы открыть его. Найдите пустой метод `buttonPressed()`, созданный программой Xcode.

Для того чтобы наши кнопки отличались одна от другой, будем использовать параметр `sender`. Мы извлечем название нажатой кнопки из параметра `sender`, создадим строку на основе этого значения и присвоим его тексту метки. Изменим метод `buttonPressed()` так, как показано в листинге 3.7.

Листинг 3.7. Завершение метода действия

```
@IBAction func buttonPressed(sender: UIButton) {
    let title = sender.title(for: .selected)!
    let text = "\(title) button pressed"
    statusLabel.text = text
}
```

Это довольно просто. Первая инструкция этого фрагмента извлекает название кнопки из параметра `sender`. Поскольку кнопки могут иметь разные названия в зависимости от текущей ситуации, мы используем параметр `UIControlStateNormal`, чтобы указать, что нам нужно название кнопки в ее нормальном, не нажатом состоянии. Это типично для всех элементов управления (а кнопка — это один из элементов управления). Состояния элементов управления рассматриваются в главе 4.

СОВЕТ. Возможно, вы заметили, что при вызове метода `title(for:)` мы использовали аргумент `.selected`, а не `UIControlState.selected`. По правилам языка Swift аргумент должен быть одним из значений перечисления `UIControlState`, поэтому мы можем пропустить имя перечисления, чтобы сэкономить количество набираемого текста.

Следующая инструкция создает новую строку, добавляя к названию кнопки слова `button pressed`. Таким образом, если речь идет о левой кнопке, которая имеет название `Left`, то при ее нажатии эта строка программы создаст строку `Left button Pressed`. Эта новая строка присваивается свойству метки `text`. Вот так изменяется текст на метке при ее нажатии.

Тестирование приложения Button Fun

Выберите команду `Product⇒Run`. Если компилятор или редактор связей выдаст ошибки, вернитесь в окно редактирования и сравните свой код с текстом в главе. Если компиляция прошла без ошибок, то программа Xcode запустит симулятор устройства iPhone и выполнит приложение. Когда вы нажмете левую кнопку, то экран должен выглядеть так, как показано на рис. 3.15.

На первый взгляд, все в порядке, но если приглядеться, то обнаружится, что чего-то не достает. Для того чтобы увидеть, чего именно, измените текущую схему, как показано на рис. 3.16, на iPhone SE и снова запустите приложение.

Снимок экрана, показанный на рис. 3.17, свидетельствует о проблемах. Левая кнопка работает, как надо, но сдвинута вправо, а правая кнопка вообще исчезла.

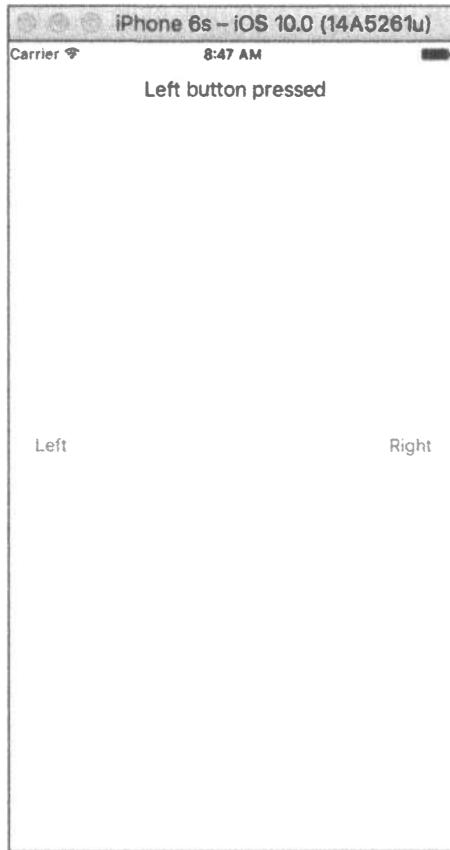


Рис. 3.15. Запуск приложение на iPhone 6s

Для того чтобы понять, почему это происходит, перейдите в среду Xcode и щелкните правой кнопкой в нижней части окна Interface Builder, чтобы выделить его и увидеть макет, а затем под окном выберите команду View As for iPhone SE, как показано на рис. 3.18. Поскольку мы настроили наш макет на устройство с более крупным экраном, при переходе на устройство с меньшим экраном некоторые элементы управления смещаются со своих позиций на новом дисплее.

Решение проблем с помощью механизма Auto Layout

Левая кнопка находится на правильном месте, а метка и другая кнопка — нет. В главе 2 мы исправили подобную проблему с помощью механизма Auto Layout. Идея механизма Auto Layout заключается в использовании ограничений, задающих место расположения элемента управления. В данном случае мы хотим добиться следующего.

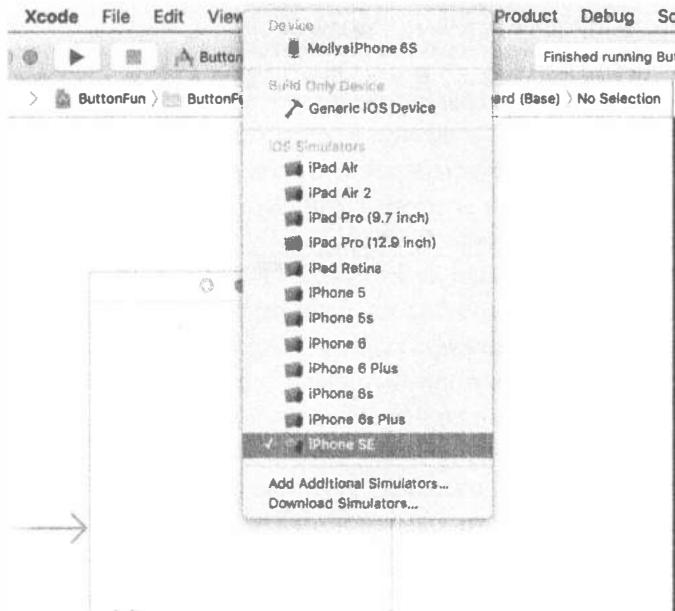


Рис. 3.16. Изменение схемы и целевого устройства на устройство с другими размерами и формой

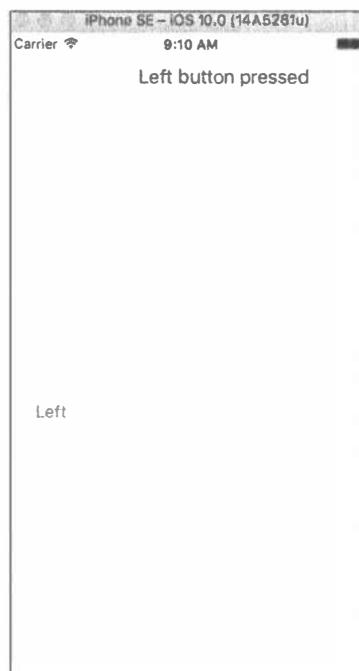


Рис. 3.17. На другом устройстве макет искажается

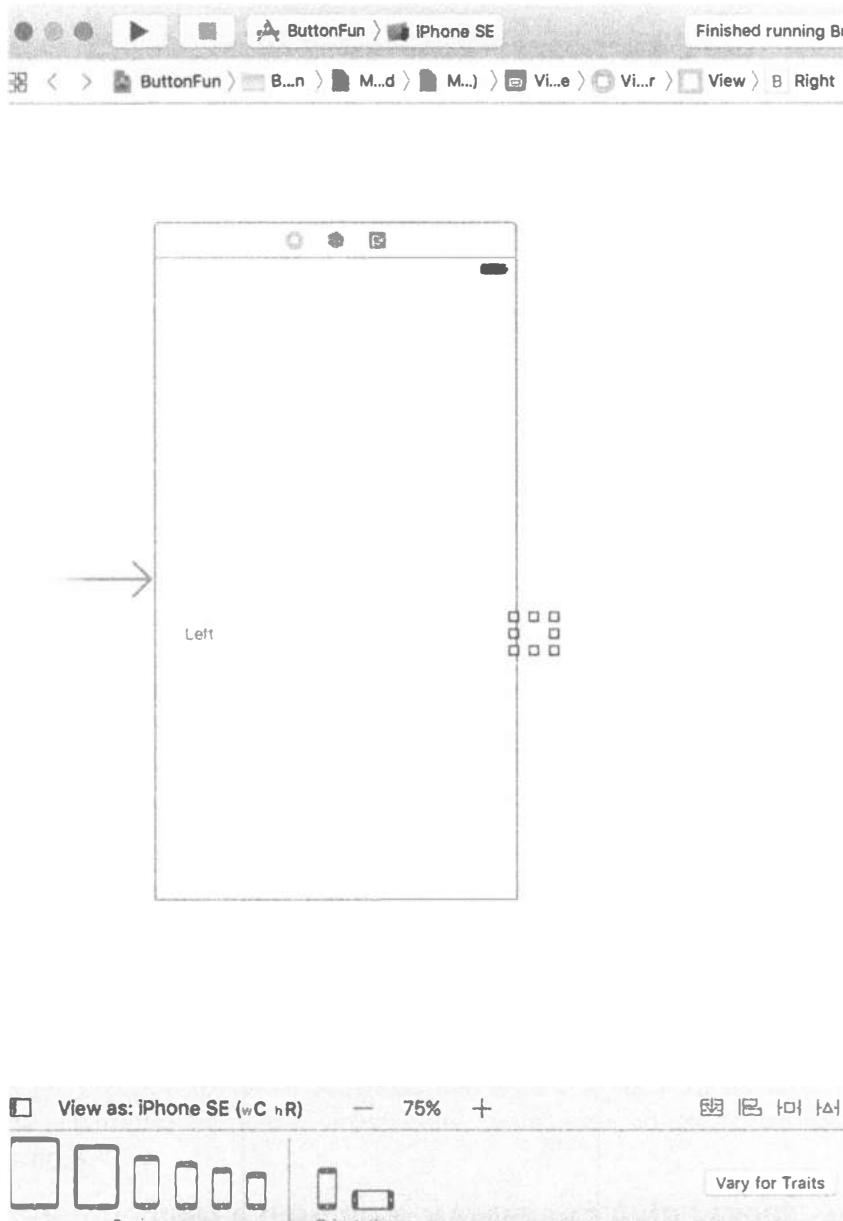


Рис. 3.18. На экране с меньшим экраном правая кнопка становится невидимой

- Кнопка **Left** должна быть отцентрована по вертикали и располагаться ближе к левому краю экрана.
- Кнопка **Right** должна быть отцентрована по вертикали и располагаться ближе к правому краю экрана.

Метка должна быть отцентрована по горизонтали и немного отступать от верхнего края экрана.

Каждое из приведенных выше утверждений содержит два ограничения — одно касается ограничения по вертикали, а другие — по горизонтали. Если применить эти три ограничения ко всем трем нашим представлениям, то механизм Auto Layout автоматически разместит эти элементы в правильных местах любого экрана. Как же нам это сделать? Мы можем добавить ограничения механизма Auto Layout в представления, создав экземпляры класса `NSLayoutConstraint`. В некоторых ситуациях это единственный способ создания правильного макета, но в данном случае (как и во всех остальных примерах, приведенных в книге) правильный макет можно создать с помощью программы Interface Builder. Эта программа позволяет визуально добавлять ограничения с помощью перетаскивания и щелчков мышью. Перейдите в окно `View As`, расположенное под окном IB, и снова выберите устройство `6s` в качестве целевого, чтобы увидеть все элементы управления. Затем задайте коэффициент масштабирования, например `75%`. Механизм Auto Layout можно применять и для настройки макетов на другие устройства (см. рис. 3.19).

Начнем с позиционирования метки. Выберите пункт `Main.storyboard` в окне навигатора проекта и откройте окно `Document Outline`, чтобы увидеть иерархию представлений. Найдите пиктограмму с меткой `View`. Она символизирует контроллер главного представления по отношению к которому мы должны позиционировать другие представления. Щелкните на треугольнике раскрытия, чтобы открыть пиктограмму `View`, если она еще не открыта, и найдите две кнопки (с метками `Left` и `Right`) и метку. Проведите соединительную линию от метки к родительскому представлению, как показано на левой панели рис. 3.20.

Перетаскивая указатель мыши с одного представления на другое, вы сообщаете программе Interface Builder, что хотите применить к ним ограничения механизма Auto Layout. Отпустите кнопку мыши — и на экране появится серое всплывающее окно с множеством команд (рис. 3.20). Каждая из этих команд представляет собой отдельное ограничение. Выбрав любую из этих команд, вы применяете соответствующее ограничение. Однако, как нам известно, нам необходимо применить к метке два ограничения, причем оба они есть в списке команд всплывающего меню. Для того чтобы применить сразу несколько ограничений, необходимо нажать и удерживать клавишу `Shift`, выбирая соответствующие команды. Итак, нажмите клавишу `Shift` и выберите команды `Center Horizontally in Container` и `Vertical Spacing to Top Layout Guide`. Для того чтобы эти ограничения были действительно применены, щелкните мышью в любом месте за пределами всплывающего меню или нажмите клавишу `<Return>`. После этого созданные вами ограничения появятся в разделе `Constraints` в окне `Document Outline`, а также будут представлены визуально в раскладовке, как показано на рис. 3.21.

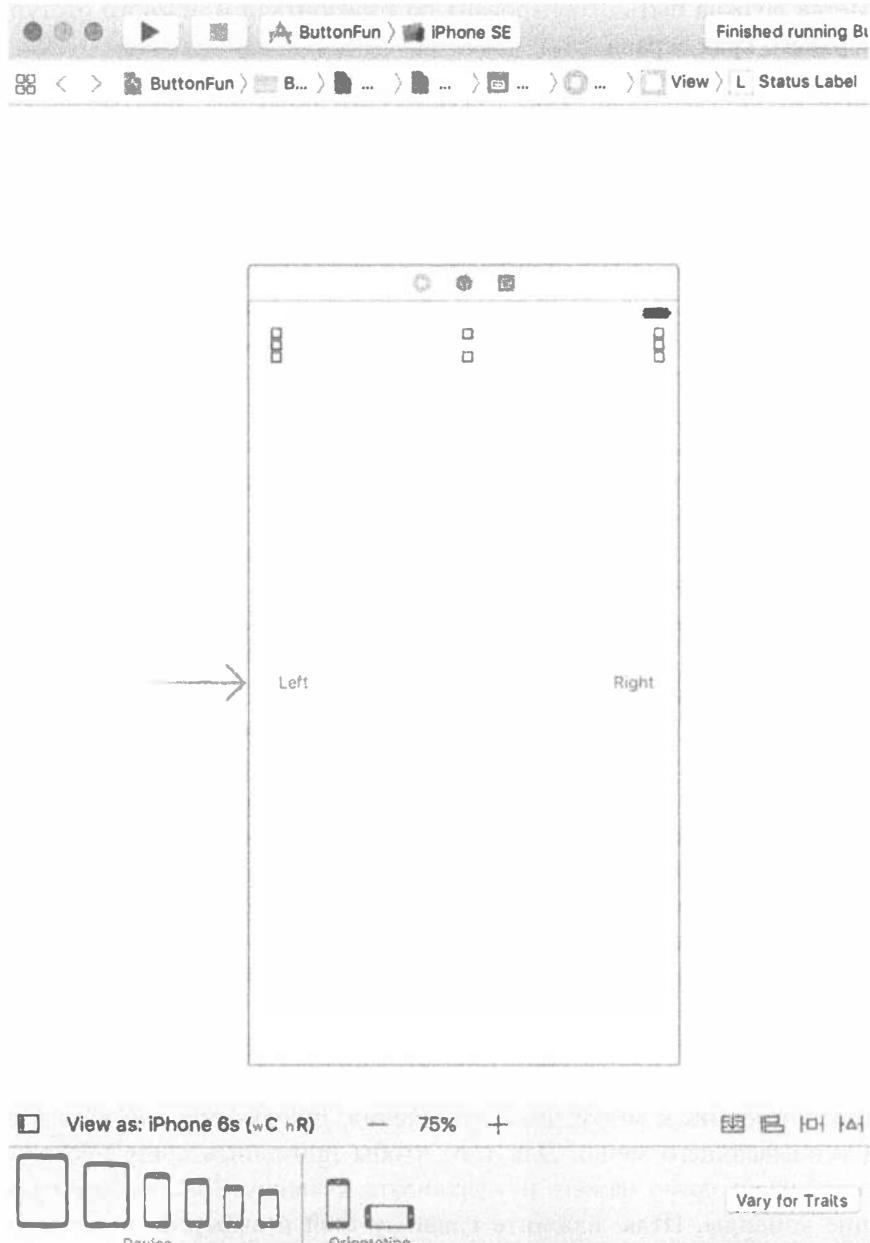


Рис. 3.19. Для настройки макета на другие устройства с помощью механизма Auto Layout мы используем то устройство, для которого мы изначально создавали свое приложение

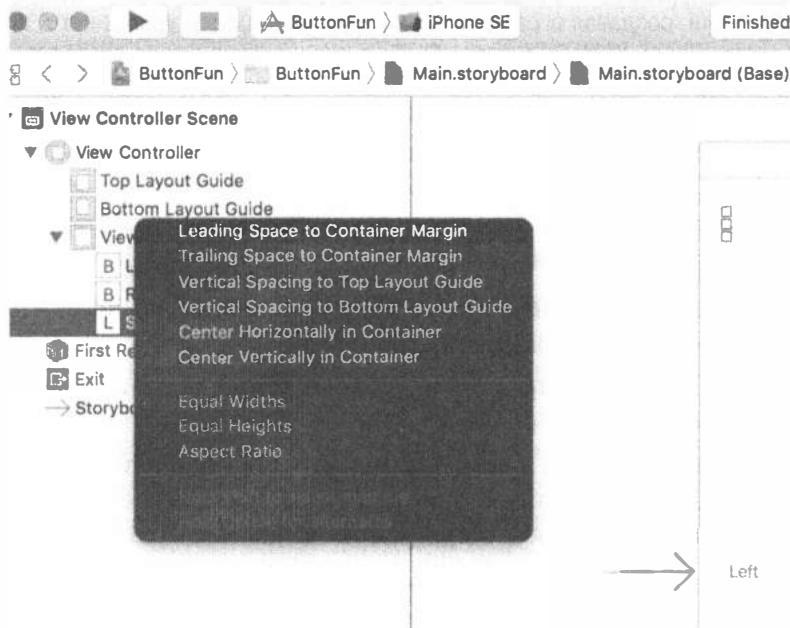


Рис. 3.20. Позиционирование метки с помощью ограничений Auto Layout



Рис. 3.21. Два ограничения Auto Layout, применяемые к метке

ПОДСКАЗКА. Если, создавая ограничение, вы сделали ошибку, удалите его, щелкнув на его представлении в окне Document Outline, или удалите его из раскладовки, нажав клавишу <Delete>.

Вероятно, вы заметили, что кнопка имеет оранжевый контур. Этим цветом программа Interface Builder отмечает возникновение проблемы в механизме Auto Layout. Существует три вида типичных проблем, о которых сообщает программа Interface, рисуя оранжевый контур.

- ➊ Вы задали недостаточное количество ограничений для того, чтобы точно указать позицию или размер представления.
- ➋ Вы задали неоднозначные ограничения, т.е. они задают размер или позицию не единственным образом.
- ➌ Ограничения являются правильными, но позиция и/или размер представления во время выполнения программы не совпадает с соответствующей позицией и/или размером в программе in Interface Builder.

Получить более подробную информацию о проблеме можно, щелкнув на оранжевом треугольнике в окне Activity View в навигаторе проблем (см. левую часть рис. 3.21). В результате вы увидите строку “Frame for ‘Label’ will be different at run time” (Рамка объекта ‘Label’ во время выполнения приложения будет другой), сообщающую о проблеме третьего вида. Это сообщение можно стереть, сделав так, чтобы программа Interface Builder переместила метку на правильную позицию и задала ее правильные размеры. Для этого обратите внимание на редактор раскладовок, расположенный в правом нижнем углу. Вы видите четыре кнопки, показанные на рис. 3.22.

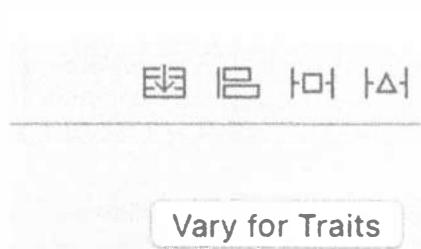


Рис. 3.22. Кнопки механизма Auto Layout в правом нижнем углу редактора раскладовок

Вы можете узнать, что делает каждая из этих кнопок, задержав указатель мыши над ними. Левая кнопка связана с элементом управления UIStackView, которую мы будем рассматривать в главе 10. Переходя слева направо, перечислим их функции.

1. Кнопка Align позволяет выровнять выбранное представление относительно другого представления. Щелкнув на этой кнопке, вы увидите всплывающее

меню, содержащее разные варианты выравнивания. Одна из этих кнопок — **Horizontal Center in Container** — соответствует ограничению, которое уже применялось к метке в окне Document Outline. Как правило, одну и ту же функцию механизма Auto Layout в программе Interface Builder можно выполнить несколькими способами. По мере чтения книги вы узнаете альтернативные способы выполнения задач, связанных с использованием механизма Auto Layout.

2. Всплывающее меню для кнопки **Pin** содержит команды, позволяющие задавать позицию представления относительно других представлений и применять ограничения размеров. Например, можно задать ограничение, требующее, чтобы высота одного представления совпадала с высотой другого.
3. Кнопка **Resolve Auto Layout Issues** позволяет решать проблемы, связанные с макетом. Меню, соответствующее этой кнопке, содержит команды, позволяющие удалять все ограничения, установленные для представления (или всю раскладовку), выяснять, какие ограничения пропущены, и добавлять их, а также уточнять рамки одного или нескольких представлений, которые должны применяться на этапе выполнения приложения.

Изменить рамку метки можно, выбрав ее в окне Document Outline или в раскладовке и щелкнув на кнопке **Resolve Auto Layout Issues**. Всплывающее меню для этой кнопки содержит две идентичные группы операций (рис. 3.23).

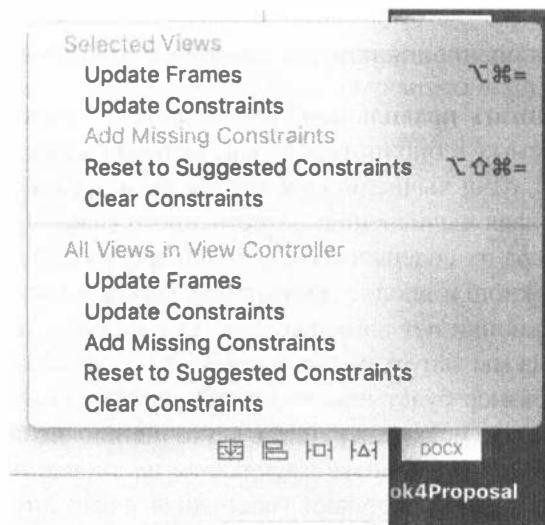


Рис. 3.23. Всплывающее меню для кнопки **Resolve Auto Layout Issues**

ПОДСКАЗКА. Если все команды всплывающего меню недоступны, щелкните на метке в окне Document Outline, чтобы обеспечить возможность их выбора.

Команды из верхней части меню относятся только к текущему представлению, а команды из нижней части меню — ко всем представлениям, связанным с контроллером представлений. В данном случае нам просто необходимо исправить рамку метки, поэтому мы выберем команду *Update Frames* в верхней части меню. После этого выберем оранжевый контур и треугольник предупреждения в окне *Activity View*, потому что теперь во время выполнения программы метка будет находиться на правильной позиции и иметь правильный размер. Фактически ширина метки будет уменьшена до нуля и будет представлена в раскладовке как маленький пустой квадратик (рис. 3.24).



Рис. 3.24. После исправления контура ширина кнопки уменьшилась до нуля

Можно ли это считать правильным? Оказывается, можно. Многие из представлений, поставляемых в библиотеке UIKit, включая класс *UILabel*, позволяют механизму Auto Layout вычислять их размер по их реальному содержимому. Это возможно благодаря вычислению естественного (*natural*), или действительного (*intrinsic*), размера их содержимого. С точки зрения действительного размера ширина и высота кнопки вполне достаточны, чтобы на ней поместился текст ее названия. Пока у кнопки нет названия, высота и ширина кнопки действительно равны нулю. Когда мы запустим приложение и щелкнем на одной из кнопок, ее действительный размер будет изменен и мы увидим весь текст.

Исправив недостатки метки, перейдем к уточнению позиций двух кнопок. Выберите кнопку *Left* на раскладовке и щелкните на кнопке *Align* в правом нижнем углу окна редактора раскладовки (последняя слева кнопка на рис. 3.22). Мы хотим, чтобы кнопка была отцентрована по вертикали, поэтому выберем команду *Vertical Center in Container* во всплывающем меню и щелкнем на кнопке *Add 1 Constraint* (рис. 3.25).

Это же ограничение необходимо применить к кнопке *Right*, поэтому выберем ее и повторим описанный процесс. В этом случае программа Interface Builder выявит несколько новых проблем, которые будут выделены оранжевым цветом

в раскладовке и отмечены треугольником предупреждения в окне Activity View. Щелкните на этом треугольнике, чтобы узнать о причинах ошибок, обнаруженных навигатором проблем (рис. 3.26).

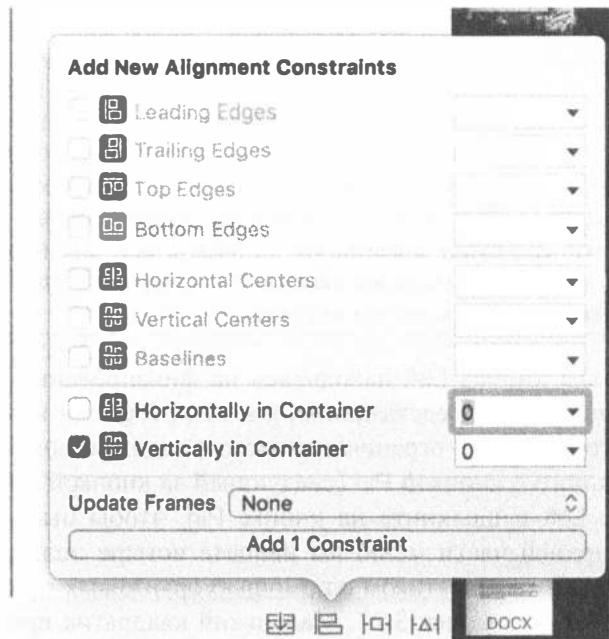


Рис. 3.25. Центрование представления по вертикали с помощью всплывающего меню Align

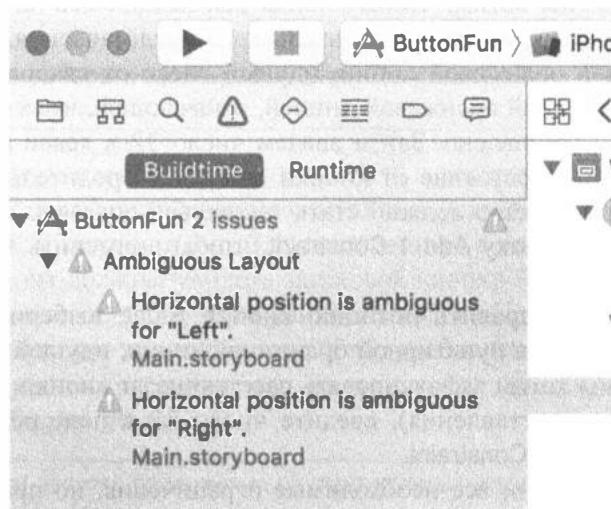


Рис. 3.26. Предупреждение программы Interface Builder об отсутствии ограничений

Программа Interface Builder предупреждает о том, что горизонтальные позиции обеих кнопок заданы неоднозначно. Фактически у нас нет ограничений, управляющих горизонтальными позициями кнопок, поэтому предупреждение не должно нас удивлять.

ЗАМЕЧАНИЕ. Задавая ограничения Auto Layout, вы будете часто получать подобные предупреждения. Они необходимы для того, чтобы вы задали полный набор ограничений. По завершении процесса разметки вы не должны получить ни одного предупреждения. Большинство примеров в этой книге содержат инструкции для задания ограничений разметки. Добавляя эти ограничения, вы, конечно, должны учитывать предупреждения, но беспокоиться стоит, только если они появляются после завершения всего процесса разметки. Это будет значить, что вы пропустили какой-то шаг, выполнили его неправильно или в книгу вкрадась ошибка! В последнем случае, пожалуйста, сообщите об ошибке по адресу www.apress.com.

Мы хотим, чтобы кнопка *Left* находилась на фиксированном расстоянии от левого края родительского представления, а кнопка *Right* — на том же расстоянии от его правого края. Эти ограничения можно задать с помощью всплывающего меню, связанного с кнопкой *Pin* (следующей за кнопкой *Align* на рис. 3.22). Выберите кнопку *Left* и щелкните на кнопке *Pin*, чтобы открыть ее всплывающее меню. В верхней части меню вы найдете четыре поля редактирования, связанных с маленьким квадратиком с помощью оранжевых пунктирных линий, показанных на левой части рис. 3.27. Маленький квадратик представляет кнопку, для которой задаются ограничения. Четыре поля редактирования позволяют задать расстояние от кнопки до ближайших соседей над и под ней, а также слева и справа от нее. Пунктирная линия означает, что соответствующего ограничения пока нет. Мы хотим, чтобы кнопка *Left* находилась на фиксированном расстоянии от левого края своего родительского представления, поэтому щелкнем на пунктирной оранжевой линии, идущей влево от квадрата. В этом случае она станет сплошной оранжевой линией, означающей, что соответствующее ограничение уже установлено. Затем введем число 32 в левое поле редактирования, чтобы задать расстояние от кнопки *Left* до его родительского представления. Всплывающее меню должно стать таким, как показано на правой части рис. 3.22. Нажмите кнопку *Add 1 Constraint*, чтобы применить это ограничение к данной кнопке.

Для того чтобы исправить позицию кнопки *Right*, выберите ее, нажмите кнопку *Pin*, щелкните на пунктирной оранжевой линии, идущей вправо от квадрата (поскольку мы хотим зафиксировать расстояние от кнопки до правого края ее родительского представления), введите число 32 в поле редактирования и нажмите кнопку *Add 1 Constraint*.

Теперь мы применили все необходимые ограничения, но предупреждения в окне *Activity View* не исчезли. Анализ показывает, что на этапе выполнения приложения кнопки будут располагаться неправильно. Для того чтобы исправить эту проблему, необходимо нажать кнопку *Resolve Auto Layout Issues*. В результате

откроется всплывающее меню, в котором надо выбрать команду **Update Frames** в нижнем разделе. Это ограничение будет касаться рамок всех представлений в настраиваемом контроллере представлений.



Рис. 3.27. Выравнивание представления по горизонтали с помощью всплывающего меню Pin

ПОДСКАЗКА. Иногда во всплывающем меню не все команды являются доступными.

В этом случае выберите пиктограмму **View Controller** в окне **Document Outline** и попытайтесь снова.

Теперь предупреждения должны исчезнуть и разметка будет, наконец, завершена. Запустите свое приложение на симуляторе устройства iPhone — и увидите результат, очень похожий на рис. 3.1, помещенный в начале главы. Коснувшись правой кнопки, вы должны увидеть заголовок кнопки **Right button pressed**. Коснувшись левой кнопки, вы должны увидеть заголовок кнопки **Left button pressed**. Запустите приложение на симуляторе iPad, и обнаружите, что макет все еще работает, хотя кнопки расположены далеко одна от другой, потому что экран iPad шире, чем экран iPhone.

ПОДСКАЗКА. При запуске приложения на имитируемых устройствах с большими экранами иногда невозможно увидеть весь экран сразу. Этую проблему можно исправить, выбрав команду **Window⇒Scale** в меню программы iOS Simulator и выбрав коэффициент масштабирования для своего экрана.

Взглянув на рис. 3.1, вы увидите, что одну вещь мы пропустили. Заглавие выбранной кнопки должно выводиться на экран полужирным шрифтом, а пока мы видим обычный текст. Немного позже мы исправим это с помощью класса `NSAttributedString`, а сначала ознакомимся с другой полезной функцией среды Xcode — предварительным просмотром макета.

Предварительный просмотр макета

Вернитесь в программу Xcode и выберите узел `Main.storyboard`, а затем откройте окно помощника редактора, если оно закрыто (как это сделать, показано на рис. 3.6). В левой половине панели быстрых переходов, расположенной в верхней части окна помощника редактора, вы увидите, что в данный момент выбран атрибут `Automatic` (если вы не изменили его на `Manual`, чтобы выбрать файл в окне помощника редактора). Щелкните на сегменте панели быстрых переходов, чтобы открыть всплывающее меню, и увидите несколько команд, последняя из которых называется `Preview`. Если установить на нее указатель мыши, появится меню, содержащее имя раскладовки приложения. Щелкните на нем, чтобы открыть раскладовку в окне `Preview Editor`.

Когда откроется окно `Preview Editor`, вы увидите внешний вид приложения на устройстве iPhone в книжной ориентации. Это всего лишь предварительный просмотр, поэтому никаких реакций на касание кнопок нет, а значит, вы не увидите метку. Если вы переместите мышь на область, расположенную ниже области предварительного просмотра, где написано `iPhone 6s`, появится элемент управления, позволяющий поворачивать телефон и переводить экран в альбомную ориентацию. Этот элемент управления изображен в левой части рис. 3.28. Щелкнув на нем, вы выполните вращение телефона по часовой стрелке.

Благодаря механизму `Auto Layout` при вращении телефона кнопки перемещаются так, чтобы их центровка и расстояния были такими, как в книжной ориентации. Если метка остается видимой, то ее позиция также будет правильной.

Кроме того, помощник предварительного просмотра можно использовать для того, чтобы увидеть, что произойдет, если запустить приложение на другом устройстве. В левом нижнем углу окна помощника предварительного просмотра (и на рис. 3.28) вы увидите кнопку `+`. Щелкните на ней, чтобы открыть список устройств, а затем выберите пункт `iPad`, чтобы иметь возможность предварительного просмотра экрана `iPad` в окне помощника предварительного просмотра. Экран `iPad` занимает много места, поэтому, возможно, вам придется закрыть окна `Document Outline` и `Utility View`, чтобы можно было видеть и экран `iPhone`, и экран `iPad`. Если вы все еще не видите полный экран устройства `iPad`, то можете масштабировать окно `Preview Assistant`, используя разные способы. Проще всего дважды щелкнуть на панели `Preview Assistant` — и оно станет значительно меньше. Если вы хотите задать конкретный коэффициент масштабирования, то можете выполнить жест щипка на мультисенсорной панели (к сожалению, мышь `Magic Mouse` эту возможность не поддерживает). Окна предварительного просмотра экранов `iPhone` и `iPad`, уменьшенные так, чтобы их было видно

полностью, показаны на рис. 3.29. Обратите внимание на то, что механизм Auto расположил кнопки правильно. Поверните окно предварительного просмотра экрана iPad, чтобы убедиться в том, что приложение правильно работает в альбомном режиме.

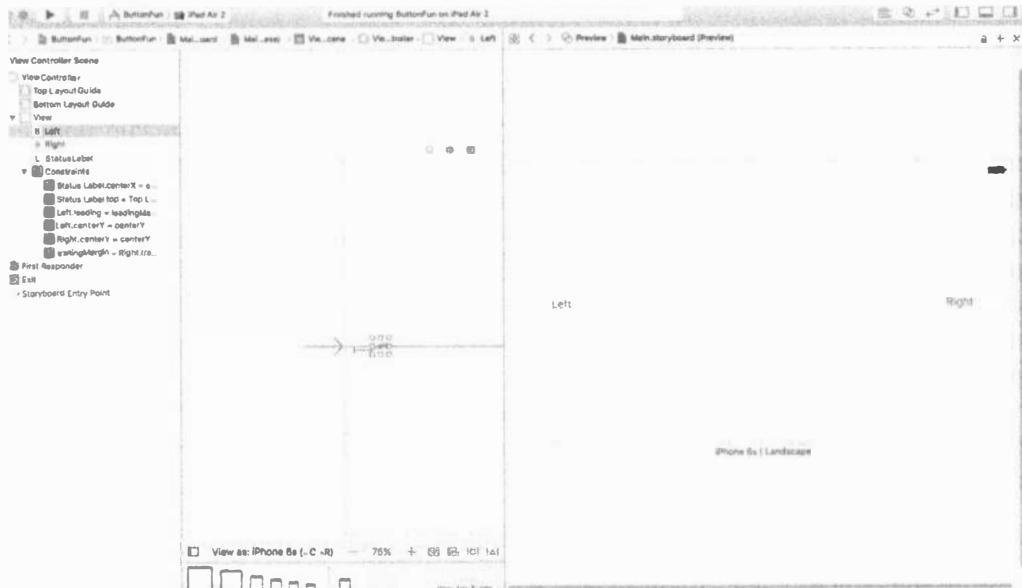


Рис. 3.28. Предварительный просмотр разметки на экране устройства iPhone альбомной ориентации

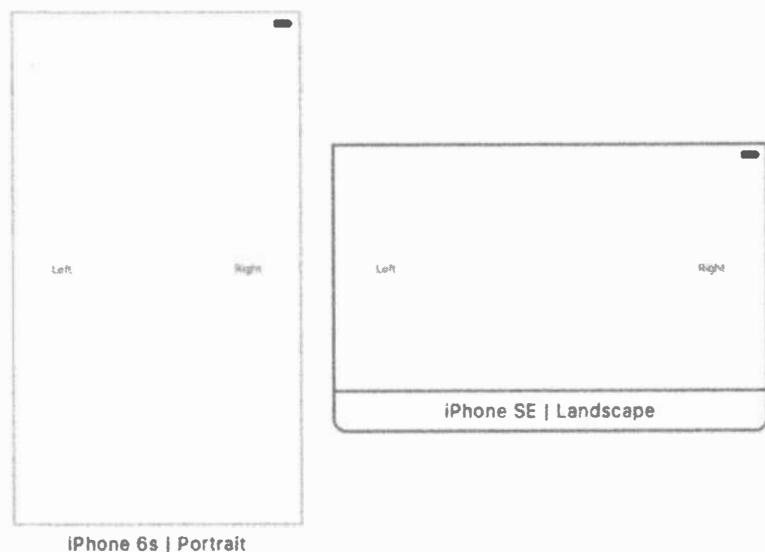


Рис. 3.29. Одновременный предварительный просмотр экранов iPhone и iPad

ЗАМЕЧАНИЕ. Когда мы работали над книгой, панель Preview Assistant некорректно отображала макеты для iPad с экранами 9,7 и 12,9 дюйма. Возможно, это связано с ошибками бета-версии Xcode 8. Однако выполнение соответствующих приложений на симуляторе показывает, что они работают корректно.

Изменение стиля текста

Класс NSAttributedString позволяет добавлять информацию о формате, например о шрифтах и выравнивании абзацев. Эти метаданные можно применять ко всей строке, причем разные атрибуты можно применять к разным частям интерфейса. Для того чтобы понять, как работает класс NSAttributedString, достаточно вспомнить, как форматируется текст в текстовом редакторе. Большинство элементов управления в библиотеке UIKit позволяют использовать строки с атрибутами. В случае класса UILabel, который мы используем в своем приложении, можно просто создать строку с атрибутами и передать ее метке с помощью ее свойства attributedText.

Итак, выберем файл ViewController.swift и обновим метод buttonPressed() так, как показано в листинге, удалив перечеркнутую строку и добавив строки, приведенные в листинге 3.8.

Листинг 3.8. Обновление метода buttonPressed() для полужирного шрифта

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var statusLabel: UILabel!

    @IBAction func buttonPressed(_ sender: UIButton) {
        let title = sender.title(for: .selected)!
        let text = "\(title) button pressed"
        let styledText = NSMutableAttributedString(string: text)
        let attributes = [
            NSFontAttributeName:
                UIFont.boldSystemFont(ofSize: statusLabel.font.pointSize)
        ]
        let nameRange = (text as NSString).range(of: title)
        styledText.setAttributes(attributes, range: nameRange)

        statusLabel.attributedText = styledText
    }
}
```

Сначала новый код создает строку с атрибутами, в частности объект класса NSMutableAttributedString, на основе строки, которую вы хотите вывести на экран. Нам нужна строка с атрибутами, допускающая изменения, потому что мы собираемся изменять ее атрибуты.

Далее мы создаем словарь для хранения атрибутов, которые будут применяться к строке. На самом деле у нас пока только один атрибут, поэтому словарь содержит единственную пару “ключ–значение”. Ключ `NSFontAttributeName` позволяет задать шрифт для части строки с атрибутами. Значение, которое мы передаем, иногда называют “полужирный системный шрифт”. Оно означает, что размер шрифта строки с атрибутами должен совпадать с размером шрифта, который в данный момент используется для метки. Такое задание шрифта является более гибким, чем указание шрифта по названию, поскольку система сама знает, как правильно использовать полужирный шрифт.

Теперь мы попросим строку `plainText` сообщить нам диапазон (состоящий из начального индекса и длины) подстроки, содержащей название метки. Применим эти атрибуты к строке с атрибутами и передадим ее метке. Рассмотрим строку, задающую координаты строки заголовка.

```
let nameRange = (text as NSString).range(of: title)
```

Обратите внимание на то, что тип переменной `plainText` приводится из типа `String` языка Swift в тип `NSString` каркаса Core Foundation. Это необходимо, потому что оба класса, `String` и `NSString`, имеют метод `range(of: String)`. Метод класса `NSString` задает диапазон в виде объекта `NSRange`, потому что именно его ожидает метод `setAttributes()` в следующей строке программы.

Теперь можно щелкнуть на кнопке `Run`; вы увидите, что приложение показывает название нажатой кнопки с помощью полужирного шрифта, как на рис. 3.1.

Использование делегата приложения

Теперь, когда наше приложение работает, прежде чем переходить к новой теме, уделим несколько минут изучению исходного файла, который мы еще не просматривали: `AppDelegate.swift`. Этот файл является реализацией делегата приложения (`application delegate`).

Делегаты широко используются в каркасе Cocoa Touch. Они представляют собой классы, решающие определенные задачи от имени другого объекта. Делегат приложения позволяет выполнять определенные действия в заранее установленное время от имени класса `UIApplication`. Каждое приложение для системы iOS имеет один и только один экземпляр класса `UIApplication`, обеспечивающий выполнение приложения и реализующий его функциональные возможности, такие как направление входных данных соответствующему классу контроллера. Класс `UIApplication` является стандартной частью библиотеки `UIKit` и выполняет свою работу практически незаметно, так что в большинстве случаев о нем не приходится беспокоиться.

В определенные точно заданные моменты времени в ходе выполнения приложения класс `UIApplication` вызывает установленные методы делегата, при условии, что делегат, реализующий этот метод, действительно существует. Например,

если у вас есть код, который должен сработать непосредственно перед завершением программы, можете реализовать метод `applicationWillTerminate()` в делегате своего приложения и поместить этот код в него. Данный тип делегирования позволяет вашему приложению реализовать обычное поведение без создания подкласса класса `UIApplication` и даже без информации о том, как он работает.

Щелкните на файле `AppDelegate.swift` в окне навигатора проекта, и увидите содержимое заголовочного файла делегата своего приложения (листинг 3.9).

Листинг 3.9. Первоначальный код делегата приложения

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?
```

Выделенная полужирным шрифтом часть строки означает, что класс поддерживает протокол `UIApplicationDelegate`. Нажмите и удерживайте клавишу `<Option>`. Ваш курсор имеет вид ножниц. Переместите курсор на слово `UIApplicationDelegate`. Он имеет вид знака вопроса, а слово `UIApplicationDelegate` будет выделено как ссылка браузера (рис. 3.30).

```
9 import UIKit
10
11 @UIApplicationMain
12 class AppDelegate: UIResponder, UIApplicationDelegate {
13
14     var window: UIWindow?
15
```

Рис. 3.30. После того как вы нажали клавишу `<Option>` в среде Xcode и указали на символ в нашем коде, этот символ стал выделенным, а курсор принял вид знака вопроса

Продолжая удерживать нажатой клавишу `<Option>`, щелкните на этой ссылке. Откроется маленькое всплывающее окно с кратким описанием протокола `UIApplicationDelegate` (рис. 3.31).

Проходя по всплывающему меню сверху вниз, вы найдете две ссылки (см. рис. 3.32).

Обратите внимание на две ссылки, расположенные в нижней части всплывающего окна документации. Щелкните на ссылке `More`, чтобы увидеть полную документацию для этого символа, или на ссылке `Declared`, чтобы увидеть определение символа в заголовочном файле. Аналогичный трюк работает и для имен классов, протоколов и категорий, а также для методов, отображаемых на панели редактирования. Просто дважды щелкните на слове, и программа Xcode найдет для вас это слово в браузере документации.

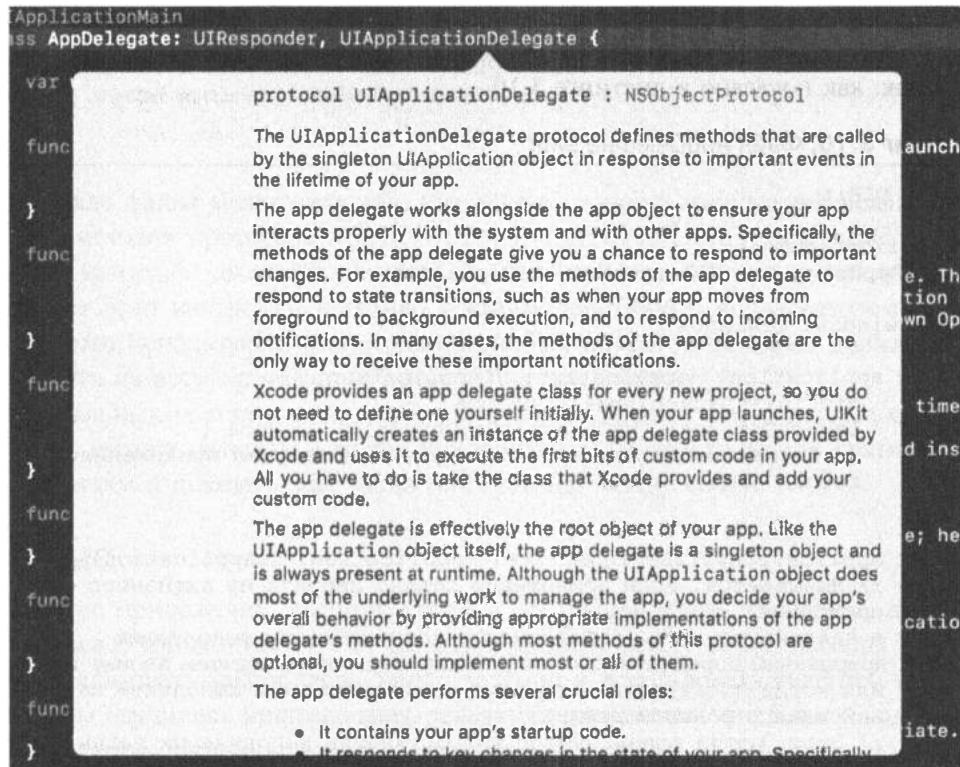


Рис. 3.31. После того как вы нажали клавишу <Option> и щелкнули на ссылке `UIApplicationDelegate` в исходном коде, программа Xcode открыла панель Quick Help с описанием требуемого протокола

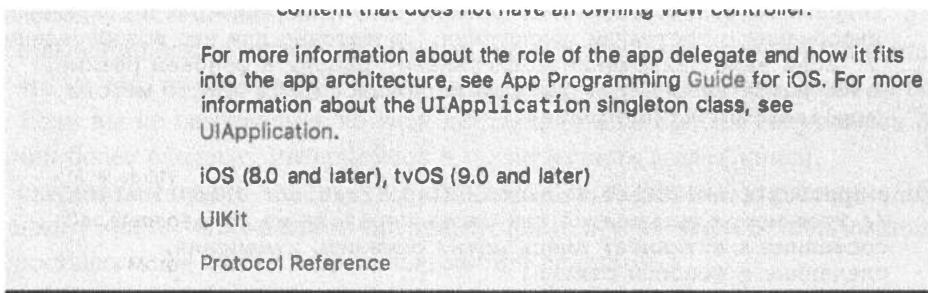


Рис. 3.32. Ссылки на дополнительную информацию о выделенном элементе

Умение быстро находить нужную информацию о протоколе в документации, безусловно, важно, но еще важнее иметь возможность видеть определение протокола. Именно здесь вы можете узнать, какие методы делегата приложения можно реализовать и когда эти методы будут вызваны. Вероятно, стоит потратить время на изучение описания этих методов.

Вернитесь в окно навигатора проекта и щелкните на файле AppDelegate.swift, чтобы увидеть делегат приложения. Содержимое файла должно выглядеть так, как показано в листинге 3.10.

Листинг 3.10. Файл AppDelegate.swift

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    func application(_ application: UIApplication,
                    didFinishLaunchingWithOptions
        launchOptions: [NSObject: AnyObject]?) -> Bool {
        // Точка замещения для настройки после запуска приложения.
        return true
    }

    func applicationWillResignActive(_ application: UIApplication) {
        // Вызывается, если приложение должно перейти из активного
        состояния
        в неактивное. Эта необходимость возникает при выполнении
        прерываний определенного типа (например, при входящем звонке или SMS)
        или когда пользователь выходит из приложения и выполняет какие-то
        действия в фоновом режиме.
        // Этот метод используется для остановки выполняемых задач,
        отключения таймеров и замедления прорисовки кадров OpenGL ES, а также
        в играх для организации паузы.
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        // Этот метод используется для освобождения общих ресурсов,
        сохранения пользовательских данных, обнуления таймеров и сохранения
        информации о состоянии приложения, достаточно для его возобновления
        // Если ваше приложение поддерживает работу в фоновом режиме,
        этот метод вызывается при выходе пользователя вместо метода
        applicationWillTerminate:.
    }

    func applicationWillEnterForeground(application: UIApplication) {
        // Этот метод вызывается как часть перехода из фонового
        состояния в активное; здесь можно отменить изменения,
        сделанные в фоновом режиме.
    }

    func applicationDidBecomeActive(application: UIApplication) {
        // Возобновляет выполнение остановленных задач (или запускает
        еще не стартовавшие), оставляя приложение неактивным. Если приложение
        работало в фоновом режиме, пользовательский интерфейс может быть
        прорисован заново.
    }
}
```

```
func applicationWillTerminate(application: UIApplication) {
    // Вызывается во время прекращения работы приложения.
    Сохраняет данные, если есть возможность.
    См. также applicationDidEnterBackground:.
}
```

В начале файла можно увидеть, что делегат нашего приложения реализовал один из методов протокола `application(_: didFinishLaunchingWithOptions:)`, который, как легко догадаться, выполняется, как только приложение завершает этап настройки и готово к взаимодействию с пользователем. Этот метод часто используется для создания любых объектов, которые должны существовать на всем протяжении выполнения приложения.

В остальных частях книги, особенно в главе 15, мы узнаем, насколько важную роль играют делегаты в жизни приложений. Мы просто хотели кратко описать делегаты и показать, как тесно они связаны между собой.

Резюме

Простое приложение, рассмотренное в настоящей главе, позволило вам ознакомиться с концепцией MVC, создать и связать между собой выходы и действия, реализовать контроллеры представлений и использовать делегаты приложений. Вы научились инициировать действия при нажатии кнопки и узнали, как изменить метку кнопки во время выполнения программы. Несмотря на простоту созданного приложения, основные концепции, рассмотренные нами, совпадают с концепциями, лежащими в основе всех других элементов управления в системе iOS, а не только кнопок. Способ использования кнопок и меток, продемонстрированный в главе, прекрасно работает и со всеми другими элементами управления в системе iOS.

Чрезвычайно важно, чтобы вы поняли, что и почему мы делали в этой главе. Если это не так, то вернитесь к началу и повторяйте все действия, пока не поймете. Если вы не разобрались во всех деталях, то еще больше запутаетесь при создании более сложных интерфейсов в последующих главах книги.

В следующей главе мы изучим стандартные элементы управления iOS. Вы также узнаете, как выдавать предупреждения и уведомления пользователям и предоставлять им выбор с помощью листов действий.

ГЛАВА 4



Новые упражнения с интерфейсом

В главе 3 мы обсудили концепцию MVC и написали приложение, воплощающее ее в жизнь. Вы ознакомились с выходами и действиями и использовали их для связывания кнопки с текстовой меткой. В настоящей главе мы планируем создать приложение, которое повысит уровень ваших знаний об элементах управления (рис. 4.1).

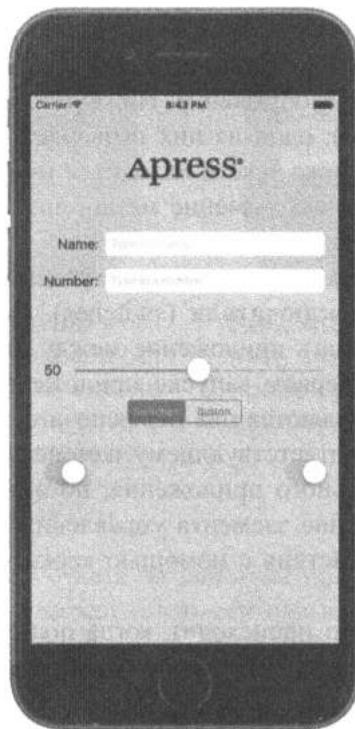


Рис. 4.1. Приложение с несколькими элементами управления

Мы создадим графическое представление, ползунок, два разных поля редактирования, сегментированный элемент управления, несколько переключателей и кнопку системы iOS (до версии 7). Мы покажем, как задать и определить значения разных элементов управления; как использовать список действий, чтобы заставить пользователя сделать выбор, а также предупреждения, обеспечивающие обратную связь с пользователем. Вы ознакомитесь с состояниями элементов управления и научитесь использовать растягиваемые изображения для создания кнопок желаемого размера.

Поскольку в этой главе используется много разных терминов, обозначающих элементы пользовательского интерфейса, мы поступим немного иначе, чем в двух предыдущих главах, а именно: разобьем наше приложение на части, разработаем каждую из них по отдельности, настроим и выполним в среде Xcode и симуляторе iOS, тестируя каждую часть, прежде чем перейти к следующей. Разделяя процесс построения сложного интерфейса на более мелкие части, мы упрощаем задачу и приближаемся к реальному процессу проектирования приложений. Цикл “кодирование–компиляция–отладка” занимает большую часть рабочего времени разработчика программного обеспечения.

Наше приложение будет использовать только одно представление и контроллер, но, как показано на рис. 4.1, это представление намного сложнее приложения, написанного в главе 3.

Логотип на экране устройства iPhone называется **графическим представлением** (image view). В этом приложении графическое представление просто выводит на экран статическое изображение. Ниже логотипа расположены два **поля редактирования** (text field): одно из них позволяет вводить буквы и цифры, а второе — только цифры. Ниже текста находится **ползунок** (slider). Когда пользователь перемещает ползунок, значение метки, следующей за ним, изменяется и всегда отображает его значение.

Под ползунком расположены **сегментированный элемент управления** (segmented control) и два **переключателя** (switches). Сегментированный элемент управления будет переключать приложение между двумя разными типами элементов управления. При первом запуске приложения под сегментированным элементом управления появляются два переключателя. Изменение значения одного из них приводит к соответствующему изменению другого. Это не совсем то, чего мы хотим от реального приложения, но это позволяет продемонстрировать, как изменить значение элемента управления программным путем и как анимировать некоторые действия с помощью каркаса Cocoa Touch, не прикладывая никаких усилий.

На рис. 4.2 показано, что происходит, когда пользователь нажимает сегментированный элемент управления. Переключатели исчезают и заменяются кнопками.

Если нажата кнопка **Do Something**, на экране всплывает **список действий** (action sheet), с помощью которого приложение спрашивает у пользователя, что он имел в виду, когда нажимал кнопку (рис. 4.3).

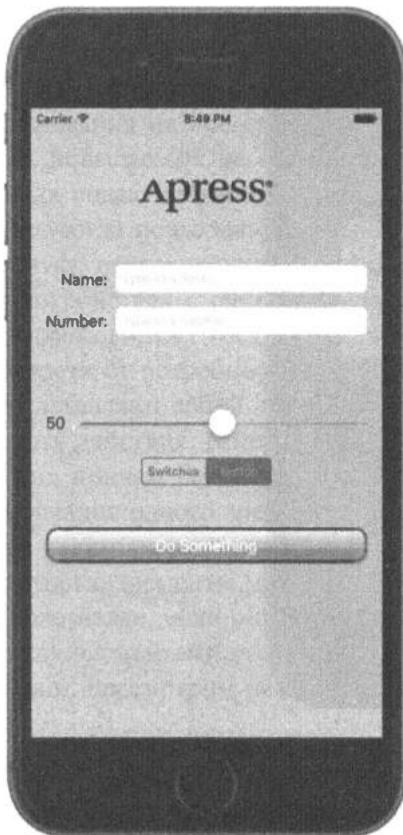


Рис. 4.2. В результате нажатия сегментированного контроллера, расположенного слева, на экране появляются два переключателя, а после нажатия сегментированного контроллера, расположенного справа, появляется кнопка

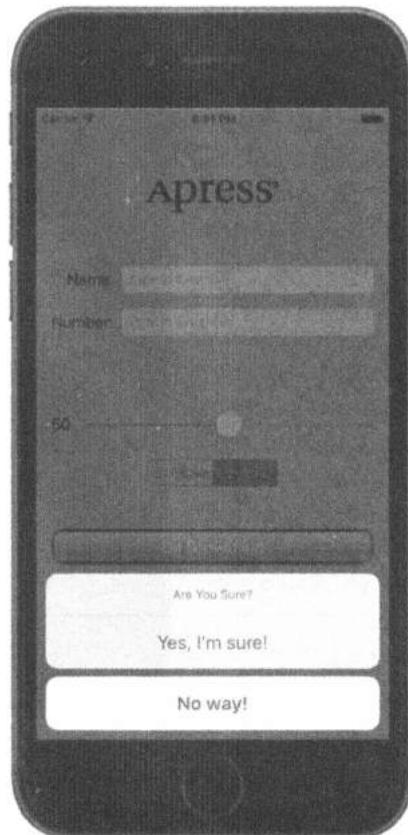


Рис. 4.3. Для того чтобы заставить пользователя дать ответ, приложение использует список действий

Это стандартный способ ответа на действие пользователя, которое является потенциально опасным или может иметь серьезные последствия. Он дает пользователям возможность остановить потенциально опасное развитие событий. Если пользователь выберет команду **Yes, I'm Sure!**, приложение выведет на экран предупреждение, сообщающее, что все в порядке (рис. 4.4).

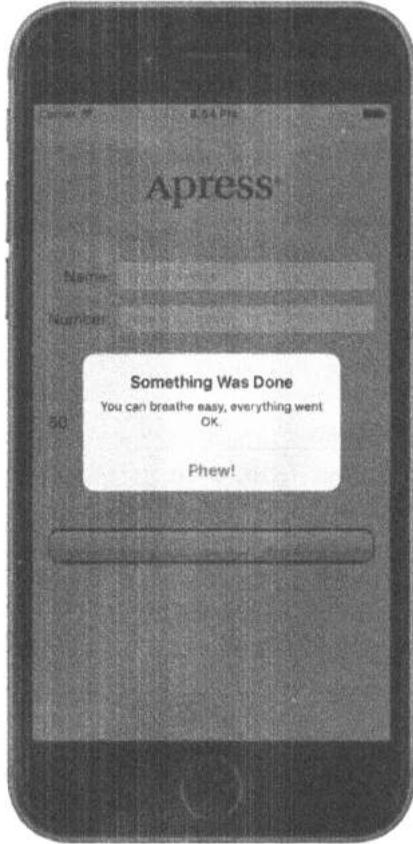


Рис. 4.4. Для уведомления пользователя о важных событиях используются предупреждения. Здесь показано предупреждение, сообщающее, что все идет хорошо

Активные, статические и пассивные элементы управления

Существуют три основные категории элементов управления интерфейсом: активные, статические (неактивные) и пассивные. Кнопки, использованные в предыдущей главе, представляют собой классический пример активных элементов управления. Вы нажимаете их, и что-то происходит — обычно выполняется часть программы.

Хотя многие элементы управления, которые вы будете использовать, непосредственно вызывают какой-нибудь метод действия, не все элементы управления работают именно так. Графическое представление, которое мы реализуем в этой главе, всего лишь демонстрирует изображение; несмотря на то что его можно настроить на запуск действий, пользователь ничего не сможет с ним делать. Поля редактирования и изображения часто используются таким образом.

Некоторые элементы управления носят пассивный характер. Они просто хранят какое-то значение, введенное пользователем, пока вы его не прочитаете. Эти элементы управления не инициируют никаких действий, но пользователь может взаимодействовать с ними и изменять их значения. Классический пример пассивного элемента управления — поле редактирования на веб-странице. Несмотря на то что может существовать некий код проверки, который выполняется, когда вы переключаетесь на поле редактирования, подавляющее большинство полей редактирования на веб-странице просто являются контейнерами для хранения данных, посылаемых на сервер, когда вы щелкаете на кнопке Submit. Самы поля редактирования не являются инициаторами выполнения какого-либо кода, но, когда пользователь щелкает на кнопке Submit, данные, которые в них содержатся, отправляются по назначению.

На устройствах с системой iOS большинство элементов управления можно использовать во всех трех режимах как в активном, так и в пассивном режимах в зависимости от потребностей пользователя. Все элементы управления в системе iOS представляют собой подклассы класса UIControl и поэтому способны инициировать действия. Многие элементы управления могут также использоваться пассивно, причем все они могут быть пассивными и невидимыми. Например, использование одного элемента управления может приводить к инициированию другого неактивного элемента управления, который становится активным. Однако некоторые элементы управления, такие как кнопки, действительно не могут быть полезными, если они не активны.

Между элементами управления в системах iOS и Mac существует разница в поведении. Рассмотрим несколько примеров.

- Благодаря мультисенсорному интерфейсу все элементы управления в системе iOS могут инициировать несколько действий в зависимости от того, как выполнено прикосновение к ним. Пользователь может инициировать разные действия, скользя пальцем поперек элемента управления или просто касаясь его.
- Можно инициировать одно действие, когда пользователь прикасается к кнопке, и другое — когда отнимает палец от кнопки.
- Один элемент управления может инициировать несколько действий в ответ на одно событие. Например, в ответ на прикосновение к внутренней области кнопки можно инициировать два разных действия, т.е. оба действия будут выполнены, когда пользователь отнимет палец от этой кнопки.

ЗАМЕЧАНИЕ. Несмотря на то что элементы управления в системе iOS могут вызывать сразу несколько методов, вероятно, было бы лучше реализовать один метод действия, который делает все, что необходимо при определенном состоянии элемента управления. Обычно такая возможность не нужна, но ее следует иметь в виду, работая с программой Interface Builder. Установление новой связи между событием и действием не разрывает предыдущую связь с другим действием этого же элемента

управления. Это может привести к неожиданным отклонениям в работе вашего приложения, поэтому, устанавливая связи между событиями и действиями, следует проявлять осторожность.

Другое важное различие между системами iOS и Mac является следствием того факта, что, как правило, устройства с системой iOS не имеют физической клавиатуры (если, конечно, вы не присоединили внешнюю клавиатуру). Стандартная клавиатура системы iOS является обычным представлением, заполненным рядом кнопок. Ваша программа, скорее всего, никогда не будет взаимодействовать с клавиатурой системы iOS напрямую.

Создание приложения Control Fun

Откройте среду Xcode и создайте новый проект с именем Control Fun. Мы снова собираемся использовать шаблон Single View Application, чтобы создать проект точно так же, как в предыдущих двух главах.

Создав проект, найдите изображение, которое мы будем использовать в качестве нашего графического представления. Это изображение должно быть импортировано в среду Xcode, чтобы быть доступным для использования в программе Interface Builder, поэтому сначала импортируем его. Вы найдете три файла — apress_logo.png, apress_logo@2x.png и apress_logo@3x.png — в папке 04 – Logos, содержащей исходные файлы проекта. Эти файлы представляют собой стандартную версию и две версии Retina одного и того же изображения. Мы добавим эти файлы в каталог ресурсов и предоставим приложению самостоятельно выбирать соответствующее изображение на этапе выполнения. Если вы будете использовать изображение, выбранное самостоятельно, убедитесь, что его размеры соответствуют размеру доступного пространства, а файл имеет расширение .png. Его размер должен быть не больше 100 пикселей в высоту и 300 пикселей в ширину, чтобы можно было заполнить верхнюю часть представления на самом узком экране устройства iPhone без изменения его размеров. На более крупных экранах используются версии удвоенного и утроенного размеров.

Находясь в среде Xcode, выберите папку Assets.xcassets в окне навигатора проекта, откройте папку 04 – Logos в окне Finder и выберите все три графических изображения. Затем перетащите эти изображения в область редактирования Xcode и отпустите кнопку мыши. Среда Xcode использует имена изображений для того, чтобы определить, что вы добавляете три версии изображения с именем apress_logo, и остальную работу сделает автоматически (рис. 4.5). В левом столбце области редактирования под элементом AppIcon появится элемент apress_logo. Теперь имя apress_logo можно использовать в коде или в программе Interface Builder для ссылки на данный набор графических изображений и их загрузки во время выполнения приложения.

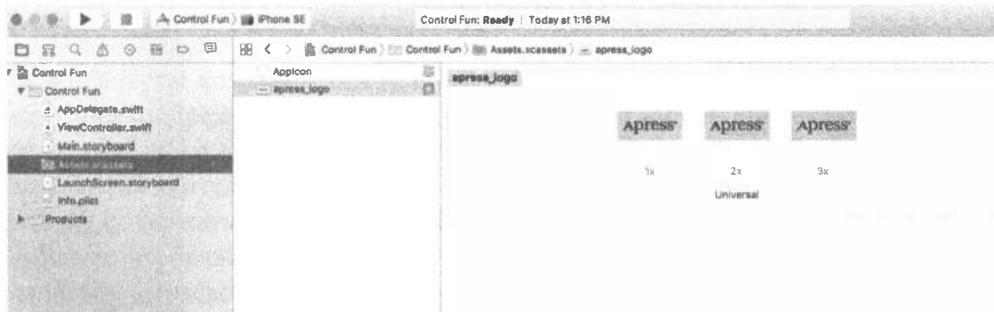


Рис. 4.5. Добавление изображений apress_logo в проект Xcode

Реализация графического представления и полей редактирования

Добавив изображение в проект, реализуем пять элементов интерфейса в верхней части экрана: графическое представление, два поля редактирования и две метки (рис. 4.6).



Рис. 4.6. Графическое представление, метки и поля редактирования реализуются в первую очередь

Добавление графического представления

Перейдите в окно навигатора проекта, щелкните на файле Main.storyboard, чтобы открыть его в среде Interface Builder. Вы увидите знакомый белый фон и единственное квадратное представление, на котором можно разместить интерфейс вашего приложения. Как и в предыдущей главе, под окном IB выберите в списке View as: пункт iPhone 6s.

ЗАМЕЧАНИЕ. Этот раздел, расположенный ниже макета, является новшеством в среде Xcode 8 и называется View Dimension. Он позволяет выбрать, как мы будем видеть сцену, работая с макетом.

Если библиотека объектов не открыта, выполните команду View⇒Utilities⇒Show Object Library. Прокрутите примерно четверть списка и найдите пункт ImageView (рис. 4.7). Запомните, что библиотека объектов открывается с помощью третьей пиктограммы в верхней части панели библиотеки.

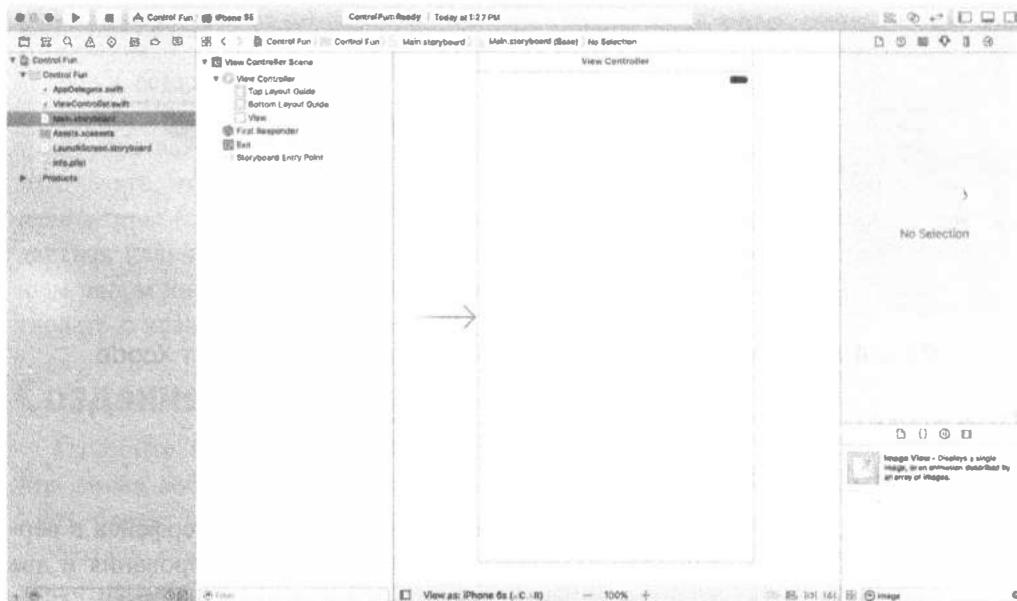


Рис. 4.7. Элемент Image View в библиотеке объектов программы Interface Builder

Перетащите графическое представление в окно редактора раскладовок и оставьте его поблизости от верхнего края, как показано на рис. 4.8. Пока не стоит беспокоиться о точном позиционировании; в следующем разделе мы еще вернемся к этому вопросу.

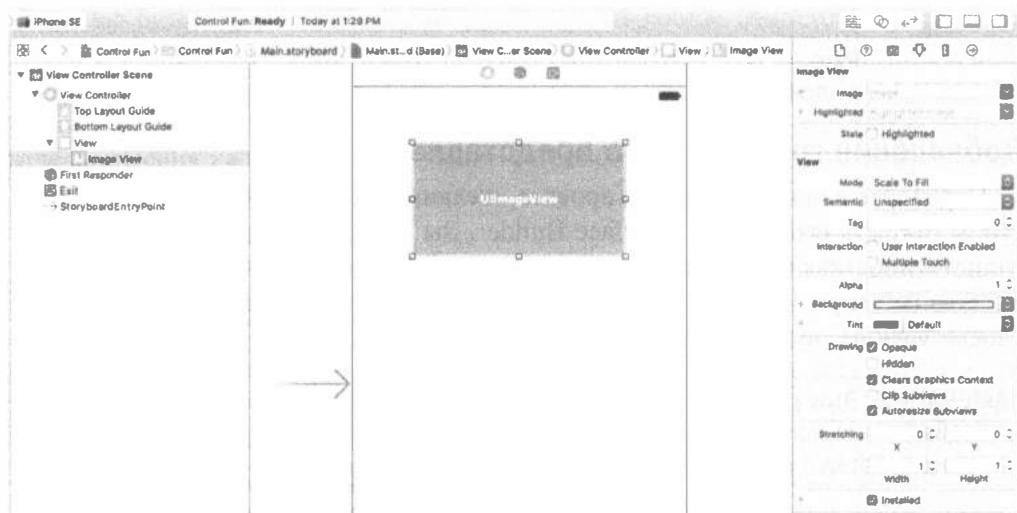


Рис. 4.8. Добавление объекта класса UIImageView в раскладовку

Выбрав графическое представление, вызовите инспектор атрибутов объекта, нажав комбинацию клавиш **<Option+⌘+4>**. В результате отобразятся редактируемые атрибуты класса UIImageView. Самым важным атрибутом нашего представления изображений является верхний элемент в окне инспектора, имеющий имя **Image**. Если щелкнуть на маленькой стрелке, находящейся справа от поля, на экране появится всплывающее меню с доступными изображениями, которые должны содержать все изображения, добавленные вами в свой проект Xcode. Выберите изображение, добавленное минуту назад. Ваше изображение должно появиться в графическом представлении, как показано на рис. 4.9.

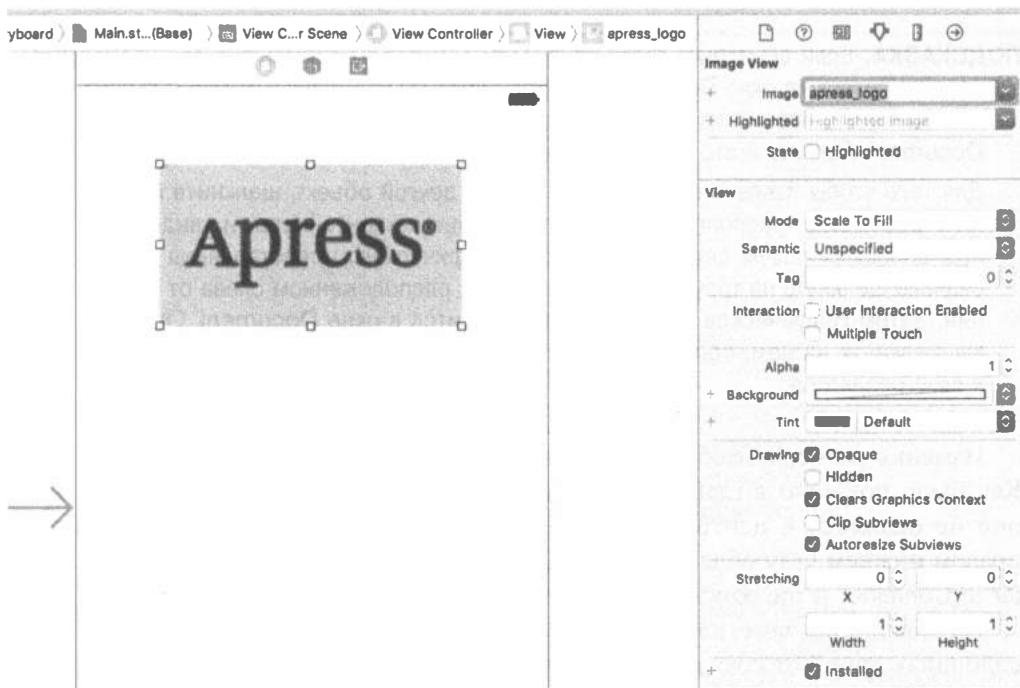


Рис. 4.9. Инспектор атрибутов графического представления. Мы выбрали наше изображение в меню **Image** в верхней части окна инспектора, а затем заполнили графическое представление нашим изображением

Изменение размеров графического представления

Оказывается, размеры используемого нами изображения не совпадают с размерами графического представления, в котором оно находится. Среда Xcode по умолчанию масштабирует изображение так, чтобы заполнить все графическое представление. Для этого достаточно в окне инспектора атрибутов установить для параметра **Mode** значение **Scale to Fill**. Масштабировать изображения лучше до запуска приложения, потому что этот процесс занимает время работы процессора. В данном случае мы вообще не собираемся масштабировать наше

графическое представление, поэтому просто изменим его размеры, чтобы они совпадали с размерами изображения. Для начала измените атрибут *Mode* на *Center*, означающий, что изображение не должно масштабироваться и должно быть расположено в центре представления. Затем подгоните размеры графического представления под размеры изображения. Для этого выберите графическое представление, чтобы на экране появились контуры и размерные линии, а затем нажмите комбинацию клавиш *<⌘+=>* или выберите команду *Editor*⇒*Size to Fit Content*. Если комбинация клавиш не дала эффекта или команда была заблокирована, снова выберите графическое представление, перетащите его немного в сторону и повторите попытку.

ПОДСКАЗКА. Если вы испытываете трудности с выбором элемента в окне редактора, переключитесь на окно *Document Outline*, щелкнув на маленьком треугольнике, расположенном в левом нижнем углу. Затем щелкните на элементе, выбранном в окне *Document Outline*, и этот элемент окажется выбранным в окне редактора.

Для того чтобы извлечь объект, вложенный в другой объект, щелкните на треугольнике раскрытия, расположенном слева от объекта-контейнера, и увидите вложенный объект. В нашем случае для выбора графического представления необходимо сначала щелкнуть на треугольнике раскрытия, расположенном слева от представления. Затем графическое представление появится в окне *Document Outline*. Когда вы щелкнете на нем, соответствующее графическое представление будет выбрано в окне редактора.

Изменив размеры изображения, переместите его в окончательное положение. Как было показано в главе 3, перетаскивайте графическое представление, пока оно не окажется в центре, щелкните на пиктограмме *Align*, расположенной в правом нижнем углу области редактирования, установите флагок *Horizontal Center in Container* и щелкните на кнопке *Add 1 Constraint*.

Возможно, вы заметили, что программа Interface Builder проводит несколько сплошных линий от края представления к краю его родительского представления (не путайте их с пунктирными голубыми линиями, которые появляются при перетаскивании элементов по макету) или от одного родительского представления к другому. Эти сплошные линии представляют добавленные вами ограничения. Если щелкнуть на добавленном ограничении, то на экране появится сплошная оранжевая линия, вертикально проходящая через все главное представление, как показано на рис. 4.10.

Эти сплошные линии представляют ограничения. Оранжевый цвет означает, что позиция графического представления и/или его размеры заданы не полностью и необходимо задать дополнительные ограничения. Выяснить причину этой проблемы можно, щелкнув на оранжевом треугольнике в окне *Activity View*. В данном случае среда Xcode сообщает нам, что необходимо установить вертикальное ограничение для графического представления. Это можно сделать прямо сейчас, использовав приемы, описанные в главе 3, или подождав, пока мы не исправим все ограничения в нашем макете позднее.

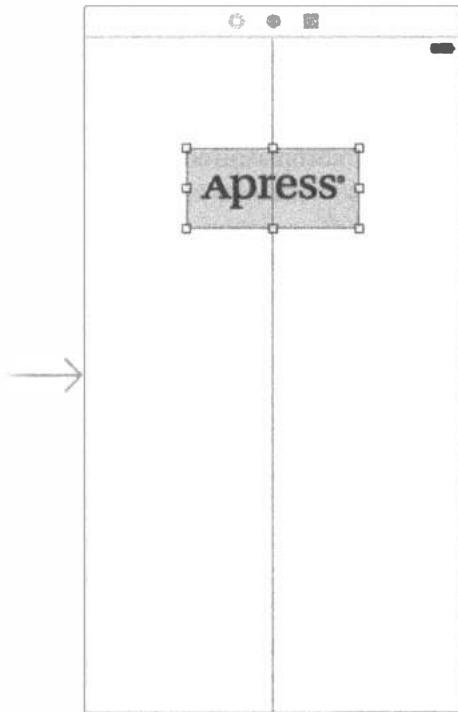


Рис. 4.10. Изменив размеры графического представления так, чтобы они соответствовали размерам изображения, мы перетащили его в требуемое место с помощью голубых линий разметки и создали ограничение, чтобы закрепить его в центре

ПОДСКАЗКА. Перетаскивание и масштабирование представлений в программе Interface Builder может оказаться сложной задачей. Не забывайте об окне Document Outline, которое открывается после щелчка на маленьком треугольнике, расположенном в нижней части окна редактора. Во время масштабирования удерживайте нажатой клавишу <Option>. Программа Interface Builder нарисует на экране вспомогательные красные линии, которые позволяют намного быстрее понять, где должно находиться изображение. Этот трюк не срабатывает при перетаскивании, потому что клавиша <Option> предполагает программе Interface Builder создать копию перетаскиваемого объекта. Однако, если выбрать команду Editor⇒Canvas⇒Show Bounds Rectangles, вокруг элементов вашего интерфейса будут нарисованы линии, чтобы их легче было увидеть. Эти линии можно отключить, снова выбрав команду Show Bounds Rectangles.

Настройка атрибутов представления

Выберите свое графическое представление, а затем переключите внимание на инспектор атрибутов. Под разделом *Image View* находится раздел *View*. Легко

догадаться, что в этом разделе сначала выводятся атрибуты выбранного объекта, за которыми следуют более общие атрибуты, применимые к родительскому классу выбранного объекта. В данном случае родительским классом класса `UIImageView` является класс `UIView`, поэтому следующий раздел называется `View`. Он содержит атрибуты, которыми обладает любое представление.

Атрибут *Mode*

Первым элементом в окне инспектора представлений является всплывающее меню `Mode`. В этом меню можно задать способ, которым представление будет отображать свое содержимое на экране. Оно определяет режим выравнивания изображения в представлении и то, надо ли его масштабировать. Можно выбрать любой вариант демонстрации изображения `apress_logo`, чтобы посмотреть, как оно будет выглядеть, но не забудьте в конце оставить значение `Center`.

Как указывалось ранее, выбирая любой вариант, предусматривающий масштабирование изображения, вы увеличиваете вычислительные затраты, поэтому лучше избегать таких ситуаций и задавать правильные размеры изображений еще до их импортирования. Если же вы хотите вывести на экран одно и то же изображение, но с разными размерами, то лучше сделать несколько разных копий этого изображения, чем вынуждать устройство с системой iOS масштабировать его во время работы.

Атрибут *Semantic*

Непосредственно под меню `Mode` расположен атрибут `Semantic`. Этот атрибут появился в версии iOS 9 и позволяет указывать, как должно прорисовываться представление в локализации, предусматривающей чтение справа налево, например на иврите или арабском языке. По умолчанию представление не имеет этого атрибута, но его можно установить, выбрав соответствующее значение. Более подробную информацию можно найти в документации по Xcode, в которой описывается свойство `semanticContentAttribute` в классе `UIView`.

Атрибут *Tag*

Следующий по порядку элемент — `Tag` — заслуживает внимания, хотя мы не будем использовать его в этой главе. Все подклассы класса `UIView`, включая все представления и элементы управления, имеют свойство `tag`, которое является обычным числовым значением (дескриптором), отображаемым рядом с представлением изображения. Этот атрибут предназначен только для пользователя; система никогда не устанавливает и не изменяет его значение. Если вы установили значение атрибута `Tag` для элемента управления или представления, то можете быть уверены, что он всегда будет иметь заданное вами значение, пока вы сами его не измените.

Дескрипторы обеспечивают простой и независимый от языка способ идентификации объектов интерфейса. Допустим, в вашем интерфейсе есть пять разных кнопок, каждая из которых имеет собственную метку, и вы хотите использовать один метод, выполняющий действие для всех пяти кнопок. В этом случае вам, вероятно, необходимо как-то различать кнопки при вызове этого метода. В отличие от меток, дескрипторы никогда не изменяются, поэтому, если вы зададите значение дескриптора в программе Interface Builder, то сможете использовать его в качестве быстрого и надежного инструмента для проверки элемента управления, передаваемого в качестве аргумента `sender` методу, выполняющему действие.

Флажки *Interaction*

Эти два флажка относятся к взаимодействию с пользователем. Первый флажок — `User Interaction Enabled` — определяет, может ли пользователь вообще что-либо делать с объектом. Для большинства элементов управления этот флажок установлен, потому что иначе элемент управления никогда не сможет инициировать методы, выполняющие действия. Однако для графических представлений этот флажок по умолчанию сброшен, потому что очень часто эти элементы управления используются для отображения статической информации. Поскольку мы планируем лишь вывести рисунок на экране, нет необходимости устанавливать этот флажок.

Последний флажок — `Multiple Touch` — определяет, способен ли элемент управления получать события многократного прикосновения. Такие события допускают использование сложных жестов, таких как щипок, который применяется для увеличения масштаба во многих приложениях для системы iOS. О жестах и событиях многократного прикосновения речь пойдет в главе 18. Так как графическое представление не допускает никакого взаимодействия с пользователем, нет причин для включения обработки событий многократного прикосновения, поэтому этот флажок остается сброшенным.

Ползунок *Alpha*

Следующий элемент инспектора называется `Alpha`, и с ним следует быть осторожным. Элемент `Alpha` определяет степень прозрачности вашего изображения, т.е. как отображать на экране элементы, находящиеся под этим изображением. Степень прозрачности определяется числом с плавающей точкой от 0.0 до 1.0. Число 0.0 означает полную прозрачность, а 1.0 — полную непрозрачность. Если ввести число, меньшее единицы, то ваше устройство iOS нарисует это представление полупрозрачным, так что вы сможете увидеть элементы, находящиеся под ним. Если степень прозрачности меньше единицы, то, даже если под изображением ничего нет, вы заставите процессор выполнить дополнительные вычисления, связанные с определением прозрачности. Поэтому не задавайте этот показатель меньшим единицы, если у вас нет веских причин поступить иначе.

Атрибут *Background*

Следующий по порядку элемент под названием *Background* определяет цвет фона для представления. Для графических представлений это имеет значение только тогда, когда изображение не заполняет представление полностью или какая-то его часть является прозрачной. Поскольку наше изображение полностью заполняет представление, изменение этого атрибута не будет иметь никаких визуальных последствий, так что его можно не трогать.

Атрибут *Tint*

Следующий элемент управления задает оттенок цвета переднего плана выбранного представления. Этот же цвет некоторые представления используют для своей прорисовки. Сегментированный элемент управления, который мы будем использовать далее, использует цвет переднего плана, а объект класса `UIImageView` — нет.

Флажки *Drawing*

Под атрибутом *Tint* расположен ряд флажков *Drawing*. Первый из них имеет метку *Opaque*. Убедитесь, что он установлен по умолчанию. Если нет, установите его. Он сообщает системе iOS, чтобы она ничего не рисовала под вашим представлением, и позволяет ей оптимизировать методы рисования для ускорения процесса прорисовки.

Вы можете поинтересоваться, почему необходимо установить флажок *Opaque*, если мы уже задали атрибут *Alpha* равным 1.0, чтобы сделать представление непрозрачным. Значение атрибута *Alpha* относится только к тем частям изображения, которые должны быть нарисованы, а если изображение не полностью занимает представление или в нем есть дыры, образованные *альфа-каналом* (*alpha channel*), то объекты, расположенные под ним, будут показаны независимо от значения атрибута *Alpha*. Установив флажок *Opaque*, мы сообщаем системе iOS, что под данным представлением ничего рисовать не надо, поэтому нет необходимости тратить время процессора на элементы, расположенные под объектом. Мы можем безопасно устанавливать флажок *Opaque*, поскольку ранее выбрали команду *Size to Fit*, которая вынуждает представление изображений изменить размеры так, чтобы они совпали с размерами изображения, которое оно содержит.

Флажок *Hidden* означает именно то, что вы подумали. Если он установлен, то пользователь не может видеть данный элемент управления. Скрыть элемент управления иногда бывает полезно. Позднее мы покажем, когда следует скрывать переключатели и кнопки, но в подавляющем большинстве случаев этот флажок не устанавливается. Мы можем оставить его в состоянии, предусмотренном по умолчанию.

Следующий флажок — *Clears Graphics Context* — устанавливается редко. Если он установлен, то система iOS, прежде чем нарисовать сам элемент управления,

нарисует всю область, покрыгую элементом управления, как прозрачную и черную. И снова заметим, что он отключен, отчасти потому, чтобы не снижать производительность приложения, а отчасти потому, что он нужен довольно редко. Оставим этот флагок сброшенным (скорее всего, по умолчанию он окажется установленным).

Флагок **Clip Subviews** довольно интересен. Если ваше представление имеет дочерние представления и они не полностью заполняют родительское представление, то данный флагок определяет, как будут прорисовываться эти представления. Если флагок **Clip Subviews** установлен, то нарисована будет только та часть дочерних представлений, которая лежит в пределах родительского представления. Если флагок **Clip Subviews** сброшен, то дочерние представления будут прорисованы полностью, даже если они выходят за пределы родительского представления.

По умолчанию флагок **Clip Subviews** должен быть сброшен. Может показаться, что на самом деле все должно быть наоборот и дочернее представление не должно прорисовываться где угодно. Однако вычисление отрезаемой области и отображение только части дочерних представлений — довольно затратная операция с математической точки зрения, а в большинстве случаев дочерние представления не выходят за пределы родительских. Если по каким-то причинам это вам действительно необходимо, можете установить флагок **Clip Subviews**, но по умолчанию он сброшен, чтобы не тормозить работу приложения.

Последний флагок в этом разделе — **Autoresizing Subviews**. Он сообщает системе iOS, чтобы она масштабировала все дочерние представления при масштабировании родительского представления. Оставим его установленным. Поскольку мы не собираемся масштабировать представление, этот флагок не играет никакой роли.

Раздел Stretching

Следующий раздел называется **Stretching**. Он регламентирует перерисовку прямоугольных представлений при изменении их размеров на экране. Идея заключается в том, чтобы не растягивать все содержание представления равномерно, а фиксировать внешние края представления, например скошенные грани кнопок, чтобы они выглядели так же, как в центральной части области растягивания.

Четыре числа с плавающей точкой определяют, какая часть прямоугольника может растягиваться. Эта величина определяется координатами левой верхней точки представления и размером растягиваемой области. Все эти числа лежат в диапазоне от 0.0 до 1.0, выражая долю размера всего представления. Например, если вы хотите оставить 10% каждого края нерастяжимыми, задайте значения X и Y равными 0.1, а Width и Height — 0.8. В данном случае оставим значения X и Y равными 0.0, а Width и Height — 1.0. Скорее всего, вам редко придется изменять эти параметры.

Добавление полей редактирования

Завершив подготовку графического представления, добавим в него поля редактирования. Захватите поле редактирования из библиотеки, перетащите его в раскладовку. Используя голубые линии разметки для выравнивания по правому краю, поместите поле немного ниже графического представления (рис. 4.11).

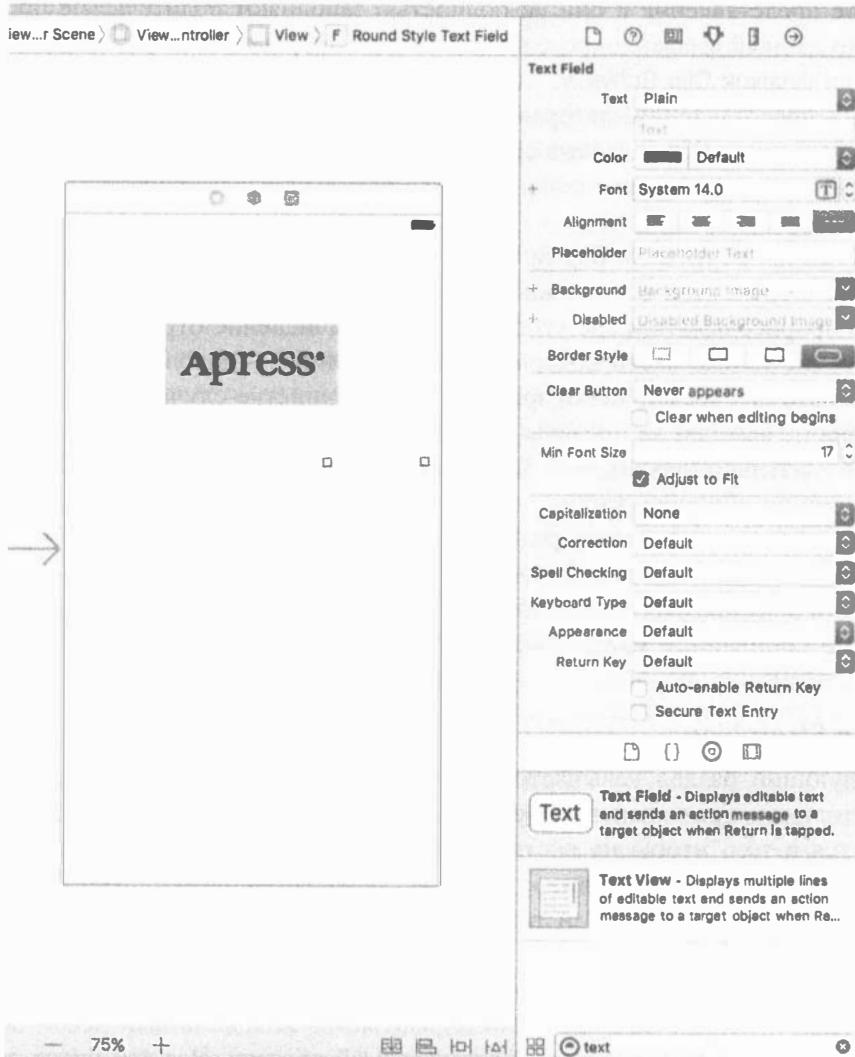


Рис. 4.11. Мы перетащили поле редактирования из библиотеки и разместили его на представлении непосредственно под изображением, используя голубые линии разметки

Затем захватите метку из библиотеки и перетащите ее так, чтобы выровнять по левой границе изображения и по верхнему краю поля редактирования.

Обратите внимание на несколько голубых линий разметки, которые появляются вокруг метки при ее перемещении. Они облегчают выравнивание метки по отношению к полю редактирования с помощью верхней или нижней границы, а также середины метки. Мы собираемся выровнять метку и поле редактирования по тексту, ориентируясь на базовую линию, проходящую по нижнему краю текста через все поле редактирования (рис. 4.12).

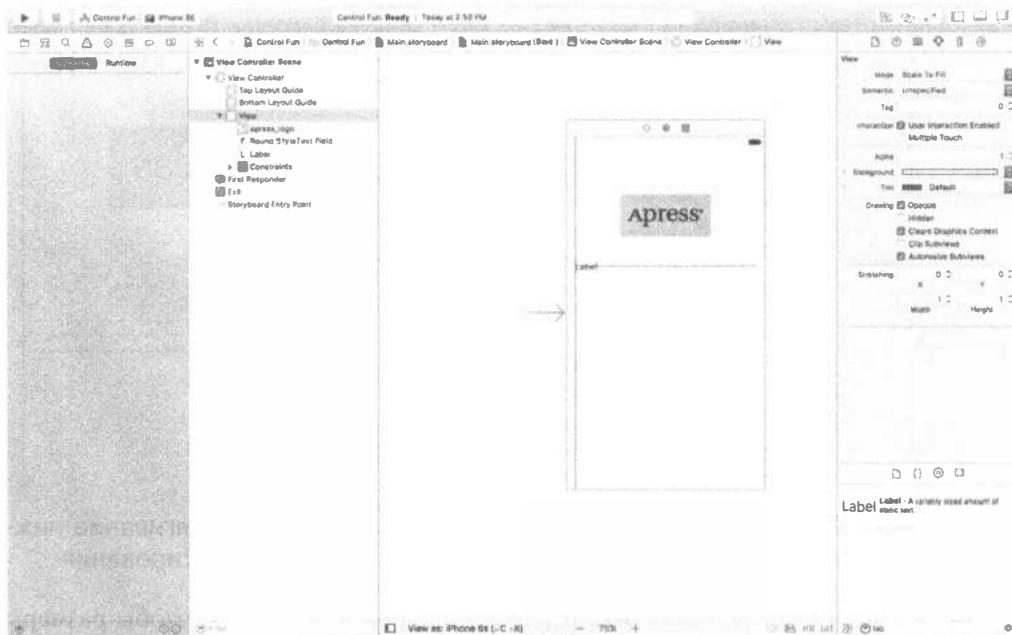


Рис. 4.12. Выравнивание метки и поля редактирования по базовой линии

Дважды щелкнув на метке, измените ее имя **Name**: (не забудьте поставить двоеточие) и нажмите клавишу <Return>, чтобы зафиксировать внесенные изменения.

Затем перетащите другое поле редактирования из библиотеки на представление и, используя голубые линии разметки, поместите его под первым полем редактирования (рис. 4.13).

Разместив второе поле редактирования, захватите в библиотеке другую метку и поместите ее в левой части ниже существующей метки. Для выравнивания метки по второму полю редактирования снова используйте голубую линию разметки. Дважды щелкните на новой метке и измените ее имя на **Number**: (не забудьте поставить двоеточие).

Растяните нижнее поле редактирования влево, чтобы оно полностью доходило до правого края метки. Почему мы начинаем с нижнего поля редактирования, а не с верхнего? Потому что хотим, чтобы оба текстовых поля имели одинаковые размеры, а нижняя метка длиннее, чем верхняя.

Один раз щелкните на нижнем поле редактирования и перетаскивайте левую точку изменения размера влево, пока не появится голубая линия разметки, сообщающая о том, что приблизиться к метке еще больше невозможно (рис. 4.14). Эта линия разметки почти незаметна — ее длина равна высоте самого поля редактирования, так что будьте внимательны.

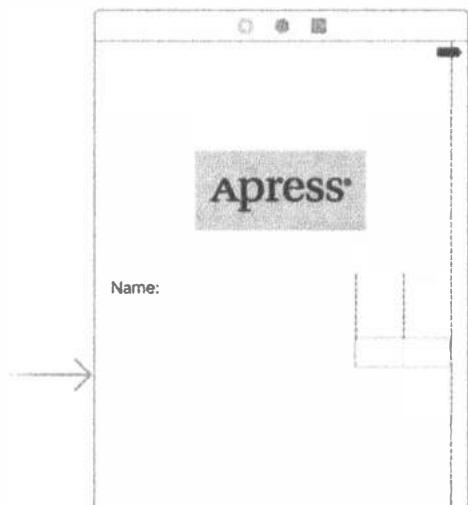


Рис. 4.13. Добавление второго поля редактирования

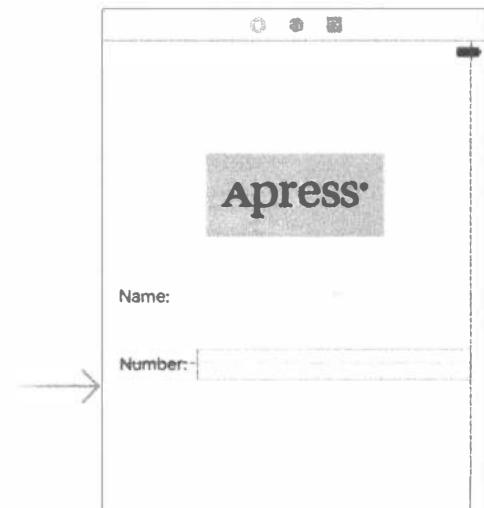


Рис. 4.14. Растворение нижнего поля редактирования

Теперь точно так же растяните верхнее поле редактирования, чтобы размеры обоих полей совпадали. Эту задачу также облегчают голубые линии разметки.

Мы сделали с полями редактирования практически все, что требовалось, за исключением одной детали. Вернитесь к рис. 4.1. Разве метки `Name:` и `Number:` выровнены по правому краю? Нет, в данный момент они выровнены по левому краю. Для того чтобы обе эти метки выровнять по правому краю, щелкните на метке `Name:` и, удерживая нажатой клавишу `<Shift>`, щелкните на метке `Number:`, чтобы обе метки оказались выбранными. Нажмите комбинацию клавиш `<Option+⌘+4>`, чтобы открыть окно инспектора атрибутов, и убедитесь, что раздел `Label` открыт. Если он не открыт, то щелкните на кнопке `Show`, расположенной справа от заголовка. Теперь воспользуйтесь элементом управления `Alignment` для выравнивания метки, а затем задайте ограничение, регламентирующее, что оба поля всегда должны иметь одинаковый размер, щелкнув на кнопке `Add 1 Constraint`. В окне `Activity View` должен появиться оранжевый индикатор предупреждения в виде треугольника, а в окне навигатора проблем должны появиться предупреждения о проблемах с макетированием. Пока их можно игнорировать — мы исправим их позднее.

После этого интерфейс должен очень напоминать интерфейс, показанный на рис. 4.1. Единственное отличие заключается в том, что поля редактирования на рис. 4.1 содержат светло-серый текст, а поля на нашем интерфейсе — нет. Этот недостаток мы сейчас исправим. Щелкните на верхнем поле и нажмите комбинацию клавиш **<Option+⌘+4>**, чтобы открыть окно инспектора атрибутов (рис. 4.15). Поле редактирования — один из наиболее сложных и наиболее часто используемых элементов управления в системе iOS. Рассмотрим его настройки, начиная с самой верхней строки окна инспектора. Убедитесь, что вы выделили именно поле редактирования, а не метку или другой элемент.



Рис. 4.15. Инспектор поля редактирования со значениями по умолчанию

Установки инспектора полей редактирования

В первом разделе метка `Text` связана с двумя элементами управления, позволяющими задать вид текста в поле редактирования. Первый элемент — это раскрывающийся список, позволяющий сделать выбор между обычным текстом и текстом с атрибутами (разными шрифтами и т.п.). В главе 3 мы использовали эти атрибуты, чтобы текст отображался полужирным шрифтом. В данном случае пока оставим атрибут равным `Plain`. Непосредственно под ним можно ввести значение, заданное для поля редактирования по умолчанию. Все, что вы наберете в этом поле, будет отображаться в поле редактирования при запуске вашего приложения.

Сразу за первым разделом следует ряд элементов управления, позволяющих задавать шрифт и его цвет. Оставим атрибут `Color` по умолчанию черным. Меню `Color` разделено на две части: в правой можно выбрать заранее заданные цвета, а в левой — собственный цвет.

Раздел `Font` разделен на три части. Правая часть — это элемент управления, позволяющий увеличивать или уменьшать размер текста на один пункт. Левая часть позволяет вручную редактировать название и размер шрифта. И наконец пиктограмма с буквой Т в квадрате открывает контекстное окно, в котором можно задать различные атрибуты шрифта. Оставим для атрибута `Font` значение `System 14.0`, заданное по умолчанию. В зависимости от системы это значение может меняться.

Под этими разделами расположены три кнопки для управления выравниванием текста в поле редактирования. Оставим текст выровненным по левому краю (по умолчанию). Для этого надо нажать левую кнопку.

Раздел `Placeholder` позволяет задать текст (заполнитель), который будет отображаться в поле редактирования серым цветом только тогда, когда в поле редактирования не введено никаких значений. Если на экране мало места или вы хотите подсказать пользователю, что именно следует вводить в поле редактирования, то заполнитель можно использовать в качестве метки. Введите в качестве заполнителя поля редактирования текст `Type in a name`.

Следующие два поля, `Background` и `Disabled`, используются, только если вам необходимо настроить внешний вид поля редактирования, хотя это совершенно необязательно и часто просто приводит к потере времени. Пользователи привыкли к полям редактирования определенного вида. По этой причине мы оставим эти поля заполненными их значениями по умолчанию.

Далее следуют четыре кнопки с меткой `Border Style`, позволяющие изменять стиль прорисовки границ полей редактирования. Вы можете выбрать любой из четырех стилей, но по умолчанию считается выбранной правая кнопка. Она задает стиль поля редактирования, который наиболее популярен среди пользователей приложений для системы iOS, поэтому, когда наигралась со стилями, выберите крайнюю справа кнопку.

Параметр `Clear Button` позволяет указать, должна ли появляться на экране кнопка очистки содержимого. Она представляет собой букву X, расположенную

в конце поля редактирования. Кнопка очистки содержимого обычно используется в полях поиска и других полях, содержимое которых часто изменяется. Как правило, его не выводят в полях, содержащих постоянные данные, поэтому это значение параметра Clear Button следует оставить равным Never Appears.

Флажок Clear When Editing Begins определяет, что произойдет, когда пользователь прикоснется к указанному полю. Если флажок установлен, то значение, ранее введенное в поле редактирования, будет удалено, и пользователь начнет работу с пустым полем редактирования. Если же флажок сброшен, то предыдущее значение останется в поле, и пользователь сможет его отредактировать. Убедитесь, что этот флажок сброшен.

Следующий раздел начинается с элемента управления, позволяющего задать минимальный размер шрифта. Оставим значение этого поля заданным по умолчанию. Флажок Adjust to Fit определяет, следует ли уменьшать размер текста при уменьшении размеров поля редактирования. Если флажок установлен, то весь текст будет виден в поле редактирования, даже если его слишком много и он выходит за выделенное пространство. Справа от этого флажка расположено поле редактирования, позволяющее задать минимальный размер шрифта. Независимо от размера поля, шрифт не может быть меньше заданного минимального размера. Задавая минимальный размер шрифта, вы гарантируете, что текст всегда можно будет прочитать.

Следующий раздел определяет внешний вид клавиатуры и ее поведение при использовании поля редактирования. Поскольку в поле будет вводиться имя, установим для параметра Capitalize значение Words. В результате каждое слово будет автоматически начинаться с прописной буквы, что обычно и требуется при вводе имен.

Четыре следующих раскрывающихся списка — Correction, Spell Checking, Keyboard Type и Appearance — можно оставить без изменений. Подумайте о них пару минут, чтобы понять смысл их содержания.

Следующее меню, Return Key, относится к клавише, расположенной в правом нижнем углу клавиатуры. Ее метка изменяется в зависимости от того, что вы делаете. Если вводите текст в поле запросов поисковой машины Safari, например, то она называется Google. В приложении, подобном этому, где поле редактирования делит экран с другими элементами управления, правильным выбором будет значение Done.

Если установлен флажок Auto-enable Return Key, то клавиша <Return> будет отключена, пока в поле редактирования не будет введен хотя бы один символ. Оставим этот флажок сброшенным, потому что мы разрешаем, чтобы поле редактирования оставалось пустым, если пользователь так хочет.

Флажок Secure определяет, должен ли введенный символ отображаться в поле редактирования. Этот флажок следует устанавливать, если поле редактирования используется для ввода пароля. Оставим его сброшенным.

Следующий раздел (для того чтобы его увидеть, возможно, необходимо выполнить вертикальную прокрутку экрана) позволяет установить общие атрибуты элемента управления, унаследованные от класса `UIControl`, но они обычно не относятся к полям редактирования и, за исключением флагка `Enabled`, не влияют на внешний вид поля. Мы хотим, чтобы поля редактирования были доступными и пользователь мог взаимодействовать с ними, поэтому оставим все как есть.

Последний раздел в окне инспектора должен быть вам хорошо знаком. Он идентичен разделу с таким же названием в инспекторе представления изображений, который мы описывали выше. Существуют атрибуты, унаследованные от класса `UIView`, а поскольку все элементы управления являются подклассами класса `UIView`, все они совместно используют этот раздел атрибутов. Установите флагок `Opaque` и сбросьте флагки `Clears Graphics Context` и `Clip Subviews` (причины мы объясним позже).

Установка атрибутов для второго поля редактирования

Щелкните на нижнем поле редактирования (под меткой `Number:`) в раскладке и вернитесь в окно инспектора атрибутов. Введите в поле `Placeholder` строку `Type in a number`. Флагок `Clear When Editing Begins` должен быть сброшен. Немного ниже откройте всплывающее меню `Keyboard`. Поскольку мы хотим, чтобы пользователь вводил только цифры, а не буквы, выполните команду `Number Pad`. В результате пользователь увидит на экране iPhone клавиатуру, содержащую только цифры. Это означает, что он не сможет вводить буквы, символы или что-либо другое, кроме цифр. Мы не обязаны устанавливать значение атрибута `Return Key` для цифровой клавиатуры, потому что на ней нет такой клавиши. Следовательно, все остальные параметры в инспекторе следует оставить заданными по умолчанию. Как и раньше, установите флагок `Opaque` и сбросьте флагки `Clears Graphics Context` и `Clip Subviews`. На устройстве iPad команда `Number Pad` включает виртуальную цифровую клавиатуру, когда пользователь активизирует поле редактирования, при этом он может переключить клавиатуру обратно в алфавитный режим. Это означает, что в реальном приложении вы должны проверить, что пользователь ввел правильное число в поле `Number`.

ПОДСКАЗКА. Если хотите предотвратить ввод в поле редактирования любых символов, кроме чисел, то создайте класс, реализующий метод `textView(_ textView: shouldChangeTextInRange: replacementText: text)` протокола `UITextViewDelegate`, и сделайте его делегатом текстового представления. Это несложно, но описание этого процесса выходит за рамки нашей книги.

Добавление ограничений

Прежде чем продолжать работу, мы должны настроить несколько ограничений для нашего макета. Когда вы перетаскиваете одно представление в другое, находясь в окне `Interface Builder` (как мы только что сделали), программа `Xcode` автоматически не создает для него никаких ограничений. Однако система

макетирования требует полного набора ограничений, поэтому во время компиляции программа Xcode создает набор стандартных ограничений, описывающих макет. Эти ограничения зависят от позиции всех объектов в родительском представлении. В зависимости от того, где они расположены — ближе к левому или правому краю представления, — они прикрепляются к левому или правому краю. Аналогично в зависимости от того, к какому краю они ближе расположены — верхнему или нижнему, — они прикрепляются к верхнему или нижнему краю. Если объект расположен точно в центре, то обычно он фиксируется в центре.

Для того чтобы еще больше запутать ситуацию, программа Xcode может применить автоматические ограничения и прикрепить все новые объекты к одному или нескольким “соседям” в рамках одного и того же родительского представления. Это может соответствовать вашим желаниям, а может и нет, поэтому лучше самостоятельно создать полный набор ограничений в программе Interface Builder до компиляции вашего приложения, и в последних двух главах мы показали, как это сделать.

Итак, посмотрим на то, что мы имеем. Для того чтобы увидеть все ограничения, относящиеся к конкретному представлению, выберите его и откройте инспектор размеров. Выбрав любую метку, поле редактирования или ползунок, вы увидите, что инспектор размеров выводит сообщение о том, что для выбранного представления не установлено ни одного ограничения. Графический пользовательский интерфейс, который мы создаем, пока имеет только ограничения, связывающие центры по горизонтали графического и контейнерного представлений. Щелкните на контейнерном или графическом представлении, чтобы увидеть эти ограничения в окне инспектора.

Нам необходим полный набор ограничений, точно описывающий все представления и элементы управления на макете на этапе компиляции. К счастью, создать его довольно просто. Выберите все представления и элементы управления, нажав клавишу `<Control>` и обведя указателем контур вокруг них, начиная с верхнего левого угла контейнерного представления против часовой стрелки в направлении правого верхнего угла. Если вы увидите, что при этом представление начало перемещаться, отпустите кнопку мыши, переместите указатель глубже внутрь представления и попробуйте снова. Выбрав все элементы, выполните команду `Editor⇒Resolve Auto Layout Issues⇒Add Missing Constraints`. После этого вы увидите, что все представления и элементы управления соединены между собой и с контейнерным представлением тонкими голубыми отрезками. Каждый из этих отрезков представляет ограничение. Это намного лучше, чем позволить программе Xcode создать автоматические ограничения, потому что теперь у вас есть возможность модифицировать каждое ограничение при необходимости. Примеры такой модификации мы приведем по мере изложения.

ПОДСКАЗКА. Для того чтобы применить ограничения ко всем представлениям, принадлежащим контроллеру, можно выбрать контроллер представлений в окне Document Outline и выполнить команду `Editor⇒Resolve Auto Layout Issues⇒Add Missing Constraints`.

Теперь, когда все необходимые ограничения заданы, мы можем удалить предупреждения, появившиеся в окне навигатора проблем. Для этого выберите контроллер представления в окне Document Outline и выполните команду Editor⇒Resolve Auto Layout Issues⇒Update Frames в меню Xcode. В результате предупреждения о проблемах с макетированием должны исчезнуть.

Создание и присоединение выходов

Наше приложение почти готово к испытанию. Первая часть интерфейса у нас есть. Осталось создать и присоединить выходы. Графическому представлению и меткам интерфейса выходы не нужны, потому что мы не будем изменять их во время выполнения программы. Однако два поля редактирования являются массивными элементами управления, хранящими данные, которые будут использоваться в нашем коде, поэтому для них выходы необходимы.

Как вы, вероятно, помните, программа Xcode позволяет создавать и присоединять выходы одновременно с помощью помощника редактора, окно которого должно быть открыто.

Выберите свой файл раскладовки в навигаторе проекта. Если места на экране недостаточно, выполните команду View⇒Utilities⇒Hide Utilities, чтобы скрыть вспомогательную панель. На панели быстрых переходов в окне помощника редактора выберите файл ViewController.swift (рис. 4.16).

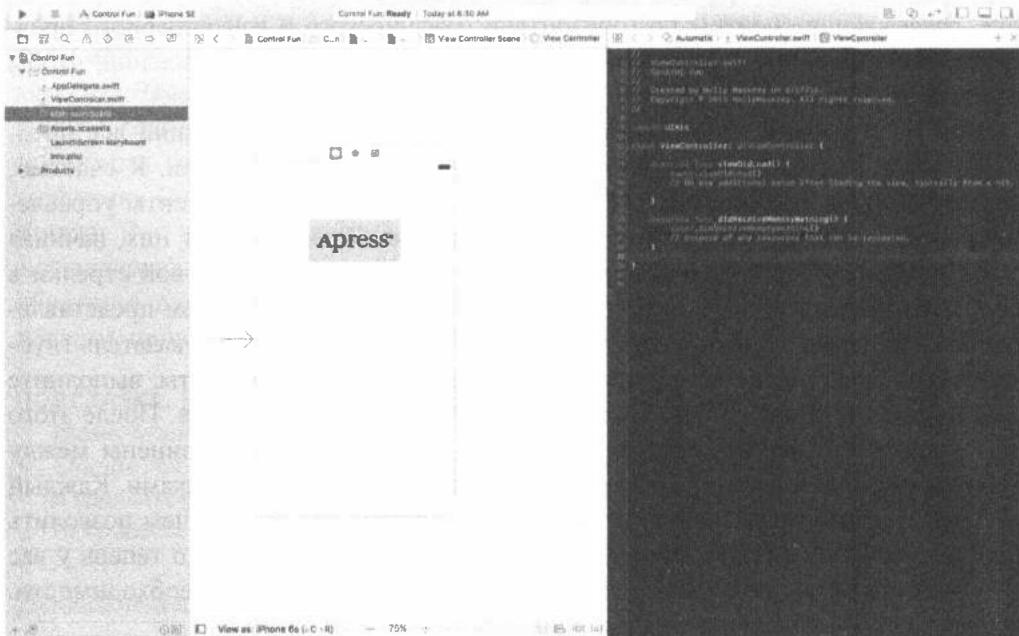


Рис. 4.16. Область редактирования раскладовки при включенном помощнике. Справа показана область помощника, в которой находится файл ViewController.swift

Теперь соединим объекты. Удерживая клавишу <Control>, перетащите указатель от представления, расположенного прямо над файлом ViewController.swift, в точку, расположенную под строкой ViewController. Появится серое всплывающее окно, содержащее строку **Inserts Outlet, Action, or Outlet Collection** (рис. 4.17). Отпустите кнопку мыши, и откроется меню, которое вы уже видели в предыдущей главе. Мы хотим создать выход с именем nameField, поэтому введите строку nameField в поле **Name**, а затем нажмите клавишу <Return> или щелкните на кнопке **Connect**.

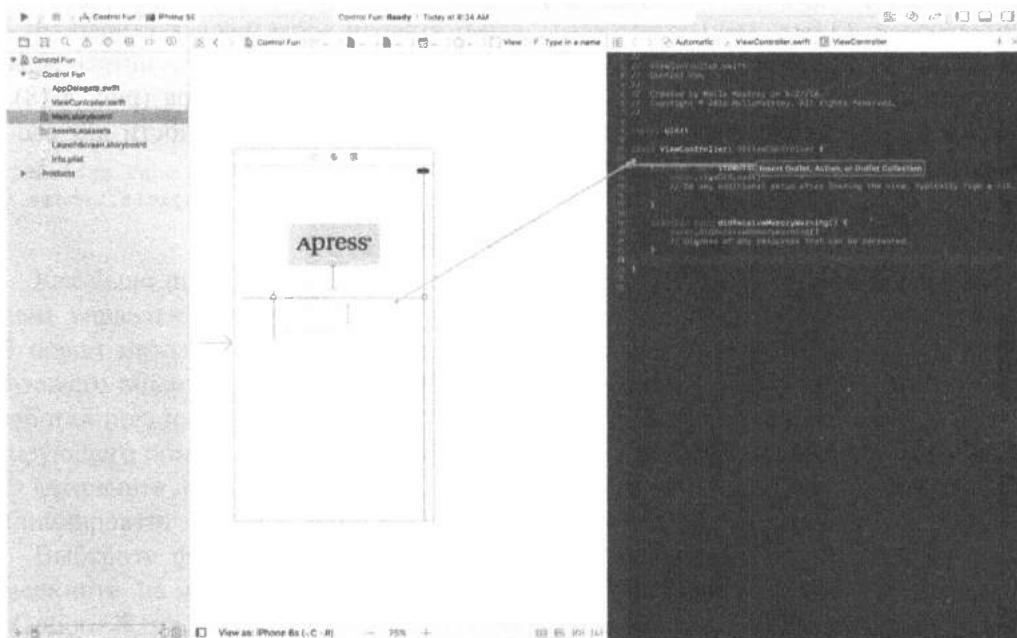


Рис. 4.17. Для того чтобы создать и присоединить выход nameField, мы включили помощник редактора и перетащили указатель из исходного кода в соответствующее поле редактирования, удерживая клавишу <Control>

Теперь у нас есть свойство nameField в классе ViewController, которое связано с верхним полем редактирования. Сделайте то же самое со вторым полем редактирования, создав для него выход и связав его со свойством numberField. После этого код будет выглядеть так, как показано в листинге 4.1.

Листинг 4.1. Связанные поля редактирования

```
class ViewController: UIViewController {
    @IBOutlet weak var nameField: UITextField!
    @IBOutlet weak var numberField: UITextField!
```

Закрытие клавиатуры

Посмотрим, как работает наше приложение. Выполните команду Product⇒Run. Приложение должно попасть в симулятор iOS. Щелкните на поле редактирования Name, чтобы на экране появилась обычная клавиатура.

ПОДСКАЗКА. Если клавиатура в симуляторе не появляется, значит, симулятор настроен на работу с клавиатурой подключенного устройства. Для того чтобы исправить ситуацию, сбросьте флагок у команды Hardware⇒Keyboard⇒Connect Hardware Keyboard в меню симулятора iOS и повторите запуск приложения.

Когда на экране появится цифровая клавиатура, введите имя и коснитесь поля редактирования Number. Должна появиться цифровая клавиатура (рис. 4.18). Каркас Cocoa Touch предоставляет все функциональные возможности для свободного добавления полей редактирования в свой интерфейс.

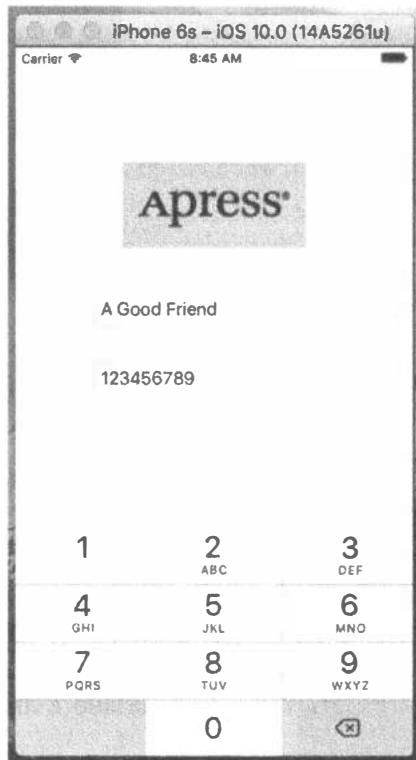


Рис. 4.18. Клавиатура появляется автоматически, когда вы касаетесь либо поля редактирования, либо поля цифры

Но осталась одна маленькая проблема. Как убрать клавиатуру с экрана? Попытайтесь. Как видите, ничего не происходит.

Закрытие клавиатуры при нажатии кнопки Done

Поскольку клавиатура является виртуальным, а не физическим устройством, нам следует обдумать операции, которые необходимы для того, чтобы она исчезла с экрана, когда пользователь заканчивает работать с ней. Когда пользователь нажмет кнопку Done, расположенную на виртуальной клавиатуре, будет сгенерировано событие Did End On Exit, и в это время мы должны сообщить полю редактирования, чтобы оно передало управление и клавиатура исчезла с экрана. Для этого необходимо добавить в наш класс контроллера метод, выполняющий действие.

Выберите в навигаторе проекта файл ViewController.swift и добавьте следующие строки (листинг 4.2).

Листинг 4.2. Метод для закрытия клавиатуры

```
@IBAction func textFieldDoneEditing(sender: UITextField) {
    sender.resignFirstResponder()
}
```

Как было показано в главе 2, первым реагирующим объектом является элемент управления, с которым пользователь взаимодействует в данный момент. В новом методе мы снимаем наш элемент управления с роли первого реагирующего объекта, передавая ее предыдущему элементу управления, с которым работал пользователь. Когда поле редактирования получает статус первого реагирующего объекта, клавиатура, связанная с ним, исчезает с экрана.

Сохраните файл ViewController.swift и вернитесь к раскладовке, чтобы инициировать заданное действие из обоих полей редактирования.

Выберите файл Main.storyboard в окне навигатора проекта, один раз щелкните на поле редактирования Name и нажмите комбинацию клавиш <Option+⌘+6>, чтобы вызвать инспектор связей. На этот раз мы не хотим генерировать событие Touch Up Inside, использованное в предыдущей главе. Вместо него сгенерируем событие Did End On Exit, поскольку именно оно возникает, когда пользователь нажимает кнопку Done на виртуальной клавиатуре.

Перетащите указатель от кружочка, расположенного справа от события Did End On Exit, к желтой пиктограмме View Controller и соедините ее с действием textFieldDoneEditing. Повторите эту операцию для второго поля редактирования, сохраните файл и нажмите комбинацию клавиш <⌘+R>, чтобы снова запустить приложение.

Когда появится симулятор, щелкните на поле Name, наберите в нем что-нибудь и нажмите кнопку Done. Скорее всего, клавиатура исчезнет, как вы и хотели. Отлично! А что вы скажете о поле Number? Где кнопка Done для нее (см. рис. 4.18)?

Не все схемы клавиатуры имеют кнопку Done. Мы, конечно, можем заставить пользователя коснуться поля редактирования Name, а затем нажать кнопку Done, но это не очень удобно. А мы хотим, чтобы пользователю было удобно работать с нашим приложением. Посмотрим, как выйти из этой ситуации.

Закрытие клавиатуры прикосновением к фону

В большинстве приложений компании Apple для системы iOS, имеющих поля редактирования, нажатие представления за пределами активных элементов управления приводит к закрытию клавиатуры. Попробуем реализовать эту функциональную возможность в нашем приложении.

Ответ удивит вас своей простотой. Наше представление изображений имеет свойство `view`, унаследованное от класса `UIViewController` и соответствующее главному представлению в раскладовке. Оно указывает на экземпляр класса `UIView`, который функционирует как контейнер для всех элементов пользовательского интерфейса. Иногда его называют контейнерным представлением (*container view*), потому что основным его назначением является хранение всех других представлений и элементов управления. Со всех точек зрения это контейнерное представление является фоном для нашего пользовательского интерфейса. Нам необходимо лишь распознать, когда пользователь коснется его. В главе 18 будет описано несколько способов, позволяющих это сделать. В классе `UIResponder`, производном от класса `UIView`, есть методы, которые вызываются каждый раз, когда пользователь касается представления одним или несколькими пальцами, проводит пальцами по нему или отнимает пальцы от представления. Один из этих методов можно переопределить (в частности, метод, который вызывается, когда пользователь отнимает палец от экрана) и добавить в него код. Другой способ подразумевает добавление **механизма распознавания жестов** (*gesture recognizer*) в контейнерное представление. Этот механизм прослушивает события, которые генерируются, когда пользователь взаимодействует с представлением, и пытается понять, что делает пользователь. В главе 18 описываются несколько механизмов распознавания жестов, соответствующих разным последовательностям действий. Нам нужен механизм распознавания прикосновения, сигнализирующий о событии, когда пользователь прикасается пальцем к экрану, а затем отнимает его от экрана за разумно короткое время.

Для использования механизма распознавания жестов необходимо создать объект, настроить его, связать его с представлением, за которым мы хотим следить в ожидании событий, связанных с прикосновением, и добавить метод действия в класс контроллера представления. Этот метод будет вызываться, когда будет распознано прикосновение. Механизм распознавания жестов можно создать и настроить в коде или в программе Interface Builder. В данном случае мы будем использовать Interface Builder, потому что это проще. Вернитесь в раскладовку и откройте окно библиотеки объектов. Затем найдите в нем объект механизма распознавания прикосновения, перетащите его на раскладовку и оставьте в контейнерном представлении. Во время работы механизм распознавания жестов остается невидимым, поэтому вы не увидите его в раскладовке, но он появится в окне Document Outline, как показано на рис. 4.19.

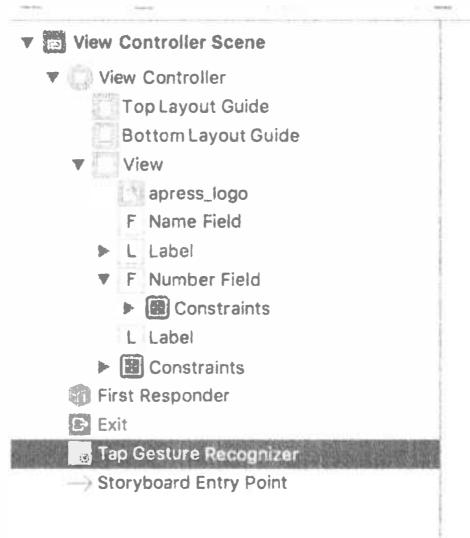


Рис. 4.19. Механизм распознавания прикосновений в окне Document Outline

Когда вы выберете механизм распознавания прикосновений, в окне инспектора атрибутов появится набор настроек, показанный на рис. 4.20.



Рис. 4.20. Атрибуты механизма распознавания прикосновений

Поле **Taps** задает количество прикосновений, которые пользователь должен выполнить, чтобы механизм их распознал, а поле **Touches** определяет, сколько пальцев должно быть задействовано в прикосновении. По умолчанию предусмотрено, что пользователь один раз прикоснется к экрану одним пальцем, поэтому остальные поля можно оставить неизменными. Остальные атрибуты нас также вполне устраивают, так что нам остается лишь связать механизм

распознавания с методом действия. Для этого откройте файл `ViewController.swift` в окне помощника редактора, нажмите клавишу `<Control>` и перетащите указатель с механизма распознавания в окне Document Outline на строку, расположенную над закрывающей скобкой в файле `ViewController.swift`. Отпустите кнопку мыши, когда увидите обычное всплывающее окно, подобное показанному на рис. 4.16. В этом окне необходимо изменить тип соединения на `Action`, а имя метода — на `onTapGestureRecognized`. В результате среда Xcode добавит метод действия и свяжет его с механизмом распознавания прикосновения. Этот метод будет вызываться каждый раз, когда пользователь прикоснется к главному представлению. На осталось лишь добавить код, чтобы закрыть клавиатуру, если она была открыта. Мы уже знаем, как это сделать, поэтому изменим код так, как показано в листинге 4.3.

Листинг 4.3. Код закрытия клавиатуры в механизме распознавания прикосновений

```
@IBAction func onTapGestureRecognized(sender: AnyObject) {
    nameField.resignFirstResponder()
    numberField.resignFirstResponder()
}
```

Этот код просто устанавливает у обоих полей редактирования статус первого отвечающего объекта. Можно совершенно безопасно вызывать метод `resignFirstResponder()` из элемента управления, который не является первым отвечающим объектом, поэтому его можно вызывать из обоих полей редактирования, не проверяя, какое из них является первым отвечающим объектом. Соберите и выполните свое приложение еще раз. На этот раз клавиатура должна исчезнуть не только после прикосновения к кнопке `Done`, но и после того, как пользователь прикоснется к любому месту экрана за пределами активного элемента управления. Именно такая реакция является наиболее естественной.

Добавление ползунка и метки

Добавим в приложение ползунок и метку, значение которой будет изменяться в зависимости от позиции ползунка. Выберите файл `Main.storyboard` в окне навигатора проекта, чтобы добавить несколько элементов управления в пользовательский интерфейс приложения. Найдите ползунок в библиотеке объектов и разместите его под полем редактирования `Number`, оставив немного места. Щелкните на вновь добавленном ползунке, чтобы выбрать его, а затем нажмите комбинацию клавиш `<Option+⌘+4>`, чтобы вернуться к окну инспектора, если оно еще не открыто. Это окно должно выглядеть так, как показано на рис. 4.21.

Ползунок позволяет выбрать число в определенном диапазоне. С помощью инспектора установите минимальное значение равным 1, максимальное значение — равным 100, а начальное значение — 50. Установите флажок `Events Continuous Update`. Он обеспечит непрерывный поток событий при изменении значения ползунка.



Рис. 4.21. Инспектор, демонстрирующий атрибуты ползунка

Перетащите метку в окно и поместите ее слева от ползунка, используя голубые линии разметки для выравнивания по верхнему краю ползунка и левому краю представления (рис. 4.22).

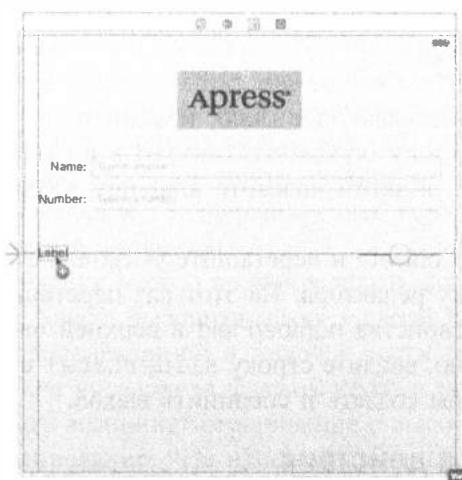


Рис. 4.22. Размещение метки ползунка

Дважды щелкните на вновь добавленной метке и измените ее текст с Label на 100. Это наибольшее значение, которое может принимать ползунок, и вы

можете использовать его для определения правильной ширины ползунка. Поскольку строка 100 короче, чем строка `Label`, вы должны изменить размеры метки, захватив среднюю правую точку изменения размера и перетащив ее влево. Постарайтесь остановиться до того, как размер шрифта станет уменьшаться. Если это все же произойдет, перетащите точку изменения размера вправо и продолжайте ее тянуть, пока не восстановится исходный размер шрифта. Можно также использовать функциональную возможность “вписать в форму”, которую мы уже обсуждали ранее. Для этого следует нажать комбинацию клавиш `<⌘+=>` или выбрать команду `Editor⇒Size to Fit Content`.

Затем измените размер ползунка, щелкнув на нем и перетащив влево левую точку изменения размера так, чтобы голубые линии разметки указали, где следует остановиться.

Добавив два дополнительных элемента управления, мы должны задать соответствующие ограничения Auto Layout. Это можно легко сделать, выбрав пиктограмму `View Controller` в окне `Document Outline` и выполнив команду `Editor⇒Resolve Auto Layout Issues⇒Add Missing Constraints`. Среда Xcode автоматически настроит эти ограничения, чтобы они соответствовали позициям всех элементов управления на экране.

Создание и связывание действий и выходов

Нам осталось лишь связать выход и действие с этими двумя элементами управления. Нужен выход, указывающий на метку, чтобы можно было изменять значение метки при движении ползунка. Кроме того, нужен метод действия, чтобы вызывать его при изменении положения ползунка. Вызовите помощник редактора для редактирования файла `ViewController.swift`, а затем нажмите клавишу `<Control>` и перетащите указатель от ползунка к точке кода, расположенной под методом `onTapGestureRecognized()` в окне помощника редактора. Когда появится всплывающее окно, измените значение поля `Connection` на `Action`, наберите строку `onSliderChanged` в поле `Name`, наберите в поле `Type` строку `UISlider`, а затем нажмите клавишу `<Return>`, чтобы создать и соединить выход.

Нажмите клавишу `<Control>` и перетащите указатель от вновь созданной метки (“100”) к помощнику редактора. На этот раз перетащите указатель в точку, расположенную ниже свойства `numberField` в верхней части файла. Когда появится всплывающее окно, введите строку `sliderLabel` в поле `Name` и нажмите клавишу `<Return>`, чтобы создать и соединить выход.

Реализация метода действия

Несмотря на то что программа Xcode создала и связала наш метод действия, мы все еще должны написать код самого метода. Добавьте в метод `onSliderChanged()` код, показанный в листинге 4.4.

Листинг 4.4. Изменение метки в зависимости от позиции ползунка

```
@IBAction func onSliderChanged(_ sender: UISlider) {
    sliderLabel.text = "\(lroundf(sender.value))"
}
```

Метод `lroundf()` при вызове из стандартной библиотеки языка C получает текущее значение ползунка и округляет его до ближайшего целого числа. Остальной код создает строку, содержащую это число, и присваивает ее метке.

Это обеспечивает реакцию контроллера на перемещение ползунка. Однако для того, чтобы быть точными, необходимо убедиться, что метка правильно отражает значение ползунка еще до того, как пользователь его коснется. Для этого в метод `viewDidLoad()` необходимо добавить строку `sliderLabel.text = "50"`.

Этот метод выполняется сразу после начала загрузки приложения из файла раскладовки, но до того, как оно будет отображено на экране. Добавленная строка гарантирует, что пользователь сразу увидит правильное начальное значение.

Сохраните файл. Затем нажмите комбинацию клавиш `<⌘+R>`, запустите приложение в симуляторе iPhone и испытайте ползунок. При его перемещении вы должны увидеть изменение текста на метке, происходящее в реальном времени. Еще один фрагмент приложения готов. Однако, если перетащить ползунок влево (сделав его значение меньше 10) или до упора вправо (положив его значение равным 100), мы увидим странные явления. По мере уменьшения значения ползунка до одной цифры метка будет сжиматься горизонтально, но, достигнув значения 3, она станет расширяться. Теперь, кроме текста, который содержит метка, мы не увидим саму метку и не обнаружим изменение ее размера, хотя увидим, что ползунок действительно изменяет его, делая метку то больше, то меньше. Тем самым поддерживается постоянное расстояние между ним и меткой.

Это просто побочный эффект программы Interface Builder, помогающей нам создавать гибкие и точно реагирующие пользовательские интерфейсы. Мы уже упоминали об ограничениях и теперь видим, как они работают. Когда мы изменили размер ползунка так, чтобы он почти касался метки, программа Interface Builder создавала ограничение, поддерживающее горизонтальное расстояние между этими элементами постоянным.

К счастью, это поведение можно переопределить собственным ограничением. Вернитесь в среду Xcode, выберите метку в своей раскладовке и щелкните на пиктограмме Pin, расположенной в нижней части раскладовки. В открывшемся всплывающем окне установите флажок `Width` и щелкните на кнопке `Add 1 Constraint`. В результате возникнет ограничение с высоким приоритетом, сообщающее системе макетирования: “Не изменяйте ширину этого ползунка”. Если теперь вы нажмете клавиши `<⌘+R>`, чтобы собрать и выполнить приложение снова, то увидите, что ползунок больше не растягивается и не сжимается при его перетаскивании.

В книге есть еще несколько примеров ограничений и их применения. Однако настало время перейти к реализации переключателей.

Реализация переключателей, кнопки сегментированного элемента управления

Вернемся к среде Xcode. Эти метания туда и обратно могут показаться немного странными, но переходы от редактирования кода в среде Xcode к настройке интерфейса в программе Interface Builder с эпизодическими остановками для проверки приложения в симуляторе системы iOS — обычная практика при разработке программ.

В нашем приложении предусмотрены два переключателя, представляющие собой маленькие элементы управления, имеющие только два состояния: включен и выключен. Кроме того, мы добавим сегментированный элемент управления, который будет скрывать и отображать переключатели. Наряду с этим элементом управления добавим кнопку, которая появляется на экране, когда пользователь нажимает правую часть сегментированного элемента управления.

Вернитесь к раскладовке, перетащите сегментированный элемент управления из библиотеки объектов в окно View немного ниже ползунка и отцентруйте его по горизонтали (рис. 4.23).

Дважды щелкните на слове First на сегментированном элементе управления и измените строку First на строку Switches. Сделав это, повторите эту процедуру для сегмента Second, переименовав его в Button (рис. 4.24) и перетащите этот элемент управления в центр.

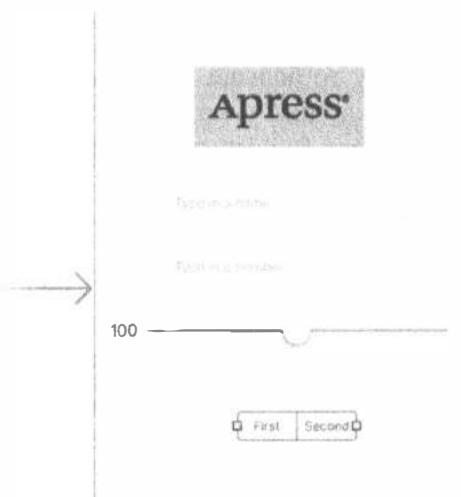


Рис. 4.23. Размещение сегментированного представления в раскладовке

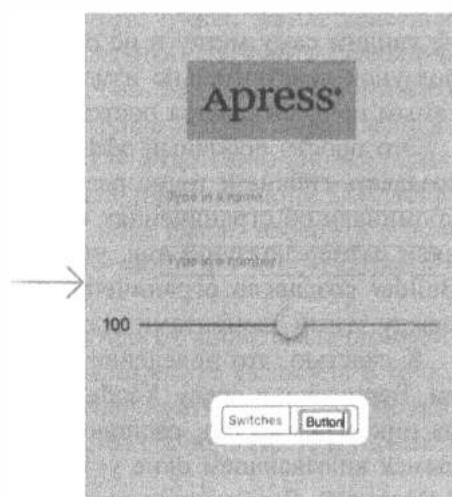


Рис. 4.24. Переименование сегментов

Добавление переключателей с метками

Захватите переключатель в библиотеке, поместите его на представление чуть ниже сегментированного элемента управления и выровняйте по левому краю. Перетащите на представление второй переключатель и выровняйте его по правому краю, расположив на одной линии с первым переключателем, как показано на рис. 4.25.

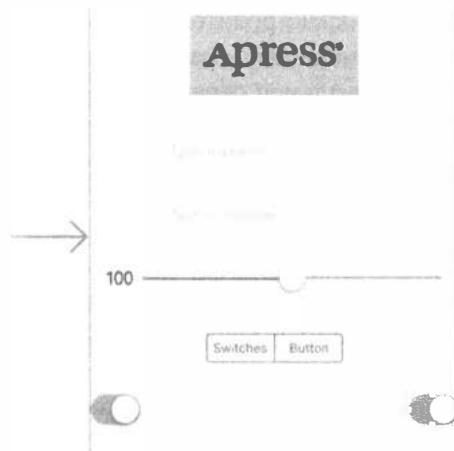


Рис. 4.25. Добавление переключателей в представление

ПОДСКАЗКА. Удерживая нажатой клавишу <Option> и перетаскивая объект в программе Interface Builder, вы создаете копию этого элемента. Если вам необходимо создать несколько экземпляров одного и того же объекта, то будет быстрее перетащить из библиотеки только один объект, а затем с помощью указанного приема сделать столько копий, сколько потребуется.

Для трех добавленных нами ограничений необходимы ограничения разметки. На этот раз мы добавим ограничения вручную. Сначала выберите сегментированный элемент управления и зафиксируйте его в центре, щелкнув на пиктограмме Align, установив флажок **Horizontally in Container** и щелкнув на кнопке Add 1 Constraint. Затем выберите сегментированный элемент управления, нажмите клавишу <Control> и перетаскивайте указатель вверх до тех пор, пока фон главного представления не станет голубым. Отпустите кнопку мыши и выполните команду **Vertical Spacing to Top Layout Guide**, чтобы зафиксировать расстояние от сегментированного элемента управления до верхнего края представления.

Перейдем к переключателям. Нажмите клавишу <Control>, перетащите указатель влево вверх по диагонали и отпустите кнопку мыши. Нажав и удерживая клавишу <Shift>, выполните команды **Leading Space to Container Margin** и **Vertical Spacing to Top Layout Guide** из всплывающего меню, а затем нажмите клавишу

<Return> или щелкните мышью где-нибудь за пределами меню, чтобы применить ограничения. Сделайте то же самое со вторым переключателем, но на этот раз перетащите указатель вверх и вправо по диагонали и выполните команды *Trailing Space to Container Margin* и *Vertical Spacing to Top Layout Guide*. После того как вы примените ограничения, программа Xcode предложит вам разные варианты в зависимости от направления, в котором вы провели соединительную линию. Если вы перетащили объект по горизонтали, вам предложат прикрепить элемент управления к левому или правому краю родительского представления, а если по вертикали — то к верхнему или нижнему краю. В данном случае нам нужны одно горизонтальное и одно вертикальное ограничения для каждого переключателя, поэтому мы вели соединительные линии по диагонали.

Связывание и создание выходов переключателя и действий

Перед тем как добавить кнопку, мы собираемся связать выходы с обоими переключателями. Кнопка, которую мы добавим позднее, фактически будет делить одно и то же место с переключателями, затрудняя связывание, поэтому мы хотим сделать это заранее. Поскольку кнопка и переключатели никогда не появляются на экране одновременно, разместить их в одном и том же месте — не проблема.

Используя помощник редактора, нажмите клавишу <Control> и перетащите курсор от переключателя к точке, расположенной ниже последнего выхода в файле *ViewController.swift*. Когда появится всплывающее окно, назовите выход *leftSwitch* и нажмите клавишу <Return>. Повторите эту процедуру для другого переключателя, назвав выход *rightSwitch*.

Теперь выберите левый переключатель, снова щелкнув на нем. Нажмите клавишу <Control> и перетащите курсор в окно помощника редактора. На этот раз остановитесь над скобкой в конце объявления класса. Когда появится всплывающее окно, измените значение поля *Connection* на *Action*, присвойте методу имя *onSwitchChanged()*, задайте атрибут *Type* его аргумента *sender* равным *UISwitch* и нажмите клавишу <Return>, чтобы создать новое действие. Повторите эту процедуру для правого переключателя, но вместо создания нового действия проведите соединительную линию к ранее созданному методу *onSwitchChanged()*. Как и в предыдущей главе, будем использовать один метод для обоих переключателей.

В заключение нажмите клавишу <Control> и перетащите курсор от сегментированного элемента управления к помощнику редактора в точку, расположенную под методом *onSwitchChanged()*. Вставьте новый метод *toggleControls()*, как вы делали это раньше, но на этот раз задайте атрибут *Type* его параметра *sender* равным *UISegmentedControl*.

Реализация действий переключателя

Сохраните раскладовку и щелкните на файле *ViewControlllew.swift*, открытом в окне помощника редактора. Найдите метод *onSwitchChanged()* и добавьте в него строки, приведенные в листинге 4.5.

Листинг 4.5. Метод onSwitchChanged()

```
@IBAction func onSwitchChanged(_ sender: UISwitch) {
    let setting = sender.isOn
    leftSwitch.setOn(setting, animated: true)
    rightSwitch.setOn(setting, animated: true)
}
```

Метод `onSwitchChanged()` вызывается при нажатии одного из двух переключателей. В этом методе мы получаем значение свойства `isOn` аргумента `sender`, представляющего нажатый переключатель, и используем это значение для установки обоих переключателей. Идея заключается в том, чтобы задание этого значения для одного переключателя изменяло состояние другого переключателя, обеспечивая их синхронизацию.

Аргумент `sender` всегда будет принимать одно из двух значений: `leftSwitch` или `rightSwitch`. В этом случае может возникнуть вопрос: зачем нам два раза присваивать значения? Ответ: из практических соображений. Это проще, чем каждый раз устанавливать значение для обоих переключателей, чтобы определять, какой из них сделал вызов. Как только переключатель вызовет наш метод, правильное значение уже будет установлено и повторное присвоение того же самого значения ничего не изменит.

Добавление кнопки

Вернитесь в программу Interface Builder и перетащите элемент управления `Button` из библиотеки на представление. Поместите его прямо поверх крайней слева кнопки и выровняйте по левому краю так, чтобы ее центр находился на одной линии с обоими переключателями (рис. 4.26).

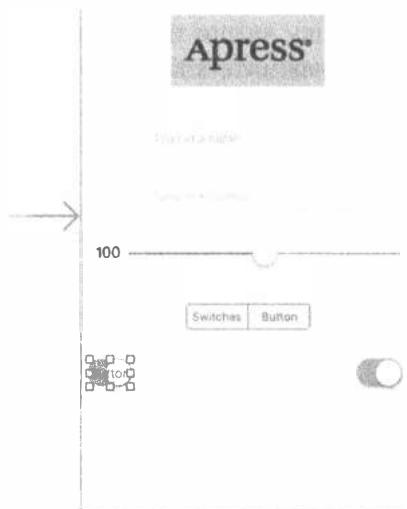


Рис. 4.26. Добавление кнопки поверх существующих переключателей

Захватите среднюю точку изменения размера на правой границе кнопки и перетаскивайте ее до тех пор, пока не достигнете голубой линии разметки, обозначающей правое поле. Кнопка должна полностью покрыть оба переключателя. Однако, поскольку по умолчанию кнопка считается прозрачной, мы по-прежнему будем видеть переключатели (рис. 4.27).

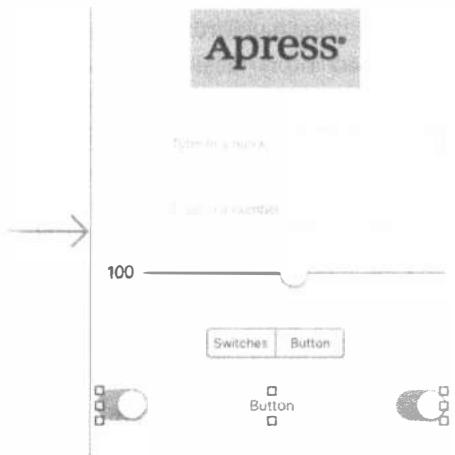


Рис. 4.27. После изменения размеров кнопка должна полностью скрыть оба переключателя

Дважды щелкните на кнопке и присвойте ей метку Do Something.

Для этой кнопки нужны ограничения Auto Layout. Мы прикрепим ее к правому краю и обеим сторонам главного представления. Нажмите клавишу <Control> и перетаскивайте указатель вверх от кнопки, пока фон представления не станет голубым, а затем отпустите кнопку мыши и выполните команду Vertical Spacing to Top Layout Guide. Теперь нажмите клавишу <Control> и перетаскивайте указатель по горизонтали, пока фон главного представления снова не станет голубым, а затем выполните команду Leading Space to Container Margin. Этот вариант станет доступным, только если вы переместите указатель достаточно далеко влево от кнопки, поэтому если вы его не увидите, то попробуйте снова и перетащите указатель влево, далеко за пределы кнопки. В заключение нажмите клавишу <Control>, перетаскивайте указатель вправо от кнопки до тех пор, пока фон главного представления не станет голубым, и выполните команду Trailing Space to Container Margin. Запустите приложение, чтобы увидеть результат.

Добавление изображения на кнопку

Если сравнить реальное приложение с приложением на рис. 4.2, то можно заметить разницу. Ваша кнопка Do Something выглядит не так, как на рисунке. Это объясняется тем, что начиная с версии iOS 7 кнопка, предусмотренная по умолчанию, выглядит очень просто: обычный фрагмент текста без контура,

границы, цветного фона и других декоративных атрибутов. Это полностью соответствует принципам дизайна, принятого компанией Apple для iOS 7 и более поздних версий, но иногда хочется использовать собственные кнопки, и мы планируем создать свой вариант.

Большинство кнопок на устройствах, работающих под управлением системы iOS, нарисованы с помощью изображений. Мы предусмотрели изображения для нашего примера в папке 04 – Button Images. Находясь в среде Xcode, выберите папку Assets.xcassets в окне навигатора проекта (этот каталог ресурсов мы уже использовали для добавления логотипа компании Apress) и перетащите оба изображения из папки 04 – Button Images, открытой в окне Finder, в область редактирования, расположенную в окне Xcode. Эти изображения будут включены в ваш проект и немедленно станут доступными для вашего приложения.

Растягивающиеся изображения

Взглянув на изображения для добавленных нами кнопок, вы, вероятно, будете удивлены их размерами. Они слишком маленькие и не способны заполнить поверхность кнопок, добавленных в раскадровку. Это объясняется тем, что изображения считаются **растягивающимися** (stretchable). Библиотека UIKit прекрасно умеет растягивать изображения и заполнять ими любое пространство. Растягивающееся изображение — это изображение, допускающее изменение размеров, которое “знает”, как целенаправленно изменить свои размеры так, чтобы достичь желаемого внешнего вида. Для данного шаблона кнопки мы не хотим равномерно растягивать ее границы по всему изображению. **Оконечные элементы** (edge insets) — это части изображения, измеряемые в пикселях, размеры которых нельзя изменять. Мы хотим, чтобы фаска вокруг краев оставалась неизменной независимо от того, какой размер имеет кнопка, поэтому необходимо задать размер ее окончных элементов.

Раньше эту часть работы приходилось программировать. Сначала надо было измерить размер изображения в пикселях, используя графический редактор, а затем задать это число в качестве размера окончных элементов. В среде Xcode 6 это больше делать не нужно! Теперь можно визуально создать нарезку любого изображения в своем каталоге ресурсов! Именно это мы собираемся сделать.

Откройте каталог ресурсов Assets.xcassets в программе, а в нем выберите элемент whiteButton. В нижней части области редактирования вы увидите кнопку с меткой Show Slicing. Щелкните на ней, чтобы начать процесс нарезки, который просто поместит кнопку Start Slicing поверх вашего изображения. Щелкните на этой кнопке, и увидите три новые кнопки, позволяющие задать вид нарезки: по вертикали, по горизонтали или в обоих направлениях. Выберите среднюю кнопку, чтобы выполнить нарезку в обоих направлениях. Программа Xcode быстро проанализирует ваше изображение, а затем найдет разделы, расположенные на однозначно определенном расстоянии от краев, а также вертикальные и горизонтальные срезы, которые следует повторить. Эти границы представлены в виде пунктирных линий (рис. 4.28). Если изображение сложное,

его можно подправить, перетащив с помощью мыши; в нашем случае автоматические окончательные элементы работают прекрасно.

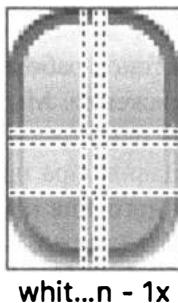


Рис. 4.28. Нарезка белой кнопки по умолчанию

Выберите элемент `blueButton` и выполните его автоматическую нарезку. Готово! Эти графические элементы готовы к использованию.

Вернитесь в раскладовку и щелкните один раз на кнопке `Do Something`. Выделив кнопку, нажмите комбинацию клавиш `<Option+⌘+4>`, чтобы открыть инспектор атрибутов. Перейдите в окно инспектора атрибутов и с помощью всплывающего меню измените тип с `System` на `Custom`. В этом инспекторе можно задать изображение и фон своей кнопки. Мы хотим использовать фон для демонстрации растягивающегося изображения, поэтому откройте меню `Background` и выполните команду `whiteButton`. Вы увидите, что вся поверхность кнопки заполнена белым изображением.

Теперь мы хотим использовать голубую кнопку для выбора внешнего вида нажатой кнопки. О состояниях кнопки мы поговорим в следующем разделе, а пока просто обратите внимание на второе всплывающее меню — `State Config`. Объект класса `UIButton` может иметь много состояний, каждое из которых имеет свой текст и изображения. До сих пор мы настраивали состояние по умолчанию, поэтому выполните во всплывающем меню команду `Highlighted`, чтобы приступить к настройке этого состояния. Вы увидите, что всплывающее меню `Background` стало пустым; щелкните на нем и выполните команду `blueButton`.

Состояния элемента управления

Каждый элемент управления в системе iOS имеет пять возможных состояний и всегда в отдельный момент времени пребывает только в одном из них.

- **По умолчанию (default).** Чаще всего элементы управления находятся в состоянии, которое задано по умолчанию. Именно в этом состоянии пребывает элемент управления, если он не находится в другом состоянии.
- **Сфокусированное (focused).** В системах, использующих механизм фокусирования, элементы управления переходят в сфокусированное состояние,

когда получают фокус. Сфокусированное состояние изменяет внешний вид элемента управления, чтобы сигнализировать, что он получил фокус. Этот внешний вид отличается от подсвеченного или выделенного. Дальнейшее взаимодействие с элементом управления может перевести его в подсвеченное или выделенное состояние.

- ⌘ **Подсвеченное (highlighted).** Подсвеченное состояние — это состояние, в котором находится элемент управления, используемый в данный момент. Кнопка переходит в это состояние, когда пользователь прикасается к ней пальцем.
- ⌘ **Выбранное (selected).** Только некоторые элементы управления поддерживают выбранное состояние. Оно обычно используется для того, чтобы обозначить, что элемент управления включен или выбран. Выбранное состояние похоже на подсвеченное, но элемент управления может по-прежнему находиться в выбранном состоянии, даже если пользователь больше не использует этот элемент управления непосредственно.
- ⌘ **Недоступное (disabled).** Элементы управления находятся в недоступном состоянии, если они были отключены с помощью переключателя `Enabled` в программе Interface Builder или их свойство `isEnabled` имеет значение `NO`.

Некоторые элементы управления системы iOS имеют атрибуты, принимающие разные значения в зависимости от их состояния. Например, задав одно изображение для свойства `isDefault`, а другое — для `isHighlighted`, мы сообщаем системе iOS, чтобы она использовала одно изображение, когда пользователь касается кнопки пальцем, а другое — в остальных случаях. По существу, при этом мы настраиваем два фоновых состояния кнопки в раскладовке.

ЗАМЕЧАНИЕ. В предыдущих изданиях книги перечислялись четыре состояния: `Normal`, `Highlighted`, `Disabled` и `Selected`, которым соответствовали значения перечисления в языке Objective-C `UIControlStateNormal`, `UIControlStateHighlighted`, `UIControlStateEnabled` и `UIControlStateSelected`. Описание этих значений можно найти в литературе, посвященной ранним версиям среды Xcode (до Xcode 8) и языка Swift (до Swift 3).

Связывание выходов и действий кнопки

Нажмите клавишу `<Control>` и перетащите указатель от новой кнопки в окно помощника редактора, в точку, расположенную ниже последнего выхода в верхнем разделе файла. Как только появится всплывающее окно, создайте новый выход с именем `doSomethingButton`. Затем нажмите клавишу `<Control>` и перетащите указатель от кнопки к точке, расположенной над закрывающей фигурной скобкой в конце файла. На этот раз вместо выхода создайте метод действия `onButtonPressed()` и установите атрибут `Type` равным `UIButton`.

Сохраните результаты работы, вернитесь в среду Xcode и протестируйте приложение. Вы увидите, что сегментированный элемент управления работает, но это ничего нам не дает. Нам нужно определить какую-то логику, в соответствии с которой кнопки и переключатели будут отображаться и скрываться.

Кроме того, изначально кнопки должны быть скрыты. Мы не сделали этого раньше, потому что эта задача немного сложнее, чем соединение выходов и действий. Теперь, когда вы уже можете соединять выходы и методы, скроем кнопку. Будем показывать кнопку, когда пользователь нажмет правую половину сегментированного элемента управления, но в начале работы кнопка должна быть скрыта. Нажмите комбинацию клавиш **<Option+⌘+4>**, чтобы открыть инспектор атрибутов, найдите раздел **View** и щелкните на флашке **Hidden**. В окне Interface Builder кнопка по-прежнему будет видимой.

Реализация действия сегментированного элемента управления

Сохраните раскладовку и щелкните на файле `ViewController.swift`. Найдите метод `toggleControls()`, который создала программа Xcode, и добавьте в него новый код, приведенный в листинге 4.6.

Листинг 4.6. Метод, показывающий и скрывающий переключатели

```
@IBAction func toggleControls(_ sender: UISegmentedControl) {
    if sender.selectedSegmentIndex == 0 { // Выбран раздел switches
        leftSwitch.isHidden = false
        rightSwitch.isHidden = false
        doSomethingButton.isHidden = true
    } else {
        leftSwitch.isHidden = true
        rightSwitch.isHidden = true
        doSomethingButton.isHidden = false
    }
}
```

Этот код проверяет свойство `selectedSegmentIndex` аргумента `sender`, который сообщает о том, какой переключатель вызвал метод. Первый раздел, `switches`, имеет индекс 0. Этот факт мы отметили в комментарии, чтобы позднее об этом не забыть. В зависимости от того, какой переключатель был выбран, соответствующие элементы управления будут появляться или исчезать.

Прежде чем выполнить приложение, воспользуемся несложным трюком, чтобы сделать его немного лучше. В версии iOS 7 компания Apple внедрила несколько новых парадигм графического пользовательского интерфейса. Одна из них — строка состояния в верхней части экрана в приложении для версии iOS 7 является прозрачной, так что содержание приложения просвечивает сквозь нее. В данный момент пиктограмма Apress бросается в глаза на фоне белого фона, поэтому желательно окрасить желтым цветом все приложение. Выберите

в файле Main.storyboard главное представление содержимого и нажмите клавиши <Option+⌘+4>, чтобы открыть инспектор атрибутов. Щелкните на пункте для выбора цвета с меткой **Background** (который в данный момент содержит белый прямоугольник,) чтобы открыть стандартный селектор цветов OS X. Одна из функциональных возможностей этого селектора цветов заключается в том, что он позволяет выбрать любой цвет, который вы видите на экране. Открыв селектор цветов, щелкните на графическом представлении Apress в раскладовке. Затем щелкните на пиктограмме с изображением лупы в левом верхнем углу селектора цветов и снова на графическом представлении Apress. В верхней части селектора цветов сразу после увеличительного стекла должен появиться цвет фона для изображения Apress. Для того чтобы сделать его основным цветом фона в главном представлении содержимого, выберите главное представление в окне **Document Outline**, а затем щелкните на желтом цвете в селекторе цветов. Закройте селектор цветов.

Цвета фона и изображения Apress могут немного отличаться, но на симуляторе или на устройстве они будут совпадать. Эти цвета могут не совпадать в программе Interface Builder из-за того, что система OS X автоматически адаптирует цвета к дисплею, который вы используете. На устройстве iOS и в симуляторе этого не происходит.

Запустите приложение, и увидите, что желтый цвет заполняет весь экран, так что строка состояния и содержимое приложения визуально различить невозможно. Если содержимое приложения не требует прокрутки на экране или другое содержимое предусматривает наличие панели навигации или других элементов управления в верхней части экрана, этот способ может оказаться удобным. Он позволяет демонстрировать полноэкранное содержимое без перекрытия со строкой состояния.

Если вы все сделали правильно, то сможете переключаться между кнопками и парами переключателей, используя сегментированный элемент управления, а если вы нажмете какой-нибудь переключатель, то второй переключатель также изменит свое значение. Однако кнопка по-прежнему ничего не делает. Прежде чем реализовать ее, поговорим немного о списках действий и сигналах.

Реализация списка действий и сигнала

Списки действий (*action sheets*) и **предупреждения** (*alerts*) обеспечивают обратную связь с пользователем.

- » Списки действий используются для того, чтобы заставить пользователя выбрать один из двух или более вариантов. Список действий появляется в нижней части экрана и предлагает пользователю выбрать одну из нескольких кнопок (см. рис. 4.3). Пользователь не может продолжать использовать приложение, пока не нажмет одну из кнопок. Списки действий часто используются для подтверждения потенциально опасных или необратимых действий, таких как удаление объекта.

- Предупреждения появляются в синем прямоугольнике с закругленными углами, расположенным в центре экрана (см. рис. 4.4). Как и списки действий, предупреждения вынуждают пользователя ответить на вопрос, прежде чем продолжить работу. Предупреждения обычно используются для информирования пользователя о чем-то важном или неожиданном и, в отличие от списка действий, могут предлагать пользователю нажать только одну кнопку, хотя, если необходимо предоставить выбор, могут содержать несколько кнопок.

ЗАМЕЧАНИЕ. Представление, вынуждающее пользователя сделать выбор, прежде чем продолжить работу, называется модальным (modal view).

Демонстрация списка действий

Перейдем к файлу ViewController.swift и реализуем метод, выполняющий действие кнопки. Приведем изменения, которые необходимо внести в пустой метод onPressed(), созданный программой Xcode автоматически (листинг 4.7).

Листинг 4.7. Демонстрация списка действий

```

@IBAction func onPressed(_ sender: UIButton) {
    let controller = UIAlertController(title: "Are You Sure?",
                                      message:nil, preferredStyle: .actionSheet)

    let yesAction = UIAlertAction(title: "Yes, I'm sure!",
                                 style: .destructive, handler: { action in
        let msg = self.nameField.text!.isEmpty
            ? "You can breathe easy, everything went OK."
            : "You can breathe easy, \(self.nameField.text),"
            + "everything went OK."
        let controller2 = UIAlertController(
            title:"Something Was Done",
            message: msg, preferredStyle: .alert)
        let cancelAction = UIAlertAction(title: "Phew!",
                                         style: .cancel, handler: nil)
        controller2.addAction(cancelAction)
        self.present(controller2, animated: true,
                     completion: nil)
    })

    let noAction = UIAlertAction(title: "No way!",
                                style: .cancel, handler: nil)

    controller.addAction(yesAction)
    controller.addAction(noAction)

    if let ppc = controller.popoverPresentationController {
        ppc.sourceView = sender
        ppc.sourceRect = sender.bounds
    }
}

```

```
present(controller, animated: true, completion: nil)
}
```

Что именно мы сделали? В методе `onButtonPressed()` мы выделили память и инициализировали объект класса `UIAlertController`, который будет представлять список действий или сигнал.

```
let controller = UIAlertController(title: "Are You Sure?",  
                                 message:nil, preferredStyle: .actionSheet)
```

Первый параметр — это заголовок, который будет выведен на экран. Взгляните еще раз на рис. 4.3, чтобы увидеть заголовок, который выводится над списком действий. Следующий аргумент — это сообщение, которое выводится под заголовком более мелким шрифтом. В этом примере мы не используем сообщение, поэтому вместо него передаем параметр `nil`. Последний параметр указывает, хотим ли мы, чтобы контроллер выводил на экран сигнал (значение `UIAlertControllerStyle.alert`) или список действий (`UIAlertControllerStyle.actionSheet`). Поскольку нам нужен список действий, мы передаем параметр `UIAlertControllerStyle.actionSheet`.

Контроллер сигнала по умолчанию не имеет никаких кнопок — вы должны самостоятельно создать объект класса `UIAlertAction` для каждой кнопки, которую хотите добавить в контроллер. Ниже приводится часть кода, создающего две кнопки для нашего списка действий (листинг 4.8).

Листинг 4.8. Создание кнопок для списка действий

```
let yesAction = UIAlertAction(title: "Yes, I'm sure!",  
                           style: .destructive, handler: { action in  
    // Код пропущен — см. ниже.  
})  
  
let noAction = UIAlertAction(title: "No way!",  
                           style: .cancel, handler: nil)
```

Для каждой кнопки необходимо задать заголовок, стиль и обработчик, который вызывается при ее нажатии. Существуют три возможных стиля.

- ⌘ Стиль `UIAlertActionStyle.destructive` должен использоваться, если кнопка является триггером разрушительного, опасного или необратимого действия, например удаления или перезаписи файла. Заголовок кнопки в этом стиле прорисовывается красным цветом и полужирным шрифтом.
- ⌘ Стиль `UIAlertActionStyle.default` используется для обычных кнопок, таких как **OK**, когда действие не является разрушительным. Заголовок таких кнопок выводится на экран с помощью обычного синего шрифта.
- ⌘ Стиль `UIAlertActionStyle.cancel` используется для кнопки **Cancel**. Ее заголовок прорисовывается полужирным синим шрифтом.

В заключение добавим в контроллер кнопки.

```
[controller addAction:yesAction];
[controller addAction:noAction];
```

Для того чтобы сигнал или список действий стал видимым, необходимо попросить текущий контроллер представления предъявить контроллер сигнала. Вот как можно предъявить список действий (листинг 4.9).

Листинг 4.9. Предъявление списка действий

```
if let ppc = controller.popoverPresentationController {
    ppc.sourceView = sender
    ppc.sourceRect = sender.bounds
}
presentViewController(controller, animated: true, completion: nil)
```

Первые четыре строки определяют, где появится список действий. В этих строках мы получаем контроллер всплывающего представления от контроллера сигнала и задаем свойства sourceView и sourceRect. Вскоре мы вернемся к этим свойствам. В заключение мы делаем представление видимым, вызывая метод контроллера представления present(_:_animated:completion:) и передавая ему контроллер сигнала в качестве контроллера, который должен быть предъявлен. Если в роли предъявляемого контроллера используется контроллер представлений, его представление временно заменяет текущее представление. Если в роли предъявляемого контроллера выступает контроллер сигнала, то список действий или сигнал частично перекрывает текущее представление; остаток представления затеняется прозрачным фоном, сквозь который можно видеть нижележащее представление, работать с которым невозможно, пока вы не переключите контроллер представлений.

Перейдем к настройке контроллера всплывающего представления. На устройстве iPhone список действий всегда всплывает снизу, как показано на рис. 4.3, но на устройстве iPad он имеет **всплывающее окно** (popover), которое представляет собой небольшой закругленный прямоугольник со стрелкой, указывающей на другое представление, ставшее причиной его появления. Список действий в симуляторе iPad показан на рис. 4.29.

Как видите, стрелка всплывающего окна показывает на кнопку Do Something. Это вызвано тем, что мы установили свойство sourceView в контроллере всплывающего

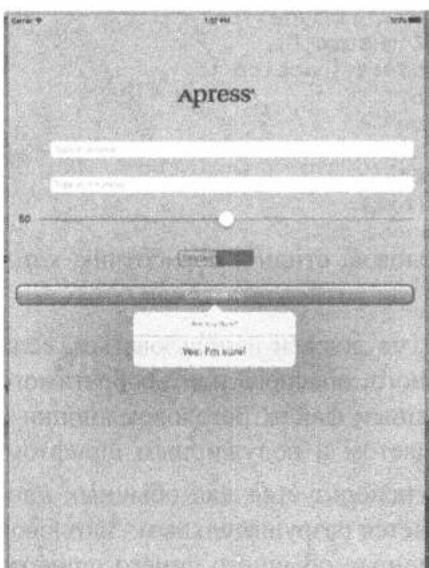


Рис. 4.29. Список действий на экране устройства iPad Air

окна так, чтобы стрелка указывала на эту кнопку, а свойство `sourceRect` связано с контуром этой кнопки (листинг 4.10).

Листинг 4.10. Задание свойств `sourceView` и `sourceRect`

```
if let ppc = controller.popoverPresentationController {
    ppc.sourceView = sender
    ppc.sourceRect = sender.bounds
}
```

Обратите внимание на конструкцию `if let` — она необходима, потому что на устройстве iPhone контроллер сигнала не предъявляет список действий в виде всплывающего окна, поэтому для свойства `popoverPresentationController` установлено значение `nil`.

На рис. 4.29 всплывающее окно появляется под кнопкой, которая его вызвала, но это можно изменить, задав свойство контроллера представления всплывающего окна `permittedArrowDirections`, которое является маской допустимых направлений для стрелки. Следующий код перемещает всплывающее окно над кнопкой с помощью присвоения этому свойству значения `UIPopoverArrowDirection.down`, как показано в листинге 4.11.

Листинг 4.11. Создание кнопок для списка действий

```
if let ppc = controller.popoverPresentationController {
    ppc.sourceView = sender
    ppc.sourceRect = sender.bounds
    ppc.permittedArrowDirections = .down
}
```

Если вы сравните рис. 4.29 и 4.3, то увидите, что на экране устройства iPad нет кнопки *No Way!*. Контроллер представления не использует на устройстве iPad кнопки со стилем `UIAlertStyle.cancel`, потому что пользователи привыкли убирать всплывающее окно, касаясь экрана за пределами этого окна.

Вывод предупреждения

Когда вы нажимаете кнопку *Yes, I'm Sure!*, вы хотите вывести на экран сигнал с сообщением. При нажатии кнопки, добавленной в контроллер сигнала, список действий (или сигнал) выключаются, а его обработчик вызывается со ссылкой на объект класса `UIAlertAction`, из которого кнопка была создана (листинг 4.12).

Листинг 4.12. Вывод на экран предупреждения

```
let yesAction = UIAlertAction(title: "Yes, I'm sure!",
    style: .destructive, handler: { action in
        let msg = self.nameField.text!.isEmpty
            ? "You can breathe easy, everything went OK."
            : "You can breathe easy, \(self.nameField.
                text),"
            + "everything went OK."})
```

```

        let controller2 = UIAlertController(
            title:"Something Was Done",
            message: msg, preferredStyle: .alert)
        let cancelAction = UIAlertAction(title: "Phew!",
            style: .cancel, handler: nil)
        controller2.addAction(cancelAction)
        self.present(controller2, animated: true,
            completion: nil)
    )
}

```

Сначала в блоке обработчика создается новая строка, которая будет выводиться на экран. В реальном приложении здесь можно предусмотреть конкретные действия. Пока мы лишь имитируем какие-то действия и уведомление пользователя с помощью сигнала. Если пользователь введет имя в верхнем поле редактирования, мы запомним его и будем использовать в сообщении, в противном случае сгенерируем следующее сообщение:

```

let msg = self.nameField.text!.isEmpty
    ? "You can breathe easy, everything went OK."
    : "You can breathe easy, \(self.nameField.text),"
    + " everything went OK."

```

Следующие несколько строк вам должны быть знакомы. Представления сигнала и списка действия создаются и используются аналогично. Мы всегда начинаем с создания объекта класса UIAlertController:

```

let controller2 = UIAlertController(
    title:"Something Was Done",
    message: msg, preferredStyle: .alert)

```

Здесь мы снова передаем заголовок сообщения. Но на этот раз передаем более подробное сообщение, а именно — созданную нами строку. Последний параметр задает стиль. В данном случае мы выбрали стиль UIAlertControllerStyle.alert, потому что хотим, чтобы на экран выводился сигнал, а не список действий. Затем мы создаем объект класса UIAlertAction для кнопки отмены сигнала и добавляем его в контроллер:

```

let cancelAction = UIAlertAction(title: "Phew!",
    style: .cancel, handler: nil)
controller2.addAction(cancelAction)

```

В заключение выводим сигнал на экран с помощью контроллера представления сигнала:

```
self.present(controller2, animated: true, completion: nil)
```

Созданный сигнал показан на рис. 4.4. Легко видеть, что наш код не пытается получить и настроить контроллер представления сигнала в виде всплывающего окна. Это объясняется тем, что сигналы появляются в маленьком закругленном представлении в центре экрана устройства iPhone или iPad.

Сохраните файл `ViewController.swift`, а затем соберите, запустите и протестируйте завершенное приложение.

Резюме

Это была большая глава. Надеемся, что вы не были шокированы лавиной новой информации, хотя мы использовали большое количество элементов управления и показали множество деталей реализации. Вы получили большой опыт работы с выходами и действиями и увидели, как можно использовать иерархическую природу представления. Вы узнали о состояниях элементов управления, а также о списках действий и предупреждениях.

В приложении `Control Fun` есть много интересного. Поработайте с ним. Измените значения, поэкспериментируйте с кодом, и увидите, какую роль играют разные параметры в программе `Interface Builder`. Мы не в состоянии продемонстрировать все возможные варианты элементов управления в системе iOS, но это приложение представляет собой хорошую отправную точку и охватывает много базовых понятий.

В следующей главе мы рассмотрим, что происходит, когда пользователь вращает устройство iOS, меняя его ориентацию с книжной на альбомную, и наоборот. Возможно, вам известно, что многие приложения меняют внешний вид в зависимости от ориентации устройства, и мы покажем, как это можно реализовать в своих приложениях.

ГЛАВА 5



Вращение устройства

Устройства iPhone и iPad представляют собой изумительные изобретения с точки зрения внешнего вида, удобства использования и функциональности. Инженеры компании Apple исумели втиснуть максимум функциональных возможностей в очень маленькую коробочку. Например, в этих устройствах существует механизм, позволяющий использовать их как в книжной (вытянутой в высоту), так и в альбомной (растянутой в ширину) ориентации, а также изменять ориентацию при повороте устройства. Проявление этой функциональной возможности, которая называется **автоматическим поворотом** (*autorotation*), можно увидеть в веб-браузере системы iOS Mobile Safari (рис. 5.1). В этой главе мы подробно рассмотрим механизм автоматического поворота, а затем перейдем к реализации этой функциональной возможности в приложениях.

До появления системы iOS 8, если вы хотели спроектировать приложение, которое могло бы работать как на устройствах iPhones, так и на устройствах iPads, вы должны были создать отдельную раскладовку для устройств iPhones и отдельную раскладовку для устройств iPad. В системе iOS 8 все изменилось. Компания добавила интерфейсы прикладного программирования в библиотеку UIKit и инструменты в среду Xcode, благодаря чему стало возможно создавать приложения, которые могут работать на любых устройствах (или, используя терминологию компании Apple, адаптироваться к любому устройству), используя только одну раскладовку. Вы по-прежнему должны внимательно учитывать формфакторы разных устройств, но теперь это можно делать в одном месте. Что еще лучше, с помощью функции Preview, описанной в главе 3, можно немедленно увидеть, как будет выглядеть ваше приложение на экране любого устройства, даже не запуская симулятор. Адаптивные макеты приложений будут рассмотрены во второй части этой главы.



Рис. 5.1. Как и многие приложения для системы iOS, браузер Mobile Safari изменяет свое представление в зависимости от того, как именно пользователь держит устройство. Это позволяет максимально использовать доступное пространство на экране

Механизм автоматического поворота

Возможность работать как в книжной, так и в альбомной ориентации есть не у всех приложений. Некоторые приложения для устройства iPhone (например, Weather) поддерживают только одну ориентацию. Однако это не относится к устройству iPad, для которого компания Apple рекомендует, чтобы практически все приложения (за исключением иммерсивных приложений, таких как игры) поддерживали любую ориентацию. Большинство собственных приложений для устройства iPad, разработанных компанией Apple, прекрасно работают в обоих режимах. Многие из них используют разные ориентации, для того чтобы продемонстрировать разные представления ваших данных. Например, приложения Mail и Notes используют альбомный режим, чтобы показать список объектов (папок, сообщений или заметок) слева и выбранный объект — справа, а книжная ориентация позволяет сконцентрировать внимание пользователя на деталях только что выбранного объекта.

Для приложений iPhone можно сформулировать правило: автоматический поворот необходимо реализовывать, если он расширяет возможности пользователя. Для приложений iPad это правило звучит несколько иначе: автоматический

поворот необходимо реализовывать, если у вас нет весомых причин этого не делать. К счастью, компания Apple проделала огромную работу, чтобы скрыть сложные детали автоматического поворота в системе iOS и в пакете UIKit, поэтому реализация этой функциональной возможности в ваших приложениях для системы iOS является довольно простой.

Автоматический поворот реализуется в контроллере представления. Если пользователь поворачивает устройство, активный контроллер представления получает запрос о том, готов ли он изменить ориентацию (как это сделать, вы узнаете из настоящей главы). Если контроллер представления отвечает положительно, ориентация окна и представления приложения, а также их размеры будут изменены.

В устройствах iPhone и iPod touch представление начинает работу в книжном режиме, в котором высота экрана больше его ширины. Реальные размеры доступной области на экране разных устройств приведены в табл. 1.1 главы 1. Однако следует заметить, что если ваше приложение имеет **строку состояния** (status bar), то размер экрана, действительно доступного для вашего приложения, может быть уменьшен на 20 точек по вертикали. Страна состояния — это полоска высотой 20 точек, расположенная в верхней части экрана (см. рис. 5.1). На нее выводятся сила сигнала, время и заряд батареи.

Когда телефон переключается в альбомный режим, представление поворачивается вдоль окна приложения и изменяет размеры, чтобы заполнить все окно. Например, на устройстве iPhone 6/6s экран имеет размеры 375×667 точек в книжном режиме и 667×375 точек в альбомном. Как и прежде, размер экрана по вертикали, выделенный для приложений, имеющих строку состояния (т.е. для большинства приложений), уменьшается на 20 точек. В устройствах iPhone, работающих под управлением системы iOS 8, строка состояния в альбомной ориентации скрывается.

Точки, пиксели и дисплей Retina

У читателей может возникнуть вопрос: почему мы говорим о точках, а не пикселях? Ведь в предыдущих изданиях книги мы измеряли размеры экрана в пикселях, а не в точках. Причина заключается в том, что компания Apple разработала **дисплей Retina** (Retina display). Дисплей Retina (от англ. *сетчатка*. — *Примеч. ред.*) — это маркетинговый термин компании Apple, которым она обозначает экраны с высоким разрешением на устройствах iPhone 4 и более поздних версий. Разрешение этих экранов приведено в табл. 1.1. В большинстве случаев оно вдвое больше аппаратного, а на устройстве iPhone 6 Plus — втрое.

К счастью, в большинстве случаев нам ничего не надо делать, чтобы учсть этот факт. Работая с визуальными элементами, мы задаем размеры и расстояния между объектами в точках, а не в пикселях. Для старых устройств iPhone, iPad 2 и iPad Mini 1 точки и пиксели эквивалентны. Однако в современных моделях iPhone и iPod touch одна точка соответствует четырем пикселям (два пикселя по горизонтали и два по вертикали), а экран, например устройства iPhone 5s,

по-прежнему имеет ширину 320 точек, в то время как фактически он имеет разрешение, равное 640 пикселям по горизонтали. На устройстве iPhone 6s коэффициент масштабирования равен трем, так что каждая точка отображается в квадрат 9×9 пикселей. Считайте это “виртуальным разрешением”, которое система iOS автоматически отображает в реальное физическое разрешение. Более подробно мы поговорим об этом в главе 16.

В обычных приложениях перемещение пикселей выполняется системой iOS. Основная задача вашего приложения — обеспечить точное согласование элементов интерфейса с размерами экрана.

Способы реализации автоматического вращения

Для обработки вращения устройства можно задать корректные **ограничения** для всех объектов, образующих интерфейс. Ограничения сообщают устройству, работающему под управлением системы iOS, как должны функционировать элементы управления, когда содержащее их представление изменяет размеры. Как это связано с вращением устройства? Когда устройство вращается, размеры его экрана меняются местами, поэтому размеры окна просмотра изменяются.

Проще всего использовать ограничения, настраивая их в программе Interface Builder (IB). Она позволяет определять ограничения, описывающие изменение положения и размеров компонентов графического пользовательского интерфейса при модификации родительского представления или перемещении других представлений, окружающих эти компоненты. Мы уже касались этой темы в главе 4, а в этой главе рассмотрим ее глубже. Ограничения можно считать некими математическими уравнениями, описывающими геометрию представления, а систему представлений в iOS — блоком решения этих уравнений, который перестраивает объекты так, чтобы эти уравнения выполнялись.

Ограничения впервые появились в системе iOS 6, но в компьютерах Mac они существовали намного раньше. В системах iOS и macOS ограничения можно использовать вместо старых “пружин и растяжек”. Ограничения позволяют делать все, что делала старая технология, предоставляя много новых возможностей.

Выбор ориентации представления

Создадим простое приложение, демонстрирующее изменение ориентации. Начните новый проект типа Single View Application в среде Xcode и назовите его Orientations. Выберите в списке Devices пункт Universal и сохраните проект.

Прежде чем скомпоновать графический пользовательский интерфейс в раскладовке, необходимо сообщить системе iOS, что наше представление поддерживает автоматический поворот. Существуют два способа сделать это. Можно создать настройки приложения, которые будут задаваться по умолчанию для всех контроллеров представлений, или уточнять их для каждого отдельного контроллера. Мы будем использовать оба подхода, начиная с общих настроек контроллеров представлений.

Поддержка ориентации на уровне приложения

Сначала мы должны указать, какие виды ориентации поддерживает наше приложение. Когда откроется проектное окно Xcode, вы увидите настройки своего проекта. Если нет, щелкните на верхней строке навигатора проекта (которая содержит имя вашего проекта), а затем откройте вкладку General. Среди параметров, доступных на вкладке General, есть раздел Deployment Info, а в нем — подраздел Device Orientation со списком флажков (рис. 5.2).



Рис. 5.2. Вкладка General показывает поддерживаемые ориентации для вашего проекта наряду с остальными параметрами

Таким образом, мы указываем, какие виды ориентации поддерживает ваше приложение. Совершенно необязательно, чтобы каждое представление приложения использовало все выбранные виды ориентации. Однако, если вы собираетесь поддерживать какую-то ориентацию в каких-нибудь представлениях, выберите ее на этой вкладке. Ориентация Upside Down по умолчанию отключена. Это объясняется тем, что телефон обычно звонит, когда пользователь держит его вертикально, и, как правило, остается в этом положении во время разговора.

Откройте список Devices, расположенный над флагжками (рис. 5.3), и увидите, как настроить разные варианты ориентации на устройствах iPhone и iPad. Если вы выберете пункт iPad, то увидите, что по умолчанию выбраны все четыре ориентации, поскольку планшет может использоваться в любом положении.

ЗАМЕЧАНИЕ. Четыре кнопки, показанные на рис. 5.2 и 5.3, означают добавление и удаление записей в файле Info.plist. Если вы щелкнете на файле Info.plist в окне навигатора проекта, то увидите два пункта, Supported Interface Orientations и Supported Interface Orientations (iPad), с подпунктами для выбранных видов ориентации. Выбирая и отменяя выбор этих кнопок на вкладке General, вы добавляете и удаляете элементы из этого массива. Использование кнопок проще и позволяет делать меньше ошибок, поэтому мы настоятельно рекомендуем пользоваться ими, полагая, что вы точно знаете их назначение.



Рис. 5.3. Настройка разных видов ориентации для iPhone и iPad

В качестве устройства мы снова выберем iPhone6s. Теперь выберите файл Main.storyboard, найдите объект *Label* в библиотеке объектов и перетащите его на свое приложение, отпустив прямо в центре, как показано на рис. 5.4. Выберите текст метки и измените его на “This way up”. Изменение текста может привести к сдвигу позиции метки, поэтому верните ее в центр экрана.



Рис. 5.4. Настройка метки в книжной ориентации

Необходимо добавить ограничения Auto Layout, чтобы зафиксировать кнопку до запуска приложения, поэтому нажмите клавишу <Control> и перетаскивайте указатель вверх от метки, пока цвет фона внешнего представления не станет голубым, а затем отпустите кнопку мыши. Нажав и удерживая клавишу <Shift>, выполните команды Vertical Spacing to Top Layout Guide и Center Horizontally in Container из всплывающего меню, а затем нажмите клавишу <Return>. Теперь нажмите клавиши <⌘+R>, чтобы собрать и запустить это простое приложение. Когда оно появится в симуляторе, попробуйте повернуть устройство несколько раз, нажимая клавиши <⌘+←> или <⌘+→>. Вы увидите, что все приложение (включая добавленную метку) будет вращаться во всех направлениях, кроме направления “сверху вниз”, в соответствии с нашими настройками. Выполните приложение в симуляторе iPad, чтобы убедиться, что оно вращается во всех четырех направлениях.

Итак, мы идентифицировали виды ориентации, поддерживаемые нашим приложением, но это еще не все. Мы должны также указать для каждого контроллера представления, какие виды ориентации он поддерживает. Эти виды ориентации должны образовывать подмножество видов ориентации, перечисленных выше.

Настройка поддержки поворота

Настроим наш контроллер представления так, чтобы он поддерживал другой, меньший набор допустимых вариантов ориентации. Отметим, что глобальная конфигурация для приложения задает нечто вроде абсолютного верхнего предела для допустимых вариантов ориентации. Если глобальная конфигурация не содержит ориентации “сверху вниз”, например, то ни у одного контроллера представлений нет никаких способов заставить систему повернуть экран в направлении “сверху вниз”. В контроллере представлений можно лишь ужесточить ограничения на допустимые варианты ориентации.

Щелкните на файле `ViewController.swift` в окне навигатора проекта. Мы реализуем метод, определенный в суперклассе `UIViewController`, который позволяет нам указать допустимые варианты ориентации.

```
override func supportedInterfaceOrientations() ->
UIInterfaceOrientationMask {
    return UIInterfaceOrientationMask(rawValue:
        (UIInterfaceOrientationMask.portrait.rawValue
        | UIInterfaceOrientationMask.landscapeLeft.rawValue))
}
```

Этот метод позволяет вернуть объект класса `UIInterfaceOrientationMask` для допустимых вариантов ориентации, как это делается в языке C. Вызывая этот метод, система запрашивает у контроллера представления, поддерживает ли он переход к конкретной ориентации. В данном случае мы возвращаем значение, которое означает поддержку двух видов ориентации: альбомная ориентация по умолчанию и ориентация, полученная при повороте экрана на 90° по

часовой стрелке. Логическая операция OR (вертикальная черта) объединяет эти два объекта класса `UIInterfaceOrientationMask` и возвращает комбинированное значение.

Устройство, работающее под управлением системы iOS, может осуществлять такую поддержку четырьмя способами.

- `UIInterfaceOrientationMask.portrait.rawValue`
- `UIInterfaceOrientationMask.landscapeLeft.rawValue`
- `UIInterfaceOrientationMask.landscapeRight.rawValue`
- `UIInterfaceOrientationMask.portraitUpsideDown.rawValue`

Кроме того, существуют заранее заданные комбинации общих вариантов ориентации. Они функционально эквивалентны операции OR и позволяют сэкономить время и сделать код более понятным.

- `UIInterfaceOrientationMask.landscape.rawValue`
- `UIInterfaceOrientationMask.all.rawValue`
- `UIInterfaceOrientationMask.allButUpsideDown.rawValue`

Когда устройство, работающее под управлением системы iOS, изменит пространственную ориентацию, из активного контроллера представления будет вызван метод `supportedInterfaceOrientations()`. В зависимости от возвращаемого значения, идентифицирующего новую ориентацию, приложение решит, следует ли повернуть экран. Поскольку каждый подкласс контроллера представления может по-разному реализовать этот поворот, одни приложения могут поддерживать автоматический поворот некоторых из своих представлений, а другие нет. Запустите приложение снова и убедитесь, что теперь вы можете вращать свое устройство только в двух видах ориентации, идентификаторы которых возвращают метод `supportedInterfaceOrientations()`. Выражение `.rawValue` в конце каждого идентификатора ориентации возвращает целое значение, соответствующее проверяемой ориентации.

ЗАМЕЧАНИЕ. Вы можете вращать устройство, но само представление вращаться не будет, так что кнопка всегда будет возвращаться в верхнюю часть экрана за исключением двух видов ориентации.

ТЕХНОЛОГИЯ АВТОДОПОЛНЕНИЯ В ДЕЙСТВИИ

Вы заметили, что некоторые системные константы в устройстве iPhone всегда начинаются с одних и тех же букв? Одна из причин, по которым имена `UIInterfaceOrientationMask.portrait`, `UIInterfaceOrientationMask.portraitUpsideDown`, `UIInterfaceOrientationMask.landscapeLeft` `UIInterfaceOrientationMask.landscapeRight` начинаются с префикса `UIInterfaceOrientationMask`, заключается в желании использовать технологию автодополнения, реализованную в среде Xcode.

Возможно, вы заметили, что, когда вы набираете символы на клавиатуре, среда Xcode часто пытается дополнить слово, которое вы печатаете. Это — технология автодополнения кода (code completion) в действии.

Разработчики часто не могут запомнить все константы, определенные в системе, но могут запомнить имена групп, которым они принадлежат. Когда нужно указать ориентацию, просто введите слово `UIInterfaceOrientationMask` (или даже `UIInterface`), а затем нажмите клавишу `<Esc>`, чтобы увидеть список имен, начинающихся с этих букв. (В настройках среди Xcode можно вместо клавиши `<Esc>` для открытия этого списка выбрать другую.) Для просмотра списка можно использовать клавиши навигации, а выбор осуществляется нажатием клавиши `<Tab>` или `<Return>`. Этот способ намного быстрее просмотра значений в документации или заголовочных файлах.

Этот метод может возвращать любую маску для ориентации устройства. Вы можете заставить систему ограничить ваше представление только теми видами ориентации, которые имеют смысл для вашего приложения, но не забывайте о глобальной конфигурации! Помните, что если вы, например, отключили ориентацию “сверху вниз” в глобальной конфигурации, то ни одно из ваших представлений не будет ориентировано так, независимо от того, как объявлен метод `supportedInterfaceOrientations()` в контроллере представления.

ЗАМЕЧАНИЕ. На самом деле система iOS различает два типа ориентации. Одна из них рассмотрена выше и называется **ориентацией интерфейса** (*interface orientation*), а другая называется **ориентацией устройства** (*device orientation*). Ориентация устройства определяет то, как пользователь держит устройство в данный момент. Ориентация интерфейса определяет способ прорисовки экрана при его вращении. Если вы держите стандартное устройство iPhone вертикально, то его ориентация будет вертикальной, но ориентация интерфейса может быть любой из трех остальных возможных, поскольку приложения для iPhone обычно не поддерживают вертикальную ориентацию по умолчанию.

СОЗДАНИЕ МАКЕТА ПРОЕКТА

Находясь в среде Xcode, создайте другой новый проект по шаблону *Single View Application* и назовите его *Layout*. Щелкните на файле *Main.storyboard*, чтобы открыть раскладовку в программе Interface Builder. Атрибуты автоматического изменения размеров имеют одно хорошее свойство — они требуют очень мало кода. Нет необходимости указывать, какую ориентацию мы выбрали, как в коде контроллера представления, а остальная реализация автоматического изменения размеров может быть осуществлена непосредственно в программе Interface Builder.

Для того чтобы увидеть, как это работает, перетащите четыре метки из библиотеки на свое представление и разместите их так, как показано на рис. 5.5.

Используя пунктирные голубые линии разметки, выровняйте их по углам. В нашем примере мы используем экземпляры класса `UILabel`, чтобы продемонстрировать ограничения макета графического пользовательского интерфейса, но эти же правила относятся ко всем интерфейсным объектам.



Рис. 5.5. Добавление четырех меток в раскладовку

Дважды щелкните на каждой из меток и назовите их так: **UL** — верхняя левая (*upper-left*); **UR** — верхняя правая (*upper-right*); **LL** — нижняя левая (*lower-left button*); **LR** — нижняя правая (*lower-right button*).

Теперь посмотрим, что произойдет, если мы не зададим ни одно ограничение Auto Layout. Соберите и выполните приложение на симуляторе iPhone 5s. Когда симулятор iOS начнет работу, вы увидите лишь метки в левой части экрана, остальные метки, расположенные справа, оказываются за пределами экрана. Более того, метка в левом нижнем углу никак не реагирует на действия пользователя. Выполните команду `Hardware⇒Rotate Left`, которая будет имитировать перевод устройства iPhone в альбомный режим, и вы увидите все метки (рис. 5.6).

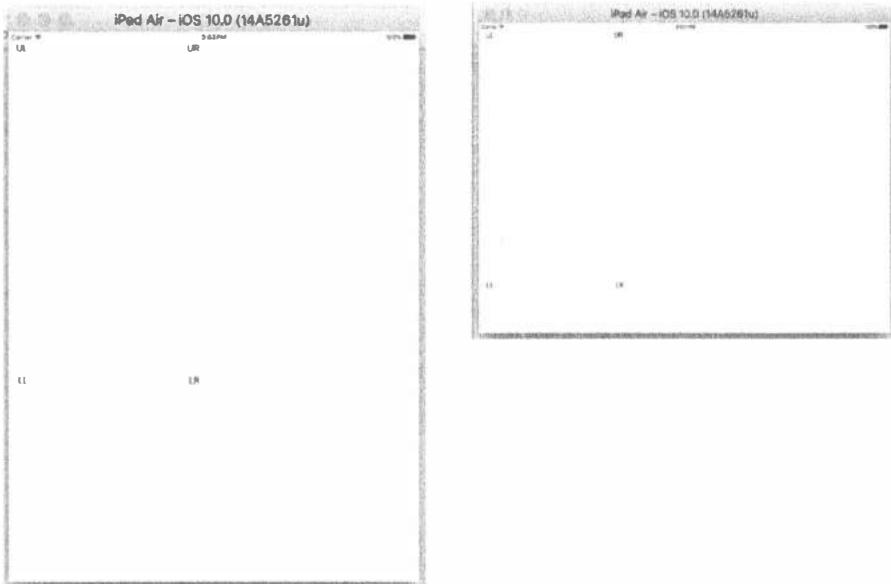


Рис. 5.6. Изменение ориентации без добавления ограничений

Как видим, все не так уж хорошо. Верхняя левая метка после вращения находится на правильной позиции, а остальные находятся на неправильных позициях или вообще не видны. Дело в том, что каждый объект интерфейса соблюдает фиксированное расстояние от левого верхнего угла представления в раскладовке. На самом деле мы хотели бы, чтобы каждая метка после вращения была прикреплена к ближайшему углу. Метки на правой стороне должны быть сдвинуты вправо, чтобы учесть новую ширину представления, а кнопки в нижней части должны быть подтянуты вверх, чтобы соответствовать новой высоте. К счастью, в программе Interface Builder это можно сделать с помощью установки ограничений.

Как указывалось ранее, программа Interface Builder может автоматически создать набор ограничений. Для этого она использует определенные правила. Для того чтобы применить их, сначала выберите все четыре метки. Для этого щелкните на одной из меток, а затем, удерживая клавишу <Shift> или <⌘>, щелкните на всех остальных метках по очереди. После этого выполните команду **Editor⇒Resolve Auto Layout Issues⇒Add Missing Constraints** (существуют два меню с такой командой — вы можете использовать любое из них). Затем щелкните на кнопке **Run**, чтобы запустить приложение на симуляторе и проверить его работу.

ЗАМЕЧАНИЕ. Существует еще один простой способ выбрать все метки — нажать клавишу <Shift> и щелкнуть на именах меток в окне Document Outline, как показано на рис. 5.7.

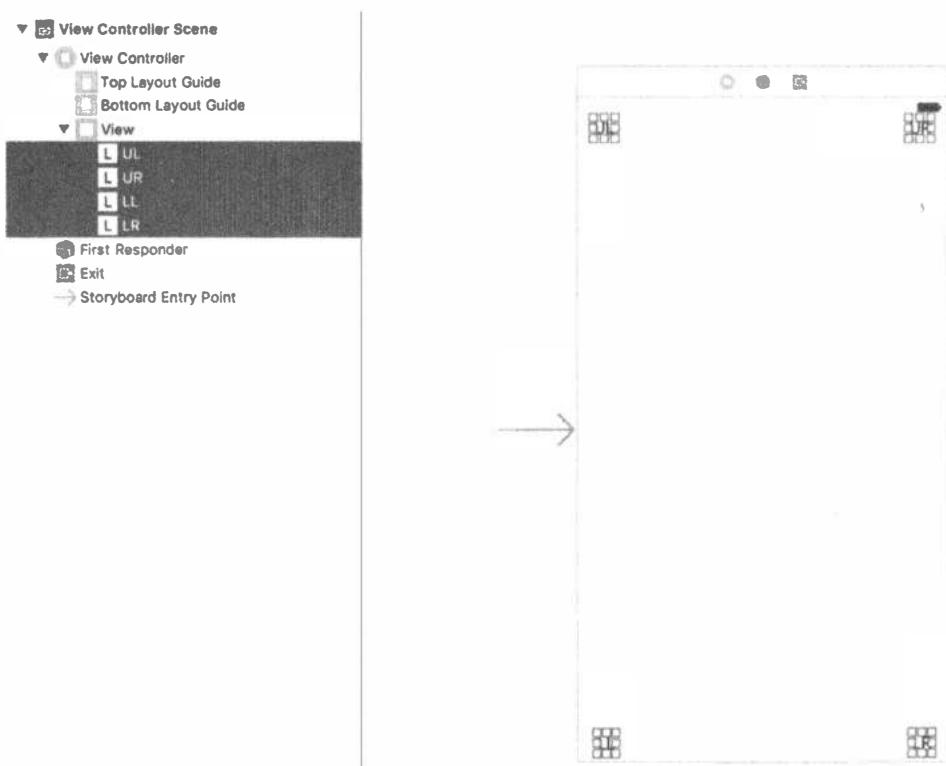


Рис. 5.7. Использование окна Document Outline (левая часть канвы Storyboard) иногда упрощает выделение и работу с несколькими объектами пользовательского интерфейса одновременно

Для того чтобы эффективно использовать ограничения, необходимо хорошо понимать, как они работают. Вернитесь в программу Xcode и выберите левую верхнюю метку, щелкнув на ней. Вы увидите пару сплошных голубых линий, присоединенных к этой метке. Одна из этих линий ведет к левому краю, другая — к верхнему. Эти голубые линии отличаются от голубых пунктирных линий разметки, которые можно увидеть при перетаскивании объектов по экрану (рис. 5.8).

Каждая из этих голубых линий представляет собой ограничение. Если нажать клавиши **<Option+⌘+5>**, то откроется **инспектор размеров** (size inspector), и вы увидите, что он содержит список ограничений. На рис. 5.9 показан типичный набор ограничений, но ограничения, создаваемые программой Xcode, зависят от конкретного места, в котором находятся метки, поэтому они могут отличаться.

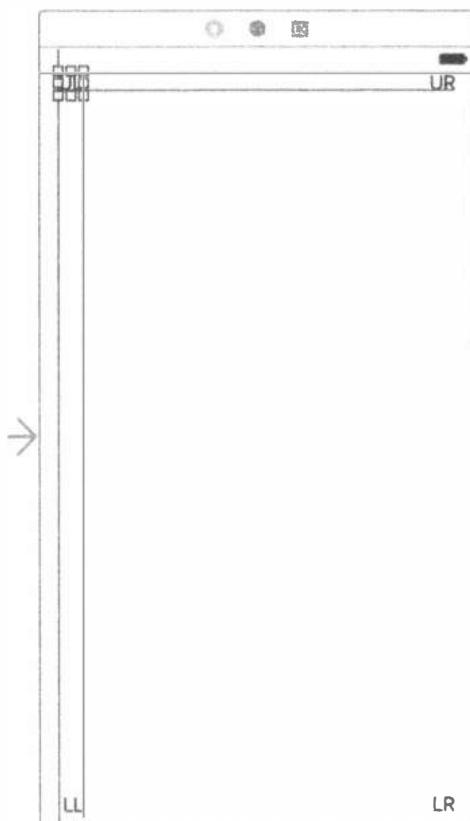


Рис. 5.8. Сплошные голубые линии, демонстрирующие ограничения, заданные для выбранного объекта

В данном случае позиция метки по отношению к родительскому представлению задается двумя ограничениями: одно из них контролирует **начальный пробел** (leading space), который обычно означает расстояние до правого края метки, а другое управляет **верхним отступом** (top space), т.е. пространством над меткой. Эти ограничения вынуждают кнопку занять равноудаленное положение по отношению к верхнему и левому краям родительского представления при изменении его размеров. Остальные два ограничения относятся к двум остальным меткам и выравнивают их по отношению к остальным. Проверьте все метки, чтобы убедиться, что ограничения удерживают метки в углах их родительского представления.

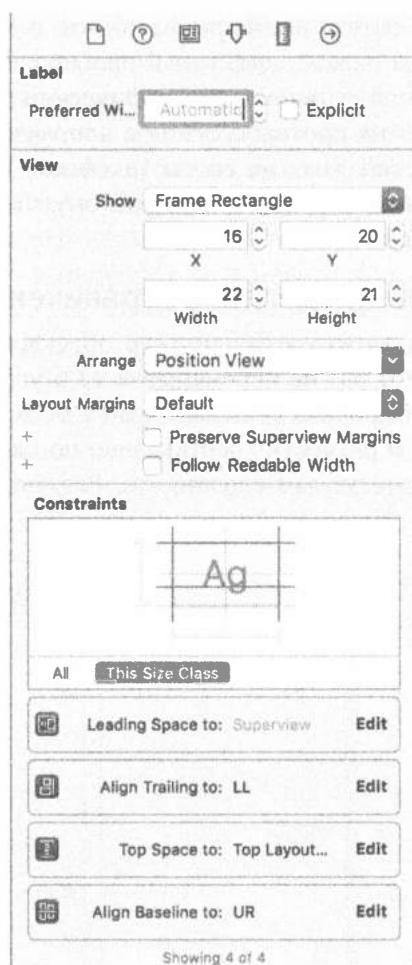


Рис. 5.9. Четыре ограничения, генерируемые программой Xcode, для фиксации метки на родительском представлении

Обратите внимание на то, что в языках, в которых текст пишется и читается справа налево, начальный пробел расположен справа, поэтому данное ограничение может испортить графический пользовательский интерфейс при переключении на противоположное направление, например, если пользователь выберет арабский язык на своем телефоне. Пока будем считать, что начальный пробел означает “пробел слева”, потому что именно это ограничение устанавливается автоматически.

Переопределение ограничений, заданных по умолчанию

Захватите в библиотеке объектов новую метку и перетащите ее на макет. На этот раз не перемещайте ее в угол, а приблизьте к левому краю представления, выровняв ее левый край с остальными метками, расположенными у левого края, и разместив вертикально по центру представления. Голубые линии разметки помогут вам сделать это. Результат показан на рис. 5.10.

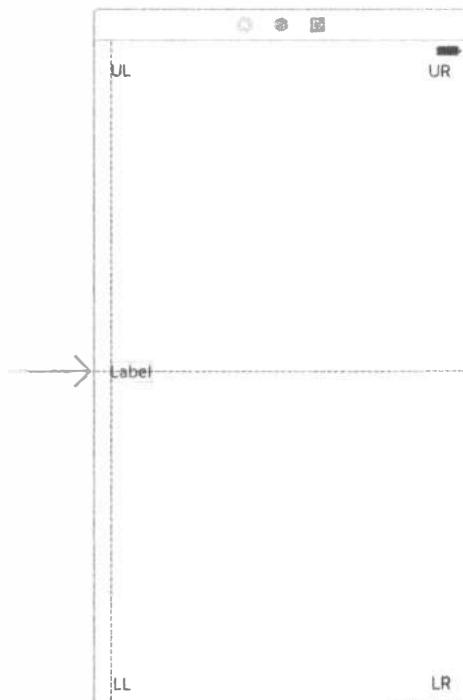


Рис. 5.10. Размещение метки Left

Добавим новое ограничение, заставляющее кнопку сохранять центральное положение по вертикали. Выберите метку и щелкните на пиктограмме Align, расположенной под раскладкой. Выберите во всплывающем меню команду Vertical Center in Container, а затем щелкните на кнопке Add 1 Constraint. Откройте инспектор размеров, нажав клавиши `<Option+⌘+5>`, и вы увидите, что теперь метка имеет ограничение, фиксирующее ее в центре по вертикали родительского

представления. Кроме того, метке нужно еще одно ограничение, фиксирующее ее по горизонтали. Выберите кнопку и выполните команду Editor⇒Resolve Auto Layout Issues⇒Add Missing Constraints из раздела All Views во всплывающем меню. Нажмите клавиши <⌘+R> и снова запустите приложение, выполните несколько поворотов и убедитесь, что кнопки занимают ожидаемые позиции.

Теперь заполним наше кольцо меток, добавив новую метку в правую часть представления, выровняв по правому краю и вертикально по метке Left. Присвоим новой кнопке имя Right. Выбрав кнопку, выровняйте ее по вертикали относительно остальных меток с помощью мыши. Мы хотим использовать автоматические ограничения, созданные программой Xcode, поэтому выполните команду Editor⇒Resolve Auto Layout Issues⇒Add Missing Constraints.

Еще раз соберите и запустите приложение, выполните несколько поворотов, и увидите, что все кнопки находятся на своих местах по отношению одна к другой (рис. 5.11). Если вы повернете экран обратно, они должны вернуться в исходное положение. Этот метод прекрасно работает во многих приложениях.



Рис. 5.11. Кнопки в новых позициях после поворота

Кнопки, занимающие всю ширину представления

Создадим несколько ограничений, гарантирующих, что кнопки будут иметь одинаковую ширину и будут растянуты на всю ширину представления даже после поворота устройства (рис. 5.12).

Нам необходимо визуально определить, достигли ли мы желаемого результата и находится ли каждая метка точно в центре своей половины экрана. Для того чтобы проверить это, временно изменим цвет фона метки. Выберите в раскаровке метки UL и UR, откройте инспектор атрибутов и перейдите в раздел View. Выберите какой-нибудь яркий цвет с помощью элемента управления Background. В результате контур каждой метки будет залит этим цветом.

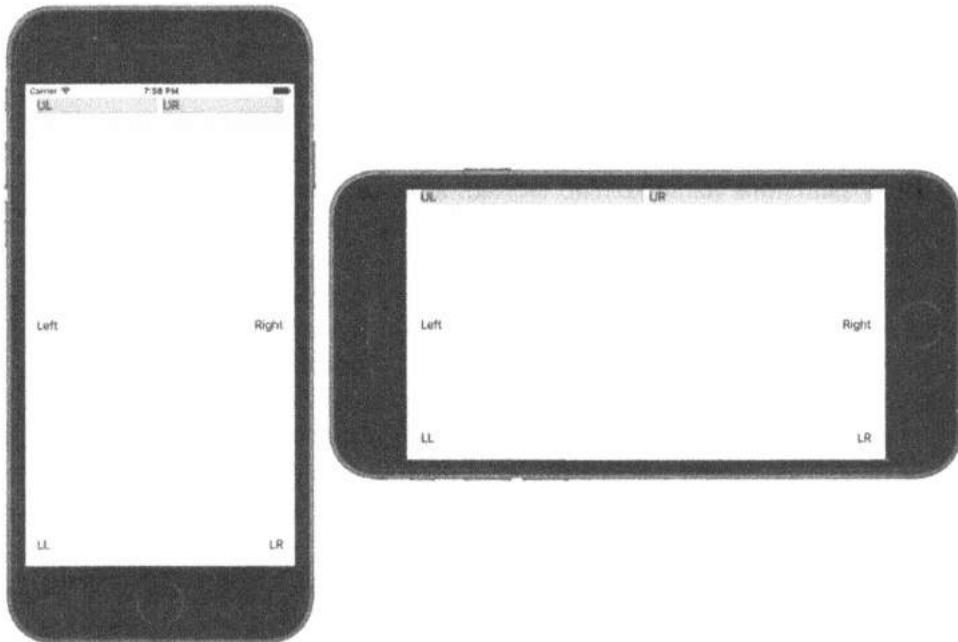


Рис. 5.12. Верхние кнопки растянуты на всю ширину экрана как в книжной, так и в альбомной ориентации

Перетащите элемент управления, изменяющий размеры метки UL, с ее правого края, приблизив его к центральной точке этого представления по горизонтали. По причинам, которые будут разъяснены позднее, особая точность здесь не нужна. Сделав это, измените размеры метки UR, перетаскивая ее левый край влево до тех пор, пока не появится перпендикулярная линия разметки, которая обозначает рекомендованное расстояние от метки до левого края родительского представления. Теперь мы добавим ограничение, благодаря которому эти метки будут заполнять всю ширину своего родительского представления. Нажмите клавишу <Control>, щелкните на метке UL и перетащите указатель на метку UR, а затем отпустите кнопку мыши. Во всплывающем меню выполните команду Horizontal Spacing и нажмите клавишу <Return>. Это ограничение, соединяющее левый край метки UR с правым краем метки UL. Соберите и запустите приложение, чтобы увидеть изменения. Поверните устройство, и, возможно, вы увидите нечто, похожее на рис. 5.13. Более длинная метка может оказаться слева или справа в зависимости от конфигурации устройства.

Это почти то, что мы хотели, но не совсем. Что же мы сделали неправильно? Мы определили ограничения, регламентирующие положение меток относительно их родительского представления, и задали расстояние между метками, но ничего не сказали о размерах этих меток. Это позволяет системе макетирования произвольно задавать размеры меток. Для того чтобы исправить эту ситуацию, добавим новое ограничение.

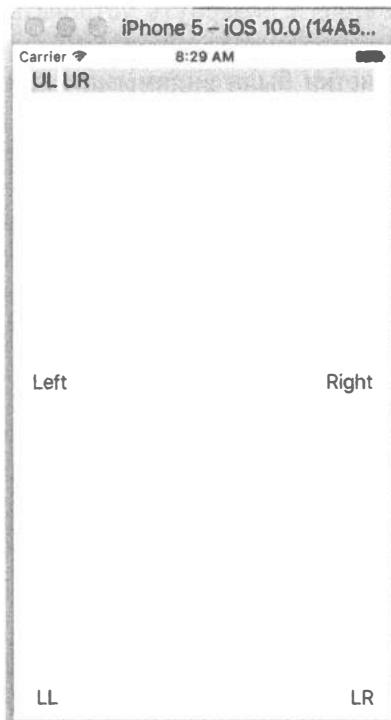


Рис. 5.13. Метки растянуты по всей ширине экрана не равномерно

Щелкните на метке UL, затем нажмите клавишу <Shift> и щелкните на метке UR. Выбрав обе метки, вы можете создать ограничение, которое будет влиять на них обеих. Щелкните на пиктограмме Pin, расположенной под раскладовкой, и установите флажок Equal Widths в появившемся всплывающем меню (как мы это делали в главе 3). Затем щелкните на кнопке Add 1 Constraint. Вы увидите новое ограничение (рис. 5.14). Под метками появятся две оранжевые линии. Это означает, что позиции и размеры меток в раскладовке не соответствуют реальным позициям и размерам меток во время выполнения приложения. Для того чтобы исправить этот недостаток, выполните команду Editor⇒Resolve Auto Layout Issues⇒Update Frames в меню Xcode. Ограничения должны стать голубыми, а размеры меток изменятся автоматически и будут иметь одинаковую ширину.

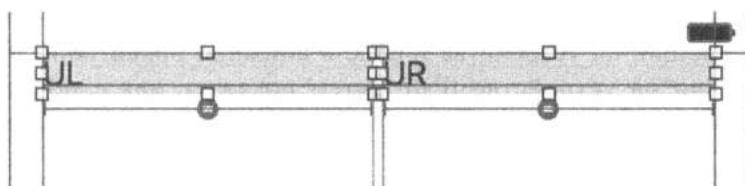


Рис. 5.14. Теперь верхние метки имеют одинаковую ширину

Если запустить приложение и повернуть устройство, то метки будут занимать всю ширину экрана, как показано на рис. 5.12.

В данном примере все метки были видимыми и допускали несколько вариантов ориентации, но большая часть экрана осталась неиспользованной. Возможно, было бы лучше, если бы остальные два ряда меток тоже заполняли всю ширину экрана или метки изменяли высоту, чтобы заполнить пустое пространство. Поэкспериментируйте с ограничениями, накладываемыми на эти шесть меток, или добавьте новые метки. Помимо приемов, описанных выше, вы можете найти действия, создающие ограничения во всплывающих меню, которые появляются, когда вы щелкаете на пиктограммах Pin и Align, расположенных под раскладкой. Если какие-то ограничения вам больше не нужны, удалите их, выбрав их и нажав клавишу <Delete>, или попытайтесь настроить их в инспекторе атрибутов. Попробуйте разные варианты, пока не почувствуете, что хорошо освоили работу с ограничениями. Мы будем использовать их на протяжении всей книги, но если вы хотите узнать больше, введите термин Auto Layout в окне документации программы Xcode.

Создание адаптивных макетов

Макет, созданный в данном простом примере, хорошо работает в книжной и в альбомной ориентации как на устройствах iPhone, так и на устройствах iPad, независимо от различий в размерах экранов. Как уже указывалось, обработка вращения устройства и создание интерфейса для устройств с разными размерами экрана по существу — одна и та же задача. В конце концов, с точки зрения приложения при вращении устройства размер экрана изменяется. В простейших случаях эти задачи решаются с помощью ограничений Auto Layout. Однако это не всегда возможно. Некоторые макеты хорошо работают в книжной ориентации и плохо в альбомной. Аналогично некоторые приложения подходят для устройства iPhone, но не для iPad. В таких случаях приходится создавать для каждого варианта отдельный макет. До появления версии iOS 8 это означало реализацию всего макета в коде с помощью многих раскладовок или сочетания этих инструментов. К счастью, компания Apple сделала возможной разработку адаптивных приложений, которые одинаково хорошо работают в обоих видах ориентации на основе всего одной раскладовки. Посмотрим, как работает этот механизм.

Создание приложения Restructure

Для демонстрации возможностей создадим пользовательский интерфейс, хорошо работающий на устройстве iPhone в книжной ориентации и плохо — в альбомной или на устройстве iPad. Затем покажем, как с помощью программы Interface Builder адаптировать макет так, чтобы он работал в любой ориентации.

Создадим новый проект Single View и назовем его Restructure. Мы собираемся разработать графический пользовательский интерфейс, состоящий

из большой области содержимого и небольшого набора кнопок, выполняющих фиктивные действия. Разметим кнопки в нижней части экрана, предоставив освободительное место для области содержимого (рис. 5.15).

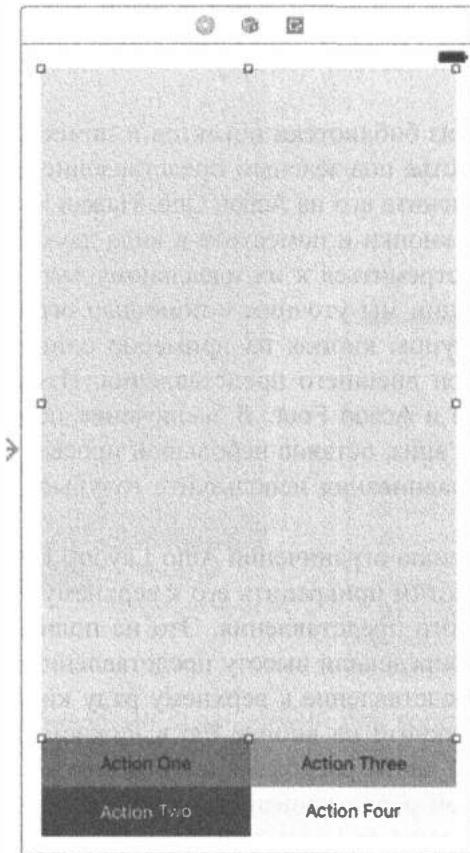


Рис. 5.15. Исходный графический пользовательский интерфейс в книжной ориентации на устройстве iPhone

ЗАМЕЧАНИЕ. Возможно, вы заметили, что некоторые устройства Apple, которые мы используем в книге для иллюстрации, имеют разные конфигурации. Одни иллюстрации (например 5.15) ближе к реальности, а другие (например 5.11) кажутся более схематичными. Эта разница не должна смущать читателей, поскольку она довольно часто встречается в технической документации, включая документацию компании Apple.

Выберите файл Main.storyboard для редактирования графического пользовательского интерфейса. Поскольку представление содержимого для нас интереса не представляет, изобразим его в виде большого цветного прямоугольника. Перетащите один объект класса UIView из библиотеки объектов в свое внешнее

представление. Выделите представление и измените его размер, оставив небольшие поля сверху, справа и слева, как показано на рис. 5.15. Теперь перейдите в окно инспектора атрибутов и с помощью раскрывающегося списка **Background** выберите другой цвет фона. Вы можете выбрать любой цвет, кроме белого, чтобы представление выделялось на белом фоне. В архивной раскладовке представление окрашено зеленым цветом, поэтому с этого момента мы будем называть его зеленым.

Перетащите кнопку из библиотеки объектов и поместите ее в левой нижней части пустого пространства под зеленым представлением. Дважды щелкните на тексте этой метки и измените его на **Action One**. Нажав клавишу **<Option>**, перетащите три копии этой кнопки и поместите в виде двух столбцов, как показано на рис. 5.15. Не стоит стремиться к их идеальному выравниванию, потому что их окончательные позиции мы уточним с помощью ограничений, но попытайтесь разместить две группы кнопок на примерно одинаковых расстояниях от соответствующих сторон внешнего представления. Измените их заголовки на **Action Two**, **Action Three** и **Action Four**. В заключение перетащите нижний край зеленого представления вниз, оставив небольшой просвет между ним и верхним рядом кнопок. Для выравнивания используйте голубые линии разметки, показанные на рис. 5.15.

Перейдем к определению ограничений **Auto Layout**. Начнем с выбора зеленого представления. Мы хотим прикрепить его к верхнему краю, а также к левому и правому краям главного представления. Это не полный набор ограничений, потому что мы еще не определили высоту представления; мы исправим этот недостаток, прикрепив представление к верхнему ряду кнопок, зафиксировав при этом сами кнопки. Щелкните на кнопке **Pin** в нижнем правом углу редактора раскладовок. В верхней части раскрывающегося списка вы увидите знакомую группу из четырех полей редактирования, окруженную небольшим квадратом. Установите флажок **Constrain to Margins**. Щелкните на красных пунктирных линиях, расположенных сверху, слева и справа от маленького квадрата, чтобы соединить представление с верхним, левым и правым краями родительского представления (рис. 5.16). Щелкните на кнопке **Add 3 Constraints**.

Теперь зададим одинаковую высоту наших кнопок, начиная с кнопки **Action One**, как показано на рис. 5.17. Мы выбрали высоту, равную 43 точкам, просто потому, что это значение уже было установлено, когда мы приступили к созданию кнопки. Для нашего примера подойдет любое значение, не слишком сильно отличающееся от 43. Задав высоту первой кнопки, повторите эти операции для остальных трех кнопок.

Если все операции выполнены правильно, все ограничения должны быть видны в окне **Document Outline**, как показано на рис. 5.18, где мы видим ограничения, задающие высоту всех четырех кнопок, а также трех сторон зеленого представления.

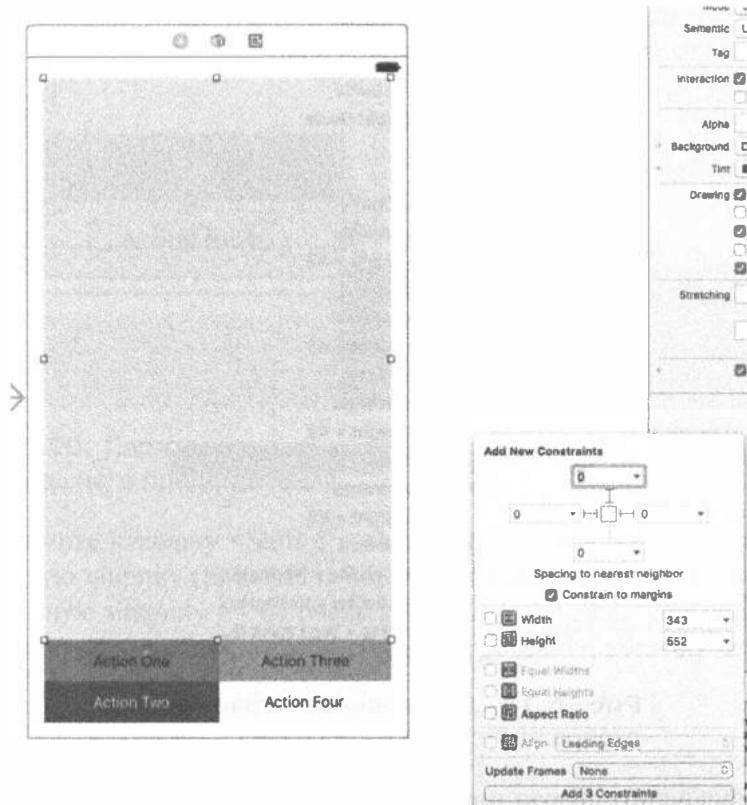


Рис. 5.16. Добавление ограничений для корректировки верхней, левой и правой сторон зеленого представления

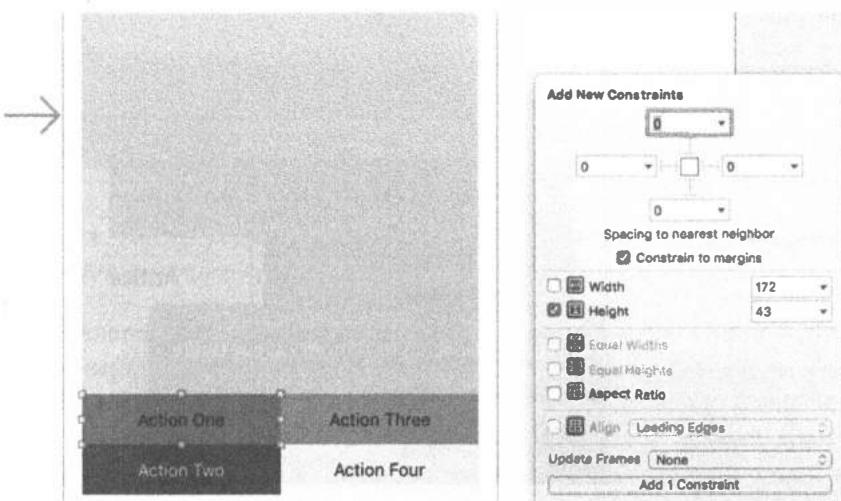


Рис. 5.17. Настройка высоты одной из кнопок

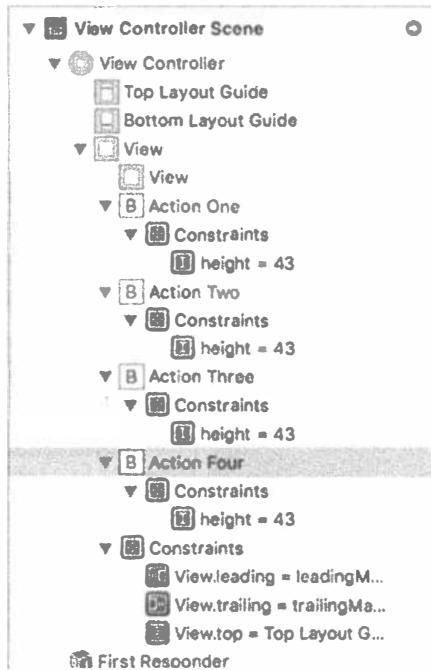


Рис. 5.18. Добавленные ограничения отображаются в окне Document Outline

Теперь прикрепите левую нижнюю (Action Two) и правую нижнюю (Action Four) кнопки к нижним углам, нажав клавишу <Control> и перетащив каждую из них в левый и правый угол соответственно. Для перемещения кнопки Action Two нажмите клавишу <Shift> и выберите два соответствующих пункта, как показано на рис. 5.19.

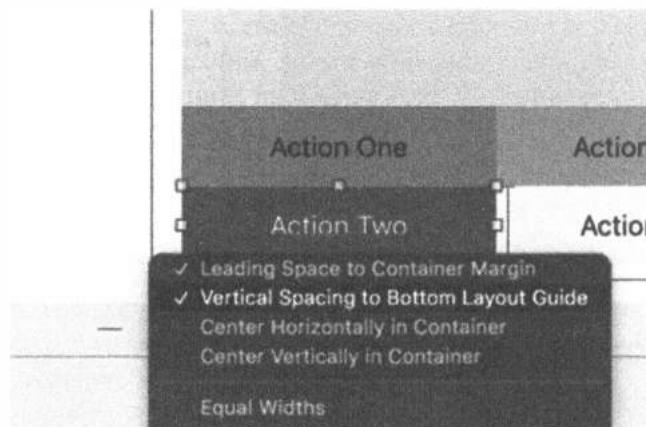


Рис. 5.19. Перетаскивание кнопки Action Two и ее прикрепление к левому нижнему углу контейнера при нажатой клавише <Control>

Выполните аналогичную операцию, перетащив клавишу Action Four в правый нижний угол при нажатой клавише <Control> и задав для нее ограничения, показанные на рис. 5.20.

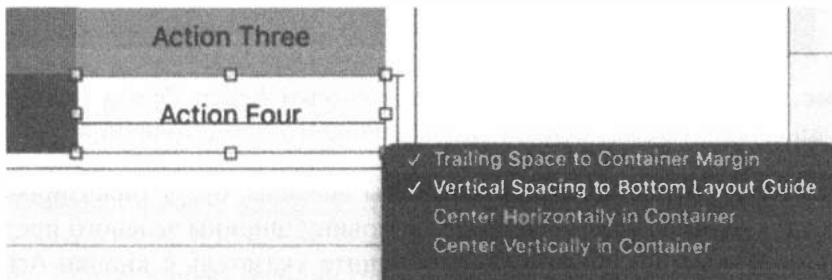


Рис. 5.20. Перетаскивание кнопки Action Four в правый нижний угол внешнего представления

Затем нажмите клавишу <Shift>, выберите все четыре кнопки и задайте для них одинаковую ширину (рис. 5.21). Обратите внимание на то, что сейчас кнопки имеют разную ширину — от очень маленькой до очень большой. Для того чтобы исправить этот недостаток, необходимо добавить дополнительные ограничения.

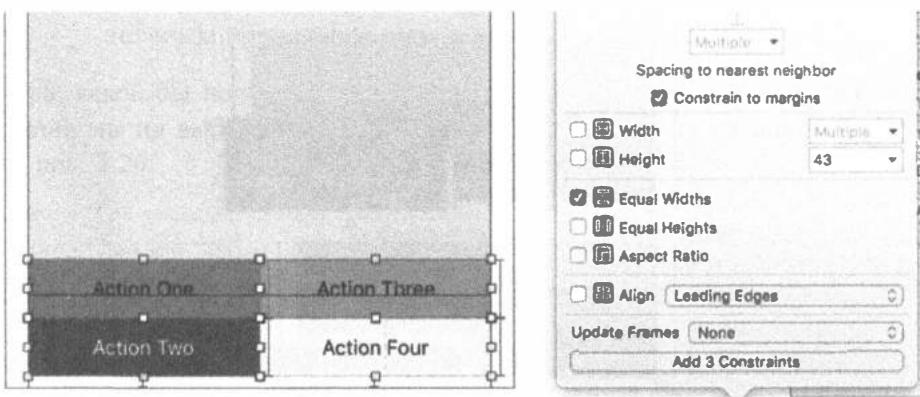


Рис. 5.21. Настройка одинаковой ширины всех кнопок, хотя конкретное значение ширины в раскладовке еще не задано

Нажмите клавишу <Control> и перетащите кнопки Action One и Action Three, образующие верхний ряд кнопок, влево и вправо соответственно, чтобы задать ограничения Leading Space to Container Margin и Trailing Space to Container Margin (рис. 5.22). В результате левый край кнопки Action One и правый край кнопки Action Three будут привязаны к краям представления с помощью одной из якорных точек.

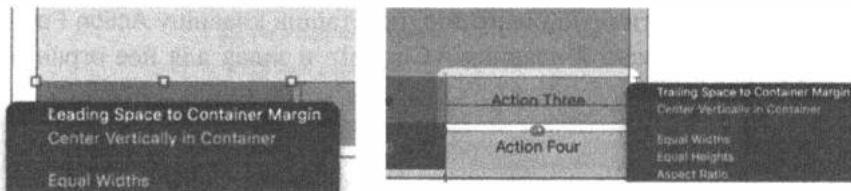


Рис. 5.22. Настройка левого края кнопки Action One и правого края кнопки Action Three по краям внешнего представления

Последние два ограничения, которые мы добавим, будут гарантировать, что кнопки будут одинаковыми и занимать половину ширины зеленого представления. Нажмите клавишу <Control>, перетащите указатель с кнопки Action One на кнопку Action Three и задайте ограничение Horizontal Spacing. Сделайте то же самое для кнопок Action Two и Action Four, как показано на рис. 5.23. Для этого переместите якорь каждого края соответствующей кнопки на левый и правый край контейнера (см. рис. 5.22). Как показано на рис. 5.21, все кнопки имеют одинаковую ширину. Задав горизонтальный отступ так, чтобы все кнопки были выровнены одна относительно другой, мы автоматически выровняем ряд кнопок по центру представления. Поскольку зеленое представление также прикреплено к левому и правому краям, кнопки окажутся посередине зеленого представления.

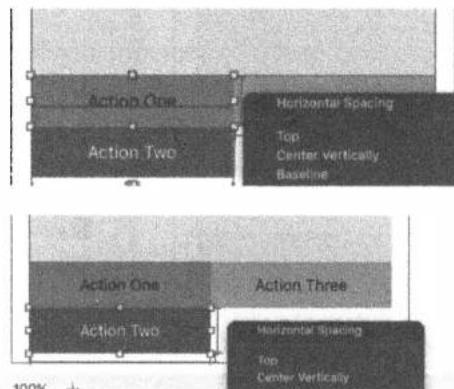


Рис. 5.23. Центровка каждого ряда кнопок относительно представления; по умолчанию при этом ширина каждой кнопки будет равна половине ширины представления

Еще немного, и мы будем готовы протестировать наше приложение на разных устройствах. Нажмите клавишу <Control> и перетащите указатель с клавиши Action Three на клавишу Action Four, чтобы задать ограничение Vertical Spacing, как показано на рис. 5.24. Сделайте то же самое для кнопок Action One и Action Two.

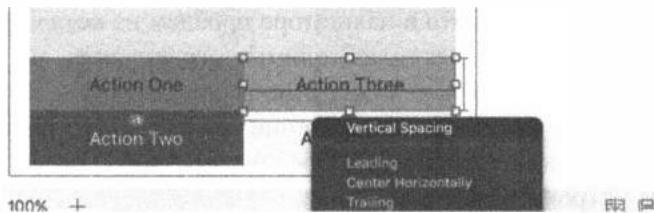


Рис. 5.24. Установка вертикального отступа между кнопками, образующими один столбец

В заключение нажмите клавишу <Control> и перетащите указатель от зеленого представления к кнопке Action One, чтобы зеленое представление и верхний ряд кнопок имели общую границу (рис. 5.25).

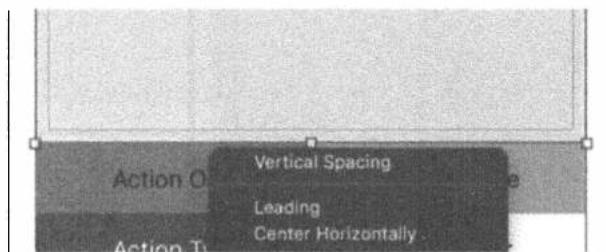


Рис. 5.25. Установка вертикального отступа между зеленым представлением и верхним рядом кнопок

Теперь, какой бы полный экран iPhone или iPad в книжной или альбомной ориентации мы ни выбрали, кнопки должны сохранять свои позиции, как показано на рис. 5.26.

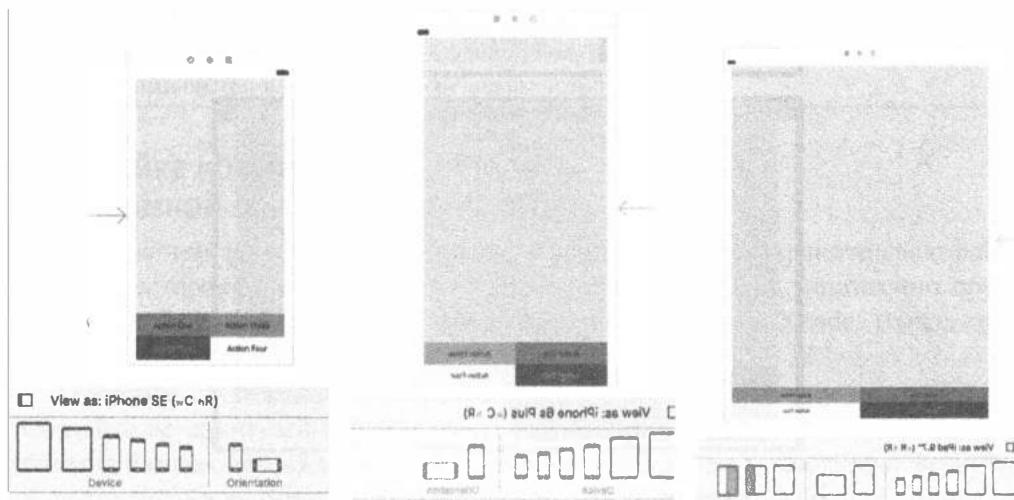


Рис. 5.26. Выбор разных устройств в книжной ориентации; кнопки должны сохранять свои позиции

Обратите внимание на то, что в навигаторе проблем не осталось никаких сообщений. Систематически добавляя необходимые ограничения, мы смогли создать макет с минимальными усилиями. Однако не всегда это будет так просто.

Осталось собрать и выполнить приложение на симуляторе iPhone. На рис. 5.27 показаны две ориентации устройства iPhone 6s, а на рис. 5.28 — две ориентации полного экрана устройства iPad Air.

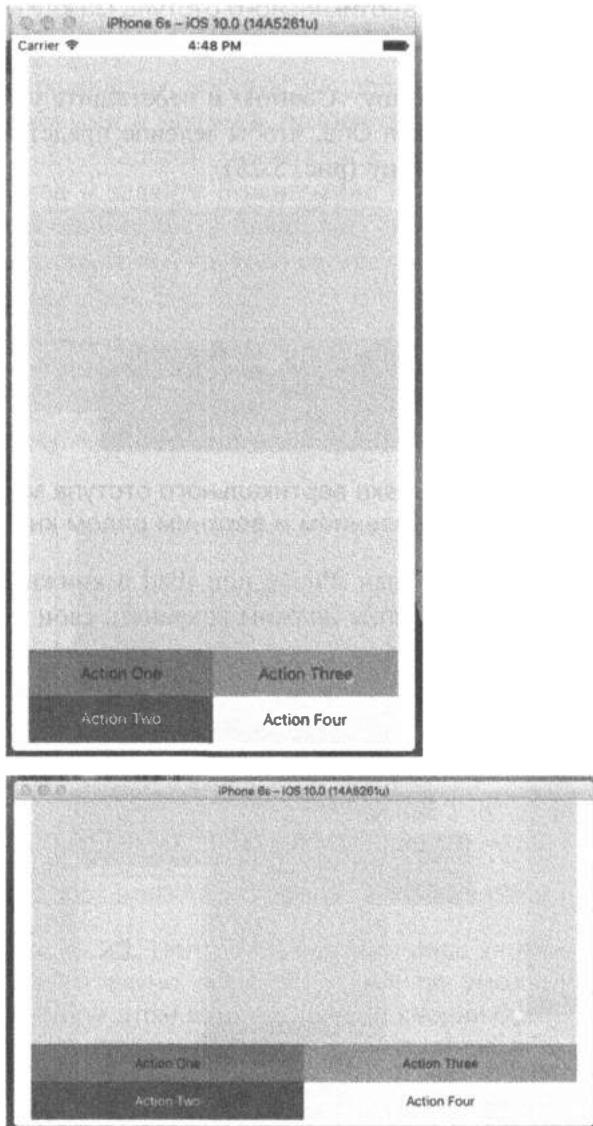


Рис. 5.27. Две ориентации устройства iPhone 6s

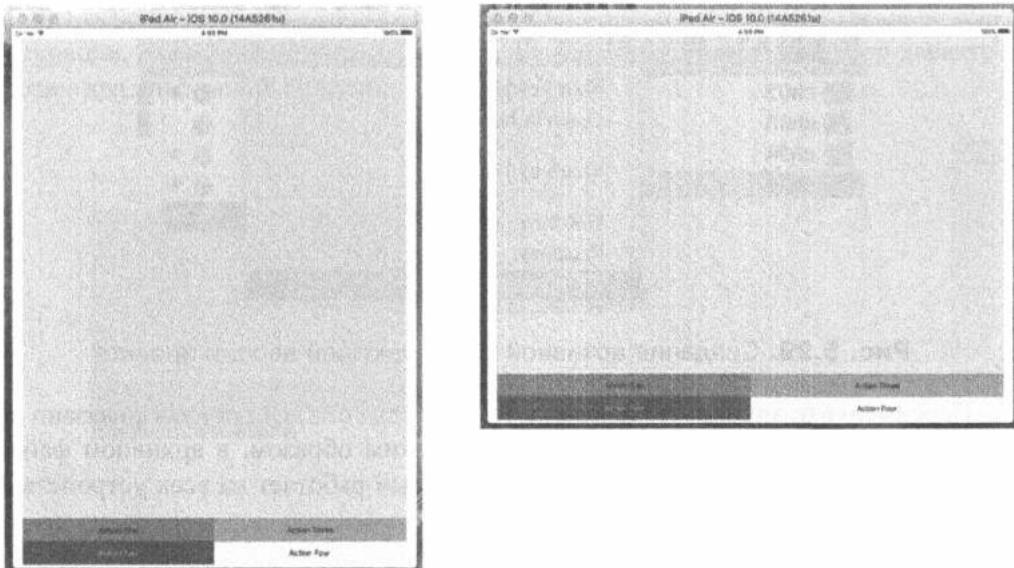


Рис. 5.28. Две ориентации устройства iPad Air

Приложение выглядит неплохо, но мы собираемся сделать еще кое-что. На экране устройства iPhone в альбомной ориентации с конфигурацией $wC\ hC$ кнопки должны выровняться в столбик у правого края, а на экране устройства iPad в любой ориентации с конфигурацией $wR\ hR$ они должны выровняться в ряд у нижнего края.

ЗАМЕЧАНИЕ. $w-$ $h-$ обозначают ширину и высоту в рассматриваемой конфигурации. В механизме Auto Layout буква С используется для обозначения компактной конфигурации, а R — обычной. Таким образом, возникают варианты $wC\ hC$, $wC\ hR$, $wR\ hC$ и $wR\ hR$. Посмотрев на панель Device Configuration, вы можете увидеть, как эти обозначения применяются к реальным физическим устройствам.

Настройка конфигурации iPhone в альбомной ориентации ($wC\ hC$)

Прежде чем задать альбомную конфигурацию $wC\ hC$, сохраните свою работу и закройте проект Xcode. Для того чтобы закрыть проект, достаточно просто щелкнуть на красном кружке в левом верхнем углу окна Xcode. Выходить из среды Xcode совсем необязательно.

Откройте окно Finder на компьютере Mac и найдите папку *Restructure*. Создайте ее архивный вариант, как показано на рис. 5.29. В результате будет создана базовая версия проекта, к которой впоследствии можно будет вернуться в любое время, если что-то пойдет не так, как ожидалось.

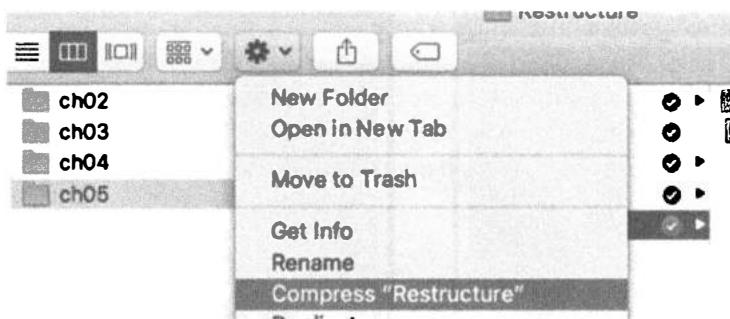


Рис. 5.29. Создание архивной копии текущей версии проекта

Переименуйте архивный zip-файл в RestructureBaseline, как показано на рис. 5.30, чтобы он имел уникальное имя. Таким образом, в архивном файле RestructureBaseline упакован проект, который работает на всех устройствах и во всех видах ориентации одинаково.



Рис. 5.30. Переименование архивной копии текущей версии проекта

Сначала изменим проект для альбомной ориентации iPhone. Выберите целевое устройство iPhone 6s и альбомную ориентацию. В правой части панели Device Configuration щелкните на кнопке Vary for Traits и обратите внимание на то, что панель стала синей, как показано на рис. 5.31. Выберите во всплывающем окне ширину и высоту. Как показано на рис. 5.31, на экран выводится информация, которая относится только к устройству iPhone и только в альбомной ориентации, поскольку мы собираемся разработать пользовательский интерфейс именно для такой конфигурации.

Теперь щелкните на всех пяти элементах пользовательского интерфейса и нажмите клавишу <Delete>. Этого не нужно бояться, потому что мы сохранили базовую версию проекта, к которой всегда сможем вернуться. Если вы этого еще не сделали, то сделайте обязательно. После этого канва будет выглядеть так, как показано на рис. 5.32. Однако, взглянув на окно Document Outline, можно увидеть, что в ней все еще отображаются элементы пользовательского интерфейса

и даже ограничения. Это объясняется тем, что они существуют в базовой конфигурации, а мы будем создавать новую конфигурацию и новый набор характеристик для альбомной ориентации wC hC.

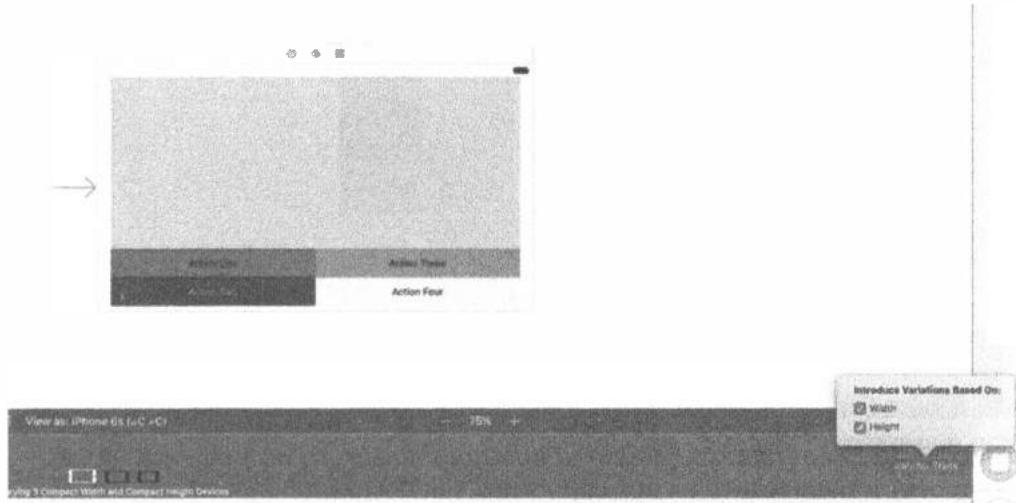


Рис. 5.31. Отправная точка для создания пользовательского интерфейса для устройства iPhone в альбомной ориентации

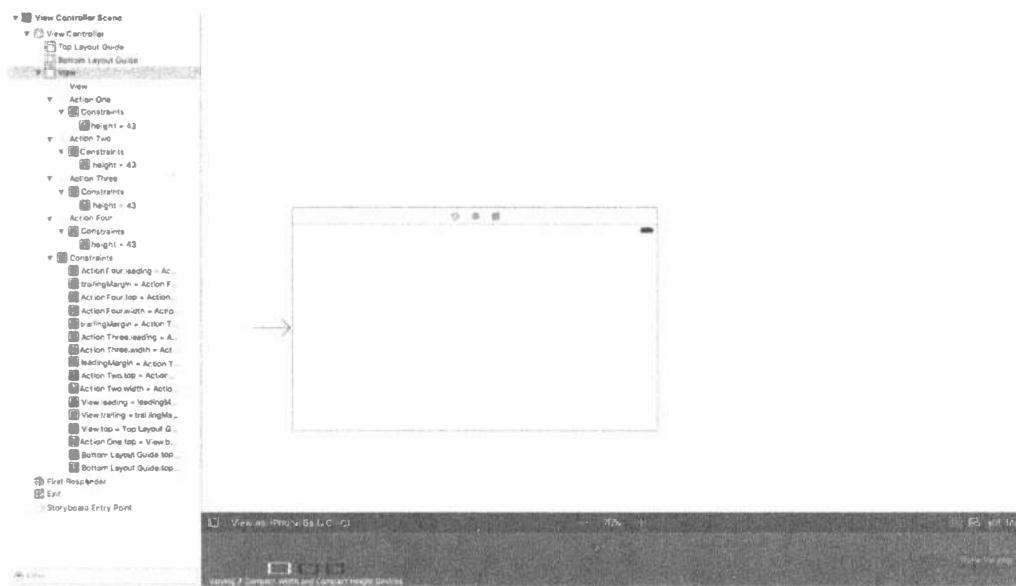


Рис. 5.32. Отправная точка для создания конфигурации устройства iPhone в альбомной ориентации. Обратите внимание на то, что в окне Document Outline по-прежнему отображаются все элементы пользовательского интерфейса и ограничения, соответствующие базовой конфигурации

Как и в базовой версии, перетащите на канву объект класса `UIView` и четыре кнопки, задав их цвет и надписи. Поместите их в позиции, указанные на рис. 5.33, но пока не устанавливайте ограничения.

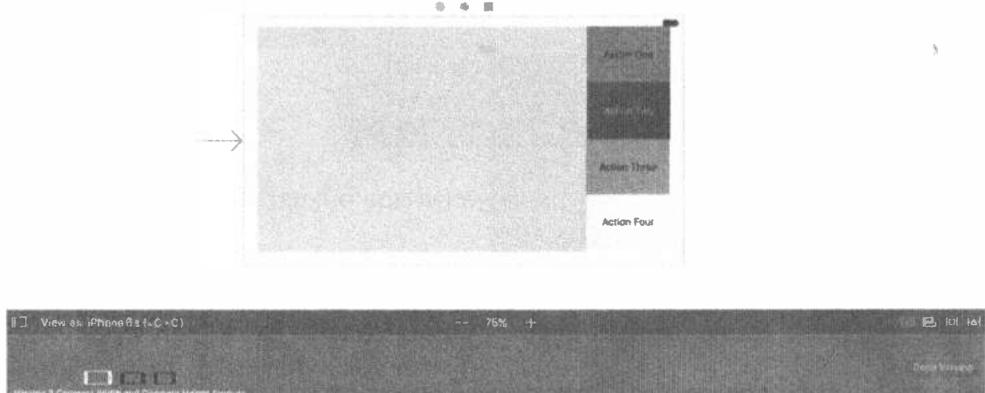


Рис. 5.33. Перетащите элементы пользовательского интерфейса на раскладовку и разместите их примерно так, как показано на рисунке

В этом разделе наши измерения должны быть немного более точными. Выберите зеленое представление. Используя инспектор размеров, задайте размер 500×340 точек, как показано на рис. 5.34. Ширина (500 точек) выбрана произвольно, а высота (340 точек) выбрана так, чтобы после деления на 4 получилось 85. Именно это значение мы выберем для высоты кнопок. Это не ограничение, а просто аспект внешнего вида раскладовки. На самом деле мы не задаем для зеленого представления никакого фиксированного размера, поскольку он может изменяться в зависимости от устройства (для Plus в большую сторону, а для SE — в меньшую).

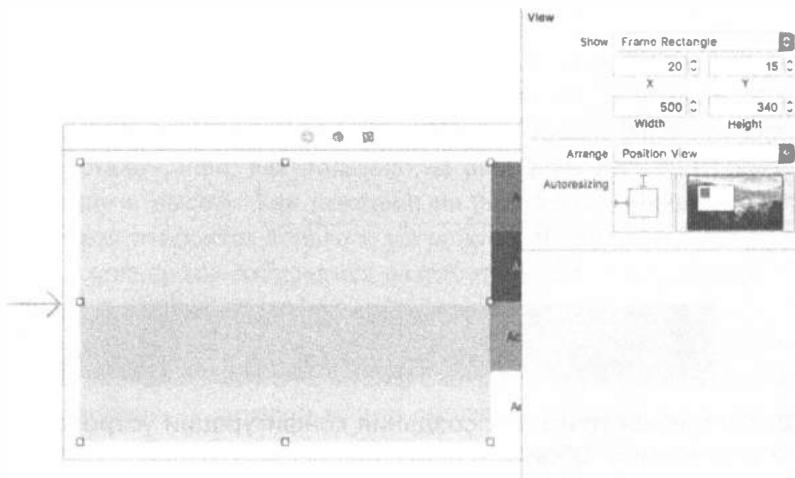


Рис. 5.34. Установка размеров зеленого представления

Выберите зеленое представление и прикрепите его к верхнему, левому и правому краям, как показано на рис. 5.35.

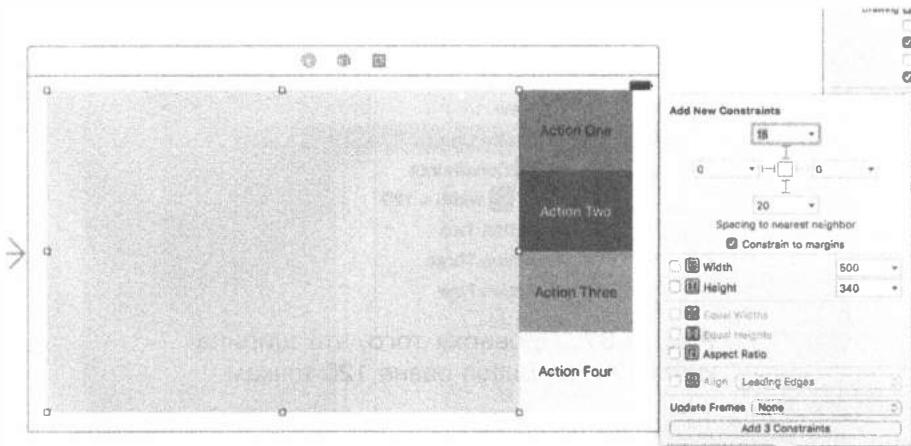


Рис. 5.35. Прикрепление зеленого представления к верхнему, левому и правому краям экрана в альбомной ориентации

Теперь мы исправим ширину кнопки *Action One*, но сделаем это немного не так, как раньше. Нажмите клавишу <Control>, когда указатель находится на кнопке *Action One*, и перетащите его из одной точки в другую, не выходя за пределы красной границы, а затем отпустите кнопку мыши. Вы увидите всплывающее меню (рис. 5.36), в котором необходимо выбрать команду *Width*. Задайте ширину кнопки *Action One* равной 120 точкам, как показано на рис. 5.37.

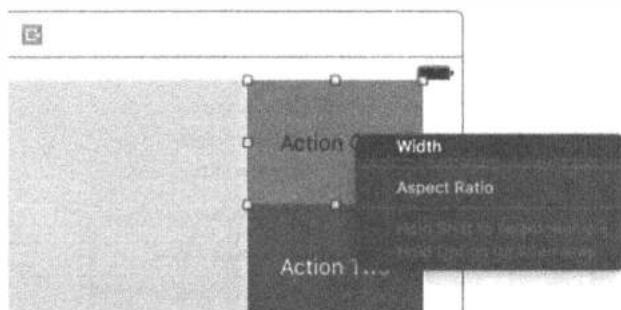


Рис. 5.36. Настройка ширины кнопки *Action Button*

Мы хотим, чтобы все четыре кнопки имели одинаковую ширину и высоту, причем ширина должна быть равной 120 точкам, а высота должна настраиваться динамически в зависимости от вертикальных размеров доступной области на экране iPhone в альбомной ориентации. Как и ранее, одинаковую высоту кнопок мы обеспечим, разместив их столбиком. Щелкните на всех четырех кнопках, удерживая клавишу <Shift>, щелкните на пиктограмме Pin, а затем задайте ограничения *Equal Width* и *Equal Heights* (рис. 5.38).

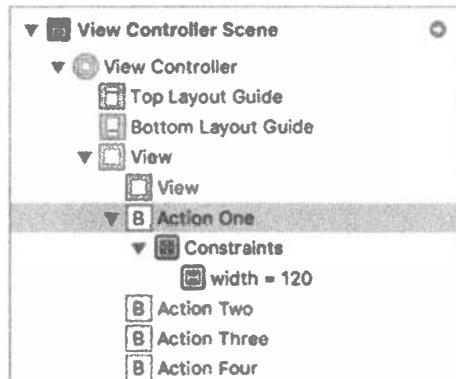


Рис. 5.37. Проверка того, что ширина кнопки Action Button равна 120 точкам



Рис. 5.38. Установка одинаковой высоты и ширины всех кнопок

ЗАМЕЧАНИЕ. Возможно, вы заметили, что, хотя для каждой из четырех кнопок установлены по два ограничения, на самом деле в проект добавлено шесть ограничений. Это объясняется тем, что свойства трех кнопок совпадают со свойствами четвертой.

Прикрепите кнопку Action One к верхнему и правому краям внешнего представления (рис. 5.39), а кнопку Action Four — к правому и нижнему краям (рис. 5.40).

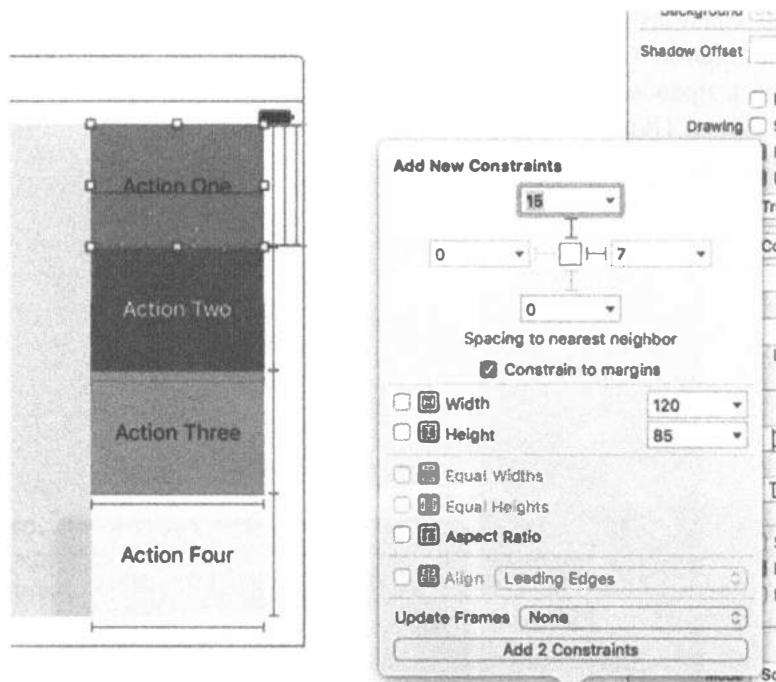


Рис. 5.39. Прикрепление кнопки Action One к верхнему и правому краям

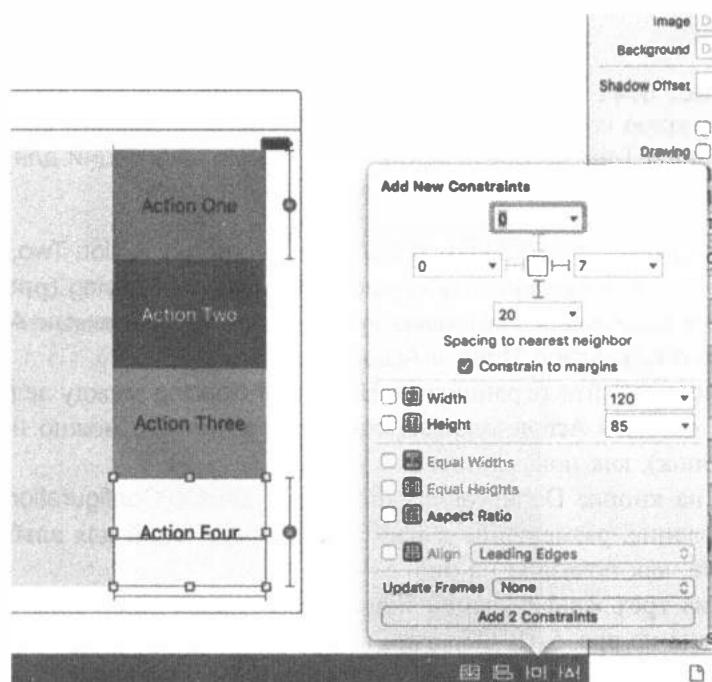


Рис. 5.40. Прикрепление кнопки Action Four к нижнему и правому краям

Нажмите клавишу <Control>, перетащите кнопку Action Two вправо и выберите ограничение Trailing Space to Container Margin так, чтобы кнопка была прикреплена к правому краю контейнера (рис. 5.41). Повторите эту операцию для кнопки Action Three.

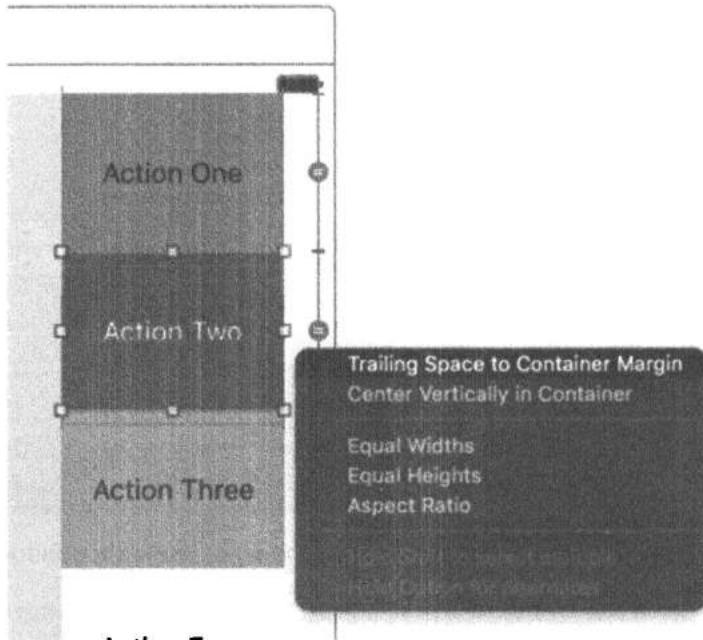


Рис. 5.41. Прикрепление кнопки Action Two к правому краю контейнера с помощью перетаскивания и команды Trailing Space to Container Margin. Операции для кнопки Action Three выполняются аналогично

Перетащите указатель с кнопки Action One на кнопку Action Two, нажав клавишу <Control>, чтобы установить ограничение Vertical Spacing (рис. 5.42). Повторите эту операцию для установки просвета между кнопками Action Two и Action Three, а также Action Three и Action Four.

В заключение задайте ограничение Horizontal Spacing между зеленым представлением и кнопкой Action One (впрочем, в этом случае можно использовать любую из кнопок), как показано на рис. 5.43.

Щелкните на кнопке Done Varying на панели Device Configuration, чтобы закончить добавление, размещение и настройку ограничений для альбомной ориентации iPhone, как показано на рис. 5.44.

В каждой из трех конфигураций iPhone wC hC кнопки должны размещаться так, как показано на рис. 5.45. Обратите внимание на то, что мы выбрали устройство iPhone 6/6s Plus, потому что оно имеет нашу базовую конфигурацию wC hC.

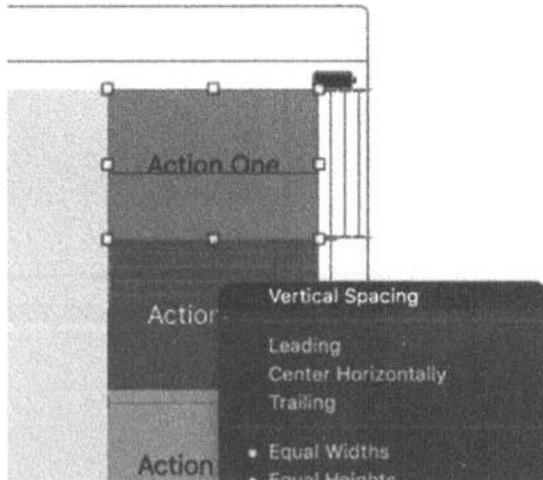


Рис. 5.42. Настройка ограничения Vertical Spacing между каждой парой кнопок в столбце. В результате высота каждой кнопки станет равной одной восьмой высоты контейнера

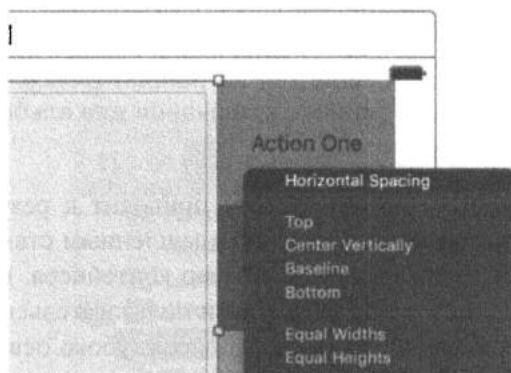


Рис. 5.43. Настройка ограничения Horizontal Spacing между зеленым представлением и рядом кнопок

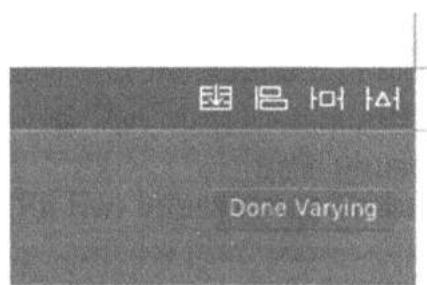


Рис. 5.44. Завершение настроек с помощью щелчка на кнопке Done Varying на панели Device Configuration

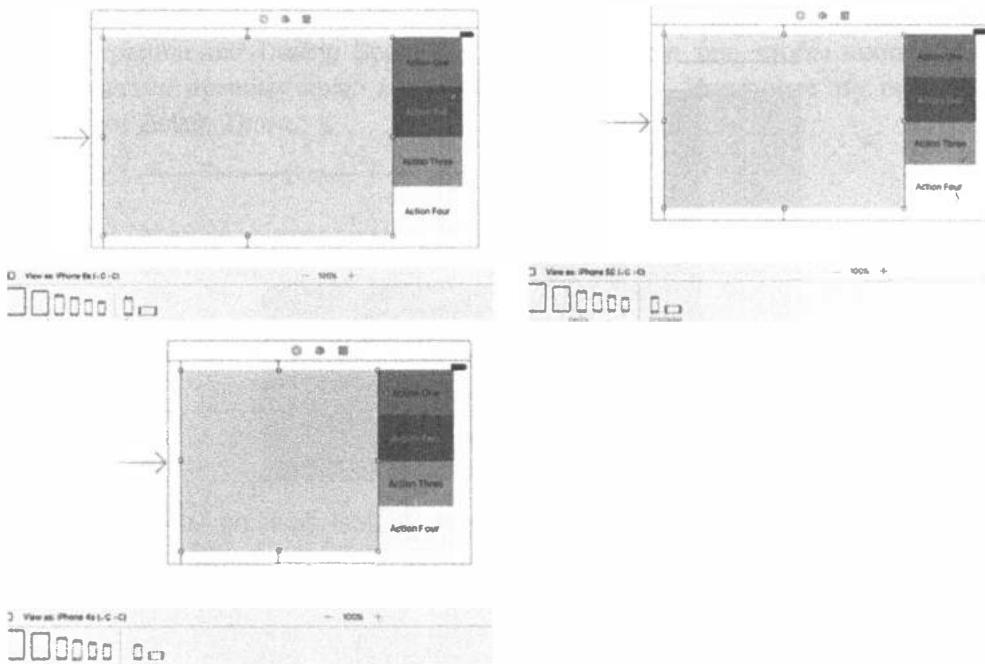


Рис. 5.45. Изменение типа устройства на панели Device Configuration показывает, что мы правильно настроили ограничения для альбомной ориентации устройств iPhone в конфигурации wC hC

Выполнение приложений на симуляторе приводит к результатам, показанным на рис. 5.46. Несмотря на то что по промышленным стандартам края представления следовало бы прижать ближе к краю контейнера, нашей целью было продемонстрировать манипуляции элементами пользовательского интерфейса на разных устройствах с разной ориентацией. Более глубоко освоив механизм Auto Layout, вы сможете точнее настраивать внешний вид своих приложений.

Прежде чем перейти к устройству iPad, нам осталось лишь сохранить текущую версию проекта, заархивировать ее и записать под уникальным именем. Как показано на рис. 5.47, для нового архива было использовано имя **Restructure_wChC.zip**, соответствующее компактной высоте и ширине. Вы можете выбирать любое имя, главное, чтобы оно содержало полезную информацию о версии проекта.

Настройка конфигурации iPad (iPhone Plus в альбомной ориентации (wR hR))

В предыдущем разделе мы рассмотрели все этапы, которые необходимо выполнить, чтобы создать базовый макет и специальные макеты для конкретной конфигурации. Для создания макета для устройства iPad нам придется выполнить все эти этапы заново.

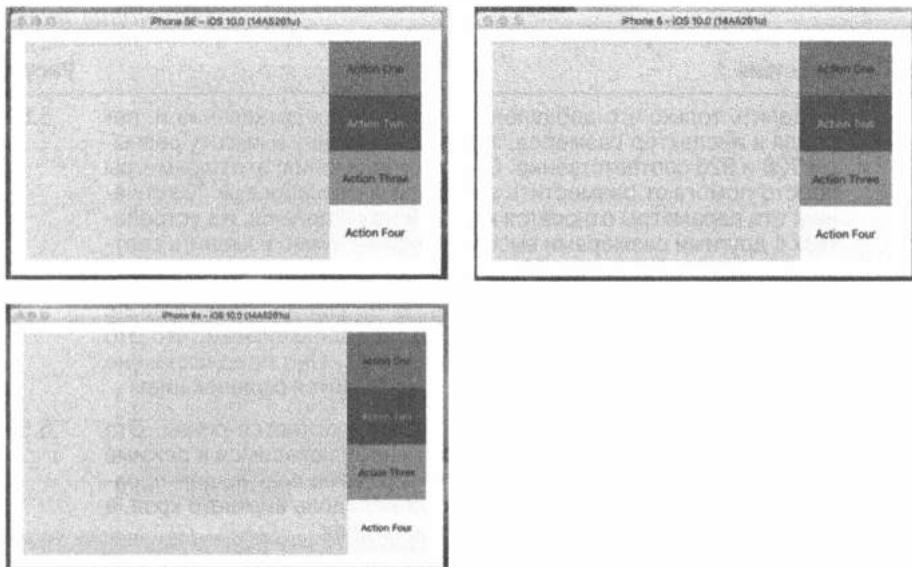


Рис. 5.46. Если все сделано правильно, то на экране любого iPhone (за исключением 6/6s Plus) макет в альбомной ориентации должен выглядеть соответствующим образом

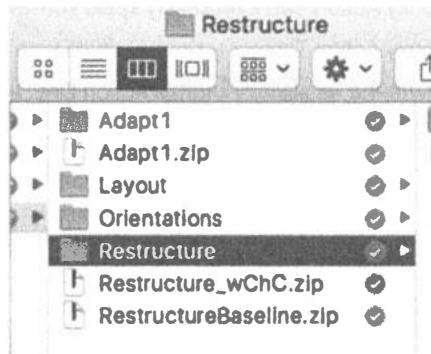


Рис. 5.47. Сохранение текущей версии проекта

Для того чтобы сэкономить место в книге, мы свели все этапы в табл. 5.1.

Таблица 5.1. Настройки для всех видов ориентации iPad и альбомной ориентации iPhone 6/6s Plus

Этап	Действие	Рисунок
1	Щелкнуть на кнопке Vary For Traits и внести изменения в зависимости от ширины	5.48
2	Удалить пять элементов пользовательского интерфейса в раскладовке. Добавить назад пять новых элементов из библиотеки объектов	5.49

Этап	Действие	Рисунок
3	Выделить только что добавленное зеленое представление и, перейдя в инспектор размеров, задать его ширину и высоту равными 728 и 926 соответственно. Это не ограничения; эти параметры просто помогают разместить элементы в раскладовке. (Замечание: эти параметры относятся к устройству iPhone 6s. На устройствах с другими размерами высоту и ширину нужно изменить соответствующим образом.)	5.50
4	Выберите кнопку Action Four и задайте ее ширину равной 182 с помощью инспектора размеров. Еще раз напоминаем, что это значение относится к устройству iPhone 6s. Оно предназначено только для визуализации элемента и не является ограничением	5.51
5	Убедитесь, что панель Device Configuration остается синей. Это свидетельствует о том, что мы по-прежнему находимся в режиме Vary for Traits. Настройте элементы интерфейса следующим образом: зеленое представление расположено вдоль верхнего края, а кнопки образуют один ряд вдоль нижнего края	5.52
6	Как в предыдущем разделе, прикрепите зеленое представление к верхнему, левому и правому краям контейнера	5.53
7	Прикрепите кнопку Action One к левому нижнему углу контейнера	5.54
8	Прикрепите кнопку Action Four к правому нижнему углу контейнера	5.55
9	Прикрепите кнопку Action Two к нижнему краю контейнера	5.56
10	Прикрепите кнопку Action Three к нижнему краю контейнера	5.57
11	Добавьте ограничение на высоту кнопки Action One, которая должна иметь фиксированное значение. Мы использовали высоту, равную 63 точкам, потому что она подходит для макета в нашей раскладовке. Это не единственная возможность; настройте высоту кнопки по своему макету	5.58
12	Нажмите клавишу <Shift> и выберите все четыре кнопки, образующие ряд, задав их высоту и ширину одинаковыми	5.59
13	Перетащите указатель с зеленого представления на кнопку Action One и задайте ограничение Vertical Spacing. В результате зеленое представление будет расположено над рядом кнопок	5.60
14	Перетащите указатель с кнопки Action One на кнопку Action Two и задайте ограничение Horizontal Spacing. Повторите эту операцию для кнопок Action Two и Action Three, а также для кнопок Action Three и Action Four. В результате кнопки будут выровнены по краям и будут занимать одну восьмую ширины контейнера	5.61
15	Щелкните на кнопке Done Varying, чтобы закончить изменение параметров	5.62

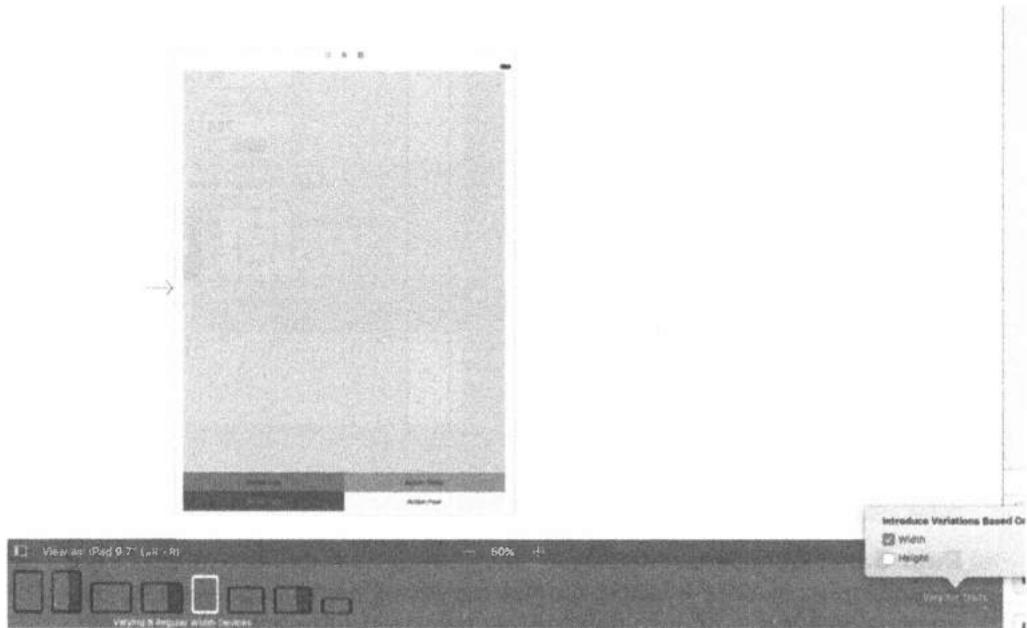


Рис. 5.48. Выберите пункт iPad, щелкните на кнопке Vary For Traits и выберите команду Width

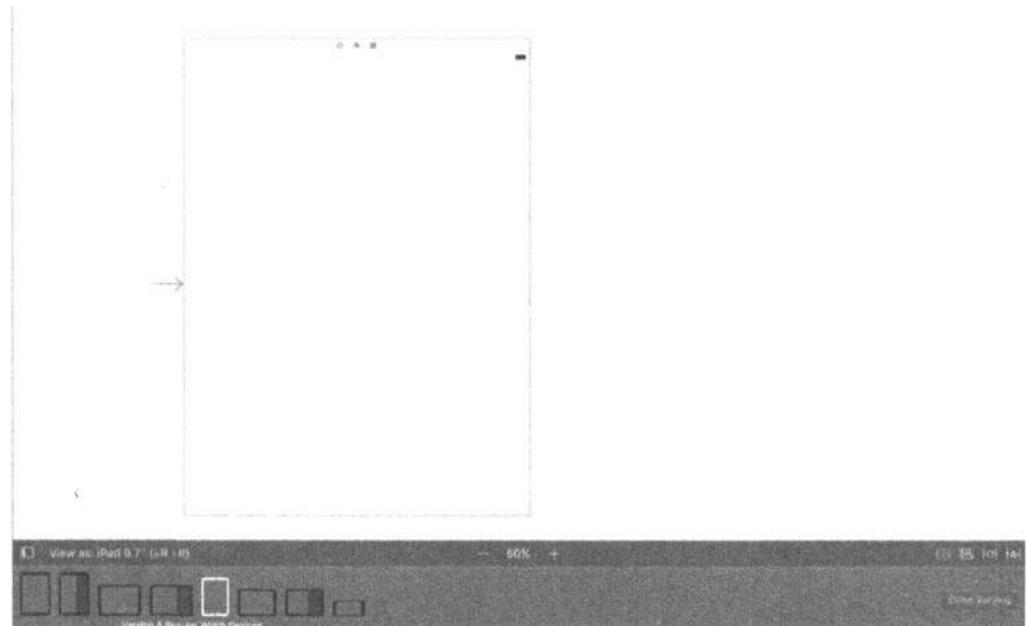


Рис. 5.49. Удалите пять элементов пользовательского интерфейса

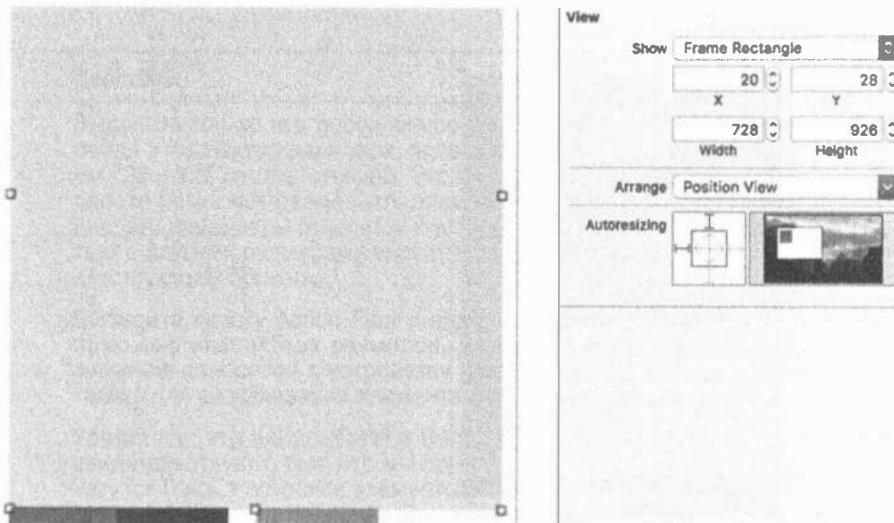


Рис. 5.50. Добавьте пять новых элементов из библиотеки объектов и задайте их высоту и ширину с помощью инспектора размеров. Это не ограничения, а просто параметры визуализации элементов в раскладовке

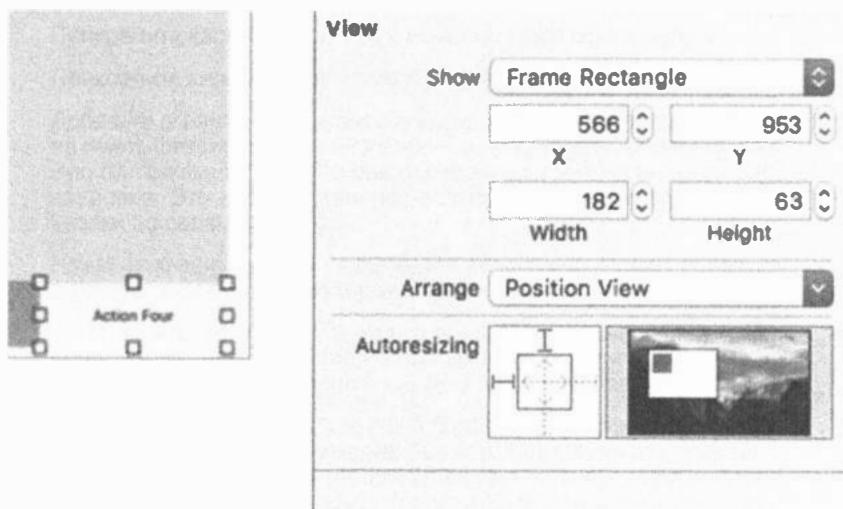


Рис. 5.51. Задайте ширину и высоту кнопки Action Four для работы в раскладовке

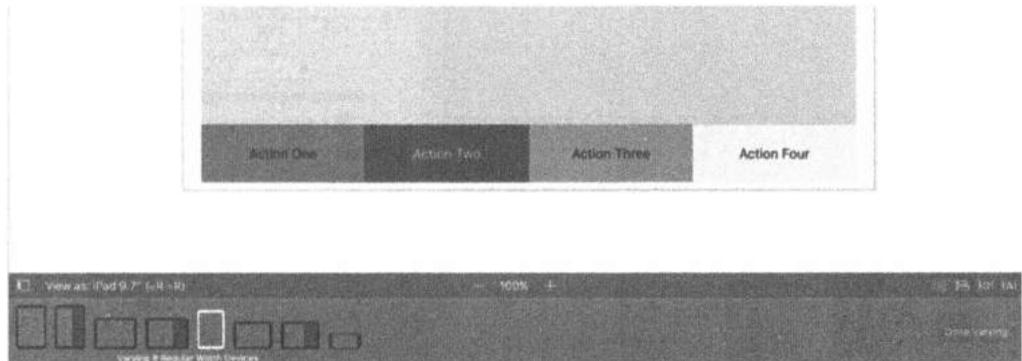


Рис. 5.52. Выровняйте кнопки, пока панель Device Configuration остается синей, позволяя работать с конкретным набором параметров



Рис. 5.53. Прикрепите зеленое представление к верхнему, левому и правому краям внешнего представления

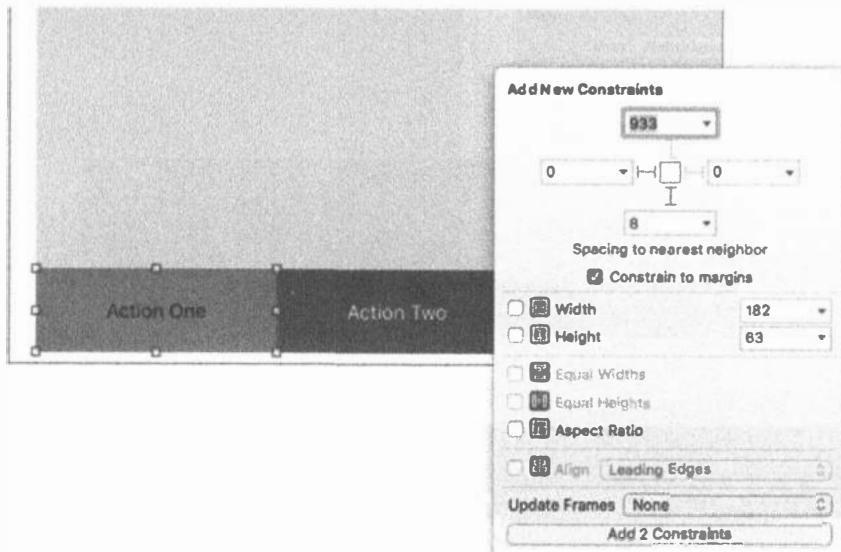


Рис. 5.54. Прикрепите кнопку Action One к левому нижнему углу внешнего представления

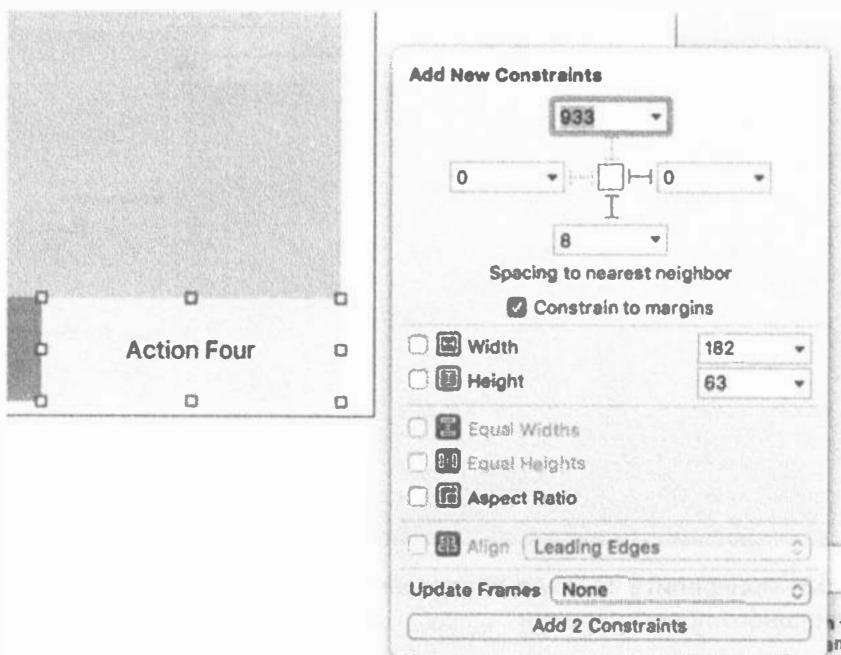


Рис. 5.55. Прикрепите кнопку Action Four к правому нижнему углу внешнего представления

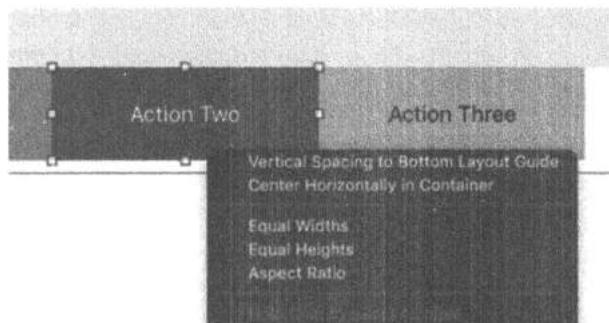


Рис. 5.56. Прикрепите кнопку Action Two к нижнему краю внешнего представления

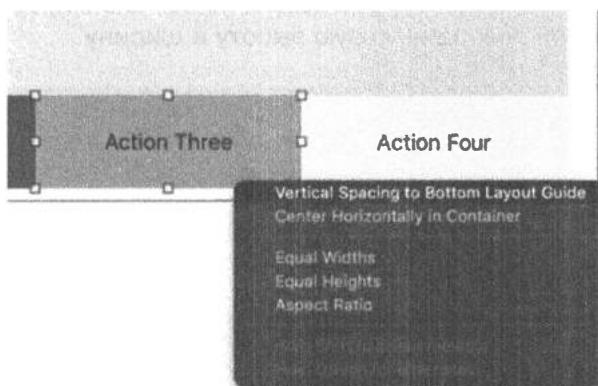


Рис. 5.57. Прикрепите кнопку Action Three к нижнему краю внешнего представления

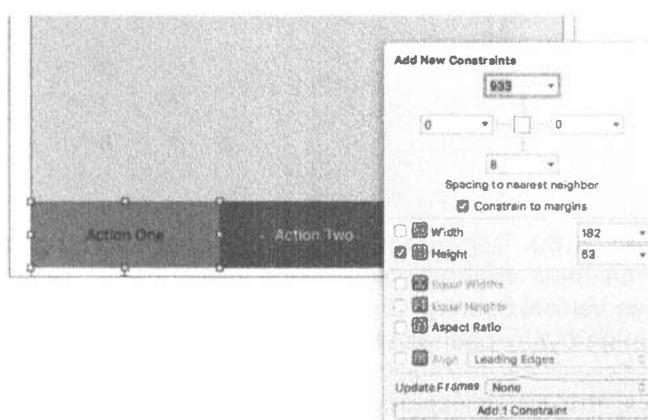


Рис. 5.58. Добавьте ограничение для установки высоты кнопки Action One равной фиксированной величине. Мы использовали высоту, равную 63 точкам, потому что она хорошо подходит для нашей раскладовки

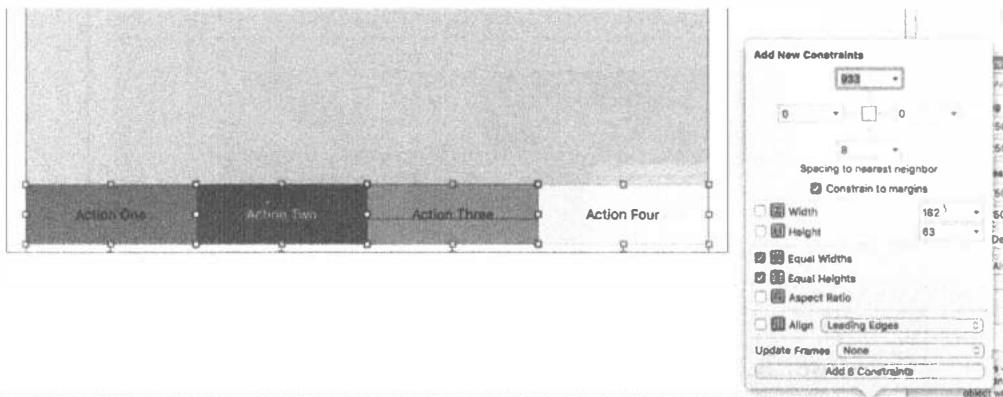


Рис. 5.59. Нажмите клавишу <Shift> и выделите все кнопки, образующие ряд, а затем установите для них одинаковую высоту и ширину

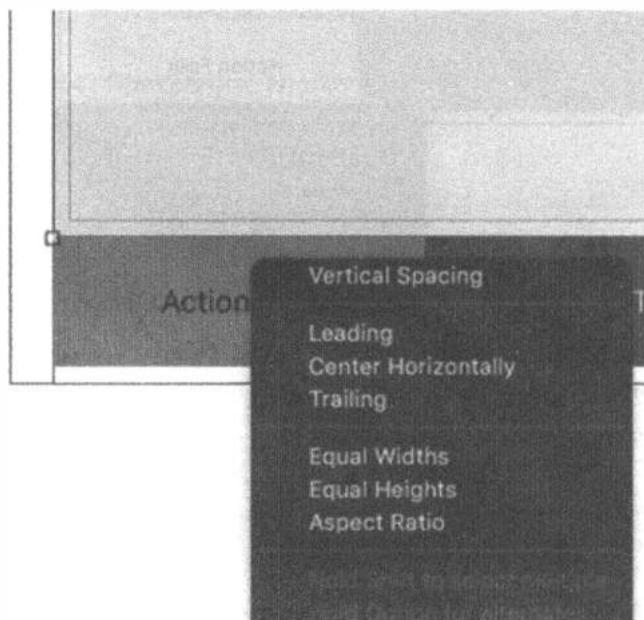


Рис. 5.60. Перетащите указатель с зеленого представления на кнопку Action One и задайте ограничение Vertical Spacing. В результате зеленое представление будет расположено над рядом кнопок

Мы надеемся, что вы разберетесь в сокращенном описании использования механизма Auto Layout. Если вам не понравится результат, можете удалить проект и вернуться к базовой версии. Пока вы не поработаете с механизмом Auto Layout хотя бы десять раз, вы, скорее всего, будете делать ошибки. В этом вы не

одиноки. Если в работе со средой Xcode и особенно с механизмом Auto Layout сделать перерыв продолжительностью несколько недель, то, скорее всего, единственным вариантом будет возврат к базовой версии и повторение пройденного.

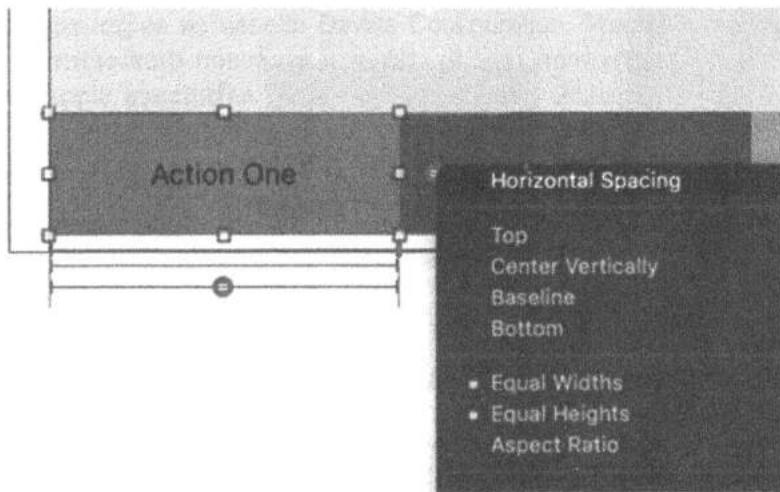


Рис. 5.61. Перетащите указатель с кнопки Action One на кнопку Action Two, удерживая клавишу <Control> и задавая ограничение Horizontal Spacing. Повторите эту операцию для кнопок Action Two и Action Three, а также для кнопок Action Three и Action Four

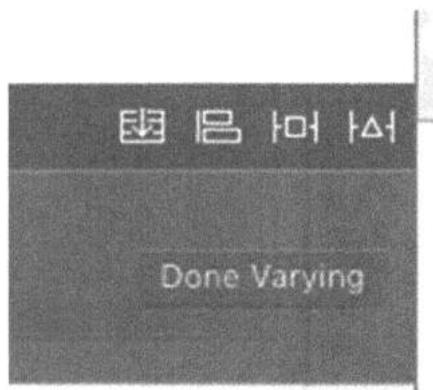


Рис. 5.62. Щелкните на кнопке Done Varying, чтобы закончить изменение параметров

Если вам все удалось, то выберите несколько разных устройств iPad и видов конфигураций, чтобы убедиться, что приложение работает так, как ожидается (рис. 5.63).

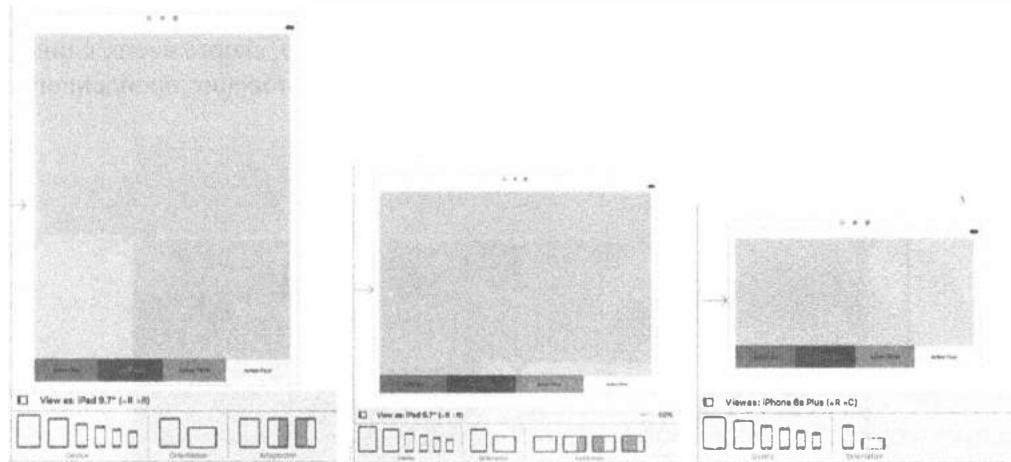


Рис. 5.63. Проверка правильной ориентации приложения на канве раскладовки

В заключение запустите симулятор для разных устройств и убедитесь, что внешний вид приложения остается правильным. На рис. 5.64 показано, как должно выглядеть ваше приложение в книжной и альбомной ориентации на устройстве iPad Air.

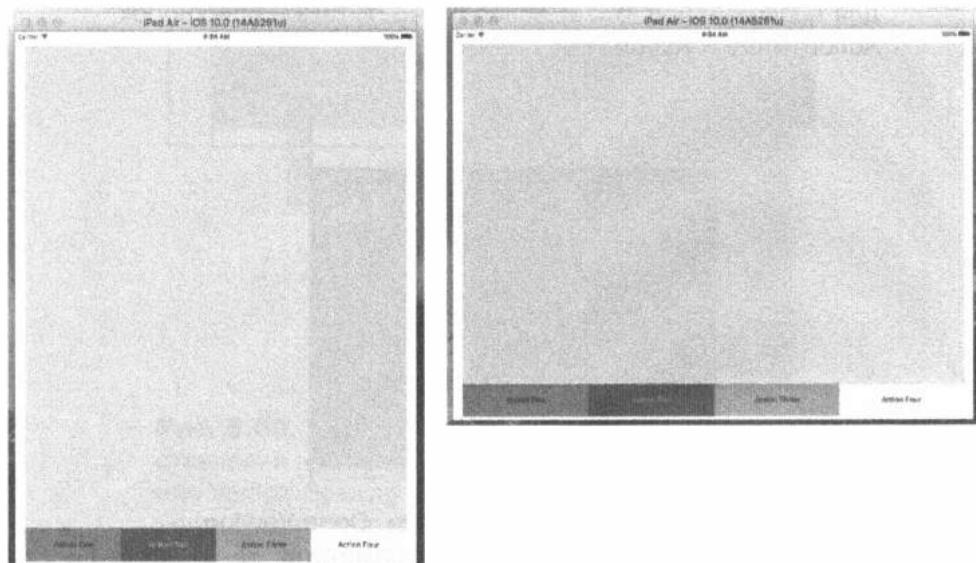


Рис. 5.64. Проверка правильной работы приложения на симуляторе с использованием разных устройств и видов ориентации

Резюме

В данной главе описаны основы механизма автоматического поворота, в большой степени основанного на ограничениях Auto Layout среды Xcode 8 и редакторе параметров на панели Device Configuration. Мы начали с основ механизма автоматического поворота и показали, что произойдет, если ориентация устройства Apple изменится. Наш первый проект, Orientations, продемонстрировал основы обработки простых поворотов устройства и позиционирования меток. Во втором проекте, Layout, мы уточнили наши знания о позиционировании меток, размещая метки по углам, а также у левого и правого краев при повороте устройства.

В заключительном проекте, Restructure, мы углубились в детали механизма Auto Layout для создания специфических конфигураций, предназначенных для конкретных устройств и видов ориентации. Поскольку механизм Auto Layout очень важен как для понимания остальной части книги, так для развития вашей карьеры, постарайтесь разобраться в нем как можно лучше. Сначала это будет трудно, но по мере приобретения опыта работать станет легче — пока компания Apple в следующем году не выпустит его новую версию.

ГЛАВА 6

Приложения с несколькими представлениями

До сих пор мы разрабатывали приложения с одним контроллером представления. Несмотря на то что с единственным представлением можно сделать довольно много, реальная мощь платформы iOS проявляется только тогда, когда вы можете переключать представления в зависимости от данных, введенных пользователем. Приложения с несколькими представлениями имеют несколько стилей, но их основной механизм остается одинаковым независимо от того, как это проявляется на экране.

В этой главе мы сосредоточим внимание на структуре приложения с несколькими представлениями и основах механизма переключения представлений содержимого, разработав “с нуля” собственное приложение с несколькими представлениями. Мы напишем свой класс контроллера, который выполняет переключение. Это позволит оценить преимущества разнообразных контроллеров нескольких представлений, поставляемых компанией Apple.

Но прежде чем начать разработку приложения, рассмотрим, чем хороши приложения с несколькими представлениями.

Основные типы приложений с несколькими представлениями

Строго говоря, мы уже работали с несколькими представлениями в предыдущих приложениях, поскольку кнопки, метки и другие элементы управления являлись подклассами класса `UIView` и могли быть отнесены к иерархии представлений. Однако когда компания Apple использует термин **представление** (*view*) в своей документации, он обычно означает либо класс `UIView`, либо один из его подклассов, который является соответствующим контроллером представления. Эти типы представлений иногда называют **представлениями содержимого** (*content views*) приложения.

Простейший пример приложения с несколькими представлениями — **служебное приложение** (utility application). Большую часть работы оно выполняет с единственным представлением, но предлагает также второе представление, с помощью которого можно конфигурировать представление приложения или предоставлять пользователю больше деталей, чем основное представление. Приложение Stocks, поставляемое вместе с устройством iPhone, — также хороший пример приложения с несколькими представлениями (рис. 6.1). Если щелкнуть на маленькой пиктограмме в правом нижнем углу, то представление конфигурации позволит вам задать список ценных бумаг, отслеживаемых этим приложением.

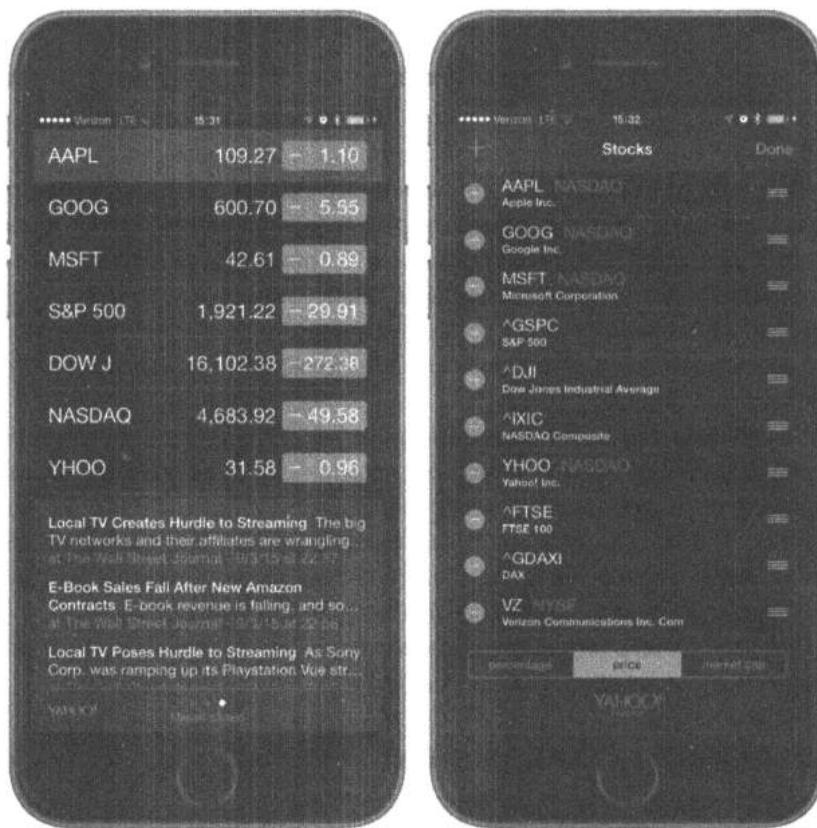


Рис. 6.1. Приложение Stocks, поставляемое с устройством iPhone, имеет два представления: одно — для вывода данных, другое — для конфигурирования списка ценных бумаг

Существует несколько **приложений, использующих панель вкладок** (tab bar applications), которые поставляются вместе с устройством iPhone, например приложения Phone (рис. 6.2) и Clock. Приложение, использующее панель вкладок, — это приложение с несколькими представлениями, которое выводит

в нижней части экрана ряд кнопок, называемых **панелью вкладок** (tab bar). Нажатие одной из этих кнопок активизирует новый контроллер представления и вывод на экран нового представления. В приложении Phone, например, касание вкладки Contacts выводит на экран представление, которое отличается от представления, которое появляется после нажатия кнопки Keypad.



Рис. 6.2. Приложение Phone — пример приложения с несколькими представлениями, использующего панель вкладок

Другой широко распространенной разновидностью приложений для устройства iPhone с несколькими представлениями являются приложения, обладающие свойствами навигационного контроллера, использующего **панель навигации** (navigation bar) для управления иерархией представлений. Ярким примером является приложение Settings (рис. 6.3). Первое представление в приложении Settings показывает пользователю ряд строк, в котором каждая строка соответствует кластеру настроек или конкретному приложению. Касание одной из этих строк вызывает новое представление, в котором можно задать конкретный

218 ГЛАВА 6 ■ ПРИЛОЖЕНИЯ С НЕСКОЛЬКИМИ ПРЕДСТАВЛЕНИЯМИ

набор настроек. Некоторые представления выводят на экран список, позволяющий пользователю заглянуть глубже. Контроллер навигации следит за тем, насколько вы углубились в иерархию представлений, и позволяет вернуться к предыдущему представлению.

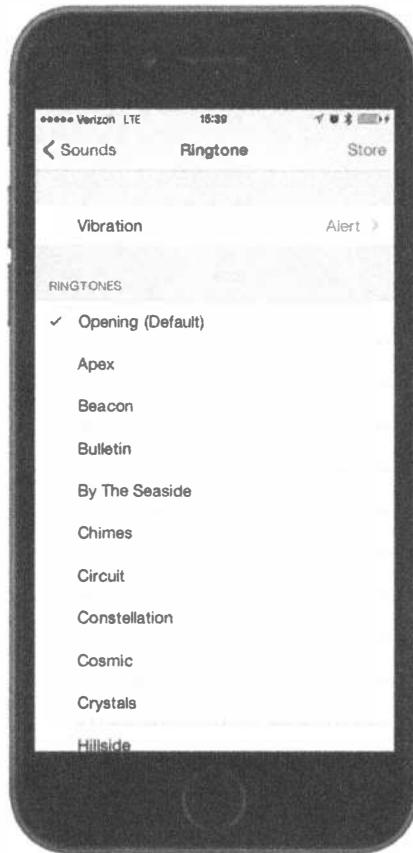


Рис. 6.3. Приложение iPhone Settings — яркий пример приложения с несколькими представлениями, использующего панель навигации

Например, если вы выбрали настройку Sounds, то увидите представление, содержащее список настроек, связанных со звуком. В верхней части этого представления расположена панель навигации, нажатие которой возвращает вас к предыдущему представлению. В настройках звука есть строка с меткой Ringtone. Коснитесь ее, и увидите новое представление, содержащее список мелодий для телефона и панель навигации, позволяющую вернуться к главному представлению Sounds (рис. 6.4). Приложение, использующее навигацию, полезно, если вы хотите иметь иерархию представлений.

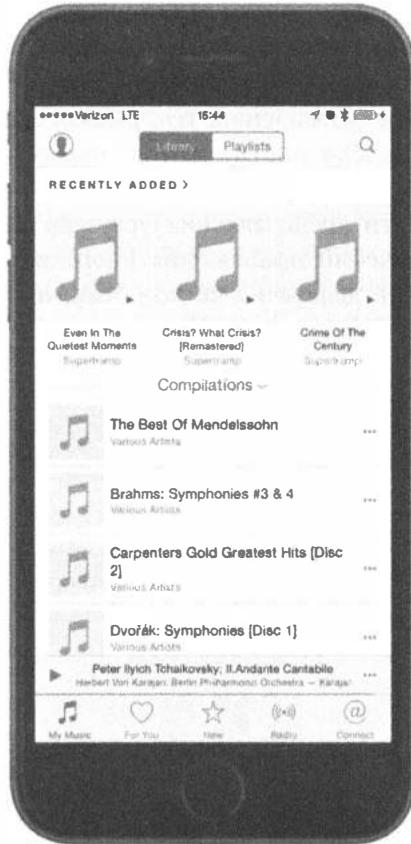


Рис. 6.4. Приложение Music использует и панель навигации, и панель вкладок

В устройстве iPad большинство приложений, основанных на навигации, таких как Mail, реализованы с помощью **разделенного представления** (*split view*), в котором элементы навигации находятся в левой части экрана, а элементы, которые пользователь выбирает для просмотра или редактирования, — в правой. Более подробно о разделенных представлениях и других элементах графического пользовательского интерфейса, характерных для устройства iPad, мы поговорим в главе 11.

Поскольку представления сами по себе имеют иерархическую природу, можно сочетать разные механизмы переключения представлений в рамках одного приложения. Например, приложение iPod для устройства iPhone использует панель вкладок для переключений между разными способами организации аудиозаписей, а контроллер навигации и связанную с ней навигационную панель — для прослушивания аудиозаписей в соответствии со сделанным выбором их группировки. На рис. 6.4 панель вкладок расположена в нижней части экрана, а панель навигации — в верхней.

Некоторые приложения используют **панель инструментов** (toolbar), которую часто путают с панелью вкладок. Панель вкладок используется для выбора только одного варианта из нескольких, а панель инструментов может содержать кнопки и другие элементы управления, которые не являются взаимоисключающими. Яркий пример панели инструментов можно увидеть в нижней части главного представления браузера Safari (рис. 6.5). Если вы сравните панель инструментов в нижней части представления браузера Safari с панелью вкладок в нижней части представлений приложения Phone или iPod, то легко их различите. Панель вкладок разделена на четко обозначенные сегменты, а панель инструментов, как правило, нет.



Рис. 6.5. В браузере Mobile Safari есть панель инструментов в нижней части экрана. Она похожа на свободную панель, позволяющую выбирать несколько элементов управления одновременно

Приложения с несколькими представлениями, относящиеся к каждому из этих типов, используют особый класс контроллера из пакета UIKit. Интерфейсы

панелей вкладок реализованы на основе класса `UITabBarController`, а интерфейс панелей навигации использует класс `UINavigationController`.

Архитектура приложения с несколькими представлениями

Приложение `View Switcher`, которое мы собираемся разработать в этой главе, выглядит очень просто, но код, который мы планируем написать, намного сложнее предыдущих программ. Приложение `View Switcher` будет состоять из трех разных контроллеров, раскладовки и делегата приложения.

При первом запуске программа `View Switcher` будет выглядеть так, как показано на рис. 6.6, с панелью инструментов, содержащей одну кнопку. Остальная часть представления будет состоять из голубого фона и кнопки, готовой для нажатия.

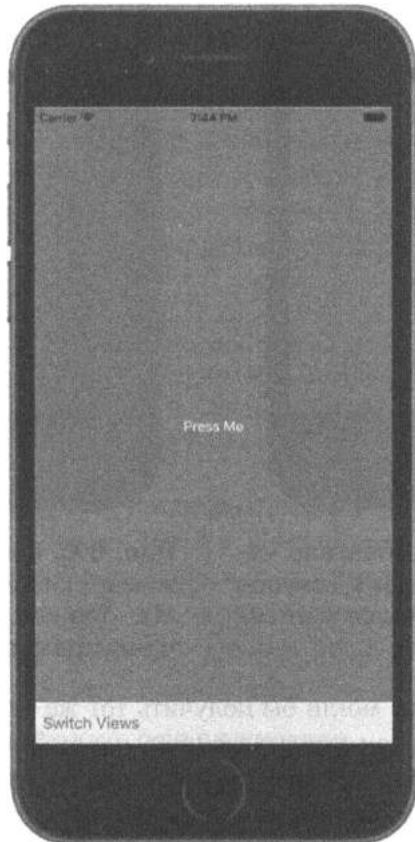


Рис. 6.6. При первом запуске приложения мы увидим голубое представление с кнопкой и панель инструментов со своей кнопкой

Когда пользователь нажмет кнопку **Switch Views**, фон окрасится в желтый цвет, а заголовок кнопки изменится (рис. 6.7).

Если нажата кнопка **Press Me** или **Press Me, Too**, на экране появится предупреждение, сообщающее, какая из кнопок была нажата (рис. 6.8).

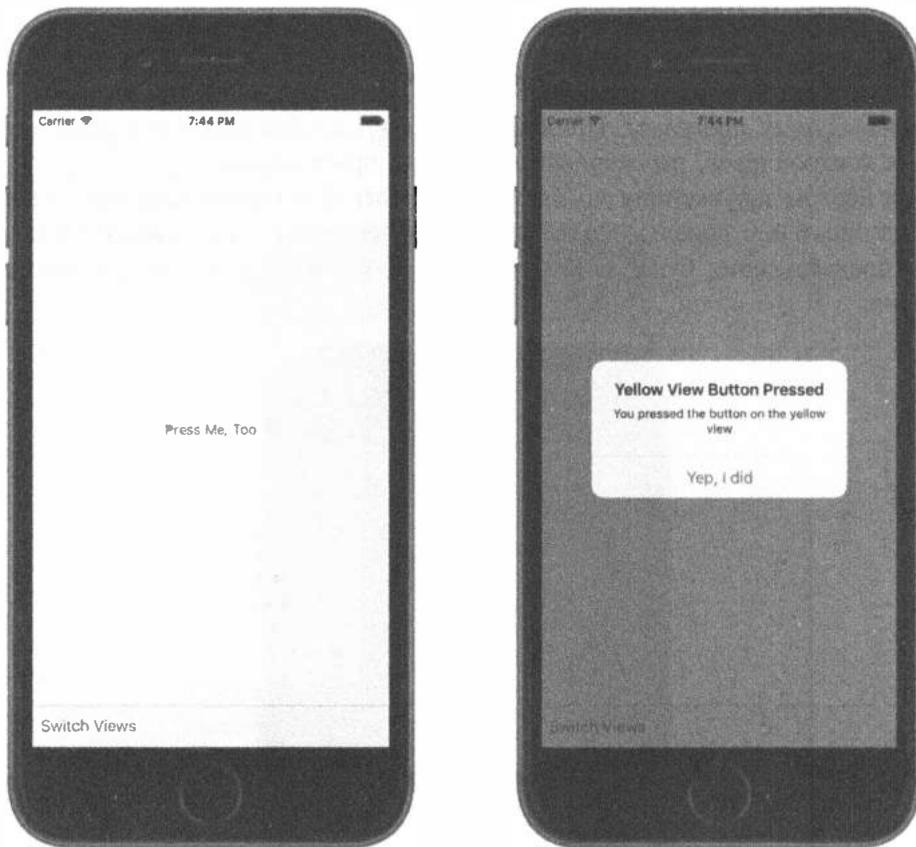


Рис. 6.7. Когда пользователь нажимает кнопку **Switch Views**, голубое представление заменяется желтым

Рис. 6.8. Когда пользователь нажимает кнопку **Press Me** или **Press Me, Too**, на экране появляется предупреждение

Несмотря на то что мы могли бы получить тот же самый эффект с помощью приложения с единственным представлением, мы выбрали более сложный путь, чтобы продемонстрировать механизм приложения с несколькими представлениями. В этом простом приложении на самом деле скрыты три разных контроллера представления: один управляет голубым представлением, второй — желтым, а третий специальный контроллер переключает эти представления, когда пользователь нажимает кнопку **Switch Views**.

Прежде чем приступить к разработке приложения, поговорим немного об устройстве приложений с несколькими представлениями для устройства iPhone. Большинство приложений с несколькими представлениями устроено точно так же.

Корневой контроллер

В приложении с несколькими представлениями ключевую роль играет раскадровка, содержащая все представления и контроллеры представлений нашего приложения. Мы создадим раскадровку, содержащую экземпляр класса контроллера, выбирающий представление, которое должно выводиться на экран в текущий момент. Назовем этот контроллер **корневым** (root controller), по ассоциации с корнем дерева или “корнем всех бед”, поскольку это первый контроллер, который видит пользователь и который загружается в память при запуске приложения. Этот корневой контроллер часто является экземпляром класса `UINavigationController` или `UITabBarController`, хотя иногда может быть наследником класса `UIViewController`.

В приложении с несколькими представлениями работа корневого контроллера сводится к тому, чтобы выбрать одно из двух или нескольких представлений и вывести его на экран в соответствующем виде в зависимости от данных, введенных пользователем. Контроллер панели вкладок, например, переключает разные представления и контроллеры представлений в зависимости от того, какой элемент панели вкладок пользователь нажимал в последний раз. Контроллер навигации делает то же самое по мере перемещения пользователя по иерархии представлений.

ЗАМЕЧАНИЕ. Корневой контроллер — это контроллер основного представления для приложения, и в этой роли именно он определяет, можно ли выполнить автоматический поворот при изменении ориентации. Однако корневой контроллер может передавать ответственность за такие решения текущему активному контроллеру.

В приложениях с несколькими представлениями большую часть экрана занимает представление содержимого, причем каждое представление содержимого имеет собственный контроллер со своими выходами и действиями. Например, в приложении с панелью вкладок при нажатии панели будет вызван контроллер панели вкладок, а при нажатии любого другого места экрана — контроллер текущего представления.

Устройство представления содержимого

В приложении с несколькими представлениями каждый контроллер представления управляет каким-то представлением содержимого, и эти представления являются основой интерфейса. В совокупности они образуют **сцену** (scene) в раскадровке. Каждое представление содержимого состоит из контроллера представления и представления, содержимое которого может быть подклассом класса `UIView`. Если только вы не собираетесь делать нечто совершенно необычное,

ваше представление содержимого будет всегда иметь соответствующий контроллер представления и иногда подкласс класса `UIView`. Хотя вы можете запрограммировать интерфейс, не используя nib-файл, лишь некоторые люди выбирают этот путь, потому что он более долгий и трудный.

В этой главе мы создадим новый класс контроллера для каждого представления содержимого. Корневой контроллер нашего приложения управляет представлением содержимого, которое состоит из панели инструментов, расположенной в нижней части экрана. Он загружает контроллер голубого представления, который, в свою очередь, выводит на экран это представление в качестве дочернего представления по отношению к представлению корневого контроллера. Когда пользователь нажимает кнопку, переключающую представления и принадлежащую корневому контроллеру (эта кнопка расположена на панели инструментов), корневой контроллер отключает голубое представление и передает управление контроллеру желтого представления, инициируя его по мере необходимости. Итак, приступим к созданию проекта — и все станет намного яснее.

Создание переключателя представлений

Для того чтобы создать наш проект, откройте среду Xcode и выполните команду `File⇒New⇒Project...` или нажмите комбинацию клавиш `<Shift+⌘+N>`. Когда откроется окно помощника, выберите пиктограмму `Single View Application` и щелкните на кнопке `Next`. На следующей странице помощника введите в поле `Product Name` строку `View Switcher`, установите значение `Language` равным `Swift` и выберите пункт `Universal` в списке `Devices`. Щелкните на кнопке `Next`. В открывшемся окне найдите место, где хотите сохранить свой проект, и щелкните на кнопке `Create`, чтобы создать папку для нового проекта.

Переименование контроллера представления

Как мы уже видели, шаблон `Single View Application` предоставляет делегат приложения, контроллер представления и раскладовку. Класс контроллера представления называется `ViewController`. В этом приложении мы будем работать с тремя контроллерами представлений, но основная логика приложения будет сосредоточена в контроллере главного представления. Он будет постоянно переключать экран с одного контроллера представления на другой. Для того чтобы разъяснить роль контроллера главного представления, его имя следовало бы уточнить, например назвать его `SwitchingViewController`. В проекте есть несколько мест, где упоминается имя класса контроллера представления. Для того чтобы изменить его имя, нам необходимо изменить информацию во всех этих местах. В среде Xcode есть прекрасная функциональная возможность под названием **рефакторинг** (*refactoring*), которая позволяет сделать это, но когда мы писали книгу, эта возможность еще не поддерживалась проектами на языке `Swift`. По этой причине мы удалим шаблонный контроллер и добавим новый.

Выберите файл ViewController.swift в окне навигатора проекта, щелкните на правой кнопке и выполните команду Delete в контекстном меню (рис. 6.9). Получив приглашение, переместите исходный файл в папку Trash.

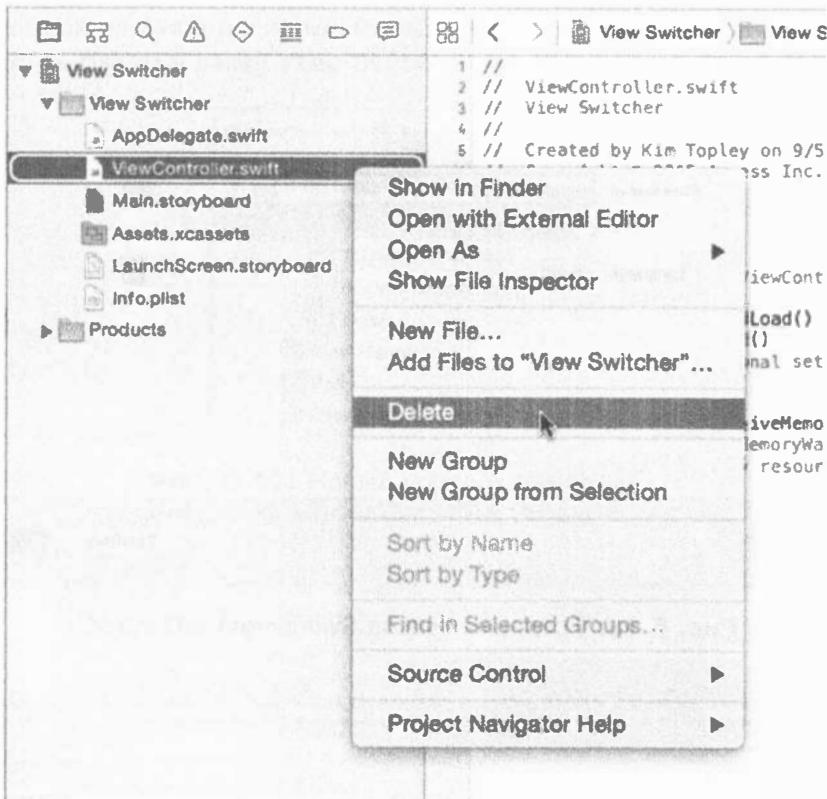


Рис. 6.9. Удаление шаблонного контроллера представления

Щелкните правой кнопкой мыши на имени группы View Switcher и выполните команду New File.... Выберите пиктограмму Cocoa Touch Class в разделе iOS Source. Назовите класс SwitchingViewController и сделайте его подклассом класса ViewController. Сбросьте флагок Also create XIB file, установите значение Language равным Swift (как показано на рис. 6.10), а затем щелкните на кнопках Next и Create.

Новый контроллер представления необходимо добавить в раскладовку. Выберите файл Main.storyboard в окне Document Outline и откройте раскладовку для редактирования. Вы увидите, что шаблон уже создал контроллер представления, и нам нужно просто связать его с нашим классом SwitchingViewController. Выберите контроллер представления в окне Document Outline и откройте окно инспектора идентичности. Измените значение Class с UIViewController на SwitchingViewController в разделе Custom Class (рис. 6.11).



Рис. 6.10. Создание класса SwitchingViewController



Рис. 6.11. Изменение класса контроллера представления в раскладовке

Теперь в окне Document Outline вы должны увидеть, что имя контроллера представления изменилось на Switching View Controller (рис. 6.12).

Добавление контроллеров представления содержимого

Для отображения представлений содержимого на экране нам нужны еще два контроллера представлений. Щелкните правой кнопкой мыши на имени группы View Switzer и выполните команду New File.... Перейдите в диалоговое окно выбора шаблонов, выберите пиктограмму Cocoa Touch Class в разделе iOS Source и щелкните на кнопке Next. Назовите новый класс BlueViewController и сделайте его подклассом класса UIViewController. Сбросьте флагок Also create

XIB file, потому что мы планируем чуть позже добавить этот контроллер в раскадровку. Щелкните на кнопках Next и Create, чтобы сохранить файлы нового контроллера представления. Повторите этот процесс, чтобы создать второй контроллер представления содержимого, и назовите его YellowViewController. Для правильной организации проекта можно перейти в окно навигатора проекта и переместить файлы в папку View Switcher, как показано на рис. 6.13.

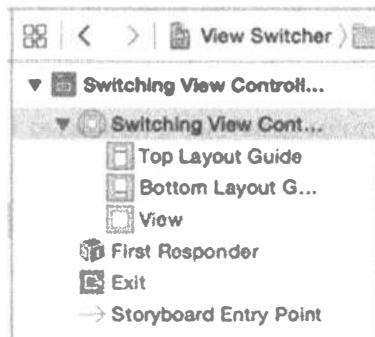


Рис. 6.12. Новый контроллер представления в окне Document Outline

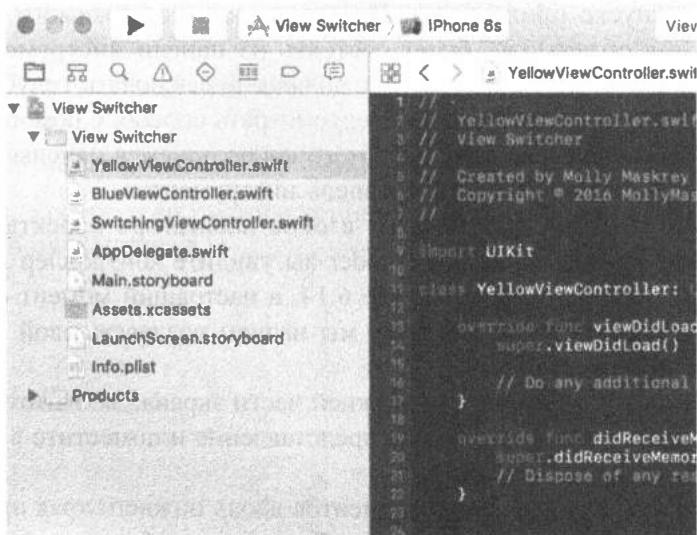


Рис. 6.13. Исходные файлы можно переместить в папку View Switcher в окне навигатора проекта

Модификация файла SwitchingViewController.swift

В классе SwitchinViewController нужен метод, выполняющий действие, чтобы переключаться между голубым и желтым представлениями. Нам вообще не нужны выходы, но все же необходимы два других указателя, по одному для

каждого контроллера, которые мы будем переключать. Они не обязаны быть выходами, поскольку мы собираемся создавать их в исходном файле, а не в раскладовке. Добавьте в файл `SwitchingViewController.swift` следующий код:

```
private var blueViewController: BlueViewController!
private var yellowViewController: YellowViewController!
```

Добавьте в конец класса следующий метод:

```
@IBAction func switchViews(sender: UIBarButtonItem) { }
```

Ранее мы добавляли методы действия с помощью перетаскивания курсора и клавиши <Control>, но теперь появилась другая возможность, поскольку программа IB может распознавать выходы и действия, уже определенные в исходном коде. Объявив необходимое действие, мы можем настроить минимальный пользовательский интерфейс для данного контроллера в нашей раскладовке.

Создание представления с панелью инструментов

Необходимо создать представление, чтобы добавить его к контроллеру `SwitchingViewController`. Напомним, что этот новый контроллер представления будет играть роль корневого контроллера, который вступает в действие при запуске приложения. Представление содержимого контроллера `SwitchingViewController` будет состоять из панели инструментов, расположенной в нижней части экрана. Она должна переключать голубое и желтое представления, поэтому необходимо предусмотреть способ, с помощью которого пользователь сможет это сделать. Для этого воспользуемся панелью инструментов с кнопкой. Итак, сконструируем панель инструментов.

Выберите файл `Main.storyboard` в окне навигатора проекта. В окне редактирования программы Interface Builder вы увидите контроллер переключающего представления. Как видно на рис. 6.14, в настоящий момент оно пустое и довольно неинтересное. Именно здесь мы начнем создавать свой графический пользовательский интерфейс.

Добавим панель инструментов в нижней части экрана. Захватите объект `Toolbar` из библиотеки, перетащите в свое представление и поместите в нижней части окна, как показано на рис. 6.15.

Мы хотим растянуть панель инструментов вдоль нижнего края представления содержимого независимо от его размера. Для этого необходимо установить три ограничения макета — одно из них прикрепит панель инструментов к нижнему краю представления, а два других прикрепят его к левому и правому краям. Для этого выберите панель инструментов в окне `Document Outline`, щелкните на кнопке `Pin` панели инструментов, расположенной под раскладовкой, и измените значения во всплывающем окне так, как показано на рис. 6.16.

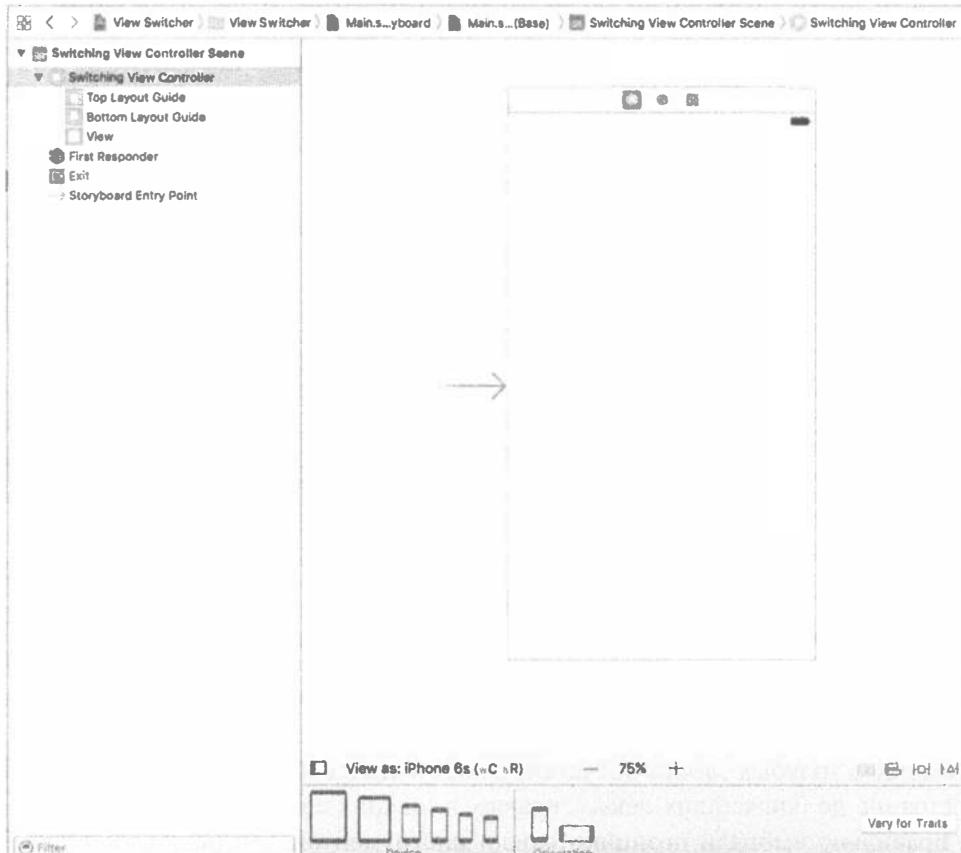


Рис. 6.14. Контроллер пустого корневого представления (Switching View Controller) в раскладовке

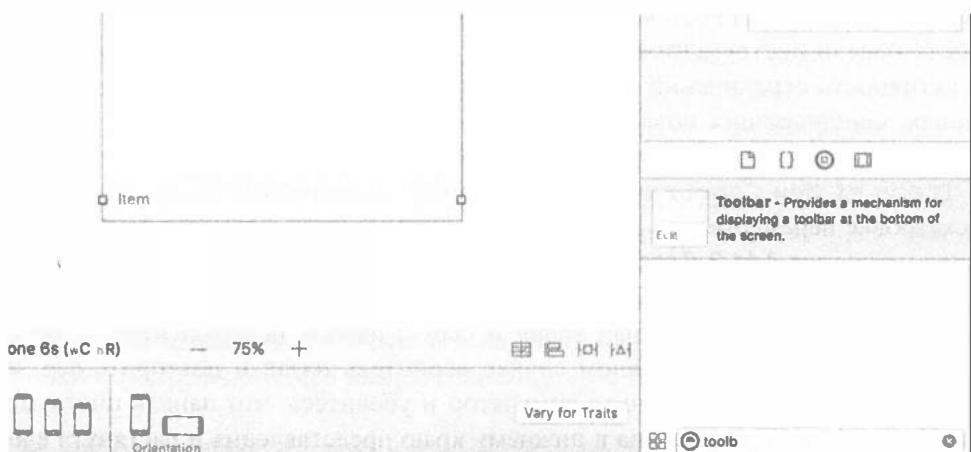


Рис. 6.15. Мы перетащили объект Toolbar на новое представление. Обратите внимание на то, что этот объект содержит одну кнопку с меткой

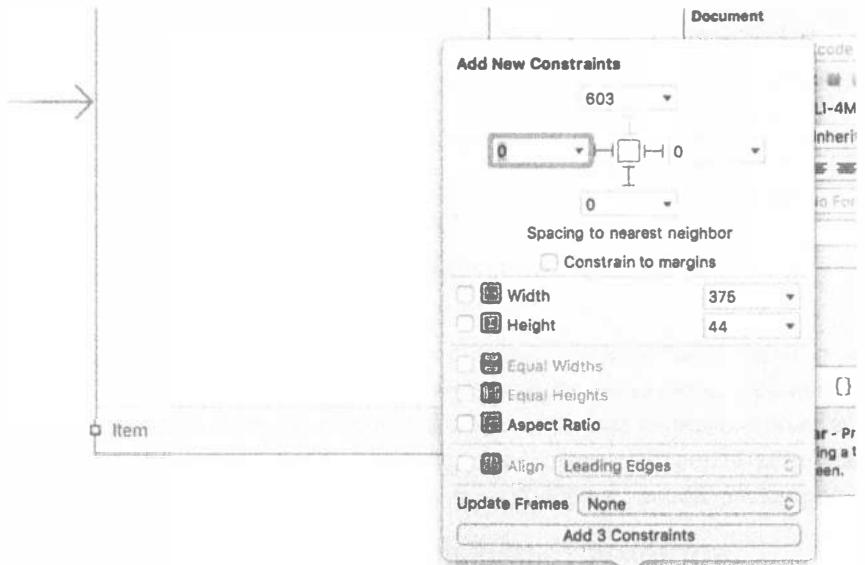


Рис. 6.16. Прикрепление панели инструментов к нижнему, левому и правому краям контейнерного представления

Сбросьте флажок **Constrain to margins**, поскольку мы хотим позиционировать панель инструментов по отношению к краям представления содержимого, а не с помощью голубых линий разметки, появляющихся возле краев. Затем задайте расстояние до ближайших левых, правых и нижних соседей равным нулю (если вы правильно выбрали позицию панели инструментов, эти расстояния уже будут равны нулю). В данном случае ближайшим соседом панели инструментов является представление содержимого. Этот факт можно обнаружить, щелкнув на маленькой стрелке в одном из флажков, регламентирующих расстояния: открывается всплывающее окно, в котором показан ближайший и все другие возможные соседи панели инструментов, — в данном случае их нет. Для того чтобы указать на активность ограничений расстояния, щелкните на трех красных пунктирных линиях, соединяющих поля редактирования с маленьким квадратом в центре, чтобы они стали сплошными. В заключение измените значение **Update Frames** на **Items of New Constraints** (чтобы представление панели инструментов в раскладовке переместилось в новое место, определенное ограничениями) и щелкните на кнопке **Add 3 Constraints**.

Щелкните на кнопке **Run** и запустите приложение на симуляторе iOS. Вы должны увидеть пустой белый экран и серую панель инструментов, содержащую одну кнопку. В противном случае вернитесь назад и повторите все, что было описано выше. Поверните симулятор и убедитесь, что панель инструментов по-прежнему прикреплена к нижнему краю представления и растянута вдоль экрана. Если нет, исправьте ограничения, относящиеся к панели инструментов.

Связывание кнопки панели инструментов с контроллером представления

Панель инструментов содержит единственную кнопку, которую мы будем использовать, чтобы предоставить пользователю возможность переключаться между разными представлениями содержимого. Дважды щелкните на кнопке в раскладовке (рис. 6.17) и измените ее название на *Switch Views*. Нажмите клавишу <Return>, чтобы подтвердить внесенные изменения. Мы можем связать кнопку на панели инструментов с нашим методом, выполняющим действие. Прежде чем сделать это, должны предупредить: кнопки на панели инструментов не похожи на элементы управления системы iOS. Они поддерживают только одно целевое действие и могут инициировать его только в четко определенный момент времени, который является эквивалентом события *Touch Up Inside*, происходящего с элементами управления системы iOS.

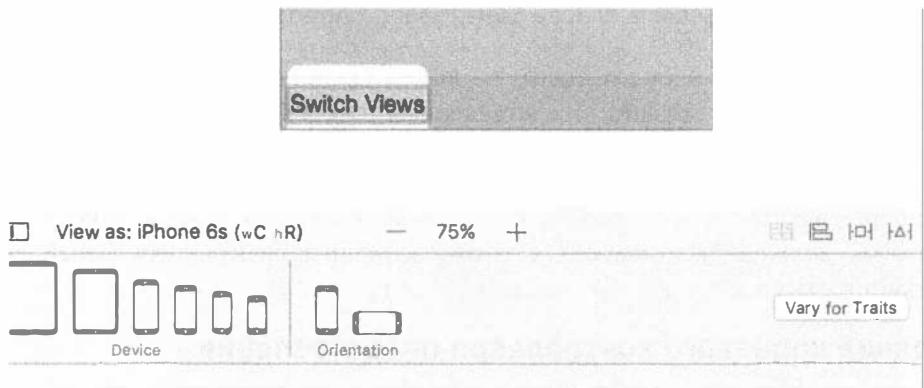


Рис. 6.17. Изменение надписи на кнопке на панели инструментов на *Switch Views*

Выбрать кнопку на панели инструментов может быть сложно. Проще всего раскрыть пиктограмму *Switching View Controller* в окне Document Outline, найти кнопку *Switch Views* и щелкнуть на ней. Выбрав кнопку *Switch Views*, нажмите клавишу <Control> и перетащите указатель от нее к пиктограмме *Switching View Controller* в верхней части сцены (рис. 6.18). Отпустите кнопку мыши и выберите действие *switchViewsWithSender*: из всплывающего меню. Если действие *switchViewsWithSender*: на экране не отображается, а вместо него вы видите указатель, который называется **делегатом**, значит, скорее всего, вы перетащили указатель от панели инструментов, а не от кнопки. Для того чтобы исправить эту ошибку, проверьте, что выбрали кнопку, а не панель инструментов, а затем повторите процедуру.



Рис. 6.18. Связывание кнопки панели инструментов с методом `switchViewsWithSender` в классе контроллера представления

Отметим еще один элемент сцены — выход представления `SwitchingViewController`. Он уже связан с представлением на сцене. Выход `view` является наследником класса `UIViewController` и предоставляет контроллеру доступ к своим элементам управления. При создании проекта программа Xcode создает как контроллер, так и его представление, и связывает их один с другим. Это все, что мы должны были сделать, поэтому сохраните раскладовку. Перейдем к реализации класса `SwitchingViewController`.

Создание корневого контроллера представления

Настало время создать корневой контроллер представления. Он предназначен для переключения между желтым и голубым представлениями, когда пользователь щелкает на кнопке `Switch Views`. Внесем изменения в файл `SwitchingViewController.swift` и модифицируем метод `viewDidLoad()`, добавив строки, приведенные в листинге 6.1.

Листинг 6.1. Код метода `viewDidLoad` контроллере корневого представления

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

    // Дополнительная настройка после загрузки представления.
    blueViewController =
        storyboard?.instantiateViewController(withIdentifier: "Blue")
        as! BlueViewController
    blueViewController.view.frame = view.frame
    switchViewController(from: nil,
                        to: blueViewController) // вспомогательный метод
}
```

ЗАМЕЧАНИЕ. Если вы введете код, показанный в листинге 6.1, в файл с расширением .swift, то получите сообщение об ошибке в строке, содержащей вызов метода switchViewController. Это объясняется тем, что мы еще не написали код вспомогательного метода. Вскоре мы это сделаем.

Модифицированный нами метод viewDidLoad замещает метод UIViewController, который вызывается при загрузке раскладовки. Откуда мы это знаем? Нажмите клавишу <Option>, дважды щелкните на имени метода и посмотрите на окно с документацией, которое появится на экране (рис. 6.19). Кроме того, можно выбрать команду View⇒Utilities⇒Show Quick Help Inspector. Этот метод определен в родительском классе UIViewController и предназначен для замещения классами, которые должны получать уведомления о том, что загрузка представления завершена.

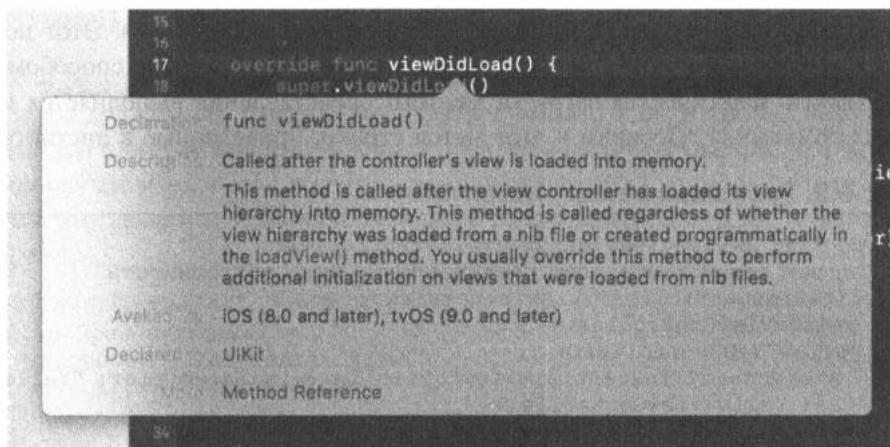


Рис. 6.19. Это окно с документацией появляется после того, как вы дважды щелкнете на методе viewDidLoad, удерживая нажатой клавишу <Option>

Этот вариант метода viewDidLoad создает экземпляр класса BlueViewController. Мы используем метод instantiateViewController(withIdentifier:) для загрузки экземпляра класса BlueViewController из раскладовки. Для доступа к конкретному контроллеру представления из раскладовки мы используем строку-идентификатор — в данном случае “Blue”, — которая задается при настройке раскладовки. После создания экземпляра класса BlueViewController мы присваиваем этот новый экземпляр свойству blueViewController.

```
blueViewController =
    storyboard?.instantiateViewController(withIdentifier: "Blue")
    as! BlueViewController
```

Затем зададим рамку голубого представления такой же, как у представления содержимого, и переключимся на контроллер голубого представления.

```
blueViewController.view.frame = view.frame  
switchViewController(from: nil, to: blueViewController)
```

Поскольку контроллер представления необходимо выполнить в нескольких местах, мы вскоре напишем для этого вспомогательный метод `switchViewController(from:, to:)`.

Почему бы нам не загрузить также желтое представление? Нам ведь все равно придется это сделать, так почему бы не сейчас? Хороший вопрос. Ответ состоит в том, что пользователь может никогда не нажать кнопку `Switch Views`. Он может просто использовать представление, которое видит на экране, а затем выйти из приложения. В этом случае нет смысла использовать ресурсы для загрузки желтого представления и его контроллера. Вместо этого мы загрузим желтое представление только тогда, когда оно действительно понадобится. Этот подход называется **ленивой загрузкой** (*lazy loading*) и стал стандартным способом экономии памяти. Фактическая загрузка желтого представления выполняется методом `switchViews()`. Добавим в этот метод строки, приведенные в листинге 6.2.

Листинг 6.2. Реализация метода switchViews

Сначала метод `switchViews()` проверяет, какое представление было включено, проверяя, не равно ли значение `yellowViewController` в родительском представлении значению `nil`. Метод возвращает значение `true`, если выполняется одно из двух условий.

- Если представление `yellowViewController` существует, но не показано пользователю, значит, оно не имеет родительского представления, потому что его в данный момент нет в иерархии представлений, и вычисляемое выражение станет равным `true`.
- Если представления `yellowViewController` не существует, потому что оно еще не было создано или было удалено из памяти, метод тоже вернет значение `true`.

Затем проверяем, равен ли объект `yellowViewController` значению `nil`.

```
if yellowViewController?.view.superview == nil {
```

Если результат равен `nil`, значит, экземпляра класса `yellowViewController` нет, и мы должны его создать. Это может произойти, если кнопка нажимается в первый раз или если система наводила порядок в памяти и удалила этот экземпляр. В этом случае мы должны создать экземпляр класса `YellowViewController` так, как мы это делали с экземпляром класса `BlueViewController` в методе `viewDidLoad`.

```
if yellowViewController == nil {
    yellowViewController =
        storyboard?.instantiateViewController(withIdentifier: "Yellow")
        as! YellowViewController
}
```

Если мы переключились на голубое представление, то должны проверить, существует ли он (поскольку он мог быть удален из памяти), и создать его, если его нет. Код, выполняющий эти операции, точно такой же, как и выше, только на этот раз он относится к голубому представлению.

```
} else if blueViewController?.view.superview == nil {
    if blueViewController == nil {
        blueViewController =
            storyboard?.instantiateViewController(withIdentifier: "Blue")
            as! BlueViewController
    }
}
```

Поскольку нам известно, что экземпляр контроллера представления существует (он создан либо системой, либо нами), задаем одинаковые рамки контроллера представления и контроллера представления содержимого и выполняем переключение с помощью метода `switchViewController(from:, to:)`, как показано в листинге 6.3.

Листинг 6.3. Переключение контроллера в зависимости от текущего представления

```
// Переключаем контроллеры представления
if blueViewController != nil
    && blueViewController!.view.superview != nil {
    yellowViewController.view.frame = view.frame
    switchViewController(from: blueViewController,
                        to: yellowViewController)
} else {
    blueViewController.view.frame = view.frame
    switchViewController(from: yellowViewController,
                        to: blueViewController)
}
}
```

Первая ветвь оператора if выполняется, если мы переключились с голубого представления на желтое, а ветвь else — если с желтого на голубое.

Помимо отказа от использования ресурсов для желтого представления и контроллера, если кнопка Switch Views никогда не была нажата, ленивая загрузка позволяет удалить из памяти представление, которое не будет показано на экране. Если объем свободной памяти опускается ниже установленного порога, система iOS вызовет метод didReceiveMemoryWarning() из класса UIViewController, который наследуется каждым контроллером представления.

Так как нам известно, что представление, загруженное в следующий раз, будет предъявлено пользователю, мы можем безопасно удалить контроллер, если он не связан с представлением, демонстрируемым на экране. Для этого в существующий метод didReceiveMemoryWarning() необходимо вставить строки, показанные в листинге 6.4.

Листинг 6.4. Безопасное освобождение памяти от ненужных контроллеров при сильных ограничениях на объем памяти

```
override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Освобождаем возобновляемые ресурсы

    if blueViewController != nil
        && blueViewController!.view.superview == nil {
        blueViewController = nil
    }
    if yellowViewController != nil
        && yellowViewController!.view.superview == nil {
        yellowViewController = nil
    }
}
```

Этот код проверяет, какое из представлений в данный момент демонстрируется пользователю, и удаляет контроллер другого представления, присваивая его свойству значение nil. В результате этот контроллер и представление, которым он управляет, будут удалены из памяти.

ПОДСКАЗКА. Ленивая загрузка — ключевой компонент управления ресурсами в системе iOS, и его следует применять всегда. В сложных приложениях с несколькими представлениями аккуратное обращение с памятью и удаление неиспользуемых объектов гарантирует безотказную работу, в противном случае приложение будет периодически давать сбои из-за исчерпания памяти.

Последний фрагмент мозаики — метод `switchViewController(from:, to:)`, отвечающий за переключение контроллера представления. Переключение контроллеров представления представляет собой двухэтапный процесс. Сначала необходимо удалить представление для текущего контроллера, а затем добавить представление для нового контроллера представления. Но это еще не все — необходимо выполнить уборку. Добавьте в программу реализацию метода, приведенного в листинге 6.5.

Листинг 6.5. Вспомогательный метод switchViewController

```
private func switchViewController(from fromVC: UIViewController?,
                                to toVC: UIViewController?) {
    if fromVC != nil {
        fromVC!.willMove(toParentViewController: nil)
        fromVC!.view.removeFromSuperview()
        fromVC!.removeFromParentViewController()
    }

    if toVC != nil {
        self.addChildViewController(toVC!)
        self.view.insertSubview(toVC!.view, at: 0)
        toVC!.didMove(toParentViewController: self)
    }
}
```

Первый блок кода удаляет текущий контроллер представления, но сначала мы разберем второй блок кода, в который добавим контроллер следующего представления. Первая строка этого блока имеет следующий вид:

```
self.addChildViewController(toVC!)
```

Этот код делает контроллер следующего представления дочерним классом контроллера переключающего представления. Контроллеры представления, такие как `SwitchingViewController`, управляющие другими контроллерами представлений, называются **контроллерами контейнерных представлений** (*container view controllers*). Стандартные классы `UITabBarController` и `UINavigationController` представляют собой контроллеры контейнерных представлений, а их код похож на код метода `switchViewController(from:, to:)`. Благодаря тому, что контроллер нового представления задается как дочерний класс класса `SwitchingViewController`, определенные события, поступающие в контроллер корневого представления, при необходимости корректно передаются дочернему контроллеру. Например, контроллер позволяет правильно обработать вращение.

Затем представление добавляется в объект класса `SwitchingViewController`:

```
self.view.insertSubview(toVC!.view, atIndex: 0)
```

Обратите внимание на то, что представление вставляется в список дочерних представлений класса `SwitchingViewController` с нулевым индексом, давая системе iOS указание вставить это представление позади всех остальных. Благодаря этому панель инструментов, созданная в среде Interface Builder, будет всегда видимой на экране, поскольку представление содержимого вставляется за ним. В заключение мы уведомляем контроллер следующего представления о том, что он был задан как дочерний класс другого контроллера.

```
toVC!.didMoveToParentViewController(self)
```

Это необходимо, если дочерний контроллер представления замещает этот метод для выполнения другой операции.

Теперь, когда мы показали, как добавляется контроллер представления, код, удаляющий контроллер представления, становится понятнее — мы просто выполняем операции в обратном порядке.

```
if fromVC != nil {
    fromVC!.willMoveToParentViewController(nil)
    fromVC!.view.removeFromSuperview()
    fromVC!.removeFromParentViewController()
}
```

Реализация представлений содержимого

Итак, код завершен, но пока мы не можем запустить приложение, потому что еще не включили контроллеры голубого и желтого представлений в раскадровку. Эти контроллеры чрезвычайно простые. Каждый из них содержит один метод действия, который выполняется при нажатии клавиши, и выходы совершенно не нужны. Голубое и желтое представления почти идентичны. Они настолько похожи, что их можно было бы реализовать с помощью одного и того же класса. Но мы создали два разных класса, потому что эта ситуация является типичной для большинства приложений с несколькими представлениями.

Два метода действия, которые мы намереваемся реализовать, просто выводят на экран предупреждение (см. главу 4). Добавим метод, представленный в листинге 6.6, в файл `BlueViewController.swift`.

Листинг 6.6. Нажатие кнопки на голубом контроллере вызывает предупреждение

```
@IBAction func blueButtonPressed(sender: UIButton) {
    let alert = UIAlertController(title: "Blue View Button Pressed",
                                 message: "You pressed the button on the blue view",
                                 preferredStyle: .alert)
    let action = UIAlertAction(title: "Yes, I did", style: .default,
                             handler: nil)
    alert.addAction(action)
    present(alert, animated: true, completion: nil)
```

Сохраните файл. Затем откройте файл YellowViewController.swift и добавьте в него метод из листинга 6.7.

Листинг 6.7. Нажатие кнопки на желтом контроллере также вызывает предупреждение

```
IBAction func yellowButtonPressed(sender: UIButton) {
    let alert = UIAlertController(title: "Yellow View Button Pressed",
        message: "You pressed the button on the yellow view",
        preferredStyle: .alert)
    let action = UIAlertAction(title: "Yes, I did", style: .default,
        handler: nil)
    alert.addAction(action)
    present(alert, animated: true, completion: nil)
```

Теперь откройте файл Main.storyboard в среде Interface Builder, чтобы внести в него несколько изменений. Сначала необходимо добавить новую сцену для класса BlueViewController. До сих пор все раскладовки, с которыми мы работали, содержали только одну пару “контроллер–представление”, но на самом деле они могут иметь несколько сцен. Перетащите из библиотеки объектов на область редактирования еще один элемент View Controller и оставьте его рядом с существующим. Теперь ваша раскладовка содержит две сцены, каждую из которых можно загружать динамически и независимо во время выполнения приложения. В строке пиктограмм, расположенной в верхней части новой сцены, щелкните на желтой пиктограмме View Controller и нажмите комбинацию клавиш <Option+⌘+3>, чтобы открыть инспектор идентичности. В разделе Custom Class атрибут Class по умолчанию имеет значение UIViewController; измените его на BlueViewController, как показано на рис. 6.20.

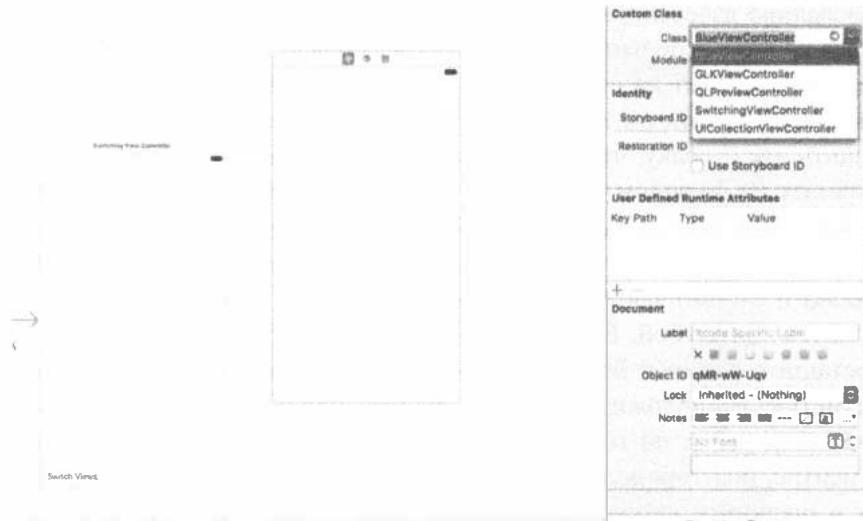


Рис. 6.20. Добавляем новый контроллер представления и связываем его с файлом класса BlueViewController

Необходимо также создать идентификатор для нового контроллера представления, чтобы наш код мог найти его в раскладовке. Под разделом *Custom Class* в окне инспектора идентичности находится поле *Storyboard ID*. Щелкните в нем и введите слово *Blue*, которое мы использовали в коде (рис. 6.21).

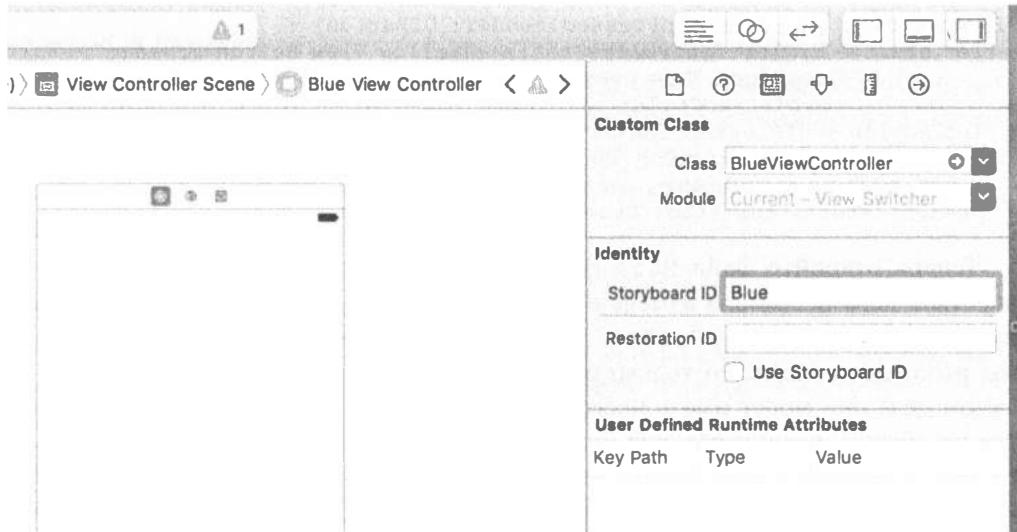


Рис. 6.21. Задаем идентификатор раскладовки в раскладовке контроллера голубого представления равным Blue

Итак, у нас есть две сцены. Ранее мы показали, как настроить приложение на загрузку раскладовки в ходе запуска, но мы ничего не рассказали о сценах. Откуда приложению известно, какое из двух представлений следует показать на экране? Ответ кроется в большой стрелке, указывающей на первую сцену (рис. 6.22). Эта строка указывает на сцену, заданную по умолчанию, с которой приложение начинает работу. Если вы хотите задать по умолчанию другую сцену, то должны перетащить эту стрелку, чтобы она указывала на желаемую сцену.

Щелкните на большом квадратном представлении на новой сцене, добавленной нами в раскладовку, а затем нажмите комбинацию клавиш *<Option+⌘+4>*, чтобы открыть инспектор атрибутов. В разделе инспектора *View* выберите цвет *Background* и с помощью селектора цвета измените цвет фона этого представления на темно-голубой. Если вам нравится этот цвет, закройте селектор цвета.

Перетащите элемент *Button* из библиотеки на представление, используя линии разметки, расположите кнопку в центре представления, выровняв ее как по вертикали, так и по горизонтали. Мы хотим, чтобы кнопка всегда оставалась в центре, поэтому должны задать соответствующие ограничения. Выберите кнопку и щелкните на кнопке *Align* под раскладовкой. Во всплывающем меню выберите команды *Horizontally in Container* и *Vertically in Container*, выберите в раскрывающемся списке *Update Frames* пункт *Item of New Constraints*, а затем

щелкните на кнопке Add 2 Constraints (рис. 6.23). Целесообразно изменить цвет фона на белый, чтобы выровнять элементы интерфейса, а затем, закончив работу, вернуть синий цвет. Кроме того, текст на кнопке следует сделать белым, чтобы он был виден на синем фоне.

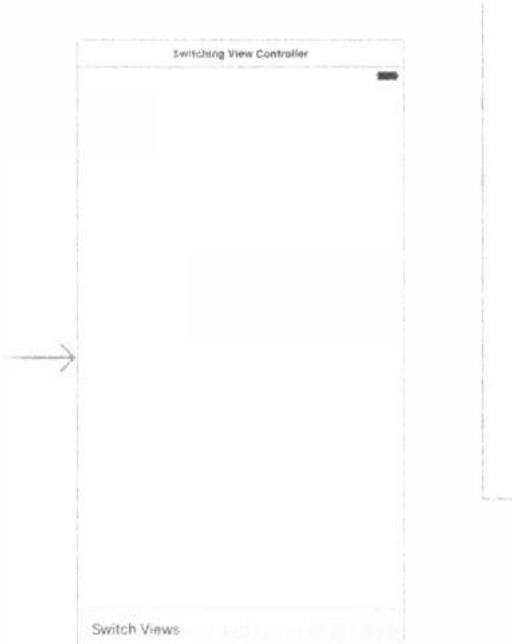


Рис. 6.22. Мы добавили вторую сцену. Большая стрелка указывает на сцену, заданную по умолчанию



Рис. 6.23. Выравнивание кнопки по центру представления

Дважды щелкните на кнопке и измените ее название на Press Me. Выбрав кнопку, перейдите в окно инспектора связей (нажав комбинацию клавиш `<Option+⌘+6>`), перетащите оттуда событие `Touch Up Inside` на желтую пиктограмму `View Controller`, расположенную в верхней части сцены, и соедините ее с методом действия `blueButtonPressedWithSender`.

Теперь необходимо сделать то же самое для представления `YellowViewController`. Перетащите элемент `View Controller` из библиотеки объектов в область редактирования. Если элементы сцен начнут тесниться, не беспокойтесь: каждая из сцен будет перекрывать другую, так что все будет в порядке! Щелкните на пиктограмме `View Controller` для новой сцены в окне `Document Outline` и с помощью инспектора идентичности измените ее класс на `YellowViewController`, а ее идентификатор `Storyboard ID` — на `Yellow`.

Затем выберите представление `YellowViewController` и перейдите в окно инспектора атрибутов. После этого щелкните на селекторе `Background`, выберите ярко-желтый цвет и закройте селектор цвета.

Перетащите объект `Button` из библиотеки на представление и с помощью линий разметки установите его в центре. Используя меню, установите ограничения по вертикали и по горизонтали, как мы делали для последней кнопки. После этого измените его заголовок на `Press Me, Too`. Выбрав кнопку, перейдите в окно инспектора связей, перетащите оттуда событие `Touch Up Inside` на желтую пиктограмму `View Controller`, расположенную в верхней части сцены, и соедините ее с методом действия `yellowButtonPressedWithSender`.

Закончив работу, сохраните раскладовку и приготовьтесь к тестированию приложения. Нажмите клавишу `Run` в среде Xcode. Ваше приложение должно запуститься и заполнить весь экран голубым цветом. При запуске приложение демонстрирует голубое представление. Когда пользователь нажмет кнопку `Switch Views`, произойдет переключение на желтое представление. Если он нажмет ее еще раз, то на экране отобразится голубое представление. Если пользователь нажмет кнопку, расположенную в центре голубого или желтого представления, на экране появится предупреждение, подтверждающее, что кнопка была нажата. Это предупреждение свидетельствует о том, что при выводе представления на экран был вызван правильный контроллер.

Тем не менее переход от одного представления к другому происходит довольно резко, поэтому нужно как-то сгладить эти переходы.

Анимация перехода

Класс `UIView` содержит несколько методов, с помощью которых можно указать, что переход следует анимировать, выбрать вид анимации и указать ее продолжительность.

Вернитесь к файлу `SwitchingViewController.swift` и замените метод `switchViews()` кодом, представленным в листинге 6.8.

Листинг 6.8. Модифицированный метод switchViews с анимацией

```

@IBAction func switchViews(sender: UIBarButtonItem) {
    // Создаем новый контроллер представления при необходимости
    if yellowViewController?.view.superview == nil {
        if yellowViewController == nil {
            yellowViewController =
                storyboard?.instantiateViewController(withIdentifier: "Yellow")
                as! YellowViewController
        }
    } else if blueViewController?.view.superview == nil {
        if blueViewController == nil {
            blueViewController =
                storyboard?.instantiateViewController(withIdentifier: "Blue")
                as! BlueViewController
        }
    }

    UIView.beginAnimations("View Flip", context: nil)
    UIView.setAnimationDuration(0.4)
    UIView.setAnimationCurve(.easeInOut)
    // Переключаем контроллеры представления
    if blueViewController != nil
        && blueViewController!.view.superview != nil {
        UIView.setAnimationTransition(.flipFromRight,
                                      for: view, cache: true)
        yellowViewController.view.frame = view.frame
        switchViewController(from: blueViewController,
                             to: yellowViewController)
    } else {
        UIView.setAnimationTransition(.flipFromLeft,
                                      for: view, cache: true)
        blueViewController.view.frame = view.frame
        switchViewController(from: yellowViewController,
                             to: blueViewController)
    }
    UIView.commitAnimations()
}

```

Скомпилируйте и выполните новый вариант. Теперь при нажатии кнопки **Switch Views** вместо мгновенной замены одного представления другим старое представление будет переворачиваться и заменяться новым.

Для того чтобы сообщить системе iOS, что мы хотим анимировать замену представлений, необходимо объявить **блок анимации** (*animation block*), используя класс `UIView`, и указать, как долго она должна продолжаться. Блоки анимации объявляются с помощью метода `presentViewController(_:animated:completion:)` из класса `UIView`:

```

UIView.PresentViewController("View Flip", context: nil)
UIView.setAnimationDuration(0.4)

```

Метод `presentViewController(_:animated:completion:)` получает два параметра. Первый из них представляет собой заголовок блока. Он используется

только в тех случаях, когда вы хотите непосредственно использовать технологию анимации Core Animation. Для наших целей достаточно задать этот параметр равным `nil`. Второй параметр — это указатель, позволяющий задать объект (или другой тип данных языка C), указатель которого мы хотели бы связать с блоком анимации. В данном случае мы использовали значение `nil`, поскольку ничего такого делать не хотим. Кроме того, мы задали продолжительность анимации, которая указывает классу `UIView`, как долго, в секундах, должна продолжаться анимация.

В заключение зададим **кривую анимации** (*animation curve*), определяющую скорость анимации. По умолчанию анимация с линейной кривой выполняется за постоянное время. Мы выбрали вариант `UIViewControllerAnimatedCurve.EaseInOut`, который означает, что анимация должна начинаться медленно, ускоряться в середине, а в конце снова замедляться. Это делает анимацию более естественной и менее механистичной.

```
UIView.setAnimationCurve(.easeInOut)
```

Далее нам необходимо указать вид перехода. На момент написания книги в системе iOS существовали пять видов перехода.

- ⌘ `UIViewControllerAnimatedTransition.flipFromLeft`
- ⌘ `UIViewControllerAnimatedTransition.flipFromRight`
- ⌘ `UIViewControllerAnimatedTransition curlUp`
- ⌘ `UIViewControllerAnimatedTransition(curlDown`
- ⌘ `UIViewControllerAnimatedTransition.none`

Мы выбрали два разных эффекта, соответствующих разным переключаемым представлениям. Левый поворот для одного переключения и правый поворот для другого имитируют ситуацию, когда представление будто опрокидывается назад и вперед. Значение `UIViewControllerAnimatedTransition.none` соответствует резкому переходу от одного контроллера представления к другому. Разумеется, если вас устраивает резкий переход, то создавать блок анимации вообще не нужно.

Использование кеша ускоряет анимацию, поскольку в начале анимации создается его снимок, который используется вместо прорисовки каждого этапа. При использовании анимации всегда следует использовать кеш, за исключением случаев, когда представление в ходе анимации не изменяется.

```
UIView.setAnimationTransition(.flipFromRight,
                             forView: view, cache: true)
```

Затем удалим текущее представление из представления контроллера, вместо того чтобы добавлять другое представление.

Когда закончим вносить изменения, обеспечивающие анимацию, вызовем метод `commitAnimations()` из класса `UIView`. Теперь все этапы выполне-

ния приложения от запуска до вызова метода `commitAnimations()` будут анимированы.

Благодаря тому, что каркас использует технологию Core Animation, мы можем реализовать сколь угодно сложную анимацию с помощью небольшого количества строк в программе.

Резюме

Создав такое приложение с несколькими представлениями “с нуля”, вы должны были получить ясное представление о его устройстве. Несмотря на то что среди Xcode содержат шаблоны проектов для большинства типичных приложений с несколькими представлениями, необходимо понимать общую структуру этих приложений, чтобы создавать их самостоятельно. Шаблонные контроллеры (`UITabBarController`, `UINavigationController` и `UIPageViewController`) очень экономят время, но иногда они просто не могут удовлетворить наши потребности.

В следующих главах мы продолжим создавать приложения с несколькими представлениями. В частности, в главе 7 мы создадим панель вкладок.

ГЛАВА 7

Панели вкладок и селекторы

В предыдущей главе вы создали свое первое приложение с несколькими представлениями. В этой главе мы сконструируем полноценное приложение с панелью вкладок с пятью различными вкладками и пятью различными представлениями содержимого. Создание этого приложения призвано закрепить материал, с которым вы ознакомились в главе 6. Мы воспользуемся этими пятью представлениями содержимого для демонстрации управляющего элемента системы iOS, с которым вы еще не сталкивались. Это — **представление селектора** (picker view) или просто **селектор** (picker). Возможно, вам не знакомо это название, но вы почти наверняка использовали селектор, если работали с устройствами iPhone или iPod более 10 минут. Селекторы представляют собой управляющий элемент с вращающейся шкалой. Они используются, например, для ввода дат в календаре или установки таймера в часах (рис. 7.1). В устройстве iPad представление селектора не столь распространено, так как больший дисплей допускает другие, более удобные способы выбора среди нескольких элементов, но даже там он используется в приложении Calendar.



Рис. 7.1. Селектор в приложении Clock

Селекторы представляют собой более сложные управляющие элементы системы iOS, чем те, с которыми вы сталкивались до сих пор, и как таковые заслуживают немного больше внимания. Селекторы могут быть настроены для отображения одной или нескольких шкал. По умолчанию они выводят текстовые элементы, но могут быть настроены и для вывода изображений.

Приложение Pickers

Приложение Pickers, создаваемое в этой главе, будет оснащено панелью вкладок. При создании приложения вы измените панель вкладок, заданную по умолчанию, так что она будет содержать пять вкладок, добавите пиктограммы к каждому элементу панели вкладок, а затем создадите ряд представлений содержимого и подключите каждое из них к вкладке.

Представления содержимого приложения будут оснащены пятью селекторами.

- ❖ **Селектор даты.** Первое представление содержимого, которое мы создадим, будет селектором даты, которое является селектором простейшего типа с точки зрения реализации (рис. 7.2). В представлении будет также кнопка, при нажатии которой появится сообщение о выбранной дате.



Рис. 7.2. На первой вкладке располагается селектор даты

Однокомпонентный селектор. Вторая вкладка оснащена селектором с единственным списком значений (рис. 7.3). Реализация этого селектора требует немного больше работы, чем селектор даты. Вы узнаете, как указывать значения, выводимые селектором, с помощью делегата и источника данных.



Рис. 7.3. Селектор выводит единственный список значений

- ❖ **Многокомпонентный селектор.** На третьей вкладке мы собираемся создать селектор с двумя отдельными колесами. Технический термин для каждого из этих колес — **компонент селектора** (picker component), так что, другими словами, здесь мы создаем селектор с двумя компонентами. Вы узнаете, как использовать источник данных и делегат, чтобы обеспечить два независимых списка данных для селектора (рис. 7.4). Каждый из компонентов этого селектора может быть изменен без воздействия на другой.
- ❖ **Селектор с зависимыми компонентами.** В четвертом представлении содержимого мы создадим еще один селектор с двумя компонентами. Но на этот раз значение, отображаемое в правом компоненте, будет меняться в зависимости от значения, выбранного в компоненте слева. В данном примере мы собираемся отображать в левом компоненте список штатов, а в правом — список почтовых индексов выбранного штата (рис. 7.5).

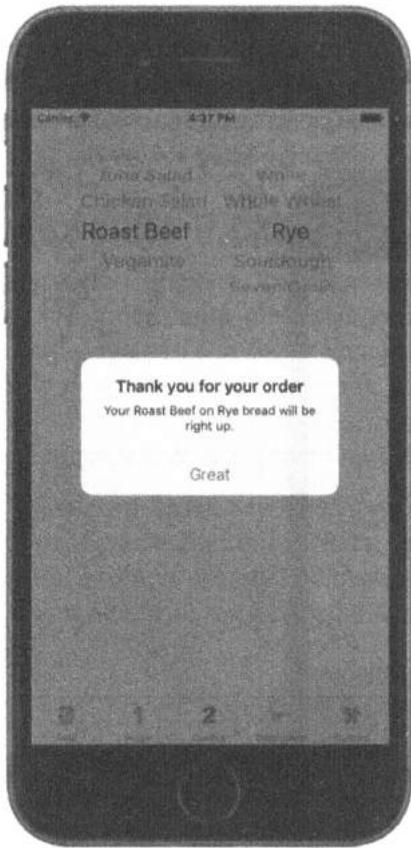


Рис. 7.4. Двухкомпонентный селектор, демонстрирующий сообщение о сделанном выборе



Рис. 7.5. В этом селекторе один компонент зависит от другого. При выборе штата в левом компоненте в правом компоненте соответствующим образом изменяется список почтовых кодов

- **Пользовательский селектор с изображениями.** С помощью пятого представления содержимого мы покажем, как добавить изображения в селектор, и сделаем это в виде небольшой игры, которая использует селектор с пятью компонентами. В нескольких местах в документации Apple селектор описывается как аналог игрового автомата. Следовательно, его можно использовать для создания маленького игрового автомата (рис. 7.6). Здесь пользователь не может изменять вручную значения компонентов, но будет иметь возможность с помощью кнопки Spin заставить пять колес выбрать новые, случайные значения. Если в строке окажутся три копии одного и того же изображения, пользователь выиграет.



Рис. 7.6. Пятый компонент — селектор, имитирующий игровой автомат

Делегаты и источники данных

Прежде чем приступить к созданию приложения, посмотрим, что же делает селекторы более сложными по сравнению с другими элементами управления, которые вы использовали до сих пор. За исключением селектора даты, невозможно просто выбрать селектор в библиотеке объектов, перенести в представление содержимого и настроить. Кроме того, необходимо обеспечить каждый селектор как **делегатом** селектора, так и **источником данных**.

Мы уже использовали делегаты приложений; их основная идея относится и к селекторам. Селектор передает часть работы делегату. Наиболее важной является задача определения того, что должно отображаться в каждой строке каждого из компонентов селектора. Селектор запрашивает у делегата строку либо представление, которое будет показано в определенном месте данного компонента. Селектор получает эти данные от делегатов.

В дополнение к делегату селекторы должны иметь источник данных. В этом случае термин *источник данных* применен не совсем верно. Источник данных говорит селектору, со сколькими компонентами он будет работать и сколько строк в каждом из его компонентов. Источник данных подобен делегату в том, что его методы вызываются в некоторые предопределенные моменты. Без источников данных и делегатов селекторы не могут делать свою работу; фактически они даже не могут быть нарисованы.

Достаточно распространенной является ситуация, когда источник данных и делегат представляют собой один и тот же объект, который хранится в файле Swift; нередко этот объект является контроллером представления, содержащего селектор. Именно этот подход мы используем в данном приложении. Контроллеры представлений для каждой из панелей нашего приложения будут источниками данных и делегатами для их селекторов.

ЗАМЕЧАНИЕ. Часто возникает вопрос: является ли источник данных селектора частью модели, представления или контроллера? Исходя из термина *источник данных* кажется, что он должен быть частью модели, но на самом деле это часть контроллера. Источником данных обычно является не объект, спроектированный для хранения данных. В простых приложениях источник данных может хранить данные, но истинная его работа заключается в получении данных из модели и их передаче селектору.

Создание приложения Pickers

Несмотря на то что среда Xcode предоставляет шаблон для приложений с панелью вкладок, мы собираемся создать наше “с нуля”. Это потребует не так уж много дополнительной работы, зато это очень хорошая практика.

Создайте новый проект, вновь выбрав шаблон Single View Application, и щелкните на кнопке **Next** на следующей странице экрана. Введите в поле **Product Name** имя **Pickers**. Убедитесь, что флагок **Use Core Data** сброшен, задайте значение **Language** равным **Swift** и выберите в списке **Devices** пункт **Universal**. Снова щелкните на кнопке **Next**. Программа Xcode предложит вам выбрать папку для хранения проекта.

Мы собираемся провести вас через процесс создания всего приложения, но если вы чувствуете в себе силы двигаться самостоятельно, сделайте это. Даже оказавшись в тупике, вы всегда сможете вернуться.

Создание контроллеров представлений

В предыдущей главе мы создали корневой контроллер представления (или, для краткости, корневой контроллер) для управления процессом обмена представлений приложения. Мы сделаем так же и на этот раз, но теперь мы не хотим создавать собственный класс корневого контроллера. Компания Apple предоставляет очень хороший класс для управления представлениями панели

вкладок, так что мы просто воспользуемся экземпляром UITabBarController в качестве корневого контроллера. Сначала нам надо создать пять новых классов в среде Xcode — пять контроллеров представления, которые будут переключать корневой контроллер. Раскройте папку Pickers в навигаторе проекта. Здесь отображаются исходные файлы, созданные программой Xcode для вашего проекта. Щелкните на папке Pickers и нажмите комбинацию клавиш **<⌘+N>** или выполните команду **File⇒New⇒File...**

Выберите на левой панели помощника по созданию нового файла пункты **iOS** и **Source**, затем выберите пиктограмму **Cocoa Touch Class** и щелкните на кнопке **Next**. На следующей странице экрана можно задать имя нового класса. Введите в поле **Class** строку **DatePickerViewController**. Поле **Subclass of** должно содержать имя **UIViewController**. Сбросьте флагок **Also create XIB file**, установите значение атрибута **Language** равным **Swift** и щелкните на кнопке **Next**.

В заключение на экране откроется окно, в котором можно выбрать папку для хранения классов. Выберите папку **Pickers**, которая уже содержит класс **AppDelegate** и несколько других файлов. Выберите также папку **Pickers** в списке **Group** и установите целевой флагок для проекта **Pickers**. После того как вы щелкнете на кнопке **Create**, в папке **Pickers** появится файл **DatePickerViewController.swift**.

Повторите эти шаги еще четыре раза, используя имена **SingleComponentPickerController**, **DoubleComponentPickerController**, **DependentComponentPickerController** и **CustomPickerController** (рис. 7.7). После щелчка на кнопке **Create** в папке **Pickers** появятся новые файлы.

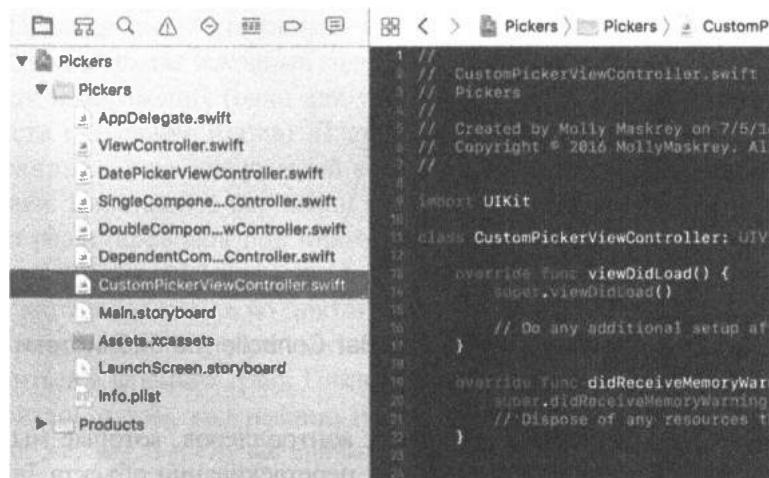


Рис. 7.7. После создания пяти классов контроллеров представлений окно навигатора проекта должно содержать все эти файлы

Создание контроллера представления панели вкладок

Мы собираемся создать контроллер представления панели вкладок. Шаблон проекта уже содержит контроллер представления `ViewController`, который является подклассом класса `UIViewController`. Для того чтобы превратить его в контроллер панели вкладок, необходимо изменить его базовый класс. Откройте файл `ViewController.swift` и вставьте в него строку, выделенную ниже полужирным шрифтом.

```
class ViewController: UITabBarController {
```

Теперь необходимо настроить контроллер панели вкладок в раскладовке, поэтому откройте файл `Main.storyboard`. Шаблон уже содержит начальный контроллер представления, который мы должны заменить, поэтому выберите его в окне `Document Outline` или в области редактирования и удалите его, нажав клавишу `<Delete>`. Найдите в библиотеке объектов элемент `Tab Bar Controller` и перетащите его в область редактирования (рис. 7.8).

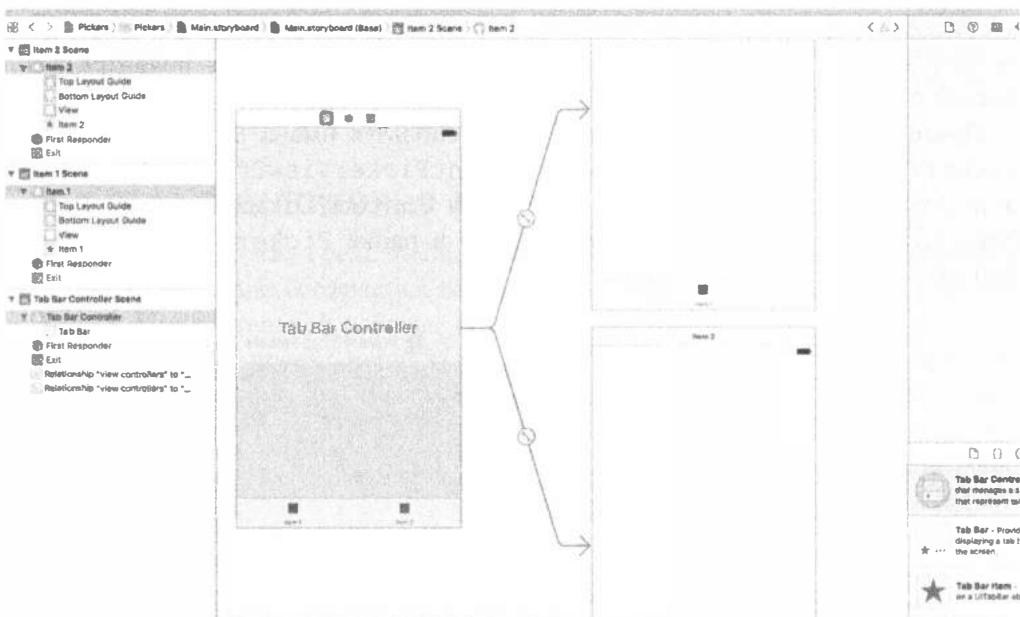


Рис. 7.8. Перетаскивание объекта `Tab Bar Controller` из библиотеки в область редактирования

Вы увидите, что, в отличие от других контроллеров, которые мы ранее перетаскивали из библиотеки объектов, при перетаскивании объекта `Tab Bar Controller` одновременно будут перемещаться три пары “представление–контроллер”, связанные между собой изогнутыми линиями. На самом деле это больше чем

обычный контроллер панели вкладок; у него есть два дочерних контроллера, уже связанных между собой и готовых к использованию.

После размещения контроллера панели вкладок в области редактирования в раскадровке появятся три новые сцены. Открыв окно структуры документа в левой части экрана, вы увидите подробный обзор сцен, содержащихся в раскадровке (рис. 7.8). Вы увидите, что изогнутые линии по-прежнему связывают контроллер панели вкладки с его дочерними представлениями. Эти линии всегда автоматически настраиваются при перемещении сцен. Положение каждой сцены в раскадровке не влияет на внешний вид приложения в ходе его работы.

Этот контроллер панели вкладок будет нашим корневым контроллером. Напомним, что корневой контроллер — это самое первое представление, которое видит пользователь, запустив приложение. Он отвечает за переключение других представлений. Поскольку мы будем связывать эти представления с кнопками панели вкладок, логично выбрать в качестве корневого именно контроллер панели вкладок. Нам необходимо сообщить системе iOS, что контроллер панели вкладок необходимо загружать из файла Main.storyboard при запуске приложения. Для этого выберите пиктограмму *Tab Bar Controller* в окне Document Outline и откройте окно инспектора атрибутов. Затем установите флагок *Is Initial View Controller* в разделе *View Controller*. Выделив контроллер представления, перейдите в окно инспектора идентичности и измените значение *Class* на *ViewController*.

На панели вкладок для представления каждой вкладки используются пиктограммы, поэтому мы должны также добавить пиктограммы, которые будем использовать, до того, как приступим к редактированию раскадровки для этого класса. Подходящие пиктограммы можно найти в архиве проектов в папке 07 – ImageSets в архиве исходных кодов. Каждая папка в папке 07 – ImageSets содержит три изображения (одно для устройств со стандартным экраном и два для устройств с экраном Retina). В окне навигатора проекта выберите папку Assets.xcassets, уже содержащий стандартное изображение для пиктограммы и заставки. Перетащите все папки из папки 07 – ImageSets в левый столбец области редактирования под пиктограмму *AppIcon* и скопируйте их в свой проект (рис. 7.9).

Если вы хотите создать свою пиктограмму, то должны придерживаться определенных правил. Используемые пиктограммы должны иметь размер 24×24 пикселя и храниться в формате .png. Пиктограммы должны иметь прозрачный фон. Не беспокойтесь о том, как именно пиктограммы будут выведены на панели вкладок. Так же, как и в случае значков приложения, iOS самостоятельно сделает все необходимые преобразования для того, чтобы пиктограмма выглядела корректно.

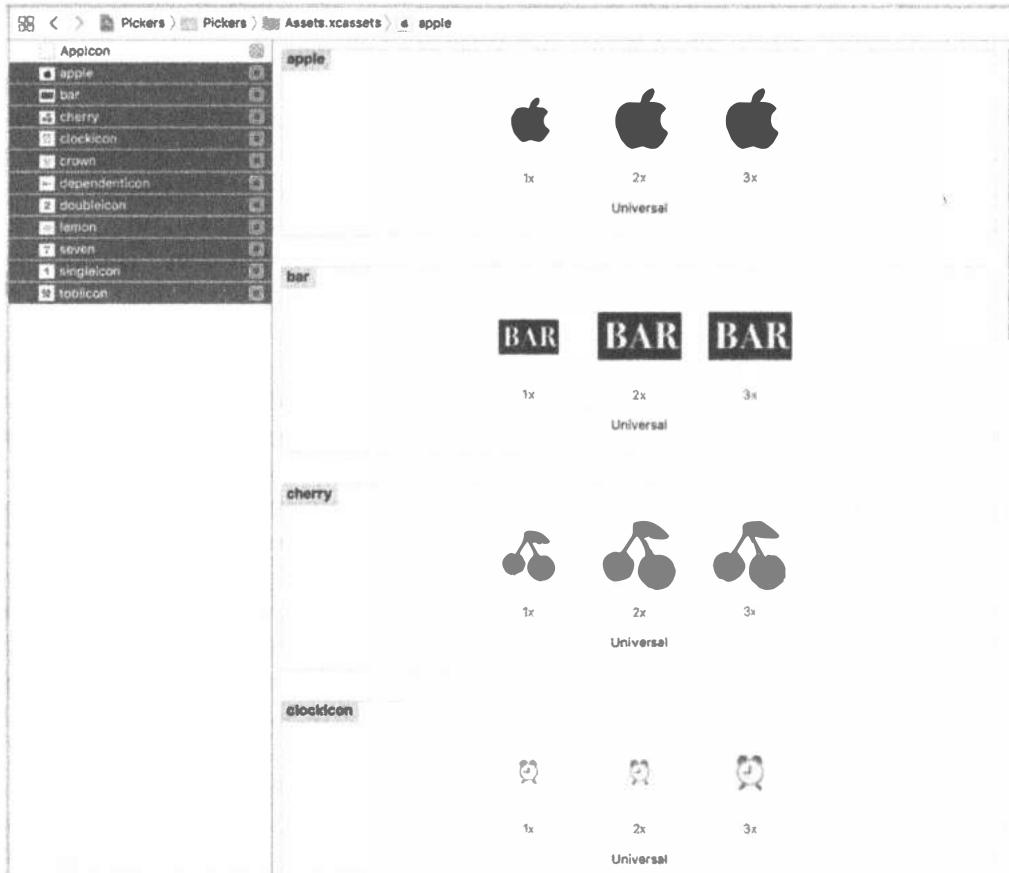


Рис. 7.9. Выделение изображений, расположенных ниже пиктограммы AppleIcon в каталоге Assets.xcassets в среде Xcode

ЗАМЕЧАНИЕ. Размер изображения 24×24 пикселя предназначен для обычных дисплеев; для дисплеев Retina на устройствах iPhone 4 и более поздних версий, а также для нового устройства iPad нужны изображения вдвое большего размера, иначе они будут выглядеть мозаично. Для устройства iPhone 6/6s необходимо изображение, которое в три раза больше оригинала. Это очень легко: любому изображению foo.png сопутствует изображение foo@2x.png, имеющее вдвое больший размер, а также изображение foo@3x.png, имеющее втрое больший размер. Инструкция UIImage(named: "foo") вернет изображение нормального или двойного размера, наилучшим образом подходящее для текущего приложения, выполняемого на устройстве.

Вернемся к раскладовке. Как видите, каждый из дочерних контроллеров представления имеет имя вроде “Item 1” в верхней части представления и одну панель в нижней части, а также простую метку, соответствующую названию

вкладки. Эти имена можно выбрать точнее, поэтому выберите контроллер представления **Item 1**, а затем щелкните на вкладке в нижней части представления или в окне **Document Outline**. Откройте окно инспектора атрибутов, и вы увидите текст для настройки атрибута **Title** элемента **Bar Item**, который в данный момент представляет собой строку **Item 1**. Замените этот текст строкой **Date** и нажмите клавишу **<Return>**. В результате текст вкладки в нижней части представления и в контроллере панели вкладок немедленно изменится. Оставаясь в окне инспектора атрибутов, щелкните на всплывающем меню **Image** и выберите пункт **clockicon**. Повторите все предыдущие этапы для второго контроллера представления, назвав его **Single** и использовав изображение **singleicon** для вкладки.

Теперь мы должны закончить контроллер панели с пятью вкладками (рис. 7.2). Каждая из этих пяти вкладок соответствует одному из селекторов. Мы сделаем это, просто добавив в раскладовку еще три контроллера представления (в дополнение к двум контроллерам, которые мы добавили вместе с контроллером панели вкладок) и соединив их между собой, чтобы контроллер панели вкладок мог их активизировать. Для начала перетащите обычный элемент **View Controller** из библиотеки объектов на раскладовку. Затем нажмите клавишу **<Control>**, перетащите указатель от контроллера панели вкладок к новому контроллеру представления, отпустите кнопку мыши и выберите контроллер представления из раздела **Relationship Segue** в маленьком всплывающем меню. Тем самым мы сообщаем контроллеру панели вкладок о том, что у него появился новый дочерний класс, поэтому панель вкладок немедленно запросит новый элемент, а новый контроллер представления получит элемент вкладки в нижней части своего представления, как и все остальные. То же самое необходимо сделать для последнего контроллера представления с заголовком **Double** и изображением **doubleicon**.

Перетащите еще два контроллера представления и соедините их с контроллером панели вкладок, как описано выше. Выберите по очереди каждую из их вкладок, назвав одну из них **Dependent** с изображением **dependenticon**, а другую **Custom** — с изображением **toolicon**. После этого у вас должен появиться один контроллер с вашей панелью инструментов и пятью связанными один с другим контроллерами представления, как показано на рис. 7.10.

Теперь зададим для каждого контроллера представления соответствующий класс контроллера. Это позволит обеспечить функциональные возможности для каждого из этих представлений. Выберите контроллер представления **Date** в окне **Document Outline** и откройте окно инспектора идентичности. В разделе **Custom Class** измените имя класса на **DatePickerViewController** и нажмите клавишу **<Return>** или **<Tab>**, чтобы подтвердить изменения (рис. 7.11).

Повторите этот же процесс для следующих четырех контроллеров представлений. В инспекторе атрибутов убедитесь в корректности установки флажков для каждого из них и введите имена **SingleComponentPickerViewController**, **DoubleComponentPickerViewController**, **DependentComponentPickerViewController** и **CustomPickerController** соответственно. Сохраните файл раскладовки.

258 ГЛАВА 7 ■ ПАНЕЛИ ВКЛАДОК И СЕЛЕКТОРЫ

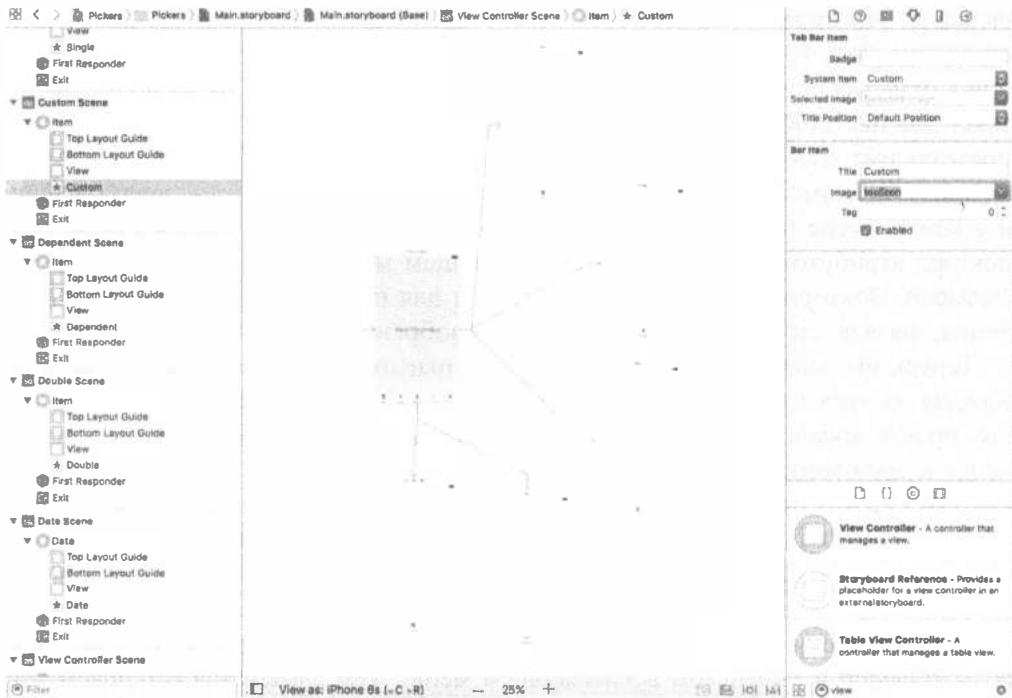


Рис. 7.10. Добавление пяти контроллеров представлений, доступ к которым обеспечивает панель инструментов из корневого контроллера представления

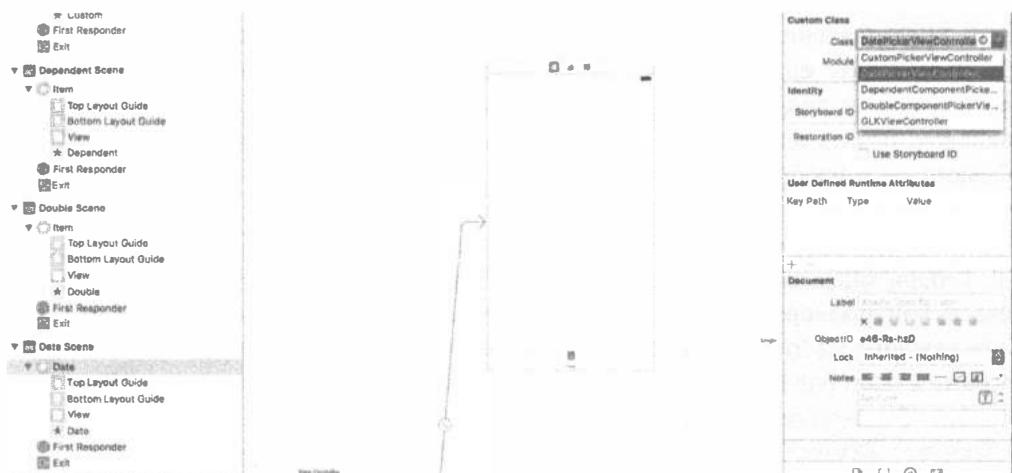


Рис. 7.11. Связывание представления Date с его контроллером

Первичный тест на симуляторе

В данный момент панели вкладок и представления содержимого должны быть работоспособными. Вернитесь в Xcode, выполните компиляцию и запуск,

и ваше приложение должно заработать, выведя панель функционирующих вкладок (рис. 7.12). Щелкните по очереди на каждой из вкладок. Каждая из них должна реагировать на выбор.

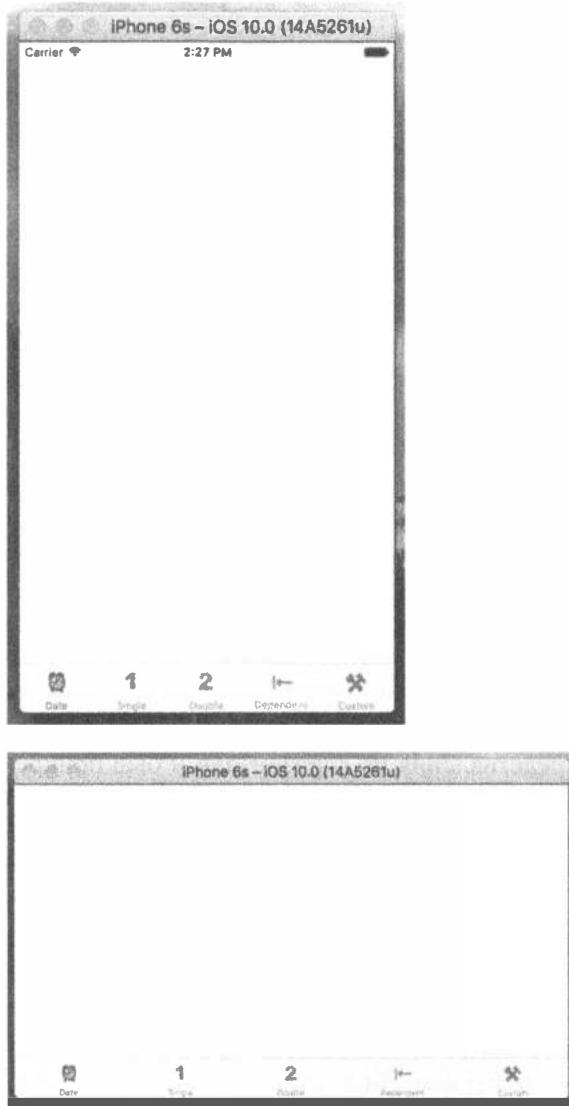


Рис. 7.12. Приложение с пятью пустыми, но работоспособными вкладками

В представлениях содержимого в настоящий момент пусто. Но если все прошло нормально, то базовый каркас приложения с несколькими представлениями настроен и работает, и мы можем приступить к разработке отдельных представлений содержимого.

ПОДСКАЗКА. Если после щелчка на одной из вкладок ваш симулятор даст сбой, то, скорее всего, вы либо пропустили какой-то шаг, либо сделали опечатку. Вернитесь назад и убедитесь в корректности всех соединений и правильности ввода всех имен классов.

Если хотите дважды убедиться, что все работает, добавьте другие метки или некоторые другие объекты к каждому представлению содержимого и вновь запустите приложение. При этом вы должны увидеть изменившееся содержимое представлений при селекторе разных вкладок.

Реализация селектора даты

Для реализации селектора даты нам требуются один выход и одно действие. Выход будет использован для получения значения от селектора даты. Действие будет запускаться кнопкой и выводить сообщение о выбранной дате. Мы создадим выход и действие с помощью программы Interface Builder, поэтому выберите файл Main.storyboard в окне навигатора проекта.

Найдите в библиотеке объектов элемент Date Picker и перетащите его в область редактирования на сцену Date Scene. Щелкните на пиктограмме Date в окне Document Outline, чтобы требуемый контроллер представления оказался на переднем плане, затем перетащите селектор даты из библиотеки объектов в верхнюю часть представления, вплотную к верхнему краю. Он может перекрывать строку состояния, потому что этот элемент управления содержит вертикальные заполнители.

Теперь применим ограничения Auto Layout, чтобы селектор даты правильно отображался на экране любого устройства. Мы хотим, чтобы селектор даты находился в центре представления по горизонтали и был прикреплен к его верхнему краю. Мы также хотим, чтобы его размер соответствовал его содержимому. Следовательно, нам нужны три ограничения. Выделив селектор даты, выберите команду Editor⇒Size to Fit в меню Xcode. Если эта команда заблокирована, то щелкните на кнопке Align в нижней части раскладовки, установите флажок Horizontally in Container, а затем щелкните на кнопке Add 1 Constraint. Щелкните на кнопке Pin (следующей за кнопкой Align). Используя четыре поля редактирования в верхней части всплывающего меню, задайте расстояние от селектора до верхнего края представления равным нулю, а затем щелкните на красной пунктирной линии под ним, чтобы она стала сплошной. В нижней части всплывающего окна установите значение Update Frames равным Items of New Constraints и щелкните на кнопке Add 1 Constraint. Размер селектора даты изменится, а сам он переместится на правильную позицию (рис. 7.13).

Щелкните на селекторе даты, если он еще не выбран, и вернитесь к инспектору атрибутов. Как показано на рис. 7.14, имеется ряд атрибутов, которые можно настроить для селектора даты. Мы оставим для большинства атрибутов их значения по умолчанию (но не бойтесь экспериментировать со значениями

по окончании работы над приложением, чтобы увидеть, что они делают). Единственное, что мы сделаем, — это ограничим диапазон селектора разумными сроками. Взгляните на заголовок **Constraints** и установите флажок **Minimum Date**, оставив значение по умолчанию 1/1/1970. Установите также флажок **Maximum Date** и значение 12/31/2200.

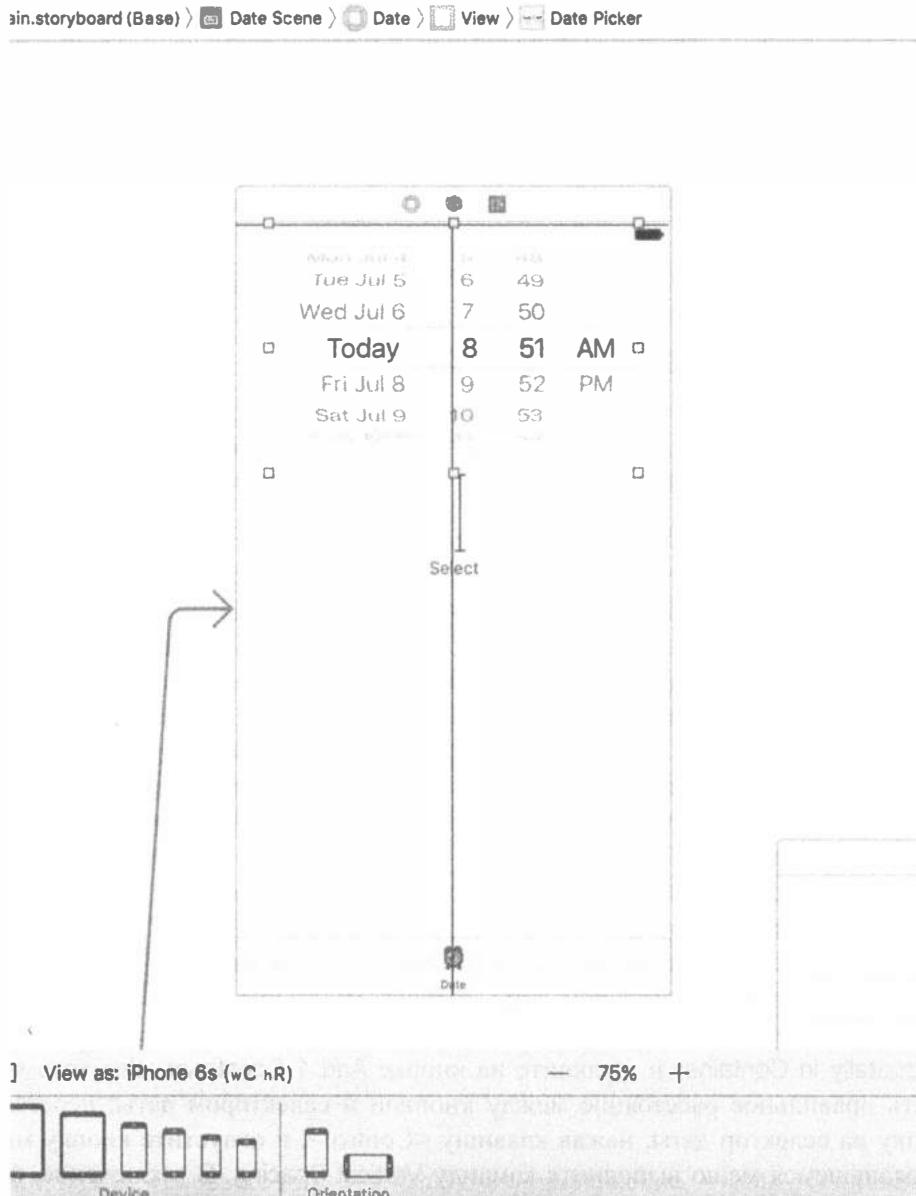


Рис. 7.13. Селектор даты, расположенный в верхней части представления

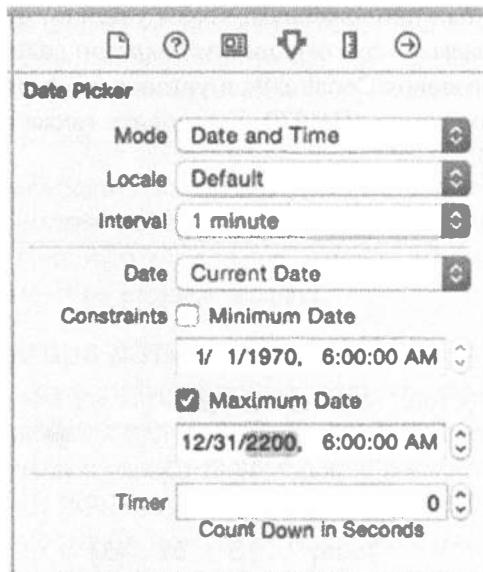


Рис. 7.14. Инспектор атрибутов для селектора даты. Мы устанавливаем минимальную и максимальную даты, оставляя для других атрибутов значения по умолчанию

Теперь свяжем этот селектор с его контроллером. Нажмите комбинацию клавиш `<Option+⌘+Enter>`, чтобы открыть окно помощника редактора, и установите панель быстрого перехода в верхней части в режим Automatic. В результате откроется файл `DatePickerViewController.swift`. Нажмите клавишу `<Control>` и перетащите указатель от селектора к пустой строке между объявлением класса и методом `viewDidLoad()` и отпустите кнопку мыши, когда появится подсказка `Insert Outlet`, `Action` или `Outlet Collection`. В появившемся всплывающем окне установите значение `Connection` равным `Outlet`, введите в поле `Name` строку `datePicker`, а затем нажмите клавишу `<Return>`, чтобы создать выход и соединить его с селектором.

Захватите элемент `Button` в библиотеке и поместите его немного ниже селектора даты. Дважды щелкните на кнопке и назовите ее `Select`. Мы хотим, чтобы эта кнопка находилась в центре представления по горизонтали и на фиксированном расстоянии под селектором даты. Выбрав эту кнопку, еще раз щелкните на кнопке `Align`, расположенной в нижней части раскладовки, установите флажок `Horizontally in Container` и щелкните на кнопке `Add 1 Constraint`. Для того чтобы задать правильное расстояние между кнопкой и селектором даты, перетащите кнопку на селектор даты, нажав клавишу `<Control>`, и отпустите кнопку мыши. В появившемся меню выполните команду `Vertical Spacing`. В заключение, щелкните на кнопке `Resolve Auto Layout Issues` в нижней части раскладовки, а затем на кнопке `Update Frames` в верхней части всплывающего окна (если этот элемент недоступен, значит, ваша кнопка уже находится в правильной позиции).

Кнопка должна переместиться в правильную позицию, и предупреждения Auto Layout больше появляться не должны.

Убедитесь, что файл DatePickerViewController.swift по-прежнему остается видимым в окне помощника редактора; если это не так, перейдите на панель быстрого перехода и с помощью элемента Manual найдите и откройте этот файл. Нажмите клавишу <Control> и перетаскивайте указатель от кнопки к строке над закрывающей фигурной скобкой в конце класса в окне помощника, пока не увидите подсказку Insert Outlet, Action и Outlet Collection. Измените значение Connection type на Action, присвойте новому действию имя onBbuttonPressed и нажмите клавишу <Return>. В результате возникнет пустой метод onButtonPressed(), который необходимо заполнить кодом, представленным в листинге 7.1.

Листинг 7.1. Код действия при выборе кнопки

```
@IBAction func onButtonPressed(_ sender: UIButton) {
    let date = datePicker.date
    let message = "The date and time you selected is \(date)"
    let alert = UIAlertController(
        title: "Date and Time Selected",
        message: message,
        preferredStyle: .alert)
    let action = UIAlertAction(
        title: "That's so true!",
        style: .default,
        handler: nil)
    alert.addAction(action)
    present(alert, animated: true, completion: nil)
}
```

В методе viewDidLoad() мы создаем новый объект класса NSDate. Созданный таким образом объект NSDate содержит текущие дату и время. Затем мы передаем эту дату для datePicker, что гарантирует, что всякий раз при загрузке представления из раскладовки селектор будет сбрасываться к текущим дате и времени (листинг 7.2).

Листинг 7.2. Настройка даты в методе viewDidLoad()

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

    // Дополнительная настройка после загрузки представления.
    let date = NSDate()
    datePicker.setDate(date as Date, animated: false)
}
```

Вот и все. Теперь создадим и запустим приложение, чтобы убедиться, что селектор даты работает правильно. Если все получилось как надо, ваше приложение при работе должно выглядеть так, как показано на рис. 7.2. При выборе кнопки Select появится предупреждение с выбранными датой и временем.

ЗАМЕЧАНИЕ. Селектор даты не позволяет указать секунды или часовой пояс. Предупреждение же отображает время с секундами и по Гринвичу (GMT). Мы могли бы добавить код для упрощения отображаемой строки, но не слишком ли большой станет наша и так немаленькая глава? Если вас интересует форматирование даты, обратитесь к классу `NSDateFormatter`.

Реализация однокомпонентного селектора

Наш следующий селектор позволяет пользователю выбирать значение из списка. В этом примере мы создадим массив для хранения значений, которые будут выводиться селектором. Селекторы сами по себе не хранят никаких данных. Вместо этого для получения выводимых данных они вызывают методы своего источника данных и делегат. Селектору совсем не интересно, где и в каком виде хранятся используемые им данные. Он запрашивает данные тогда, когда они ему нужны, и источник данных, и делегат (которые на практике часто представляют собой один и тот же объект) вместе предоставляют ему эти данные. В результате данные могут поступать, в частности, из статического списка. Данные могут быть также считаны из файла или URL и даже вычисляться “на лету”.

Для того чтобы класс селектора запрашивал данные у контроллера, необходимо, чтобы в контроллере были реализованы соответствующие методы. Для этого нужно объявить в определении класса контроллера, что он будет реализовывать ряд протоколов. Перейдите в окно навигатора проекта и щелкните на файле `SingleComponentPickerController.swift`. Наш класс контроллера будет действовать и как источник данных, и как делегат селектора, поэтому он должен поддерживать оба протокола для этих ролей. Добавьте в программу следующий код:

```
class SingleComponentPickerController: UIViewController,
    UIPickerViewDelegate, UIPickerViewDataSource {
```

После этого вы увидите в окне редактирования сообщение об ошибке. Это объясняется тем, что мы пока не реализовали требуемые методы протокола. Вскоре мы это сделаем, поэтому пока эти сообщения можно игнорировать.

Создание представления

Выберите файл `Main.storyboard`, чтобы отредактировать представление содержимого на второй вкладке на панели вкладок. Перейдите в окно `Document Outline` и щелкните на пиктограмме `Single`, чтобы контроллер представления вышел на передний план области редактирования. Перетащите из библиотеки объект `Picker View` (рис. 7.15) и добавьте его в верхнюю часть вашего представления, как вы это делали с представлением селектора даты.

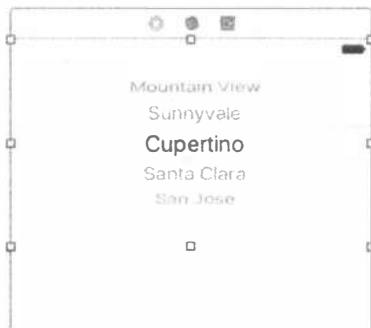


Рис. 7.15. Перетащите объект Picker View из библиотеки на второе представление

Селектор должен быть отцентрован по горизонтали и прикреплен к верхнему краю сцены. Для этого можно задать для селектора те же самые ограничения Auto Layout, которые были добавлены в приложение Date Picker в предыдущем примере. Кроме того, мы хотим, чтобы размер селектора соответствовал его содержанию, поэтому нам нужны три ограничения. Выделив селектор даты, выберите команду **Editor**⇒**Size to Fit** в меню Xcode. Если эта команда заблокирована, то щелкните на кнопке Align в нижней части раскладовки, установите флажок **Horizontally in Container**, а затем щелкните на кнопке **Add 1 Constraint**. Щелкните на кнопке **Pin** (следующей за кнопкой Align). Используя четыре поля редактирования в верхней части всплывающего меню, задайте расстояние от селектора до верхнего края представления равным нулю, а затем щелкните на красной пунктирной линии под ним, чтобы она стала сплошной. В нижней части всплывающего окна установите значение **Update Frames** равным **Items of New Constraints** и щелкните на кнопке **Add 1 Constraint**. Размер селектора даты изменится, а сам он переместится на правильную позицию (рис. 7.16).

Теперь соединим селектор с контроллером. Эта процедура напоминает предыдущую: откройте окно помощника редактора и настройте панель быстрых переходов на демонстрацию файла с расширением `SingleComponentPickerViewController.swift`. Нажмите клавишу **<Control>**, перетащите указатель от селектора к верхней части класса `SingleComponentPickerViewController` и создайте выход `singlePicker`.

Далее, выбрав селектор, нажмите комбинацию клавиш **<Option+⌘+6>**, чтобы вызвать инспектор связей. Если вы посмотрите на связи, доступные представлению селектора, то увидите, что первыми двумя элементами являются `dataSource` и `delegate`. Если вы их не видите, проверьте, выбран ли селектор! Нажмите клавишу **<Control>** и перетащите указатель от кружка, расположенного рядом с элементом `dataSource`, к пиктограмме `View Controller` в верхней части сцены раскладовки или панели `Document Outline` (рис. 7.17), а затем еще раз — от кружка, расположенного рядом с элементом `delegate`, к пиктограмме `View Controller`. Теперь данный селектор знает, что экземпляр класса `SingleComponent`

266 ГЛАВА 7 ■ ПАНЕЛИ ВКЛАДОК И СЕЛЕКТОРЫ

PickerViewController в раскладовке является его источником данных и легатом, и будет запрашивать у него выводимые данные. Иначе говоря, когда селектору потребуется информация о выводимых данных, он запросит ее у экземпляра SingleComponentPickerController, который управляет этим представлением для данной информации.

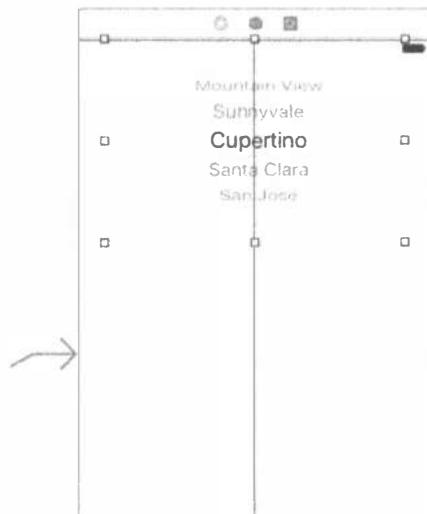


Рис. 7.16. Селектор, расположенный в верхней части представления

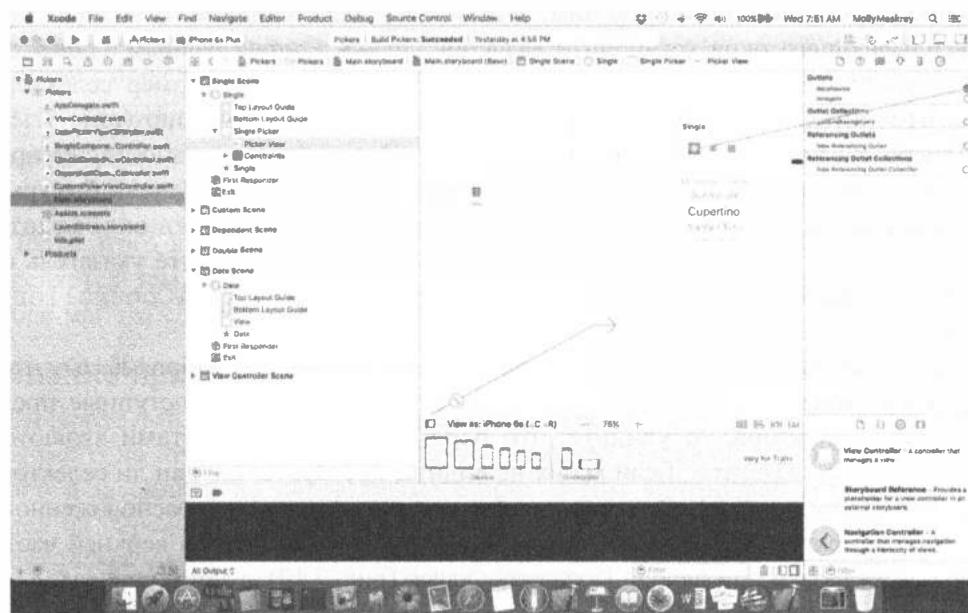


Рис. 7.17. Связывание источника данных с контроллером представления

Перетащите кнопку на представление и разместите ее непосредственно под селекторомом. Дважды щелкните на ней и дайте ей имя *Select*. Перейдите в окно инспектора связей, нажмите клавишу <Control>, перетащите указатель от кружка, расположенного рядом с событием *Touch Up Inside*, в код, показанный в окне помощника редактора, и отпустите кнопку мыши над закрывающей фигурной скобкой в конце определения класса, чтобы создать новый метод действия. Присвойте этому пустому методу имя *onButtonPressed*, и увидите, что программа Xcode автоматически его заполнит. На этом построение графического пользовательского интерфейса для второй вкладки завершено. Кнопка должна быть отцентрована по горизонтали и находиться на фиксированном расстоянии под селектором. Щелкните на кнопке *Align* в нижней части раскладовки, установите флажок *Horizontally in Container*, а затем щелкните на кнопке *Add 1 Constraint* (рис.7.18).

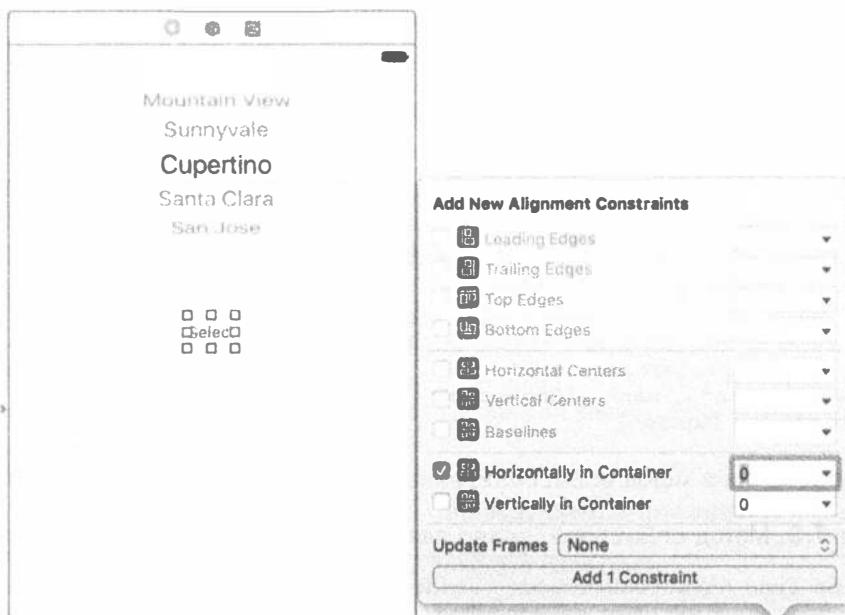


Рис. 7.18. Центрование кнопки по горизонтали относительно представления

Для того чтобы задать правильное расстояние между кнопкой и селектором даты, перетащите кнопку на селектор даты, нажав клавишу <Control>, и отпустите кнопку мыши. В появившемся меню выполните команду *Vertical Spacing* (рис. 7.19). Щелкните на кнопке *Resolve Auto Layout Issues* в нижней части раскладовки, а затем на кнопке *Update Frames* в верхней части всплывающего окна (если этот элемент недоступен, значит, ваша кнопка уже находится в правильной позиции). Кнопка должна переместиться в правильную позицию, и предупреждения Auto Layout больше появляться не должны.

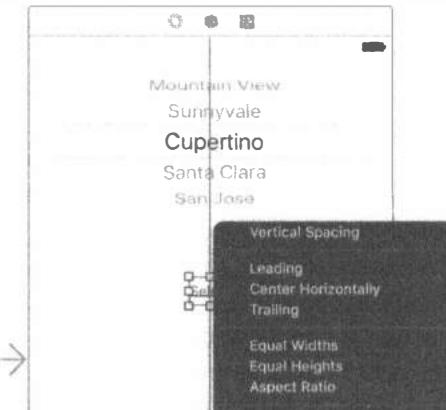


Рис. 7.19. Настройка расстояния по вертикали между кнопкой и селектором

Реализация контроллера как источника данных и делегата

Для того чтобы контроллер корректно работал в качестве источника данных и делегата, начнем с кода, с которым вы знакомы и который не должен представлять для вас никаких трудностей, а затем добавим несколько методов, с которыми вы ранее не сталкивались.

Щелкните на файле SingleComponentPickerController.swift и добавьте следующий код в его начало:

```
@IBOutlet weak var singlePicker: UIPickerView!
private let characterNames = [
    "Luke", "Leia", "Han", "Chewbacca", "Artoo",
    "Threepio", "Lando"]
```

Затем добавьте в метод onPressed() код, представленный в листинге 7.3.

Листинг 7.3. Метод onPressed для представления контроллера

```
@IBAction func onPressed(_ sender: UIButton) {
    let row = singlePicker.selectedRow(inComponent: 0)
    let selected = characterNames[row]
    let title = "You selected \(selected)!"

    let alert = UIAlertController(
        title: title,
        message: "Thank you for choosing",
        preferredStyle: .alert)
    let action = UIAlertAction(
        title: "You're welcome",
        style: .default,
        handler: nil)
    alert.addAction(action)
    present(alert, animated: true, completion: nil)
}
```

Как было показано ранее, селектор Date Picker содержит все необходимые данные, но обычный селектор поручает эту работу делегату и источнику данных. Следовательно, метод onPressed() должен спросить у селектора, какая строка выбрана, а затем попросить данные об этой строке у нашего массива pickerData. Вот как мы запрашиваем информацию о выбранной строке:

```
let row = singlePicker.selectedRow(inComponent: 0)
```

Обратите внимание на то, что мы должны указать компонент, о котором хотим получить информацию. В данном случае у селектора только один компонент (вращающаяся шкала), поэтому просто передаем в метод значение 0, являющееся индексом первого (и единственного) компонента.

В объявлении класса мы создаем массив с несколькими символьными строками, так что у нас есть данные для поставки селектору. Обычно ваши данные будут поступать из других источников типа списка свойств или запроса веб-службы. Встраивая массив элементов в код, как это сделано здесь, мы усложняем задачу его обновления или перевода приложения на другие языки. Но такой подход представляет собой быстрый и простой способ размещения данных в массиве в демонстрационных целях. Даже если вы обычно не создаете такие массивы, как здесь, вы все равно настраиваете доступ к данным в методе viewDidLoad(), чтобы не искать их на диске или в сети каждый раз, когда селектор их запросит.

ПОДСКАЗКА. Если вы не планируете создавать массивы из списков объектов в вашем коде, как мы только что сделали в viewDidLoad(), то как вы должны поступить? Вставить списки в файлы списка свойств и добавить эти файлы в свой проект. Файлы списков свойств могут изменяться без перекомпиляции исходного кода, а это означает отсутствие риска внесения новых ошибок. Можно также указать различные версии списков для разных языков, как это будет сделано в главе 22. Списки свойств более подробно описываются в главе 13.

Наконец вставьте в конец файла код, приведенный в листинге 7.4.

Листинг 7.4. Методы источника данных и делегата селектора

```
// MARK:-
// MARK: Picker Data Source Methods

func numberOfComponents(in pickerView: UIPickerView) -> Int {
    return 1
}
func pickerView(_ pickerView: UIPickerView,
               numberOfRowsInComponent component: Int) -> Int {
    return characterNames.count
}

// MARK: Picker Delegate Methods
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
               forComponent component: Int) -> String? {
    return characterNames[row]
}
```

Эти три метода необходимы для реализации селектора. Первые два метода взяты из протокола `UIPickerViewDataSource`, и оба они необходимы для всех наших селекторов (за исключением селектора даты). Вот первый из них:

```
func numberOfComponents(in pickerView: UIPickerView) -> Int {
    return 1
}
```

Селекторы могут иметь более одной вращающейся шкалы, или компонента, и с помощью этого метода селектор запрашивает, сколько компонентов он должен выводить. В данном случае мы хотим отображать только один список, поэтому возвращаем значение 1. Обратите внимание: в качестве параметра передается объект класса `UIPickerView`. Этот параметр указывает представление селектора, которое выполняет запрос, что позволяет иметь несколько селекторов с одним и тем же источником данных. В данном случае мы знаем, что у нас есть только один селектор, поэтому можем игнорировать этот аргумент (мы и так знаем, какой селектор нас запрашивает).

Второй метод источника данных используется селектором для запроса о количестве строк данных для данного компонента:

```
func pickerView(_ pickerView: UIPickerView,
               numberOfRowsInComponent component: Int) -> Int {
    return characterNames.count
}
```

Итак, мы узнали, какое представление селектора выполняет запрос и о каком компоненте оно спрашивает. Так как мы и так знаем, что у нас есть только один селектор, состоящий из единственного компонента, нас не интересуют значения передаваемых аргументов, и мы просто возвращаем количество объектов в нашем единственном массиве данных.

После этих двух методов источника данных мы реализуем один метод делегата. В отличие от методов источника данных, все методы делегата не являются обязательными. Термин *необязательный* немного некорректен, поскольку вам нужно реализовать по крайней мере один метод делегата. Обычно реализуют тот метод, который мы реализуем сейчас и здесь. Однако, если вы хотите выводить в селекторе что-то иное, а не текст, необходимо реализовать другой метод, — об этом мы поговорим, когда доберемся до пользовательского селектора.

```
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
               forComponent component: Int) -> String? {
    return characterNames[row]
}
```

В этом методе селектор просит нас предоставить данные для конкретной строки конкретного компонента. Мы получаем указатель на запрашивающий селектор, а также компонент и строку, для которых требуются данные. Поскольку наше представление имеет единственный селектор с единственным компонентом, будем игнорировать все, за исключением аргумента, указывающего строку, и использовать его для возвращения соответствующего элемента из нашего массива данных.

ЧТО ТАКОЕ ДИРЕКТИВА MARK?

Обратили ли вы внимание на следующие строки кода из файла:

```
// MARK:-  
// MARK: Picker Data Source Methods
```

Любая строка кода, которая начинается символами //, является комментарием. Комментарии, начинающиеся символами // MARK:, представляют собой директиву, которая указывает программе Xcode на необходимость изменения выпадающего меню методов и функций на панели редактора. Первая директива помещает в меню разделитель, а вторая создает текстовую запись, в которую вносится текст, содержащийся в оставшейся части строки директивы и который можно рассматривать как описывающий заголовок для группы методов в исходном коде. Некоторые из ваших классов, особенно классы контроллеров, могут оказаться довольно длинными, и контекстное меню с методами и функциями существенно облегчает навигацию по коду. Аккуратное применение директивы // MARK: — логичная организация кода позволяют использовать контекстное выпадающее меню гораздо более эффективно.

Скомпилируйте и запустите приложение снова. Затем переключитесь на вторую вкладку, помеченную **Single**, и проверьте, как работает новый пользовательский селектор, который должен выглядеть так, как показано на рис. 7.3.

Вернитесь в среду Xcode, и мы покажем, как реализовать селектор с двумя компонентами. Если считаете, что разобрались в последнем селекторе, то у вас есть отличный шанс убедиться в этом, создав селектор с двумя компонентами самостоятельно. Вы уже видели все необходимые методы, которые требуются для этого селектора, так что можете идти вперед и пробивать эту стену. Мы пока подождем вас здесь. Можете начать с рассмотрения рис. 7.4, чтобы освежить все в памяти. Когда закончите, читайте дальше, чтобы узнать, как эту проблему решали мы.

Реализация многокомпонентного селектора

Следующим мы рассмотрим селектор с двумя независимыми один от другого компонентами. Левая вращающаяся шкала содержит названия начинок бутерброда, а правая — сорта хлеба. Мы напишем такие же источник данных и делегат, как и для однокомпонентного селектора. Все, что нам надо, — это написать небольшой дополнительный код в некоторые из методов, чтобы они возвращали корректные значения и количество строк для каждого из компонентов. Щелкните на файле `DoubleComponentPickerController.swift` и добавьте следующий код:

```
class DoubleComponentPickerController: UIViewController,  
    UIPickerViewDelegate, UIPickerViewDataSource {
```

Здесь мы просто подтверждаем, что наш класс является контроллером как делегата, так и источника данных. Сохраните его и щелкните на файле Main.storyboard для работы над графическим пользовательским интерфейсом.

Создание представления

Выберите пиктограмму Double Scene в окне Document Outline и щелкните на пиктограмме View Controller, чтобы контроллер представления вышел на первый план области редактирования. Теперь добавьте представление селектора и кнопку. Измените метку кнопки на Select, а затем установите необходимые связи. Поскольку контроллеры представления в нашем приложении с точки зрения связей в раскладовке совершенно идентичны, мы не будем приводить детальную пошаговую инструкцию — за указаниями вы можете обратиться к предыдущему разделу. Ниже кратко описано, что вам нужно сделать.

1. Создайте выход doublePicker в расширении класса DoubleComponentPickerController, чтобы соединить контроллер с селектором.
2. Установите связь между элементами dataSource и delegate на представлении селектора, используя инспектор связей.
3. Установите связь между событием кнопки Touch Up Inside и новым методом действия onButtonPressed в контроллере представления, используя инспектор связей.
4. Добавьте в селектор и кнопку ограничения Auto Layout, фиксирующие их на месте.

Сохраните и закройте раскладовку, прежде чем перейти к работе с кодом. Сделайте закладку на этой странице, так как вам еще придется к ней вернуться.

Реализация контроллера

Выберите файл DoubleComponentPickerController.swift и добавьте в начало определения класса код, представленный в листинге 7.5.

Листинг 7.5. Параметры, необходимые для двухкомпонентного селектора

```
@IBOutlet weak var doublePicker: UIPickerView!
private let fillingComponent = 0
private let breadComponent = 1
private let fillingTypes = [
    "Ham", "Turkey", "Peanut Butter", "Tuna Salad",
    "Chicken Salad", "Roast Beef", "Vegemite"]
private let breadTypes = [
    "White", "Whole Wheat", "Rye", "Sourdough",
    "Seven Grain"]
```

Как видим, сначала мы определяем две константы, определяющие индекс компонента, чтобы повысить читабельность кода. Обращение к компонентам селектора выполняется по номерам, причем крайнему слева компоненту

присваивается нулевой индекс, а индексы остальных по очереди увеличиваются на единицу. Затем мы объявляем два массива, в которых хранятся данные для компонентов селектора.

Реализуем метод `onButtonPressed()`, как показано в листинге 7.6.

Листинг 7.6. Действия при нажатии выбранной кнопки

```
@IBAction func onButtonPressed(_ sender: UIButton) {
    let fillingRow =
        doublePicker.selectedRow(inComponent: fillingComponent)
    let breadRow =
        doublePicker.selectedRow(inComponent: breadComponent)

    let filling = fillingTypes[fillingRow]
    let bread = breadTypes[breadRow]
    let message = "Your \(filling) on \(bread) bread will be right up."

    let alert = UIAlertController(
        title: "Thank you for your order",
        message: message,
        preferredStyle: .alert)
    let action = UIAlertAction(
        title: "Great",
        style: .default,
        handler: nil)
    alert.addAction(action)
    present(alert, animated: true, completion: nil)
}
```

Теперь добавим в конец определения класса методы делегата и источника данных, как показано в листинге 7.7.

Листинг 7.7. Методы источника данных и делегата

```
// MARK:-
// MARK: Picker Data Source Methods
func numberOfComponents(in pickerView: UIPickerView) -> Int {
    return 2
}

func pickerView(_ pickerView: UIPickerView,
               numberOfRowsInComponent component: Int) -> Int {
    if component == breadComponent {
        return breadTypes.count
    } else {
        return fillingTypes.count
    }
}

// MARK:-
// MARK: Picker Delegate Methods
func pickerView(_ pickerView: UIPickerView,
               titleForRow row: Int, forComponent component: Int) -> String? {
    if component == breadComponent {
```

```

        return breadTypes[row]
    } else {
        return fillingTypes[row]
    }
}
}

```

Метод `onButtonPressed()` на этот раз немного сложнее, но нового для вас в нем очень мало. Мы просто указываем, о каком компоненте идет речь при запросе выбранной строки с помощью констант `breadComponent` и `fillingComponent`.

```
let fillingRow = doublePicker.selectedRow(inComponent: fillingComponent)
let breadRow = doublePicker.selectedRow(inComponent: breadComponent)
```

Здесь можно увидеть, насколько использование констант вместо 0 и 1 делает код значительно более понятным. Сам по себе метод `onButtonPressed()`, в принципе, такой же, как и написанный перед этим.

Когда мы приступим к методам источника данных, нам нужно будет кое-что изменить. В первом методе мы указываем, что наш селектор должен состоять не из одного, а из двух компонентов.

```
func numberOfComponents(in pickerView: UIPickerView) -> Int {
    return 2
}
```

В этот раз, когда запрашивается количество строк, необходимо проверить, какой компонент селектора запрашивает это количество, и вернуть корректное значение для соответствующего массива.

```
func pickerView(_ pickerView: UIPickerView,
               numberOfRowsInComponent component: Int) -> Int {
    if component == breadComponent {
        return breadTypes.count
    } else {
        return fillingTypes.count
    }
}
```

Затем в методе нашего делегата делаем то же самое. Проверяем компонент и используем массив, соответствующий запрашивающему компоненту, для извлечения и возврата правильного значения.

```
func pickerView(_ pickerView: UIPickerView,
               titleForRow row: Int, forComponent component:
               Int) -> String? {
    if component == breadComponent {
        return breadTypes[row]
    } else {
        return fillingTypes[row]
    }
}
```

Это было не так уж и трудно, не правда ли? Скомпилируйте и запустите приложение и убедитесь, что панель `Double` выглядит так, как показано на рис. 7.4.

Обратите внимание на то, что каждая вращающаяся шкала полностью независима от другой шкалы. Вращение одной никак не влияет на другую. В данном случае это вполне уместно. Но бывают ситуации, когда один из компонентов зависит от другого. Хорошим примером являются даты. При изменении месяца шкалу, которая показывает дни месяца, возможно, потребуется изменить, потому что не все месяцы имеют одно и то же количество дней. Реализовать это не очень трудно, когда вы знаете, как это делается, но это не настолько легкая вещь, чтобы разобраться в ней самостоятельно. В следующем разделе мы будем решать именно эту задачу — реализацию зависимых компонентов.

Реализация зависимых компонентов

В этом разделе мы не будем вести читателя за руку, как делали это раньше, особенно если тема уже рассматривалась. Вместо этого сосредоточимся на новом материале. Наш новый селектор будет отображать в левом компоненте список штатов США, а в правом — список почтовых индексов, соответствующих штату, выбранному в левом компоненте.

Нам понадобится отдельный список значений почтовых индексов для каждого элемента левого компонента. Мы объявим два массива, по одному для каждого компонента, так же, как делали это в прошлый раз. Нам также потребуется словарь `NSDictionary`, в котором мы собираемся хранить объект класса `NSArray` для каждого штата (рис. 7.20). Позже мы реализуем метод делегата, который будет уведомлять нас при изменении выбранного элемента. При изменении значения слева мы будем брать в словаре соответствующий массив и передавать его для использования правому компоненту. Не беспокойтесь, если вы не уловили все тонкости; более подробно все эти вопросы будут раскрыты при их реализации в коде.

Добавьте в файл `DependentComponentPickerController.swift` следующий код:

```
class DependentComponentPickerController: UIViewController,
UIPickerViewDelegate, UIPickerViewDataSource {
private let stateComponent = 0
private let zipComponent = 1
private var stateZips:[String : [String]]!
private var states:[String]!
private var zips:[String]!
```

Пришло время для создания содержания. Этот процесс будет практически идентичен соответствующему процессу для двух предыдущих представлений. Если запутаетесь, вернитесь к разделу “Создание представления” для однокомпонентного селектора и следуйте приведенным там пошаговым инструкциям. Подсказка: откройте файл `Main.storyboard`, найдите класс `DependentComponentPickerController` и повторите те же основные шаги, которые делали для всех других представлений в этой главе. Когда закончите, сохраните раскладовку и вернитесь в среду Xcode. Вы должны закончить работу, задав свойство выхода `dependentPicker`, связанного с селектором, пустой метод `onButtonPressed:`,

связанный с кнопкой, а также свойства delegate и dataSource для селектора, связанного с контроллером представления. Не забудьте добавить ограничения Auto Layout в оба представления! Закончив работу, сохраните раскладовку.

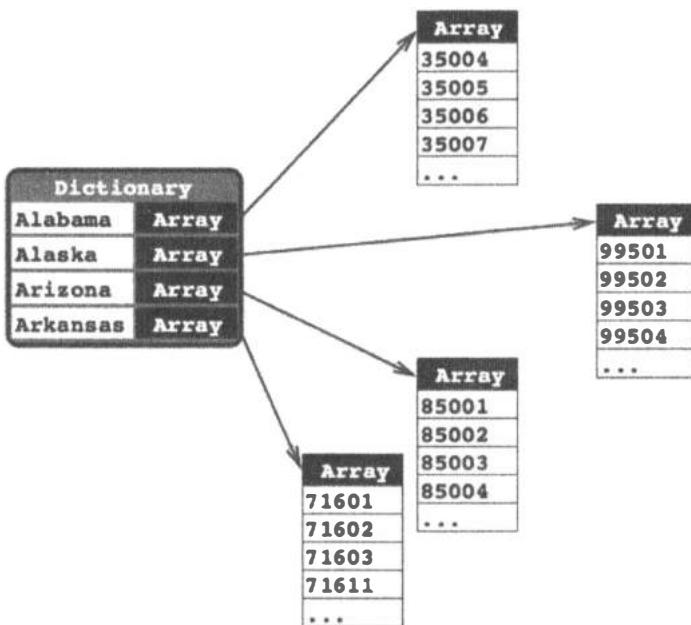


Рис. 7.20. Данные нашего приложения. Каждому штату соответствует одна запись в словаре с названием штата в качестве ключа. С этим ключом хранится экземпляр `NSArray<String>`, в котором содержатся все почтовые коды данного штата

Приступим к реализации класса контроллера. На первый взгляд, эта реализация может показаться немного сложной. Сделав один из компонентов зависящим от другого, мы добавили новый уровень сложности в класс контроллера. Хотя селектор выводит одновременно только два списка, наш класс контроллера должен знать 51 список и управлять им. Использованная здесь методика упрощает этот процесс. Методы источника данных почти идентичны тем, которые мы реализовали для представления `DoublePickerViewController`. Все дополнительные сложности обрабатываются в другом месте, между методом `viewDidLoad` и новым методом делегата — `pickerView(_:didSelectRow:inComponent:)`.

Перед тем как писать код, нам потребуются некоторые данные для вывода. До сих пор мы создавали массивы в коде, просто приводя список строк. Поскольку нам не хотелось бы вводить несколько тысяч значений, а также потому, что мы хотим показать, как правильно решать такие задачи, мы собираемся загружать данные из списка свойств. Как уже упоминалось, объекты `NSArray` и `NSDictionary` могут создаваться из свойства списков. Мы включили список свойств `statedictionary.plist` в папку 07 – Picker Data архива проекта.

Скопируйте этот файл в папку Picker в своем проекте Xcode. Если вы щелкнете на plist-файле в окне навигатора проекта, то увидите и даже сможете отредактировать данные, которые он содержит (рис. 7.21).



Key	Type	Value
Root	Dictionary	(50 items)
▶ Alabama	Array	(657 items)
▶ Alaska	Array	(251 items)
▶ Arizona	Array	(376 items)
▶ Arkansas	Array	(618 items)
▶ California	Array	(1757 items)
▶ Colorado	Array	(501 items)
▶ Connecticut	Array	(276 items)
▶ Delaware	Array	(68 items)
▶ Florida	Array	(972 items)
▶ Georgia	Array	(736 items)
▶ Hawaii	Array	(92 items)
▶ Idaho	Array	(292 items)
▶ Illinois	Array	(1375 items)
▶ Indiana	Array	(780 items)
▶ Iowa	Array	(972 items)
▶ Kansas	Array	(721 items)
▶ Kentucky	Array	(799 items)
▶ Louisiana	Array	(542 items)
▶ Maine	Array	(415 items)
▶ Maryland	Array	(468 items)
▶ Massachusetts	Array	(519 items)
▶ Michigan	Array	(987 items)
▶ Minnesota	Array	(892 items)
▶ Mississippi	Array	(447 items)
▶ Missouri	Array	(1040 items)
▶ Montana	Array	(364 items)
▶ Nebraska	Array	(590 items)
▶ Nevada	Array	(158 items)
▶ New Hampshire	Array	(238 items)
▶ New Jersey	Array	(604 items)
▶ New Mexico	Array	(366 items)
▶ New York	Array	(1677 items)
▶ North Carolina	Array	(809 items)
▶ North Dakota	Array	(392 items)
▶ Ohio	Array	(1189 items)
Item 0	String	43001
Item 1	String	43002
Item 2	String	43003
Item 3	String	43004

Рис. 7.21. Файл statestatedictionary.plist со списком штатов. Для штата Огайо открыто начало списка почтовых индексов

278 ГЛАВА 7 ■ ПАНЕЛИ ВКЛАДОК И СЕЛЕКТОРЫ

Добавьте в файл DependentComponentPickerController.swift несколько методов, которые потом мы разделим на более удобные фрагменты. Начнем с реализации метода onPressed(), показанной в листинге 7.8.

Листинг 7.8. Методы источника данных и делегата

```
@IBAction func onPressed(_ sender: UIButton) {
    let stateRow =
        dependentPicker.selectedRow(inComponent: stateComponent)
    let zipRow =
        dependentPicker.selectedRow(inComponent: zipComponent)

    let state = states[stateRow]
    let zip = zips[zipRow]

    let title = "You selected zip code \(zip)"
    let message = "\(zip) is in \(state)"

    let alert = UIAlertController(
        title: title,
        message: message,
        preferredStyle: .alert)
    let action = UIAlertAction(
        title: "OK",
        style: .default,
        handler: nil)
    alert.addAction(action)
    present(alert, animated: true, completion: nil)
}
```

Теперь добавьте в существующий метод viewDidLoad() код, показанный в листинге 7.9.

Листинг 7.9. Метод viewDidLoad()

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

    // Дополнительная настройка после загрузки представления.
    let bundle = Bundle.main
    let plistURL = bundle.urlForResource("statedictionary",
                                         withExtension: "plist")
    stateZips = NSDictionary.init(contentsOf: (plistURL)!) as! [String :
    [String]]
    let allStates = stateZips.keys
    states = allStates.sorted()
    let selectedState = states[0]
    zips = stateZips[selectedState]
}
```

Наконец добавьте в конец файла методы делегата и источника данных, приведенные в листинге 7.10.

Листинг 7.10. Методы источника данных и делегата для вывода почтовых индексов штатов

```
// MARK:-
// MARK: Picker Data Source Methods
func numberOfComponents(in pickerView: UIPickerView) -> Int {
    return 2
}

func pickerView(_ pickerView: UIPickerView,
               numberOfRowsInComponent component: Int) -> Int {
    if component == stateComponent {
        return states.count
    } else {
        return zips.count
    }
}

// MARK:-
// MARK: Picker Delegate Methods
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int,
               forComponent component: Int) -> String? {
    if component == stateComponent {
        return states[row]
    } else {
        return zips[row]
    }
}

func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int,
               inComponent component: Int) {
    if component == stateComponent {
        let selectedState = states[row]
        zips = stateZips[selectedState]
        dependentPicker.reloadComponent(zipComponent)
        dependentPicker.selectRow(0, inComponent: zipComponent,
                                  animated: true)
    }
}
```

Нет необходимости говорить о методе `onButtonPressed()`, поскольку он, по сути, такой же, как и предыдущий. Однако мы должны поговорить о методе `viewDidLoad()`. В нем происходит кое-что важное, в чем вам надо разобраться, так что садитесь поудобнее, и приступим. Первое, что нам надо сделать в этом новом методе `viewDidLoad()`, — это получить ссылку на основной пакет приложения.

```
let bundle = Bundle.main
```

Пакет (bundle) — это особый тип папки, содержимое которой имеет определенную структуру. Приложения и каркасы представляют собой пакеты, и показанный выше вызов возвращает объект пакета, который представляет наше приложение.

ЗАМЕЧАНИЕ. В последних версиях среды Xcode и библиотек iOS компания Apple предложила более удобный способ ссылаться на такие элементы, как `NSBundle` в языке Swift. Вместо выражения наподобие `let bundle = NSBundle.main()` теперь можно использовать более удобные варианты.

Одним из главных применений класса `NSBundle` является обращение к ресурсам, которые вы добавили в проект. Эти файлы будут скопированы в пакет вашего приложения при его сборке. Если мы хотим получить эти ресурсы в коде, то нам понадобится класс `NSBundle`. Основной пакет используется для получения пути интересующего нас ресурса.

```
let plistURL = bundle.URLForResource("statedictionary",
                                      withExtension: "plist")
```

Этот вызов возвращает строку с расположением `statedictionary.plist`. Затем этот путь можно использовать для создания объекта класса `NSDictionary`. Как только мы это сделаем, все содержимое списка свойств будет загружено во вновь создаваемый объект класса `NSDictionary`, который мы затем присвоим переменной `stateZips`.

```
stateZips = NSDictionary.init(contentsOf: (plistURL)!) as! [String : [String]]
```

Тип `Dictionary` в языке Swift не имеет удобного способа для загрузки данных из внешнего источника, а тип `NSDictionary` в каркасе Foundation имеет такие возможности. Этот код использует такие возможности, загружая содержимое файла `statedictionary.plist` в объект класса `NSDictionary`, который приводится к типу языка Swift `[String : [String]]`. Иначе говоря, словарь, в котором каждый ключ представляет штат и соответствующее значение, является массивом почтовых индексов, представленных в виде строк. Это соответствует структуре, показанной на рис. 7.18.

Для того чтобы заполнить массив левого компонента селектора, соберем список всех ключей нашего словаря в массиве `states`. Однако перед присваиванием отсортируем этот массив в алфавитном порядке.

```
let allStates = stateZips.keys
states = allStates.sorted()
```

Если только мы не указываем иное значение, селектор начинает работу с выбранной первой строкой (строка с индексом 0). Для того чтобы получить массив индексов, соответствующий первой строке массива штатов, мы получаем из массива объект с индексом 0. Это даст нам название штата, который будет выбран в момент запуска. Затем используем название штата для получения массива почтовых индексов для этого штата, который присвоим переменной `zips`, которая, в свою очередь, будет использоваться для передачи данных в правый компонент.

```
let selectedState = states[0]
zips = stateZips[selectedState]
```

Эти два метода источника данных практически идентичны предыдущей версии. Мы возвращаем количество строк в соответствующем массиве. То же самое верно и для первого реализованного метода делегата. Второй метод делегата является новым и в нем происходит нечто загадочное.

```
func pickerView(_ pickerView: UIPickerView,
               didSelectRow row: Int, inComponent component:
               Int) {
    if component == stateComponent {
        let selectedState = states[row]
        zips = stateZips[selectedState]
        dependentPicker.reloadComponent(zipComponent)
        dependentPicker.selectRow(0, inComponent: zipComponent,
                                   animated: true)
    }
}
```

В этом методе, который вызывается всякий раз при изменении выбранной строки в компоненте селектора, мы изучаем компоненты и проверяем, не изменился ли селектор левого компонента. Если это произошло, получаем массив индексов, который соответствует новому селектору левого компонента, и присваиваем его массиву `zips`. Затем правый компонент вновь настраивается на показ первого ряда данных и перезагружается. Метод замены массива `zips` при изменении состояния позволяет оставить прочий код почти нетронутым, таким же, каким он был в примере `DoublePicker`.

Но мы еще не закончили. Скомпилируйте и запустите приложение и проверьте работу вкладки `Dependent` (рис. 7.22). Два компонента имеют одинаковые размеры. Хотя почтовый индекс никогда не будет состоять более чем из пяти символов, для него выделяется такое же пространство, как и для названия штата. Поскольку названия таких штатов, как Миссисипи или Массачусетс, не помещаются в половину экрана на устройствах iPhone, это решение кажется далеким от идеального.

К счастью, есть еще один метод делегата, который мы можем реализовать, чтобы указать, насколько широким должен быть каждый компонент. Добавьте метод из листинга 7.11 в раздел делегата в файле `DependentComponentPickerController.swift`. Результат вы увидите на рис. 7.23.



Рис. 7.22. Вы в самом деле хотите, чтобы размер компонентов был одинаковым? Обратите внимание на обрыв длинного названия штата

Листинг 7.11. Методы источника данных и делегата для вывода почтовых индексов штатов

```
func pickerView(_ pickerView: UIPickerView,
               widthForComponent component: Int) -> CGFloat
{
    let pickerWidth = pickerView.bounds.size.width
    if component == zipComponent {
        return pickerWidth/3
    } else {
        return 2 * pickerWidth/3
    }
}
```

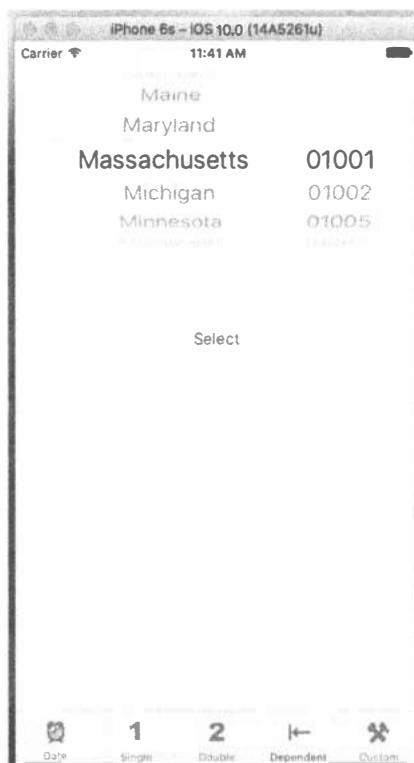


Рис. 7.23. После настройки ширины компонентов селектора пользовательский интерфейс стал намного лучше

В данном методе мы возвращаем число, представляющее собой ширину в пикселях каждого компонента, и селектор будет делать все возможное, чтобы учесть эти данные. Экспериментируя с разными значениями, можно увидеть, как распределяется пространство между компонентами при их изменении. Сохраните файл, скомпилируйте приложение и запустите его, и ваш селектор будет выглядеть так, как показано на рис. 7.5.

К этому моменту вы уже должны хорошо знать не только селекторы, но и приложения с панелями вкладок. Нам осталось показать вам еще только одну вещь, связанную с селекторами.

Создание простой игры с пользовательским селектором

Следующим нашим занятием будет создание простого игрового автомата. Вернитесь к рис. 7.6, прежде чем продолжить чтение, чтобы понимать, что мы создаем.

Подготовка контроллера представления

Добавьте следующий код в файл CustomPickerController.swift:

```
class CustomPickerController: UIViewController,  
    UIPickerViewDelegate, UIPickerViewDataSource {  
    private var images: [UIImage]!
```

Здесь мы добавляем в класс свойство массива, содержащего изображения, которые будут выводиться селектором.

Создание представления

Хотя селектор на рис. 7.6 имеет вид, отличающийся от вида созданных нами ранее селекторов, на самом деле очень мало различий в том, как мы разрабатываем наши раскладовки. Все дополнительные работы выполняются в методах делегата нашего контроллера.

Убедитесь, что вы сохранили свой новый исходный код, а затем дважды щелкните на файле Main.storyboard в навигаторе проекта и выберите пиктограмму Custom Scene для редактирования графического интерфейса. Добавьте представление селектора, метку под ним, а еще ниже — кнопку. Дайте кнопке имя Spin.

Выбрав метку, вызовите инспектор атрибутов. Установите выравнивание по центру. Затем щелкните на кнопке Text color, чтобы изменить цвет текста на что-то более яркое. Затем обратитесь к настройкам Font в инспекторе и щелкните на пиктограмме (она выглядит как буква Т в рамке), чтобы открыть селектор шрифтов. Этот элемент управления позволяет переключать системные шрифты или просто изменять их размер. Пока просто измените размер шрифта на 48 и удалите слово Label, так как мы не хотим, чтобы до момента выигрыша пользователя в метке выводился какой-либо текст.

Добавьте ограничения Auto Layout для центрования селектора, метки и кнопки по горизонтали и фиксации вертикальных просветов между ними, а также между меткой и селектором с одной стороны и верхним краем представления с другой. Вероятно, проще всего перетащить указатель мыши с кнопки в окне Document Outline при добавлении ограничений Auto Layout, потому что метка на раскладовке является пустой и ее очень трудно найти.

После этого выполните все соединения с выходами и действиями. Вам необходимо соединить выход picker контроллера представления с представлением селектора, а также выход winLabel контроллера представления с меткой. Проще всего использовать метку в окне Document Outline. Затем свяжите событие Touch Up Inside кнопки с методом действия spin(). После этого свяжите селектор с делегатом и источником данных.

Есть одна дополнительная вещь, которую следует сделать. Вызовите инспектор атрибутов для селектора. Вам нужно сбросить флагок User Interaction Enabled в нижней части настроек View, чтобы пользователь не мог нарушать правила. После того как вы сделали все описанное, сохраните изменения, внесенные в раскладовку.

ШРИФТЫ, ПОДДЕРЖИВАЕМЫЕ IOS-УСТРОЙСТВАМИ

Будьте осторожны при использовании палитры шрифтов в Interface Builder при разработке интерфейсов iOS. Программа Interface Builder позволяет назначать меткам любой шрифт, который имеется на вашем компьютере Mac, но iOS-устройства имеют очень ограниченный селектор шрифтов. Вы должны ограничиться одним из семейств шрифтов, имеющихся на целевом iOS-устройстве. Сообщение <http://iphonedevlopment.blogspot.com/2010/08/fonts-and-font-families.html> в посвященном iOS блоге Джекфа Ламарша рассказывает, как программно получить этот список.

Если говорить в двух словах, то следует создать приложение на базе представления и добавить приведенный код в метод application(_: didFinishLaunchingWithOptions:) делегата приложения.

```
for family in UIFont.familyNames() as [String] {
    println(family)
    for font in UIFont.fontNamesForFamilyName(family) {
        println("\t\(font)")
    }
}
```

Выполните проект в соответствующем симуляторе или на устройстве, и шрифты будут выведены в консольный журнал проекта.

Реализация контроллера

В реализации данного контроллера имеется масса новинок. Добавьте в метод spin() в файле CustomPickerViewController.swift код, приведенный в листинге 7.12.

Листинг 7.12. Метод spin()

```
@IBAction func spin(_ sender: UIButton) {
    var win = false
    var numInRow = -1
    var lastVal = -1
```

```

for i in 0..<5 {
    let newValue = Int(arc4random_uniform(UInt32(images.count)))
    if newValue == lastVal {
        // numInRow++ *** УЧТИТЕ: в Swift 3 инкрементация и декрементация
        // объявлены устаревшими
        numInRow += 1
    } else {
        numInRow = 1
    }
    lastVal = newValue

    picker.selectRow(newValue, inComponent: i, animated: true)
    picker.reloadComponent(i)
    if numInRow >= 3 {
        win = true
    }
}
winLabel.text = win ? "WINNER!" : " " // Обратите внимание
                                // на пробел в кавычках
}

```

ЗАМЕЧАНИЕ. Унарная инкрементация (`foo++`) и декрементация (`foo--`) в языке Swift 3 объявлены устаревшими. Вместо них рекомендуется использовать операторы `+=` и `-=`.

Вставьте в код метода `viewDidLoad()` код, приведенный в листинге 7.13.

Листинг 7.13. Модификации метода `viewDidLoad()` для настройки изображений и метки

```

override func viewDidLoad() {
    super.viewDidLoad()

    // Дополнительная настройка после загрузки представления.
    images = [
        UIImage(named: "seven")!,
        UIImage(named: "bar")!,
        UIImage(named: "crown")!,
        UIImage(named: "cherry")!,
        UIImage(named: "lemon")!,
        UIImage(named: "apple")!
    ]
    winLabel.text = " " // // Обратите внимание на пробел в кавычках
    arc4random_stir()
}

```

Наконец добавьте в код источника данных и делегата в конце объявления класса перед закрывающей фигурной скобкой текст, приведенный в листинге 7.14.

Листинг 7.14. Методы источника данных и делегата

```

// MARK:-
// MARK: Picker Data Source Methods
func numberOfComponents(in pickerView: UIPickerView) -> Int {
    return 5
}

```

286 ГЛАВА 7 ■ ПАНЕЛИ ВКЛАДОК И СЕЛЕКТОРЫ

```
func pickerView(_ pickerView: UIPickerView,
               numberOfRowsInComponent component: Int) -> Int {
    return images.count
}
// MARK:-
// MARK: Picker Delegate Methods
func pickerView(_ pickerView: UIPickerView,
               viewForRow row: Int,
               forComponent component: Int,
               reusing view: UIView?) -> UIView {
    let image = images[row]
    let imageView = UIImageView(image: image)
    return imageView
}
func pickerView(_ pickerView: UIPickerView,
               rowHeightForComponent component: Int) -> CGFloat {
    return 64
}
```

Метод spin()

Метод `spin()` запускается, когда пользователь прикасается к кнопке `Spin`. В нем мы сначала объявляем несколько переменных, которые помогут отслеживать выигрыш пользователя. Мы используем переменную `win` для того, чтобы отследить, нет ли в ряду трех одинаковых изображений. Переменная `numInRow` отслеживает, сколько одинаковых значений было в строке до сих пор, а значение предыдущего компонента хранится в `lastVal`, чтобы мы могли сравнивать текущее значение с предыдущим. Мы инициализируем переменную `lastVal` значением `-1`, потому что знаем, что такого реального значения не может быть.

```
var win = false
var numInRow = -1
var lastVal = -1
```

Далее следует цикл по всем пяти компонентам, в котором каждый из них получает новую, случайным образом выбранную строку. Поскольку нам известно, что размеры всех массивов одинаковы, мы используем значение `count` из массива `column1`.

```
for i in 0..<5 {
    let newValue = Int(arc4random_uniform(UInt32(images.count)))
```

Сравниваем новое значение с предыдущим и увеличиваем `numInRow`, если они совпадают. Если же нет, то сбрасываем значение `numInRow` в 1. Присваиваем новое значение переменной `lastVal`, чтобы можно было сравнить его в следующий раз.

```
if newValue == lastVal {
    numInRow+=1
} else {
    numInRow = 1
}
lastVal = newValue
```

После этого устанавливаем новое значение для соответствующего компонента, запрашиваем анимацию изменения и требуем от селектора перегрузки указанного компонента.

```
picker.selectRow(newValue, inComponent: i, animated: true)
picker.reloadComponent(i)
```

Последнее, что мы делаем в каждой итерации цикла, — проверяем, не получилось ли три одинаковых символа в строке и, если получилось, устанавливаем для переменной `win` значение YES.

```
if numInRow >= 3 {
    win = true
}
```

Как только мы заканчиваем работу с циклом, соответствующим образом меняем текст метки.

```
winLabel.text = win ? "WINNER!" : " "
// Обратите внимание на пробел между кавычками
```

Метод `viewDidLoad()`

Первое, что надо сделать, — это загрузить шесть различных изображений, которые мы добавили в каталог `Images.xcassets`. Мы делаем это, используя удобный метод класса `UIImage`, который называется `imageNamed()`.

```
images = [
    UIImage(named: "seven")!,
    UIImage(named: "bar")!,
    UIImage(named: "crown")!,
    UIImage(named: "cherry")!,
    UIImage(named: "lemon")!,
    UIImage(named: "apple")!
]
```

Последнее, что мы делаем в этом методе, — задаем текст метки, состоящий из одного пробела. Мы хотим, чтобы метка была пустой, но не сжималась до нуля. Задав пробел, мы обеспечим правильную высоту метки.

```
winLabel.text = " " // Обратите внимание на пробел между кавычками
```

В заключение вы вызвали метод `arc4random_stir()`, задав начальное значение генератора случайных чисел, чтобы избежать генерирования одинаковых последовательностей чисел при каждом запуске приложения.

Но что же нам делать с шестью изображениями? Если вы просмотрите только что введенный код, то увидите, что два метода источника данных выглядят примерно так же, как и ранее, но если вы посмотрите далее, на методы делегата, то увидите, что для предоставления данных селектору мы используем совершенно другой метод делегата. Тот, который мы использовали до сих пор, возвращал строку, а этот возвращает объект типа `UIView`.

Этот метод позволяет передавать селектору все, что можно изобразить, в объекте класса `UIView`. Конечно, есть ограничения на то, что будет работать таким образом и в то же время хорошо выглядеть, учитывая небольшой размер селектора. Но этот метод дает гораздо больше свободы в том, что мы выводим, хотя и требует несколько большей работы.

```
func pickerView(_ pickerView: UIPickerView,
               viewForRow row: Int,
               forComponent component: Int,
               reusing view: UIView!) -> UIView {
    let image = images[row]
    let imageView = UIImageView(image: image)
    return imageView
}
```

Этот метод возвращает один объект класса `UIImageView`, содержащий одно из изображений для символов. Для этого мы сначала получаем изображение для символа, а затем создаем и возвращаем графическое представление этого символа. Для более сложных представлений сначала целесообразно создать представ-

ление, например методом `viewDidLoad()`, а затем — заранее созданное представление в селектор представлений по запросу. Однако в нашем простом случае создание представления “на лету” работает вполне хорошо.

Все, что было первоначально одним большим куском кода, оказалось разложенным по полочкам. Запустив приложение, вы увидите экран, показанный на рис. 7.24.



Рис. 7.24. Не самое увлекательное, но вполне познавательное приложение, демонстрирующее работу селектора

Последние штрихи

У нас получилась неплохая игра, особенно если подумать о том, как мало усилий потребовалось, чтобы ее создать. Сейчас мы немного улучшим ее. Есть два замечания, которые можно высказать об игре.

- Она слишком тихая. Игровые автоматы так тихо не работают.
- Она сообщает о выигрыше еще до того, как шкала заканчивает вращаться, что не так уж важно, но тем не менее хотелось бы этого избежать. Для того чтобы увидеть этот недостаток, снова запустите приложение. Метка появляется немного раньше завершения вращения, хотя это почти не заметно.

В папке 07 – Picker Sounds архива проектов имеются два звуковых файла: `crunch.wav` и `win.wav`. Скопируйте эту папку в папку `Picker`. Эти звуки мы будем воспроизводить, когда пользователь нажимает кнопку `Spin` и когда он побеждает соответственно.

Для работы со звуками мы должны иметь доступ к классам iOS Audio Toolbox. Вставьте следующую строку в начало файла `CustomPickerController.swift`:

```
import UIKit
import AudioToolbox
```

Далее мы должны добавить выход, который будет указывать на кнопку. Пока колеса врачаются, мы хотели бы, чтобы кнопка была скрыта. Мы не хотим, чтобы пользователи могли нажимать ее в то время, когда колеса еще врачаются. Добавьте следующий код в файл `CustomPickerController.swift`:

```
class CustomPickerController: UIViewController,
    UIPickerViewDelegate, UIPickerViewDataSource {
    private var images:[UIImage]!
    @IBOutlet weak var picker: UIPickerView!
    @IBOutlet weak var winLabel: UILabel!
    @IBOutlet weak var button: UIButton!
```

После того как введете текст в файл и сохраните его, щелкните на файле `Main.storyboard`, чтобы редактировать графический пользовательский интерфейс. Откройте окно помощника редактора и файл `CustomPickerController.swift`. Нажмите клавишу `<Control>` и перетащите указатель с маленького кругочка влево вниз — на выход, только что добавленный к кнопке в раскладовке (рис. 7.25).

Теперь необходимо выполнить несколько действий по реализации нашего класса контроллера. Во-первых, нам необходимы переменные, в которых хранились бы ссылки на загружаемые звуковые файлы. Откройте файл `CustomPickerController.swift` и добавьте в него следующие новые свойства (выделенные полужирным шрифтом):

```
class CustomPickerController: UIViewController,
    UIPickerViewDelegate, UIPickerViewDataSource {
    private var images:[UIImage]!
    @IBOutlet weak var picker: UIPickerView!
    @IBOutlet weak var winLabel: UILabel!
    @IBOutlet weak var button: UIButton!
    private var winSoundID: SystemSoundID = 0
    private var crunchSoundID: SystemSoundID = 0
```

Во-вторых, необходимо добавить в класс контроллера пару методов. Добавьте два метода, приведенные в листинге 7.15, в файл `CustomPickerController.swift`.

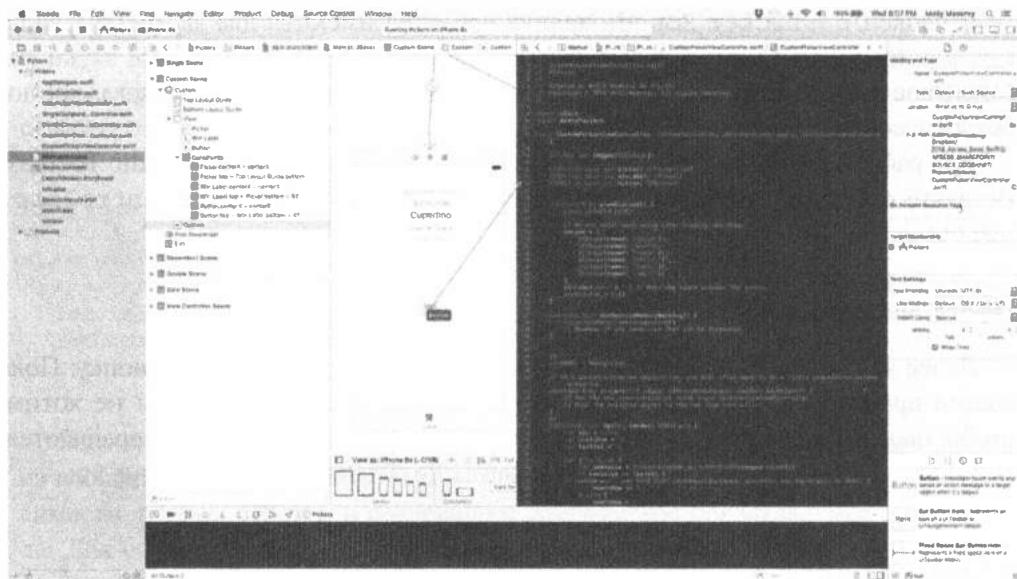


Рис. 7.25. Связывание выхода кнопки с кнопкой на канве раскладовки

Листинг 7.15. Скрытие кнопки Spin и воспроизведение звуков игрового автомата

```
func showButton() {
    button.isHidden = false
}

func playWinSound() {
    if winSoundID == 0 {
        let soundURL = Bundle.main.urlForResource(
            "win", withExtension: "wav")! as CFURL
        AudioServicesCreateSystemSoundID(soundURL, &winSoundID)
    }
    AudioServicesPlaySystemSound(winSoundID)
    winLabel.text = "WINNER!"
    DispatchQueue.main.after(when: .now() + 1.5) {
        self.showButton()
    }
}
```

Первый метод используется для того, чтобы показать кнопку. Мы собираемся скрывать кнопку после того, как пользователь нажал ее, потому что, если колеса уже вращаются, нет никакого смысла вращать их далее, пока они не остановятся.

Второй метод будет вызван, когда пользователь выиграет. Сначала мы проверяем, загружен ли звук победы. Свойства `winSoundID` и `crunchSoundID` инициализируются нулями, а идентификаторы для загруженных звуков — нет. Следовательно, для того чтобы проверить, загружен ли звук, достаточно сравнить идентификатор с нулем. Для загрузки звука сначала запрашиваем пакет,

содержащий файл `win.wav`, аналогично тому, как мы делали при загрузке списка свойств для зависимого представления селектора. Следующие три строки кода загружают звук и воспроизводят его. Затем метка получает текст `WINNER!` и вызывается метод `showButton()`, но этот вызов выполняется специальным образом — с применением метода `Dispatch_Queue(when:)`. Это очень удобная функция. Она позволяет вызвать метод через некоторое время в будущем, в данном случае — через полторы секунды, которые позволяют колесам, прежде чем результат будет сообщен пользователю, докрутиться до конечного положения. Эта функция относится к группе функций `Grand Central Dispatch (GCD)`, которую мы рассмотрим в главе 15.

ЗАМЕЧАНИЕ. Возможно, вы заметили нечто странное в способе объявления функции `AudioServicesSystemSoundID()`. Первым параметром этой функции является адрес URL. Однако он должен быть не экземпляром класса `NSURL`, а объектом класса `CFURL` (который ранее назывался `CFURLRef`), представляющим собой указатель на структуру из каркаса Core Foundation, написанного на языке C. Указатель `NSURL` является частью каркаса Foundation, написанного на языке Objective-C. К счастью, многие компоненты, написанные на языке C, имеют аналоги на языке Objective-C в каркасе Foundation, например структура `CFURL` функционально эквивалентна указателю `NSURL`. Это означает, что некоторые категории объектов, созданных на языке Swift или Objective-C, можно использовать в интерфейсах прикладного программирования, написанных на языке C, просто приводя их к соответствующим типам в языке C, используя ключевое слово `as`.

Нам также необходимо внести некоторые изменения в метод `spin()`. Мы напишем код для воспроизведения звука при выигрыше игрока (листинг 7.16).

Листинг 7.16. Модифицированный метод `spin()`

```
@IBAction func spin(sender: AnyObject) {
    var win = false
    var numInRow = -1
    var lastVal = -1

    for i in 0..<5 {
        let newValue = Int(arc4random_uniform(UInt32(images.count)))
        if newValue == lastVal {
            numInRow++
        } else {
            numInRow = 1
        }
        lastVal = newValue

        picker.selectRow(newValue, inComponent: i, animated: true)
        picker.reloadComponent(i)
        if numInRow >= 3 {
            win = true
        }
    }
}
```

```

if crunchSoundID == 0 {
    let soundURL = NSBundle mainBundle().URLForResource(
        "crunch", withExtension: "wav")! as CFURLRef
    AudioServicesCreateSystemSoundID(soundURL, &crunchSoundID)
}
AudioServicesPlaySystemSound(crunchSoundID)

if win {
    DispatchQueue.main.after(when: .now(0) + 0.5) {
        self.playwinSound()
    }
} else {
    DispatchQueue.main.after(when: .now(0) + 0.5) {
        self.showButton()
    }
}
button.hidden = true
winLabel.text = " " // Обратите внимание на пробел между кавычками
}

```

Сначала мы загружаем скрипучий звук, как это делалось ранее. Следующая строка проигрывает уже загруженный звук, чтобы игрок слышал, как крутиятся колеса. Затем, вместо метки WINNER! при выигрыше пользователя, мы идем на некоторые хитрости. Мы вызываем один из двух методов, которые только что создали, но делаем это после задержки с использованием метода DispatchQueue.main.after(when:). Если пользователь выиграл, вызываем метод playerWinSound() через полсекунды, что дает колесам время остановиться; в противном случае ждем полсекунды и снова включаем кнопку Spin. Ожидая результата, мы скрываем кнопку и стираем текст метки.

Нажмите кнопку Run в среде Xcode и щелкните на последней метке. При запуске кнопка Spin издает один звук, а в случае победы воспроизводится другой.

Резюме

Теперь вы должны комфортно чувствовать себя, работая с приложениями с панелями вкладок и селекторами. В данной главе мы “с нуля” создали полноценное приложение с панелью вкладок, содержащее пять различных представлений содержимого. Вы узнали, как использовать селекторы различных конфигураций, как создавать селекторы с несколькими компонентами и как сделать значения в одном компоненте зависящими от выбранного значения в другом компоненте. Вы также видели, как заставить селектор выводить изображения, а не просто текст.

Кроме того, вы узнали о делегатах и источниках данных селектора и увидели, как загрузить изображения, звуки, а также создать словари на основе списков свойств. Это была длинная глава, так что мы поздравляем читателей, добравшихся до ее конца. Теперь вы готовы к работе с табличными представлениями.

ГЛАВА 8



Введение в табличные представления

В нескольких следующих главах мы создадим иерархическое навигационное приложение, похожее на приложение Mail, поставляемое с iOS-устройствами. Приложения этого типа, обычно именуемые **master-detail**, позволяют пользователю переходить к вложенным спискам данных и редактировать эти данные. Но прежде чем мы сможем написать такое приложение, вам необходимо освоить концепцию табличных представлений, которые и являются целью данной главы.

Табличные представления (*table views*) — наиболее распространенный механизм, используемый для отображения списков данных. Эти объекты допускают очень подробное конфигурирование и могут быть настроены именно так, как вы хотите. Приложение Mail использует табличные представления для показа списков учетных записей, папок и сообщений, но табличные представления не ограничиваются только отображением текстовых данных. Табличные представления используются также в приложениях *Settings*, *Music* и *Clock*, несмотря на то что эти приложения имеют совершенно разный внешний вид (рис. 8.1).

Основы табличных представлений

Таблицы выводят на экран списки данных. Каждый элемент табличного списка является строкой. В системе iOS таблицы могут иметь любое количество строк, ограниченное лишь имеющейся доступной памятью. Однако таблицы iOS могут содержать только один столбец.

Табличные представления и ячейки табличного представления

Табличное представление является объектом, который выводит табличные данные и представляет собой экземпляр класса `UITableView`. Каждая видимая строка таблицы реализуется экземпляром класса `UITableViewCell` (рис. 8.2).

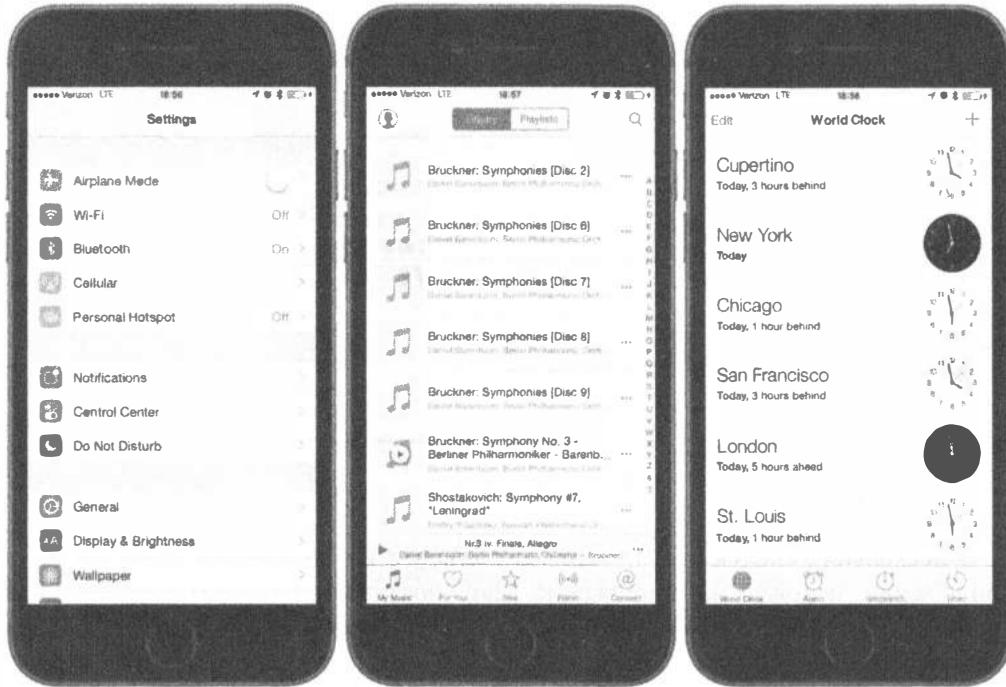


Рис. 8.1. Несмотря на разный внешний вид, приложения *Settings*, *Music* и *Clock* используют табличные представления

Табличные представления не несут ответственности за хранение табличных данных. Они хранят только те данные, которые необходимы для вывода строк, видимых в текущий момент. Табличные представления получают свои конфигурационные данные из объекта, который соответствует протоколу `UITableViewDelegate`, и строчные данные из объекта, соответствующего протоколу `UITableViewDataSource`. Вы увидите, как все это работает, в примерах наших программ, описанных в главе.

Как уже упоминалось, все таблицы реализованы в виде одного столбца. Однако приложение *Clock*, показанное на рис. 8.1, *справа*, выглядит как имеющее два столбца, но в действительности это не так — каждая строка в таблице представлена единственным объектом класса `UITableViewCell`. По умолчанию объект класса `UITableViewCell` может быть настроен для вывода изображения, некоторого текста и необязательной вспомогательной пиктограммы в правой части (мы подробно рассмотрим вспомогательные пиктограммы в главе 9).

По мере необходимости можно поместить в ячейку еще больше данных, добавив в объект класса `UITableViewCell` дочерние представления, используя один из двух основных методов: 1) путем добавления дочерних представлений программно при создании ячейки или 2) с помощью загрузки из раскадровки или `nib`-файла. Вы можете макетировать ячейку таблицы так, как вам нравится, и включить в нее любое требуемое количество дочерних представлений. Поэтому

ограничение табличных представлений одним столбцом на деле оказывается не таким страшным, как это, вероятно, выглядит на первый взгляд. Далее мы покажем, как использовать оба эти метода.



Рис. 8.2. Каждое табличное представление представляет собой экземпляр класса UITableView, а каждая видимая строка — экземпляр UITableViewCell

Сгруппированные и простые таблицы

Табличные представления бывают двух основных стилей.

- * **Сгруппированные (grouped).** Сгруппированное табличное представление содержит один или несколько разделов из строк. В каждом разделе все строки тесно связаны в одну небольшую группу; между разделами имеются ясно видимые промежутки, как показано на рис. 8.3, слева. Заметим, что сгруппированная таблица может состоять из одной группы.
- * **Простые (plain).** Это стиль, заданный по умолчанию. При данном стиле разделы располагаются немного ближе друг к другу, а заголовок каждого раздела может (необязательно) выделяться внешним видом. При использовании индекса этот стиль также именуется **индексированным** (см. рис. 8.3, справа).

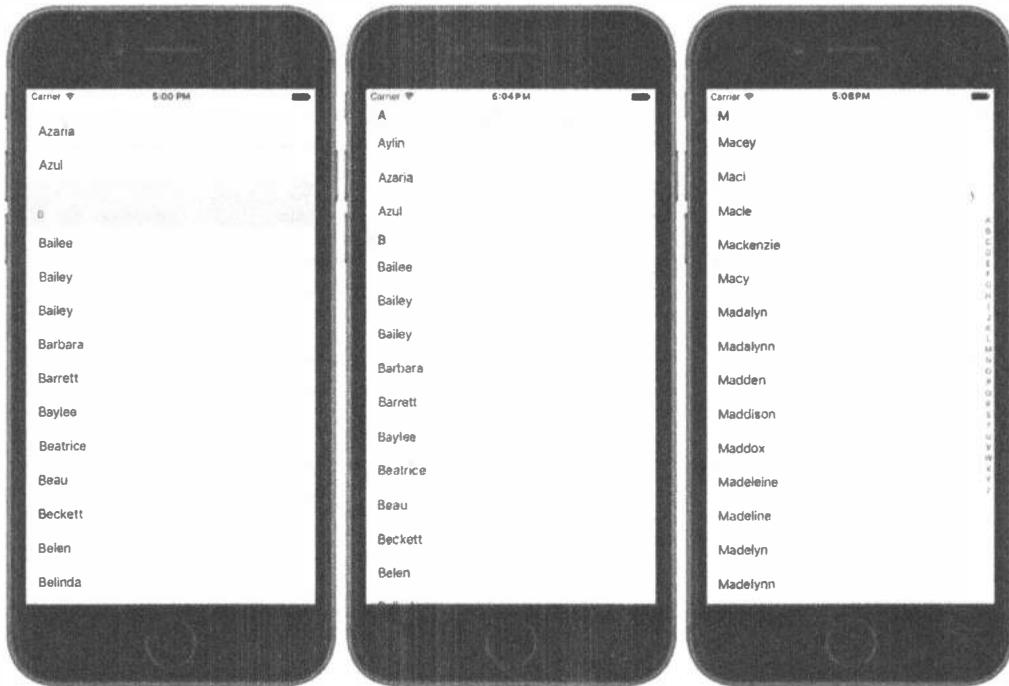


Рис. 8.3. Одно и то же табличное представление в виде сгруппированной таблицы (слева), простой таблицы без индекса (в центре) и простой таблицы с индексом, или индексированной таблицы (справа)

Если источник данных предоставляет необходимую информацию, табличное представление позволяет пользователю перемещаться по списку с использованием индекса, который отображается внизу справа.

Таблицы состоят из **разделов** (sections). В сгруппированной таблице каждый раздел визуально представляет собой группу. В индексированной таблице разделом является каждая индексированная группа данных. Например, в индексированной таблице, показанной на рис. 8.3, все имена, начинающиеся с буквы А — другой и т.д.

ПРЕДУПРЕЖДЕНИЕ. Хотя технически возможно создать сгруппированную таблицу с индексом, вы не должны этого делать. В справочнике *iPhone Human Interface Guidelines* подчеркивается, что сгруппированные таблицы не должны предоставлять индексы.

Реализация простой таблицы

Рассмотрим простейший пример табличного представления, чтобы понять, как оно работает. В данном примере мы просто выведем список текстовых значений.

Создайте новый проект в среде Xcode. В данной главе мы вернемся к шаблону Single View Application, поэтому выбираем его. Назовите ваш проект Simple Table, выберите пункт Swift в списке Language и пункт Universal в списке Devices, а затем убедитесь, что флагок Use Core Data сброшен.

Проектирование представления

Зайдите в навигатор проекта и раскройте проект верхнего уровня Simple Table и папку Simple Table. Наше приложение настолько простое, что нам не требуются ни выходы, ни действия, так что выберите файл Main.storyboard для редактирования раскладовки. Если окно View не отображается, щелкните на его пиктограмме в окне Document Outline, чтобы открыть его. Затем найдите в библиотеке объектов элемент Table View (рис. 8.4) и перетащите его в окно View.

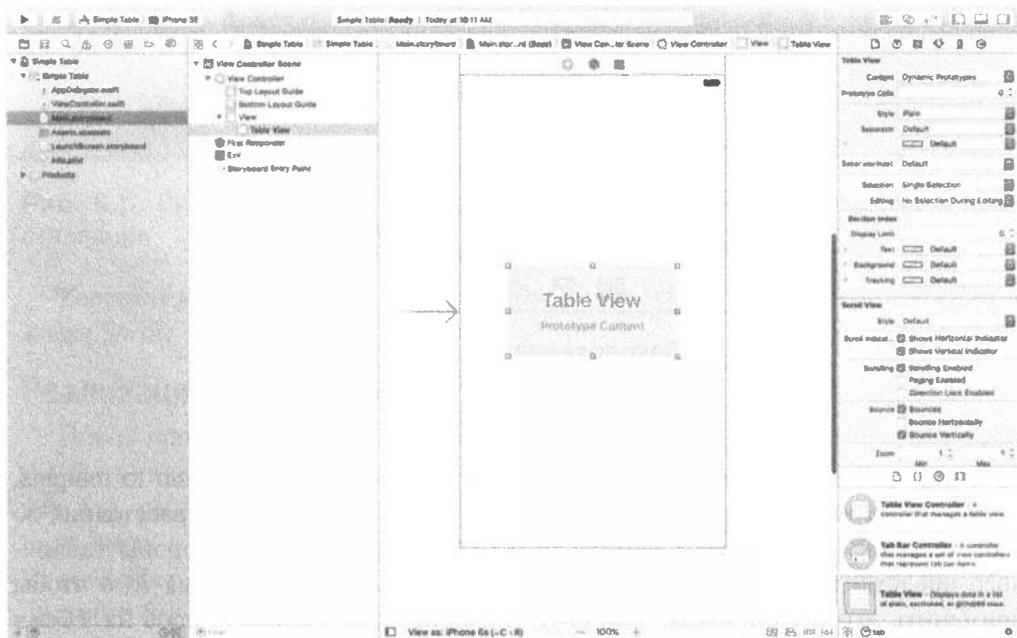


Рис. 8.4. Перетаскивание табличного представления из библиотеки на главное представление

Перетащите табличное представление в окно View и расположите его по центру родительского представления. Добавьте ограничения Auto Layout, чтобы убедиться, что табличное представление корректно расположено и имеет правильный размер, независимо от размера экрана. Выберите таблицу в окне Document Outline, а затем щелкните на пиктограмме Pin в правом нижнем углу редактора (рис. 8.5).

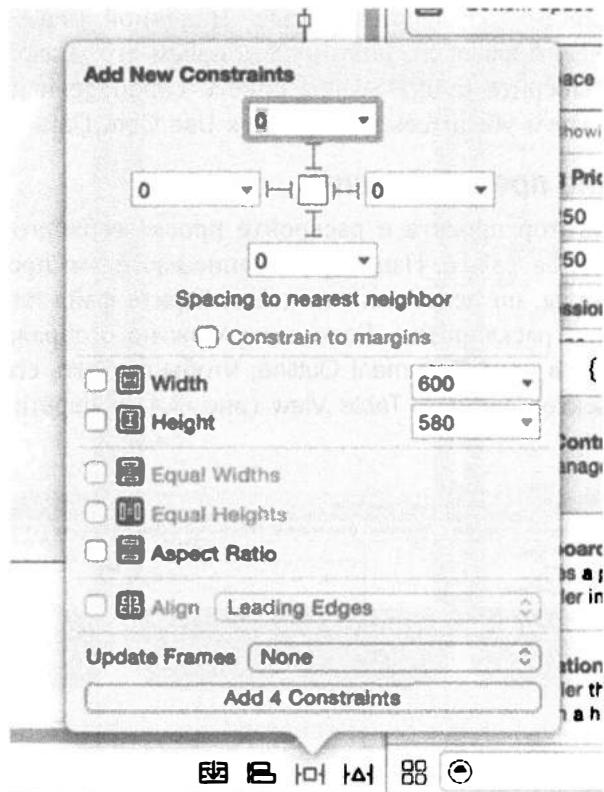


Рис. 8.5. Закрепление табличного представления таким образом, чтобы оно заполняло экран

В верхней части всплывающего окна сбросьте флагок *Constrain to margins*, щелкните на всех четырех пунктирных линиях и установите все расстояния во всех четырех полях ввода равными нулю. Это прикрепит все стороны табличного представления к сторонам родительского представления. Для того чтобы применить эти ограничения, замените значение поля *Update Frames* на *Items of New Constraints* и щелкните на кнопке *Add 4 Constraints*. Таблица должна изменить размеры и заполнить все представление.

Вновь выберите табличное представление в окне инспектора документов и нажмите комбинацию клавиш *<Option+⌘+6>*, чтобы вызвать инспектор связей. Вы увидите, что первые две доступные связи для табличного представления такие же, как и для представления селектора в предыдущей главе: *dataSource* и *delegate*. Перетащите указатель из кружка, расположенного рядом с каждой из связей, к пиктограмме *View Controller* в окне *Document Outline* или в точку, расположенную выше представления контроллера в редакторе раскладовки. Поступая так, мы делаем наш класс контроллера источником данных и делегатом для этой таблицы.

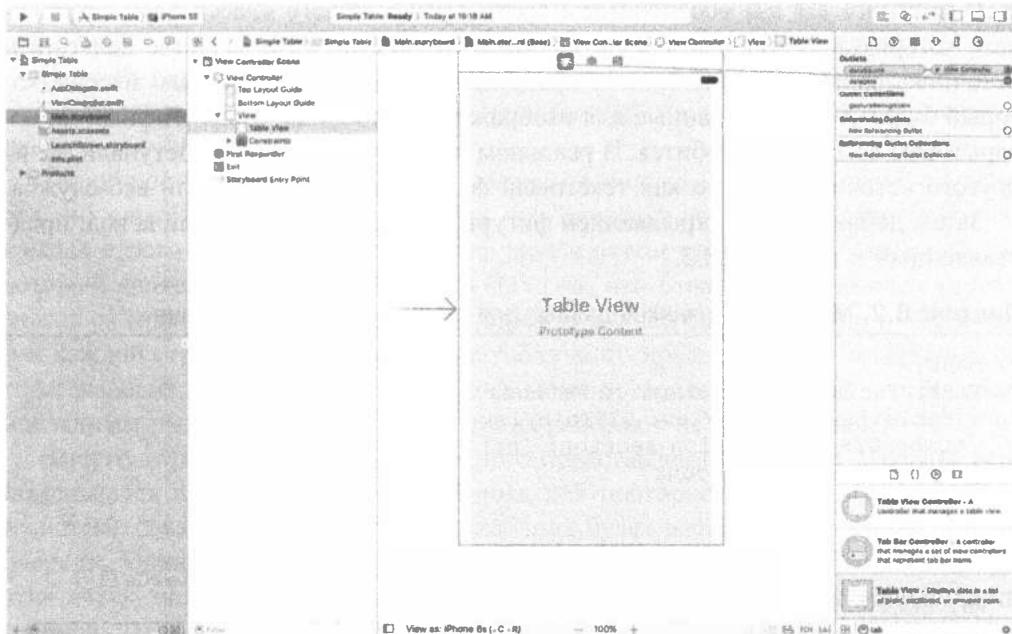


Рис. 8.6. Связывание выходов dataSource и delegate для табличного представления

Теперь мы приступим к программированию табличного представления на языке Swift.

Реализация контроллера

После прочтения предыдущей главы многие термины читателям уже должны быть понятными, и дополнительные разъяснения не требуются. Тем не менее для читателей, пропустивших предыдущую главу, мы постараемся сохранить связь с терминологией, введенной в первых главах. Щелкните на файле `ViewController.swift` и добавьте в объявление класса код, приведенный в листинге 8.1.

Листинг 8.1. Добавление кода в начало объявления класса для создания массива

```
class ViewController: UIViewController,
    UITableViewDataSource, UITableViewDelegate {
    private let dwarves = [
        "Sleepy", "Sneezy", "Bashful", "Happy",
        "Doc", "Grumpy", "Dopey",
        "Thorin", "Dorin", "Nori", "Ori",
        "Balin", "Dwalin", "Fili", "Kili",
        "Oin", "Gloin", "Bifur", "Bofur",
        "Bombur"
    ]
    let simpleTableIdentifier = "SimpleTableIdentifier"
```

В листинге 8.1 мы обеспечиваем соответствие нашего класса двум протоколам, которые необходимы для того, чтобы он действовал в качестве делегата и источника данных для табличного представления, а затем объявляем массив, который будет содержать данные для отображения в таблице, и идентификатор, который нам вскоре понадобится. В реальном приложении данные поступали бы из другого источника, такого как текстовый файл, список свойств или веб-служба.

Затем добавьте над закрывающей фигурной скобкой в конце файла код, представленный в листинге 8.2.

Листинг 8.2. Методы источников данных для табличного представления

```
// MARK:-
// MARK: Table View Data Source Methods
func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return dwarves.count
}

func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    var cell = tableView.dequeueReusableCell(withIdentifier:
        simpleTableIdentifier)
    if (cell == nil) {
        cell = UITableViewCell(
            style: UITableViewCellStyle.default,
            reuseIdentifier: simpleTableIdentifier)
    }
    cell?.textLabel?.text = dwarves[indexPath.row]
    return cell!
}
```

Эти методы являются частью протокола UITableViewDataSource. Первый из них, tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int, используется таблицей для запроса о количестве строк в конкретном разделе. Как можно ожидать, по умолчанию количество разделов равно единице, и этот метод будет вызываться для получения количества строк в этом одном разделе, составляющем список. Мы просто возвращаем количество элементов в нашем массиве.

Следующий метод, видимо, потребует определенного пояснения, так что рассмотрим его более внимательно:

```
func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
```

Этот метод вызывается табличным представлением, когда ему необходимо вывести одну из своих строк. Обратите внимание: вторым аргументом этого метода является экземпляр класса NSIndexPath. Это структура, которую табличное представление использует для того, чтобы поместить раздел и индексы строк в единый объект. Для того чтобы получить индекс строки или индекс

раздела из объекта класса `NSIndexPath`, вы вызываете либо его свойство `row`, либо свойство `section`, и оба они возвращают целочисленное значение.

Первый параметр, `tableView`, является ссылкой на создаваемую таблицу. Это позволяет нам создавать классы, которые действуют в качестве источника данных для нескольких таблиц.

Табличное представление может выводить только несколько строк за один раз, но сама таблица может хранить их значительно больше. Помните, что каждая строка в таблице представлена экземпляром класса `UITableViewCell`, который является подклассом класса `UIView`, что означает, что каждая строка может содержать дочерние представления. В случае больших таблиц хранение для каждой строки экземпляра ячейки табличного представления, независимо от того, выводится ли строка в настоящий момент, может приводить к огромным накладным расходам. К счастью, таблицы работают несколько иначе.

Вместо этого, когда ячейки табличного представления исчезают с экрана, они помещаются в очередь ячеек, доступных для повторного использования. Если в системе оказывается мало памяти, таблица будет избавляться от ячеек в этой очереди. Однако пока система имеет некоторый запас доступной памяти для этих ячеек, она будет хранить их на тот случай, если вы захотите использовать их вновь.

Каждый раз, когда ячейка табличного представления исчезает с экрана, существует довольно большая вероятность, что другая ячейка поступает на экран с другой стороны. Если эта новая строка может повторно использовать одну из ячеек, которые уже исчезли с экрана, то система сможет избежать накладных расходов, связанных с постоянным созданием и освобождением этих представлений. Для того чтобы воспользоваться преимуществами этого механизма, мы запрашиваем у табличного представления одну из свободившихся ячеек определенного типа с использованием объявленного ранее идентификатора класса `NSString`. По сути, мы запрашиваем повторно используемую ячейку типа `SimpleTableIdentifier`.

```
var cell = tableView.dequeueReusableCell(withIdentifier:  
    simpleTableIdentifier)
```

В этом примере таблица использует только один тип ячеек, но в более сложной таблице может потребоваться форматирование различных типов ячеек в зависимости от их содержания или позиции, и в этом случае придется использовать отдельный идентификатор ячеек таблицы для каждого отдельного типа ячеек.

Вполне возможно, что таблица не будет иметь никаких запасных ячеек (например, когда они изначально заполнены), поэтому после вызова мы проверяем значение переменной `cell` на равенство `nil`. Если это так, то мы вручную создаем новую ячейку табличного представления с помощью той же самой строки идентификатора. В какой-то момент неизбежно повторное использование создаваемой здесь ячейки, поэтому мы должны убедиться, что мы создаем ее с помощью `simpleTableIdentifier`.

```

if (cell == nil) {
    cell = UITableViewCell(
        style: UITableViewCellStyle.Default,
        reuseIdentifier: simpleTableIdentifier)
}

```

Вас заинтересовал класс `UITableViewCellStyle.default?`? Потерпите немного — мы доберемся до него, когда будем рассматривать стили ячеек табличного представления.

Теперь у нас есть ячейка табличного представления, которую мы можем вернуть таблице для использования. Таким образом, достаточно просто поместить в нее информацию, которая должна отображаться в этой ячейке. Отображение текста в строке таблицы — очень распространенная задача, поэтому ячейки табличного представления предоставляют свойство класса `UILabel` под названием `textLabel`, которое мы можем установить для отображения строк. Для этого нам надо получить корректную строку из нашего массива `dwarves` и использовать его для установки значения `textLabel` ячейки.

Однако, чтобы получить корректное значение, мы должны знать, какая строка таблицы запрошена. Мы получаем эту информацию из свойства строки `indexPath`. Мы используем номер строки таблицы, чтобы получить соответствующую строку из массива, присвоить ее свойству `textLabel.text` ячейки, а затем вернуть ее.

```

cell?.textLabel?.text = dwarves[indexPath.row]
return cell!

```

Скомпилируйте и запустите приложение, и увидите, как значения массива отображаются в виде таблицы (рис. 8.7, слева).

У читателей может возникнуть вопрос, зачем нужен оператор `?` в этом коде.

```

cell?.textLabel?.text = dwarves[indexPath.row]

```

Каждый случай использования оператора `?` представляет собой пример **опциональной последовательности** (optional chaining), позволяющей писать компактный код, даже если вы вызываете метод или обращаетесь к свойствам ссылки на объект, которая может быть равной `nil`. Первый оператор `?` требуется, потому что ячейка может содержать значение `nil`. Мы получим это значение, вызвав метод `dequeueReusableCellWithIdentifier()`, возвращающий значение типа `UITableViewCell?`. Разумеется, компилятор не учитывает тот факт, что мы явно проверяем, не равно ли возвращаемое значение `nil`, и создает новый объект класса `UITableViewCell`, гарантируя, что ячейка никогда не будет содержать значение `nil`, когда мы достигнем этой строки кода. Обратитесь к документации класса `UITableViewCell`, вы увидите, что свойство `textLabel` имеет тип `UILabel?`, поэтому оно тоже может быть равным `nil`. На самом деле этого тоже никогда не произойдет, потому что мы по умолчанию используем экземпляр `UITableViewCell`, который всегда включает метку. Естественно,

компилятор об этом не знает, поэтому мы используем оператор ? при разыменовании этого свойства. Эта ситуация часто встречается в программах на языке Swift.

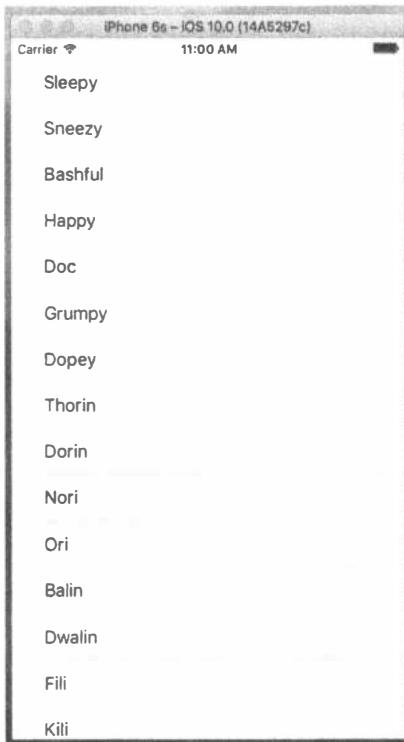


Рис. 8.7. Приложение Simple Table, демонстрирующее массив

Добавление изображения

Было бы неплохо, если бы мы могли добавить в каждую строку изображение. Можно предположить, что для этого мы должны создать подкласс UITableViewCell или добавить дочерние представления. На самом деле, если вы согласны на то, чтобы изображение находилось на левой стороне каждой строки, ничего делать не нужно. Ячейки табличного представления по умолчанию вполне могут обработать эту ситуацию. Давайте убедимся в этом.

Перетащите файлы star.png и star2.png из папки 08 – Star Image архива исходных кодов в папку Assets.xcassets проекта, как показано на рис. 8.8.

Мы планируем разместить эти пиктограммы в каждой строке табличного представления. Для этого достаточно создать объект класса UIImage для каждой из них и выполнить присваивание объекту класса UITableViewCell, когда табличное представление запросит источник данных для ячейки в каждой строке. Для этого в файле ViewController.swift модифицируйте метод tableView(_:cellForRowAtIndexPath:), как показано в листинге 8.3.

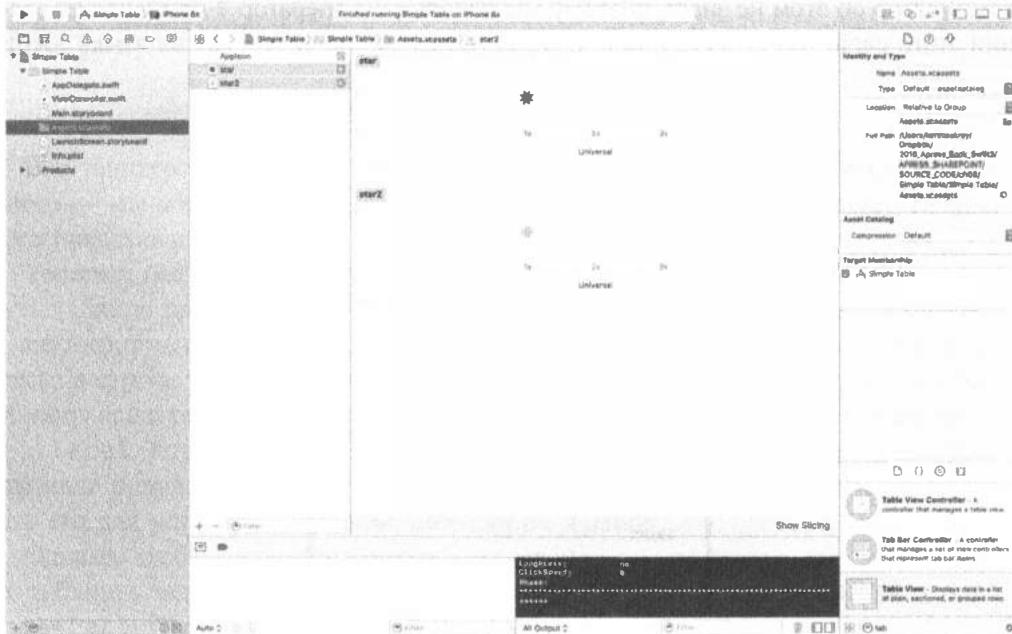


Рис. 8.8. Добавление изображений в папку Assets.xcassets

Листинг 8.3. Модификация метода для добавления изображения в каждую ячейку

```
func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    var cell = tableView.dequeueReusableCell(withIdentifier:
simpleTableIdentifier)
    if (cell == nil) {
        cell = UITableViewCell(
            style: UITableViewCellStyle.default,
            reuseIdentifier: simpleTableIdentifier)
    }
    let image = UIImage(named: "star")
    cell?.imageView?.image = image
    let highlightedImage = UIImage(named: "star2")
    cell?.imageView?.highlightedImage = highlightedImage

    cell?.textLabel?.text = dwarves[indexPath.row]
    return cell!
}
```

Это все. Каждая ячейка имеет свойство imageView типа UIImage, которое, в свою очередь, имеет свойство image, а также свойство highlightedImage. Изображение, задаваемое свойством image, появляется слева от текста ячейки и, когда ячейка выбрана, заменяется на highlightedImage, если таковое имеется. Вы просто присваиваете свойствам ячейки imageView.image и imageView.highlightedImage изображения, которые хотите отобразить.

Если сейчас вы скомпилируете и запустите приложение, то должны получить список с множеством красивых маленьких голубых звездочек слева от каждой строки (рис. 8.9). Если вы выберете любую строку, то звездочки в ней станут зелеными. Конечно, мы могли бы включить различные изображения для каждой строки в таблице или, немного потрудившись, использовать разные пиктограммы для разных категорий персонажей произведений Диснея и Толкиена.



Рис. 8.9. Мы используем свойство ячейки `imageView` для добавления изображения в каждую ячейку табличного представления

ЗАМЕЧАНИЕ. Класс `UIImage` использует механизм кеширования на основе имен файлов, поэтому он не будет загружать новое свойство изображения каждый раз при вызове `UIImage(named:)`. Вместо этого будет использоваться уже кешированная версия.

Использование стилей ячеек табличных представлений

До сих пор, работая с табличным представлением, мы использовали стиль ячейки по умолчанию (см. рис. 8.9), представленный константой стиля `UITableViewCellStyle.default`. Однако класс `UITableViewCell` включает несколько других предопределенных стилей ячеек, что позволит вам легко

добавить немного больше разнообразия в табличные представления. Эти стили используют три элемента ячеек.

- **Изображение.** Если изображение является частью указанного стиля, оно отображается слева от текста ячейки.
- **Текстовая метка.** Это основной текст ячейки. В использованном ранее стиле `UITableViewCellStyle.default` текстовая метка представляет собой единственный текст, отображаемый в ячейке.
- **Подробная текстовая метка.** Это вспомогательный текст ячейки, как правило, используемый в качестве пояснения.

Для того чтобы увидеть, на что похожи эти новые дополнения, добавьте следующий код в метод `tableView(_:cellForRowAtIndexPath:)` в файле `ViewController.swift`:

```
if indexPath.row < 7 {
    cell?.detailTextLabel?.text = "Mr Disney"
} else {
    cell?.detailTextLabel?.text = "Mr Tolkien"
}
```

Поместите этот код в текст метода непосредственно перед строкой `cell?.textLabel?.text = dwarves[indexPath.row]`.

Итак, мы задали подробный текст ячейки. Мы используем строку "Mr. Disney" для первых семи строк и "Mr. Tolkien" для остальных. Когда вы запустите этот код, каждая ячейка будет выглядеть так же, как и раньше (рис. 8.10). Это связано с тем, что мы используем стиль `UITableViewCellStyle.default`, который не использует подробный текст.



Рис. 8.10. Стиль ячеек по умолчанию выводит изображение и текст в одну строку

Теперь замените стиль `UITableViewCellStyle.default` стилем `UITableViewCellStyle.subtitle`

```
if (cell == nil) {
    cell = UITableViewCell(
        style: UITableViewCellStyle.subtitle,
        reuseIdentifier: simpleTableIdentifier)
}
```

Снова запустите приложение. В этом случае вы увидите оба текстовых элемента, один под другим (рис. 8.11).



Рис. 8.11. Стиль подзаголовка выводит подробный текст более мелким серым шрифтом под текстом метки

Замените стиль UITableViewCellStyle.subtitle стилем UITableViewCellStyle.value1 и вновь перестройте и запустите приложение. В этом случае вы увидите оба текстовых элемента в одной строке, но в разных частях ячейки (рис. 8.12).

Наконец замените стиль UITableViewCellStyle.value1 стилем UITableViewCellStyle.value2. Этот формат часто используется для отображения информации вместе с описательной меткой. При этом пиктограмма не выводится, а подробный текст помещается слева от текста метки (рис. 8.13). В этом случае подробный текст выступает в качестве метки, описывающей тип данных, хранящихся в текстовой метке.

После того как вы ознакомились с доступными стилями ячеек, вернемся к стилю UITableViewCellStyle.default, прежде чем продолжить работу. Далее в главе вы узнаете, как настроить внешний вид таблицы. Однако, прежде чем вы решите сделать это, убедитесь, что ни один из доступных стилей ячейки не позволяет сделать то, что вы хотите получить.

iPhone 6s – iOS 10.0 (14A5297c)	
Carrier	11:50 AM
✿ Sleepy	Mr Disney
✿ Sneezy	Mr Disney
✿ Bashful	Mr Disney
✿ Happy	Mr Disney
✿ Doc	Mr Disney
✿ Grumpy	Mr Disney
✿ Dopey	Mr Disney
✿ Thorin	Mr Tolkien
✿ Dorin	Mr Tolkien
✿ Nori	Mr Tolkien
✿ Ori	Mr Tolkien
✿ Balin	Mr Tolkien
✿ Dwalin	Mr Tolkien
✿ Fili	Mr Tolkien
✿ Kili	Mr Tolkien

Рис. 8.12. Стиль value1 выводит подробный текст голубыми буквами, выровненным по правой границе ячейки

iPhone 6s – iOS 10.0 (14A5297c)	
Carrier	11:53 AM
Sleepy	Mr Disney
Sneezy	Mr Disney
Bashful	Mr Disney
Happy	Mr Disney
Doc	Mr Disney
Grumpy	Mr Disney
Dopey	Mr Disney
Thorin	Mr Tolkien
Dorin	Mr Tolkien
Nori	Mr Tolkien
Ori	Mr Tolkien
Balin	Mr Tolkien
Dwalin	Mr Tolkien
Fili	Mr Tolkien
Kili	Mr Tolkien

Рис. 8.13. Стиль value2 не выводит пиктограмму и выводит подробный текст голубыми буквами слева от текстовой метки

Вы могли заметить, что мы сделали наш контроллер как источником данных, так и делегатом для этого табличного представления, но до сих пор мы фактически не реализовали ни один из методов протокола UITableViewDelegate. В отличие от представлений селектора, более простые табличные представления не требуют использования делегатов для своего функционирования. Источник данных предоставляет все необходимое для вывода таблицы. Цель делегата — настройка внешнего вида табличного представления и обработка определенного взаимодействия с пользователем. Давайте взглянем на некоторые из настроек конфигурации. Остальные настройки рассмотрим в следующей главе.

Настройка уровня отступа

Делегат может быть использован для указания того, что некоторые строки должны иметь отступ. В файле ViewController.swift добавьте к вашему коду следующий метод:

```
// MARK:-
// MARK: Table View delegate Methods

func tableView(_ tableView: UITableView,
              indentationLevelForRowAt indexPath: IndexPath) -> Int {
    return indexPath.row % 4
}
```

Этот метод задает **уровень отступа** для каждой строки на основе ее номера, так что строка 0 будет иметь нулевой уровень отступа, первая строка будет иметь уровень отступа 1 и т.д. Наличие оператора `%` приводит к тому, что отступ для четвертой строки становится нулевым и цикл повторяется. Уровень отступа — это просто целое число, которое говорит табличному представлению о необходимости сместить строку немного вправо. Чем больше это число, тем дальше вправо будет смещена строка. Вы можете использовать эту методику, например, чтобы указать, что одна строка подчинена другой строке, — точно так же, как приложение Mail показывает вложенные папки.

При повторном запуске приложения вы можете увидеть, как каждая строка в блоке из четырех строк оказывается немного правее предыдущей (рис. 8.14).



Рис. 8.14. Строки таблицы с отступами

Обработка выбора строки

Делегат таблицы может использовать два метода для обработки выбора пользователем некоторой строки. Один из методов вызывается перед выбором строки и может использоваться для предотвращения выбора строки и даже для изменения этого выбора. Реализуем этот метод и укажем, что первая строка не может быть выбрана. Добавьте приведенный далее метод в конец файла `ViewController.swift`.

```
func tableView(_ tableView: UITableView,
              willSelectRowAt indexPath: IndexPath) -> IndexPath? {
    return indexPath.row == 0 ? nil : indexPath
}
```

Этот метод получает объект `indexPath`, соответствующий элементу, который будет выбран. Наш код проверяет, какая строка была выбрана. Если это первая строка с нулевым индексом, то метод возвращает значение `nil`, указывающее, что строка не должна быть выбрана. В противном случае возвращается не модифицированное значение `indexPath` — так мы указываем, что процесс выбора можно продолжить.

Перед тем как скомпилировать и запустить приложение, реализуем также метод делегата, который вызывается после того, как строка была выбрана, и который, как правило, и представляет собой обработчик выбора. Здесь вы можете предпринимать любые действия, связанные с выбором строки пользователем. В следующей главе мы будем использовать этот метод для обработки представлений с подробным отображением, а в этой главе просто выведем сообщение о выбранной строке. Добавьте в конец файла `ViewController.swift` метод, приведенный в листинге 8.4.

Листинг 8.4. Вывод сообщения, когда пользователь касается строки

```
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    let rowValue = dwarves[indexPath.row]
    let message = "You selected \(rowValue)"

    let controller = UIAlertController(title: "Row Selected",
                                      message: message, preferredStyle: .alert)
    let action = UIAlertAction(title: "Yes I Did",
                               style: .default, handler: nil)
    controller.addAction(action)
    present(controller, animated: true, completion: nil)
}
```

Добавив этот метод, скомпилируйте и запустите приложение. Проверьте, сможете ли вы выбрать первую строку (это у вас не должно получиться), а затем выберите одну из остальных строк. Выбранная строка должна стать подсвеченной, а на экране появится сообщение о том, какая строка была выбрана (рис. 8.15).

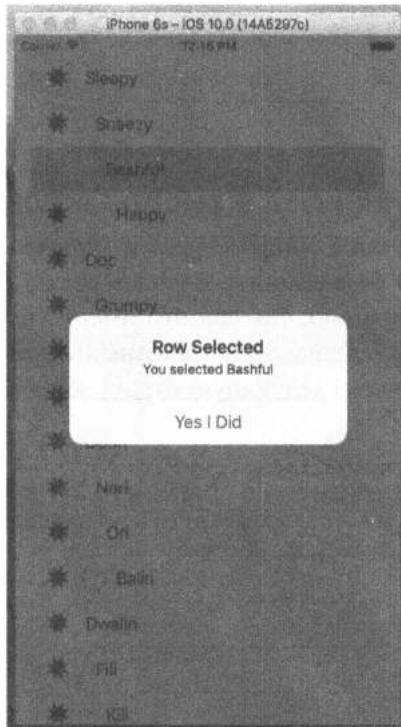


Рис. 8.15. В этом примере первую строку выбрать невозможно, а при выборе любой другой строки выводится сообщение

Вы можете изменить значение объекта `indexPath`, прежде чем передать его обратно, что приведет к выбору другой строки и/или раздела. Вряд ли вы будете прибегать к такому методу часто, так как для этого нужны веские основания. В подавляющем большинстве случаев, используя метод `tableView(_:willSelectRowAtIndexPath:)`, вы либо возвращаете переменную `indexPath` неизмененной, чтобы позволить выбор, либо возвращаетете значение `nil`, чтобы запретить его. Если вы действительно хотите изменить выбранную строку и/или раздел, используйте метод `NSIndexPath(forRow:, inSection:)`, чтобы создать новый объект класса `NSIndexPath` и вернуть его. Например, код, приведенный в листинге 8.5, гарантирует, что если вы попытаетесь выбрать четную строку, то на самом деле выберете строку, которая следует за ней.

Листинг 8.5. Возвращение следующей строки

```
func tableView(tableView: UITableView,
              willSelectRowAtIndexPath indexPath: NSIndexPath)
-> NSIndexPath? {
    if indexPath.row == 0 {
        return nil
    } else if (indexPath.row % 2 == 0) {
```

```
        return NSIndexPath(indexPath.row + 1,  
                           indexPath.section)  
    } else {  
        return indexPath  
    }  
}
```

Изменение размера шрифта и высоты строки

Предположим, мы хотим изменить размер шрифта, используемого в табличном представлении. В большинстве случаев вы не должны переопределять шрифт, заданный по умолчанию, так как это именно тот шрифт, который ожидают увидеть пользователи. Однако иногда бывают веские причины, чтобы изменить этот шрифт. Добавьте следующую строку кода в метод `tableView(_:cellForRowAtIndexPath:)`:

```
func tableView(_ tableView: UITableView,
              cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    var cell = tableView.dequeueReusableCell(withIdentifier:
simpleTableIdentifier)
    if (cell == nil) {
        cell = UITableViewCell(
            style: UITableViewCellStyle.default,
            reuseIdentifier: simpleTableIdentifier)
    }

    let image = UIImage(named: "star")
    cell?.imageView?.image = image
    let highlightedImage = UIImage(named: "star2")
    cell?.imageView?.highlightedImage = highlightedImage

    if indexPath.row < 7 {
        cell?.detailTextLabel?.text = "Mr Disney"
    } else {
        cell?.detailTextLabel?.text = "Mr Tolkien"
    }

    cell?.textLabel?.text = dwarves[indexPath.row]
    // Добавьте следующую строку
    cell?.textLabel?.font = UIFont.boldSystemFont(ofSize: 50)
    return cell!
}
```

Теперь, когда вы запустите приложение, значения в списке будут выводиться шрифтом большого размера, но не будут помещаться в строке (рис. 8.16).

Существуют два способа исправления ситуации. Для начала мы можем указать таблице, что все строки должны иметь заданную фиксированную высоту. Для этого устанавливаем ее свойство `rowHeight`:

```
tableView.rowHeight = 70
```

Если же требуется, чтобы разные строки имели разную высоту, можно реализовать метод `tableView(_:heightForRowAtIndexPath:)` делегата `UITableViewDelegate`. Добавьте приведенный ниже метод в класс контроллера.

```
func tableView(_ tableView: UITableView,
              heightForRowAt indexPath: IndexPath) -> CGFloat
{
    return indexPath.row == 0 ? 120 : 70
}
```

Мы только что указали табличному представлению, что строки должны быть высотой 70 пикселей, за исключением немного большей первой строки. Скомпилируйте и запустите приложение, и вы увидите, что строки стали существенно выше (рис. 8.17).



Рис. 8.16. Изменение шрифта, используемого для вывода ячеек табличного представления

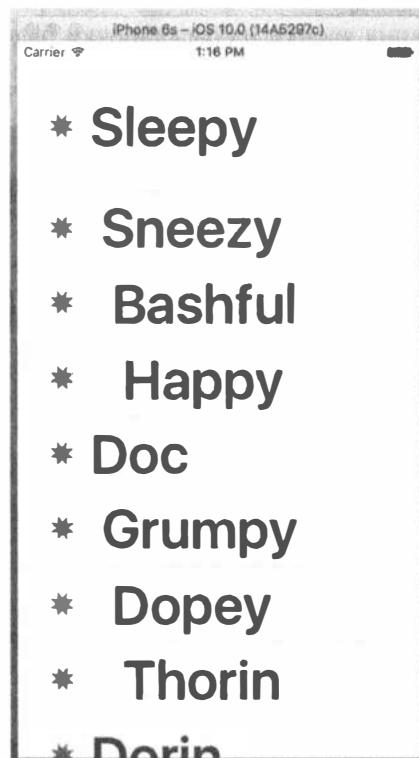


Рис. 8.17. Изменение высоты строк табличного представления с помощью делегата. Обратите внимание на несколько больший размер первой строки

Есть и другие задачи, которые может решить делегат, но большинство из них возникают, когда вы начинаете работать с иерархическими данными, а об этом мы поговорим в следующей главе. Для того чтобы получить дополнительную информацию, используйте браузер документации для изучения протокола UITableViewDelegate и посмотрите, какие еще методы доступны.

Настройка ячеек табличного представления

Можно очень многое сделать с табличными представлениями без внесения изменений, но часто требуется форматировать данные для каждой строки таким нетривиальным образом, что он просто не поддерживается классом `UITableViewCell` непосредственно. В таких случаях есть три основных подхода: один из них предусматривает программное добавление дочерних представлений в класс `UITableViewCell` при создании ячейки, второй включает загрузку множества дочерних представлений из nib-файла, а третий загружает ячейки из раскадровки. В этой главе мы рассмотрим две первые технологии, а пример создания ячейки из раскадровки будет приведен в главе 9.

Добавление дочерних представлений к ячейкам табличного представления

Для того чтобы показать, как использовать пользовательские ячейки, мы собираемся создать новое приложение с другим табличным представлением. В каждой строке будем выводить две строки информации (рис. 8.18). Наше приложение будет отображать название и цвет ряда потенциально знакомых компьютерных моделей, и мы будем выводить обе части информации в одной ячейке таблицы, добавляя дочерние представления в ячейку табличного представления.

Реализация приложения с пользовательскими табличными представлениями

Создайте новый проект в среде Xcode с использованием шаблона `Single View Application`. Назовите проект `Table Cells`. Используйте настройки предыдущего проекта. Щелкните на пункте `Main.storyboard`, чтобы отредактировать графический интерфейс в программе `Interface Builder`.

Добавьте в главное представление табличное представление и измените его так, чтобы оно занимало все представление, за исключением того, что верхняя часть таблицы должна быть выровнена по нижней части строки состояния, а не по верхней части представления. Используйте инспектор связей для задания источника данных для контроллера представления, как мы это делали для простой таблицы приложения. Затем используйте кнопку `Pin` в нижней части окна для создания ограничений между краями таблицы представления и краями его родительского элемента и строкой состояния. Вы можете использовать те же настройки, что и показанные на рис. 8.5, так как значения, которые вы укажете в полях ввода в верхней части всплывающего окна, по умолчанию являются расстояниями между табличным представлением и ее ближайшим соседом во всех четырех направлениях. В заключение сохраните раскадровку.

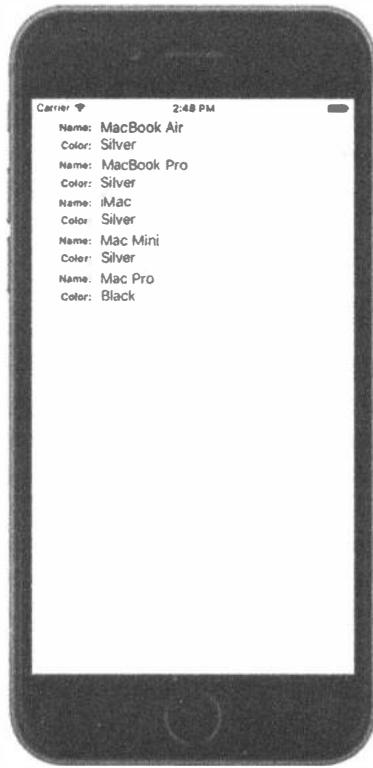


Рис. 8.18. Добавление дочернего представления в ячейку табличного представления позволяет получить многострочные ячейки

Создание подкласса UITableViewCell

До сих пор все детали макета ячеек автоматически определялись стандартным табличным представлением. В нашем коде контроллера не было запутанных деталей, регламентирующих, где должны размещаться метки и изображения, что позволяло просто передавать значения в ячейки, не заботясь об их внешнем виде. Таким образом, проблемы презентации были вынесены за пределы контроллера, что представляло собой несомненное преимущество. В нашем проекте мы создадим подкласс класса UITableViewCell для новой ячейки, который будет самостоятельно определять детали макета, чтобы наш контроллер был как можно более простым.

Добавление новых ячеек

Выберите папку Table Cells в окне навигатора проекта и нажмите комбинацию клавиш $<\text{⌘}+\text{N}>$, чтобы создать новый файл. В открывшемся окне помощника выберите пиктограмму Cocoa Touch Class из раздела iOS и щелкните на кнопке Next. На следующем экране введите строку NameAndColorCell в

качестве имени класса, выберите пункт `UITableViewCell` во всплывающем списке `Subclass of`, снова щелкните на кнопке `Next` и на последнем экране щелкните на кнопке `Create`.

Выберите файл `NameAndColorCell.swift` в окне навигатора проекта и добавьте в него следующий код:

```
class NameAndColorCell: UITableViewCell {
    var name: String = ""
    var color: String = ""
    var nameLabel: UILabel!
    var colorLabel: UILabel!
```

Здесь мы определили два свойства (`name` и `color`) интерфейса ячейки, которые наш контроллер будет использовать для передачи значений каждой ячейке. Мы также добавили пару свойств, которые будем использовать для обращения к некоторым дочерним представлениям, которые будут добавлены к нашей ячейке. Она будет содержать четыре дочерних представления, два из которых — метки с фиксированным содержимым, а два других — с содержимым, которое будет изменяться для каждой строки.

Это все свойства, которые требовалось добавить, так что перейдем к коду. Мы должны перекрыть инициализатор ячейки табличного представления `init(style:reuseIdentifier:)` и добавить код для создания представлений, которые нам надо будет выводить (листинг 8.6).

Листинг 8.6. Метод `init()` ячейки табличного представления

```
override init(style: UITableViewCellStyle, reuseIdentifier: String?) {
    super.init(style: style, reuseIdentifier: reuseIdentifier)

    let nameLabelRect = CGRect(x: 0, y: 5, width: 70, height: 15)
    let nameMarker = UILabel(frame: nameLabelRect)
    nameMarker.textAlignment = NSTextAlignment.right
    nameMarker.text = "Name:"
    nameMarker.font = UIFont.boldSystemFont(ofSize: 12)
    contentView.addSubview(nameMarker)

    let colorLabelRect = CGRect(x: 0, y: 26, width: 70, height: 15)
    let colorMarker = UILabel(frame: colorLabelRect)
    colorMarker.textAlignment = NSTextAlignment.right
    colorMarker.text = "Color:"
    colorMarker.font = UIFont.boldSystemFont(ofSize: 12)
    contentView.addSubview(colorMarker)

    let nameValueRect = CGRect(x: 80, y: 5, width: 200, height: 15)
    nameLabel = UILabel(frame: nameValueRect)
    contentView.addSubview(nameLabel)

    let colorValueRect = CGRect(x: 80, y: 25, width: 200, height: 15)
    colorLabel = UILabel(frame: colorValueRect)
    contentView.addSubview(colorLabel)
}
```

Это довольно просто. Мы создали четыре метки типа `UILabel` и добавили их в ячейку табличного представления. Ячейка табличного представления уже содержит дочернее представление класса `UIView` с именем `contentView`, которое используется для группирования всех дочерних представлений. В результате нам удалось избежать добавления меток как дочерних представлений непосредственно в ячейку табличного представления. Мы просто добавили их в объект `contentView`.

Две из этих меток содержат статический текст. Метка `nameMarker` содержит текст `Name:`, метка `colorMarker` — текст `Color:`. Это метки, которые мы не будем изменять, но обе они содержат текст, выровненный по правому краю с помощью атрибута `NSTextAlignment.right`.

Оставшиеся две метки используются для вывода данных, зависящих от строки. Напомним, что позже нам потребуется способ определения содержимого этих полей, поэтому сохраняем ссылки на них в свойствах, которые объявили ранее.

Поскольку мы переопределили назначенный инициализатор класса ячейки табличного представления, компилятор языка Swift требует от нас предоставить также реализацию инициализатора `init(coder:)`. Этот инициализатор никогда не будет вызываться в нашем примере приложения, так что просто добавим три следующие строки кода:

```
required init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

В главе 13 мы подробно рассмотрим этот инициализатор.

Теперь внесем в класс `NameAndColorCell` последние изменения — добавим логику установки свойств `name` и `color`. Изменим объявления этих свойств следующим образом:

```
var name: String = "" {
    didSet {
        if (name != oldValue) {
            nameLabel.text = name
        }
    }
}
var color: String = "" {
    didSet {
        if (color != oldValue) {
            colorLabel.text = color
        }
    }
}
```

Здесь мы добавляем код, чтобы убедиться, что при изменении значения свойства `name` или `color` свойству `text` соответствующей метки в той же ячейке таблицы присваивается это же значение.

Реализация кода контроллера

Напишем простой контроллер, отображающий данные в новых ячейках, внешний вид которых мы определили самостоятельно. Откройте файл `ViewController.swift` и добавьте в него код из листинга 8.7.

Листинг 8.7. Вывод значений в пользовательской ячейке

```
class ViewController: UIViewController, UITableViewDataSource {
    let cellTableIdentifier = "CellTableIdentifier"
    @IBOutlet var tableView:UITableView!
    let computers = [
        ["Name" : "MacBook Air", "Color" : "Silver"],
        ["Name" : "MacBook Pro", "Color" : "Silver"],
        ["Name" : "iMac", "Color" : "Silver"],
        ["Name" : "Mac Mini", "Color" : "Silver"],
        ["Name" : "Mac Pro", "Color" : "Black"]
    ]
    override func viewDidLoad() {
        super.viewDidLoad()
        // Дополнительная настройка представления после загрузки
        tableView.register(NameAndColorCell.self,
                           forCellReuseIdentifier: cellTableIdentifier)
    }
}
```

Мы сделали наш контроллер соответствующим протоколу `UITableViewDataSource`, а также добавили имя идентификатора ячейки и массив словарей. Каждый словарь содержит имя и информацию о цвете для одной строки в таблице. Название для этой строки хранится в словаре с ключом `Name`, а цвет — в словаре с ключом `Color`. Мы собираем все словари в один массив, который и является нашими данными для таблицы. Мы также добавили выход для табличного представления, так что нам надо подключить его к раскладовке. Выберите файл `Main.storyboard` в окне `Document Outline`, нажмите клавишу `<Control>` и перетащите указатель от пиктограммы `View Controller` к пиктограмме `Table View`. Отпустите кнопку мыши и выберите `tableView` из всплывающего окна для связи табличного представления с выходом.

Теперь добавьте в конец файла `ViewController.swift` код из листинга 8.8.

Листинг 8.8. Методы источника данных для табличного представления

```
// MARK: -
// MARK: Table View Data Source Methods

func tableView(_ tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
    return computers.count
}

func tableView(_ tableView: UITableView,
               cellForRowAt indexPath: IndexPath) -> UITableViewCell {
```

```

let cell = tableView.dequeueReusableCell(
   (withIdentifier: cellTableIdentifier, for: indexPath)
    as! NameAndColorCell

let rowData = computers[indexPath.row]
cell.name = rowData["Name"]!
cell.color = rowData["Color"]!

return cell
}

```

Вы уже видели эти методы в предыдущем примере — они принадлежат протоколу `UITableViewDataSource`. Обратите внимание на метод `tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)`, поскольку здесь мы используем интересную возможность: табличное представление может использовать разновидность реестра для создания при необходимости новой ячейки. Это означает, что, когда мы регистрируем все повторно используемые идентификаторы, которые собираемся использовать для табличного представления, мы всегда сможем получить доступ к свободным ячейкам. В предыдущем примере мы использовали метод `dequeueReusableCell`, который также использует реестр, но возвращает `nil`, если идентификатор, который мы передаем ему, не зарегистрирован. Возвращаемое значение `nil` используется как сигнал, что нам нужно создать и заполнить новый объект класса `UITableViewCell`. Метод, который мы используем здесь, никогда не возвращает значение `nil`:

```
dequeueReusableCell(withIdentifier: cellTableIdentifier, for: indexPath)
```

Как же он получает объект ячейки таблицы? Для этого он использует идентификатор, который мы передаем ему как ключ в реестре и добавляем запись в реестр, которая отображается на идентификатор ячейки нашей таблицы в методе `viewDidLoad()`.

```
tableView.register(NameAndColorCell.self,
    forCellReuseIdentifier: cellTableIdentifier)
```

Что произойдет, если мы передадим незарегистрированный идентификатор? В этом случае метод `dequeueReusableCell` завершится аварийно. Это плохо звучит, но в данном случае это было бы результатом ошибки, которую можно обнаружить в процессе разработки. Таким образом, нам не нужно включать код, проверяющий возвращаемое значение `nil`, так как этого никогда не произойдет.

Закончив создание новой ячейки, мы можем использовать аргумент `indexPath`, определяющий строку таблицы, в которой находится требуемая ячейка, а затем использовать эту переменную строки для определения правильного словаря для требуемой строки. Вспомните, что словарь содержит две пары “ключ–значение”: одну — для `name`, другую — для `color`.

```
let rowData = computers[indexPath.row]
```

Теперь осталось только заполнить ячейки данными из выбранной строки, используя свойства, определенные в нашем подклассе.

```
cell.name = rowData["Name"]!  
cell.color = rowData["Color"]!
```

Как вы видели ранее, установка этих свойств приводит к копированию значений в метки имени и цвета в ячейке табличного представления.

Скомпилируйте и запустите приложение. Вы должны увидеть таблицу строк, каждая из которых, в свою очередь, состоит из двух строк данных (рис. 8.19).

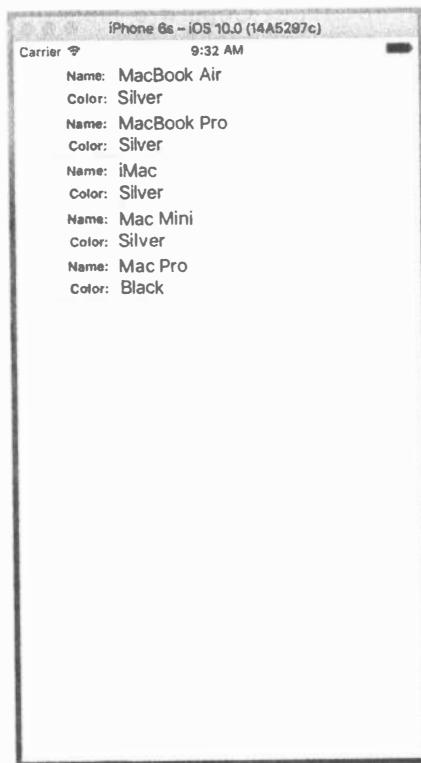


Рис. 8.19. Пользовательские ячейки табличного представления, созданные программно

Возможность добавления представлений в табличные представления обеспечивает больше гибкости, чем использование одной стандартной ячейки табличного представления, но программное создание, позиционирование и добавление всех дочерних представлений может оказаться достаточно утомительным. Было бы неплохо, если бы мы могли конструировать ячейки табличного представления графически, используя инструменты редактирования графического интерфейса пользователя в Xcode. К счастью, как уже упоминалось ранее, можно

использовать программу Interface Builder для разработки ячеек табличного представления, а затем при создании новой ячейки просто загрузить представления из раскладовки или XIB-файла.

Загрузка объекта класса UITableViewCell из XIB-файла

Мы собираемся воссоздать тот же двусторонний интерфейс, который только что создали в коде, но на этот раз используя визуальные возможности, которые среда Xcode предоставляет в программе Interface Builder. Для этого создадим новый XIB-файл, который будет содержать ячейку табличного представления, и настроим вид ее представления с помощью программы Interface Builder. Затем, когда нам будет нужна ячейка табличного представления для представления строки, вместо создания стандартной ячейки табличного представления мы просто загрузим XIB-файл, содержащий ячейку табличного представления и ее дочерние представления, и используем выход, подключенный к ячейке, чтобы найти метки и задать имя и цвет. Помимо применения программы Interface Builder, мы упростим некоторые фрагменты кода. Прежде чем продолжить, имеет смысл сделать копию проекта Table Cells, в которую можно будет вносить описываемые далее изменения. Для этого выйдите из среды Xcode и заархивируйте папку проекта под узнаваемым именем, например, Table Cells Orig.zip, что означает “исходный проект Table Cells” (рис. 8.20).

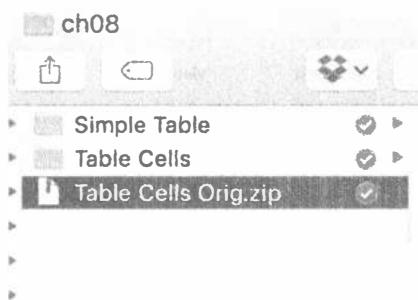


Рис. 8.20. Папку проекта можно заархивировать, чтобы создать базовую версию, в которой можно будет вернуться

Сначала внесем несколько изменений в класс NameAndColorCell в файле NameAndColorCell.swift. На первом этапе помечаем свойства nameLabel и colorLabel как выходы, так что можем использовать их в программе Interface Builder:

```
@IBOutlet var nameLabel: UILabel!
@IBOutlet var colorLabel: UILabel!
```

Теперь вспомним о настройках, которые мы делали в методе init(style: UITableViewCellStyle, reuseIdentifier: String?), в котором создавали наши метки. Все это теперь не нужно. Фактически мы должны просто удалить весь

метод, так как всю эту настройку выполняет программа Interface Builder. Поскольку мы больше не переопределяем никакие инициализаторы базового класса, можем удалить и `init(coder:)`.

После всех этих изменений класс ячейки стал еще меньше и проще. Это единственная функция, работающая с данными для меток. Теперь мы должны заново создать ячейки и их метки в программе Interface Builder.

Проектирование ячейки табличного представления с помощью программы Interface Builder

Щелкните правой кнопкой мыши на папке `Table Cells` в среде Xcode и выполните команду `New File...` из всплывающего меню. На левой панели помощника, предназначенного для создания новых файлов, щелкните на кнопке `User Interface` (выбрав ее в разделе iOS, а не в watchOS, tvOS или macOS). На правой верхней панели выполните команды `User Interface` и `Empty`, а затем щелкните на кнопке `Next` (рис. 8.21). На следующей странице экрана выберите имя файла `NameAndColorCell.xib`. Убедитесь, что главный каталог проекта выбран в браузере файлов и что группа `Table Cells` выбрана в раскрывающемся списке `Group`. Щелкните на кнопке `Create` для создания нового XIB-файла.

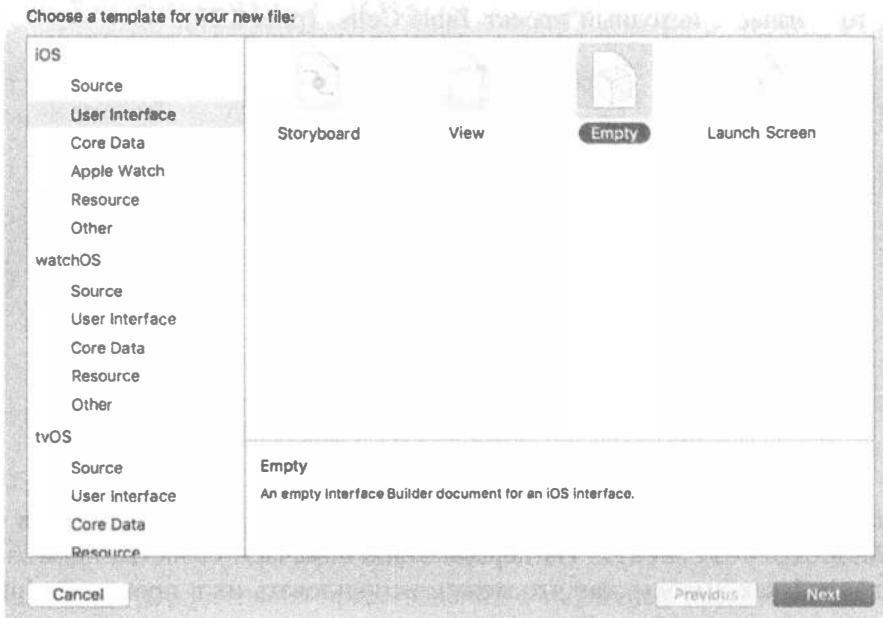


Рис. 8.21. Создание пустого файла пользовательского интерфейса, который станет XIB-файлом нашей ячейки

Выберите файл `NameAndColorCell.xib` в окне навигатора проекта, чтобы открыть этот файл для редактирования. До сих пор мы выполняли все редактирование графического интерфейса в раскладовке, а теперь используем вместо

этого nib-файл. Большинство действий похожи и будут вам очень знакомы, но есть и несколько различий. Одним из главных отличий является то, что, хотя файл раскадровки и фокусируется на сценах, которые связывают контроллер представления и само представление, в nib-файле нет никакого принудительного связывания. В действительности nib-файл часто не содержит объект реального контроллера вообще, а только прокси, который называется *File's Owner*. Если вы откроете окно Document Outline, то увидите его там над элементом *First Responder*.

Найдите в библиотеке элемент *Table View Cell* (рис. 8.22) и перетащите его в область макетирования графического интерфейса.



Рис. 8.22. Перетаскиваем элемент *Table View Cell* из библиотеки на канву

Нажмите комбинацию клавиш *<Option+⌘+4>*, чтобы перейти в окно инспектора атрибутов (рис. 8.23). Первое поле, которое вы здесь увидите, — *Identifier*. Это повторно используемый идентификатор, который мы уже применяли в нашем коде. Если это ничего вам не напомнило, вернитесь назад и найдите в главе упоминание класса *CellTableIdentifier*. Установите атрибут *Identifier* равным *CellTableIdentifier*.

Идея заключается в том, что, когда мы получаем ячейку для повторного использования (например, из-за прокрутки новой ячейки в представлении), мы хотим убедиться, что получим правильный тип ячейки. При создании объекта этой конкретной ячейки из nib-файла вы можете вызвать заполнение повторно используемого идентификатора переменной экземпляра именем, введенным в поле *Identifier*, — в данном случае им является *CellTableIdentifier*.

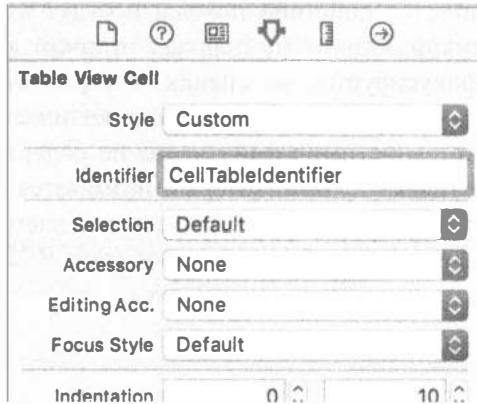


Рис. 8.23. Инспектор атрибутов ячейки табличного представления

Представьте себе сценарий, в котором вы создали таблицу с заголовком, а затем ряд средних ячеек. Если при прокрутке средняя ячейка входит в представление, важно, чтобы вы получили для повторного использования среднюю ячейку, а не ячейку заголовка. Поле Identifier позволяет соответствующим образом помечать ячейки.

Наш следующий шаг заключается в редактировании представления содержащего ячейки таблицы. Сначала выберите ячейку таблицы в области редактирования и перетащите ее нижний край, чтобы сделать ячейку немного выше. Добавьтесь высоты, равной 65. Перейдите к библиотеке, перетащите четыре элемента управления Label из библиотеки и поместите их в представление содержимого, используя рис. 8.24 в качестве ориентира. Обратите внимание на то, что метки будут находиться слишком близко к верхней и нижней направляющим, чтобы они могли оказать существенную помощь, но левая направляющая и направляющие выравнивания должны корректно служить своей цели. Заметим, что вы можете перетащить одну метку, а затем перетаскивать ее с нажатой клавишей <Option>, чтобы создавать копии, если такой подход для вас проще.



Рис. 8.24. Представление содержимого ячейки табличного представления с четырьмя метками

Дважды щелкните на верхней левой метке и измените ее текст на Name:, а текст нижней левой метки — на Color::.

Выберите обе эти метки и щелкните на маленькой кнопке с буквой T в поле Field инспектора атрибутов. В результате откроется маленькая панель, содержащая выпадающий список Font. Щелкните на ней и выберите пункт *System Bold*. Если необходимо, выберите два остальных поля меток справа, перетащите их вправо, чтобы освободить немного места, и измените размер двух других меток так, чтобы видеть только что введенный текст. Наконец измените размер двух правых меток так, чтобы они оказались растянутыми до правой направляющей. Рис. 8.25 должен дать вам представление о том, как выглядит окончательное представление содержимого ячейки.

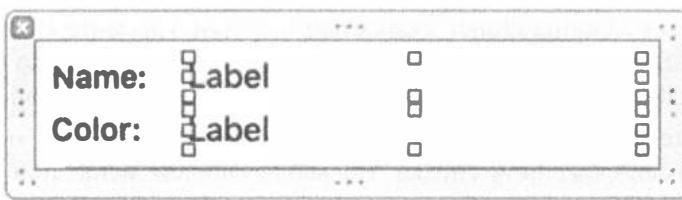


Рис. 8.25. Представление содержимого ячейки табличного представления с измененными названиями и шрифтом левых меток и с передвинутыми с изменением размеров правыми метками

Как обычно при создании нового макета, мы должны добавить ограничения Auto Layout. Общая идея заключается в том, чтобы прикрепить метки левой стороны макета к левой стороне ячейки, а метки правой стороны — к правой. Мы также должны разделить метки верхней и нижней частей ячейки по вертикали. Мы свяжем каждую метку левой стороны с соответствующей меткой справа. Ниже описано, как это сделать.

1. Щелкните на метке *Name:*, нажмите клавишу *<Shift>*, а затем щелкните на метке *Color:*. Щелкните на пиктограмме Pin, расположенной под окном редактора nib-файла, установите флажок *Equal Width* и щелкните на кнопке *Add 1 Constraint*. При этом вы увидите предупреждение механизма Auto Layout (не беспокойтесь — мы исправим ситуацию, когда добавим дополнительные ограничения).
2. Оставив две метки выбранными, откройте окно инспектора размеров и найдите раздел *Content Hugging Priority*. Если вы его не видите, отмените выбор меток и вновь выберите их. Значения в этих полях определяют, насколько метки сопротивляются расширению в дополнительное пространство. Мы не хотим, чтобы метки расширялись по горизонтали вообще, поэтому измените значение в поле *Horizontal* с 251 на 500. Любое значение, большее 251, подходит — надо лишь, чтобы оно было больше, чем значение *Content Hugging Priority* для двух правых меток, т.е. чтобы любое дополнительное горизонтальное пространство выделялось именно им.

3. Нажмите клавишу <Control> и перетащите указатель от метки Color: к метке Name:, выполните команду Vertical Spacing во всплывающем меню и нажмите клавишу <Return>.
4. Нажмите клавишу <Control> и перетаскивайте указатель вверх и влево от метки Name: по направлению к верхнему левому углу ячейки до тех пор, пока фон ячейки не станет полностью голубым. Нажав клавишу <Shift>, во всплывающем меню выполните команды Leading Space to Container Margin и Top Space to Container Margin, а затем нажмите клавишу <Return>.
5. Нажмите клавишу <Control> и перетаскивайте указатель от метки вниз и влево от метки Color: по направлению к нижнему левому углу ячейки до тех пор, пока фон ячейки не станет полностью голубым. Нажмите клавишу <Shift> и во всплывающем меню выполните команды Leading Space to Container Margin и Bottom Space to Container Margin, а затем нажмите клавишу <Return>.
6. Нажмите клавишу <Control> и перетащите указатель от метки Name: к метке, расположенной справа. Во всплывающем меню при нажатой клавише <Shift> выполните команды Horizontal Spacing и Baseline, а затем нажмите клавишу <Return>. Нажмите клавишу <Control> и перетаскивайте указатель от верхней метки вправо по направлению к правому краю ячейки, пока фон ячейки не станет голубым. Во всплывающем меню выберите Trailing Space to Container Margin.
7. Аналогично нажмите клавишу <Control> и перетащите указатель от метки Color: к метке, расположенной справа от нее. Во всплывающем меню при нажатой клавише <Shift> выполните команды Horizontal Spacing и Baseline и нажмите клавишу <Return>. Нажмите клавишу <Control> и перетаскивайте указатель от нижней метки вправо по направлению к правому краю ячейки, пока фон ячейки не станет голубым. Во всплывающем меню выполните команду Trailing Space to Container Margin и нажмите клавишу <Return>.
8. Выберите пиктограмму Content View в окне Document Outline, а затем выполните команду Editor⇒Resolve Auto Layout Issues⇒Update Frames, если она доступна. Четыре метки переместятся в свои окончательные местоположения, как показано на рис. 8.26. Если вы увидите что-то иное, удалите все ограничения в окне Document Outline и попробуйте выполнить все заново.



Рис. 8.26. Окончательная позиция метки внутри ячейки

Теперь мы должны сообщить программе Interface Builder, что ячейка табличного представления является не стандартной ячейкой, а особым подклассом. В противном случае мы не сможем связать выходы с соответствующими

ячейками. Выберите ячейку табличного представления щелчком на элементе CellTableIdentifier в окне Document Outline, откройте инспектор идентичности, нажав комбинацию клавиш $<\text{Option}+\text{⌘}+3>$, и выберите пункт NameAndColorCell в списке Class (рис. 8.27).

Затем перейдите в окно инспектора связей, нажав комбинацию клавиш $<\text{Option}+\text{⌘}+6>$, где вы увидите выходы nameLabel и colorLabel (рис. 8.28).

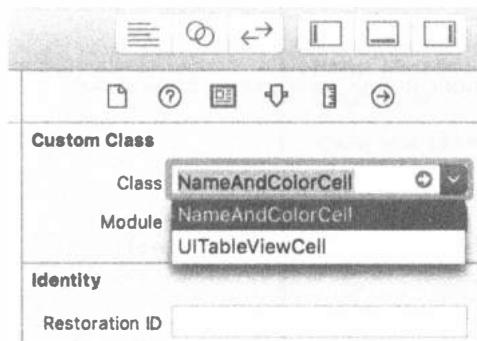


Рис. 8.27. Настройка нашего класса

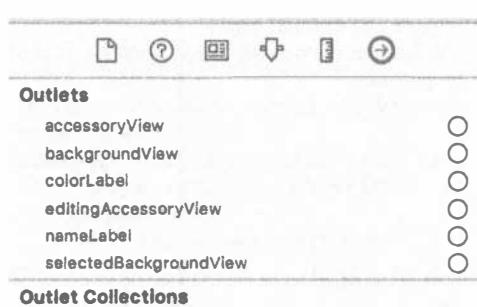


Рис. 8.28. Выходы nameLabel и nameLabel

Нажмите клавишу $<\text{Control}>$ и перетащите указатель от выхода nameLabel к верхней метке справа в ячейке таблицы и от выхода colorLabel к нижней метке справа, как показано на рис. 8.29.

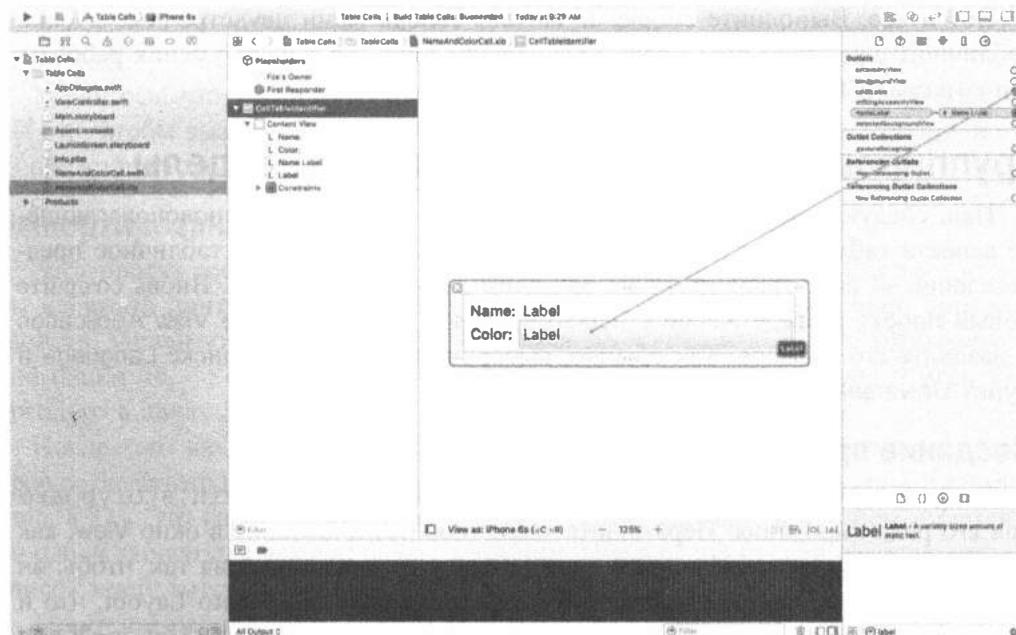


Рис. 8.29. Связывание выходов с именем и меткой цвета

Использование новой ячейки табличного представления

Для того чтобы использовать спроектированную ячейку, следует внести некоторые довольно простые изменения в метод viewDidLoad() в файле ViewController.swift (листинг 8.9).

Листинг 8.9. Модификация метода viewDidLoad() для использования новой ячейки

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка представления после загрузки, обычно из
    // nib-файла
    tableView.register(NameAndColorCell.self,
                       forCellReuseIdentifier: cellTableIdentifier)
    let xib = UINib(nibName: "NameAndColorCell", bundle: nil)
    tableView.register(xib,
                       forCellReuseIdentifier: cellTableIdentifier)
    tableView.rowHeight = 65
}
```

Так же, как класс может быть связан с повторно используемым идентификатором (как вы видели в предыдущем примере), табличное представление может отслеживать, какие nib-файлы должны быть связаны с конкретным повторно используемым идентификатором. Это позволяет регистрировать ячейки для каждого вида строк, используемых в приложении с помощью классов или nib-файла, а метод dequeueReusableCell(_:forIndexPath:) всегда будет возвращать готовую к использованию ячейку.

Вот и все. Выполните сборку и запуск. Теперь ваши двухстрочные ячейки табличного представления полностью зависят лишь от вашего умения работать с программой Interface Builder (рис. 8.30).

Группированные и индексированные разделы

Наш следующий проект посвящен изучению еще одного основополагающего аспекта таблиц. Мы по-прежнему будем использовать одно табличное представление — без иерархий, — но разделим данные на разделы. Вновь создайте новый проект в среде Xcode с использованием шаблона Single View Application и назовите его Sections. Как обычно, выберите пункт Swift в списке Language и пункт Universal в списке Devices.

Создание представления

Откройте папку Sections и дважды щелкните на файле Main.storyboard для его редактирования. Перетащите табличное представление в окно View, как мы делали это раньше. Измените верх табличного представления так, чтобы он был ниже строки состояния, и добавьте те же ограничения Auto Layout, что и в примере Table Cells. Затем нажмите комбинацию клавиш <Option+⌘+6> и соедините подключение dataSource с пиктограммой View Controller.

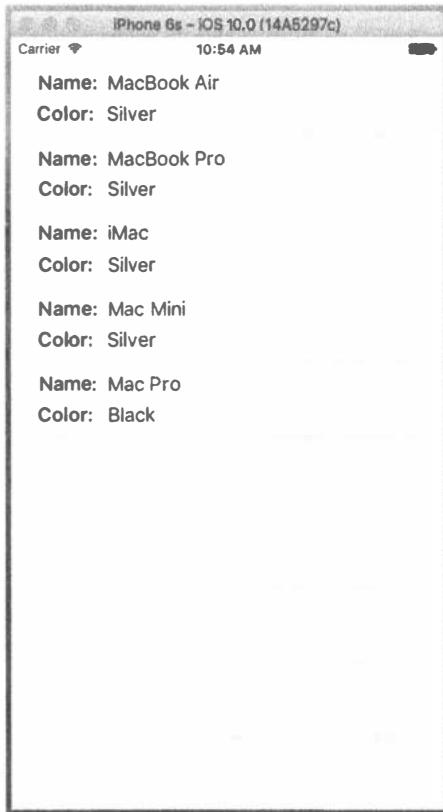


Рис. 8.30. Результаты настройки ячейки

Убедитесь, что таблица выбрана, и нажмите комбинацию клавиш <Option+ $\text{⌘}+4$ >, чтобы вызвать инспектор атрибутов. Измените значение **Style** табличного представления с **Plain** на **Grouped** (рис. 8.31). Сохраните раскладовку.

Импортирование данных

Этот проект требует для работы изрядного количества данных. Для того чтобы избежать длительного набора текста, мы предоставляем еще один список свойств, упрощающих работу. Найдите файл sortednames.plist из вложенной папки 08 – Sections Data архива исходных текстов примеров и перетащите его в папку **Sections**.

После того как файл sortednames.plist будет успешно добавлен в ваш проект, щелкните на нем, чтобы получить представление о том, как он выглядит (рис. 8.32). Это список свойств, который содержит словарь, в котором имеется по одной записи для каждой буквы алфавита. Под каждой буквой располагается список имен, которые начинаются на эту букву.

Мы будем использовать данные из этого списка свойств для заполнения табличного представления, создавая разделы для каждой буквы.

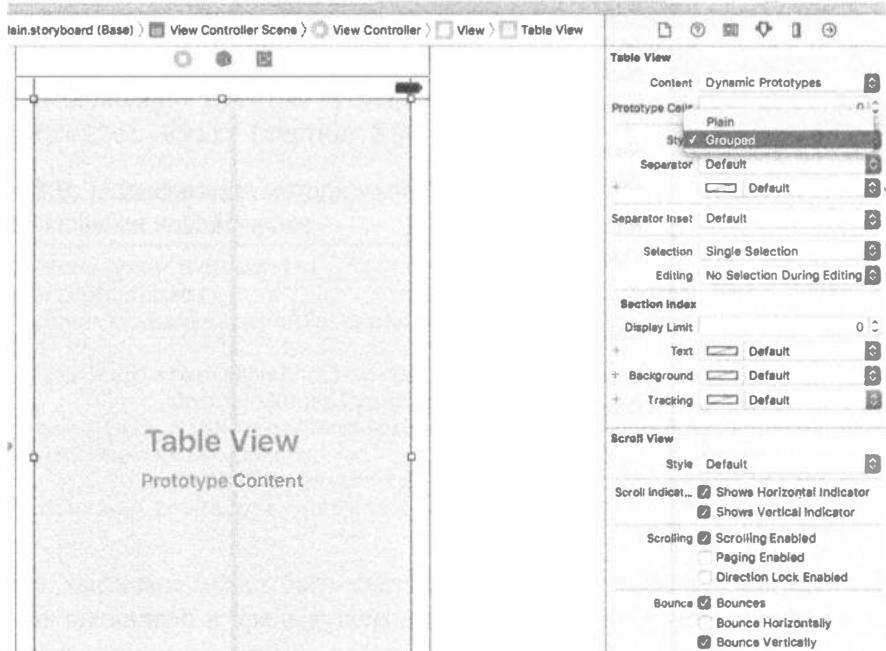


Рис. 8.31. Инспектор атрибутов для табличного представления с выбранным значением *Grouped* выпадающего меню *Style*

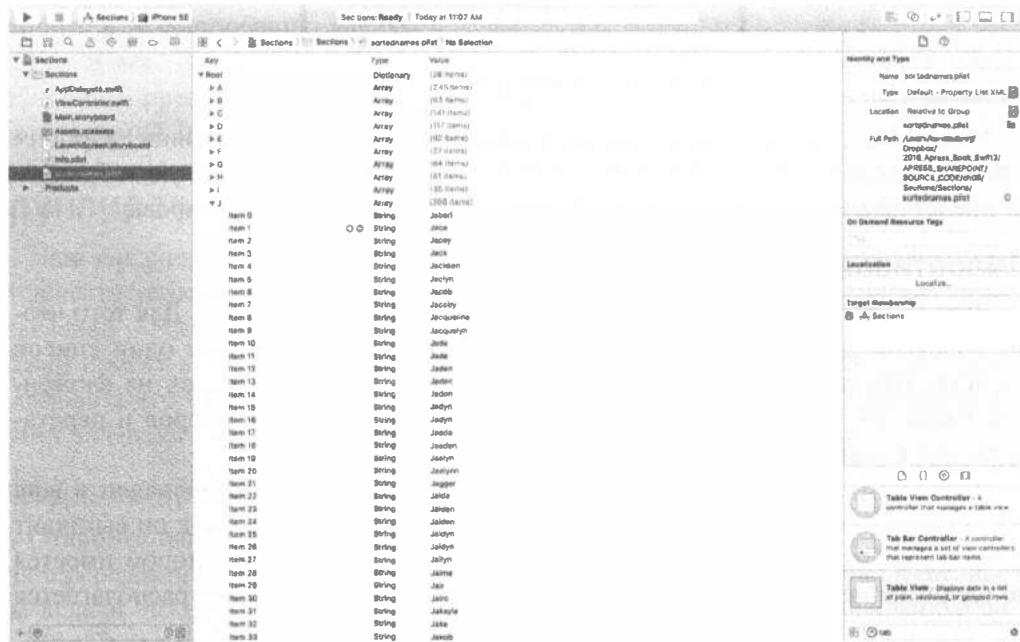


Рис. 8.32. Список свойств sortednames.plist. Здесь раскрыт список имен на букву "J"

Реализация контроллера

Щелкните на файле `ViewController.swift`. Сделайте класс отвечающим протоколу `UITableViewDataSource`, добавьте имя идентификатора ячейки таблицы и создайте пару свойств путем добавления в файл `ViewController.swift` кода, выделенного ниже полужирным шрифтом.

```
class ViewController: UIViewController, UITableViewDataSource {
    let sectionsTableIdentifier = "SectionsTableIdentifier"
    var names: [String: [String]]!
    var keys: [String]!
```

Снова откройте файл `Main.storyboard` и откройте окно помощника редактора. Если файл в окне не открывается, воспользуйтесь панелью быстрых переходов для выбора файла `ViewController.swift`. Нажмите клавишу `<Control>` и перетащите указатель от табличного представления в окно помощника редактора для создания выхода для таблицы непосредственно под определением ключевых свойств.

```
@IBOutlet weak var tableView: UITableView!
```

Теперь измените метод `viewDidLoad()` так, как показано в листинге 8.10.

Листинг 8.10. Модификация метода `viewDidLoad()`

для использования новой ячейки

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка представления после загрузки,
    // обычно из пив-файла
    tableView.register(UITableViewCell.self,
                       forCellReuseIdentifier: sectionsTableIdentifier)

    let path = Bundle.main.path(forResource:
        "sortednames", ofType: "plist")
    let namesDict = NSDictionary(contentsOfFile: path!)
    names = namesDict as! [String: [String]]
    keys = (namesDict!.allKeys as! [String]).sorted()
}
```

Большая часть этого кода не слишком отличается от того, что вы видели раньше. Прежде мы добавляли объявления свойств и для словаря, и для массива. Словарь будет хранить все наши данные, в то время как массив будет хранить разделы, отсортированные в алфавитном порядке. В методе `viewDidLoad()` мы сначала зарегистрировали класс ячейки табличного представления по умолчанию, который должен выводиться для каждой строки, с помощью нашего объявленного идентификатора. После этого мы создали экземпляр класса `NSDictionary` из добавленного в проект списка свойств и присвоили его свойству `names`, приведя его к подходящему типу словаря `Swift`. После этого мы получили все ключи из этого словаря и отсортировали их, чтобы получить

упорядоченный массив со значениями ключей из словаря в алфавитном порядке. Помните, что наши данные используют в качестве ключей буквы алфавита, так что этот массив будет содержать 26 букв в порядке от A до Z и мы будем использовать этот массив для отслеживания разделов.

Далее добавьте в конец файла код, приведенный в листинге 8.11.

Листинг 8.11. Методы источника данных для табличного представления

```
// MARK: Table View Data Source Methods
func numberOfSections(in tableView: UITableView) -> Int {
    return keys.count
}

func tableView(_ tableView: UITableView,
              numberOfRowsInSection section: Int) -> Int {
    let key = keys[section]
    let nameSection = names[key]!
    return nameSection.count
}

func tableView(_ tableView: UITableView,
              titleForHeaderInSection section: Int) -> String? {
    return keys[section]
}

func tableView(_ tableView: UITableView,
              cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: sectionsTableIdentifier, for: indexPath)
        as UITableViewCell
    let key = keys[indexPath.section]
    let nameSection = names[key]!
    cell.textLabel?.text = nameSection[indexPath.row]

    return cell
}
```

Все это методы источника данных таблицы. Первый добавленный в наш класс метод указывает количество разделов. Мы не реализовали этот метод в предыдущих примерах, потому что нас вполне устраивало значение по умолчанию, равное единице. Но на этот раз мы говорим табличному представлению, что у нас имеется по одному разделу для каждого ключа нашего словаря.

```
func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    return keys.count
}
```

Следующий метод вычисляет количество строк в конкретном разделе. В предыдущем примере у нас был только один раздел, так что мы просто возвращали количество строк в массиве. На этот раз нам необходимо разбить его по разделам. Мы

можем сделать это путем получения массива, соответствующего интересующему нас разделу, и возвращения количества элементов в этом массиве.

```
func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    let key = keys[section]
    let nameSection = names[key]!
    return nameSection.count
}
```

Метод `tableView(_:titleForHeaderInSection:)` позволяет указать необязательное значение заголовка для каждого раздела, и в нем мы просто возвращаем букву соответствующей группы.

```
func tableView(tableView: UITableView,
    titleForHeaderInSection section: Int) -> String? {
    return keys[section]
}
```

В методе `tableView(_:cellForRowIndexPath:)` мы должны извлечь из пути индекса раздел и строку, а затем использовать их для определения интересующего нас значения. Номер раздела говорит нам о том, какой массив следует получить из словаря имен, а затем на основании значения строки получить необходимое значение из указанного массива. Все остальное в этом методе в основном осталось тем же, что и в приложении простой таблицы, которое мы создали ранее в этой главе.

Скомпилируйте и запустите проект, и можете гордиться собственной крутизной. Помните, что мы изменили значение `Style` таблицы на `Grouped` и в результате получили сгруппированную таблицу с 26 разделами, которая должна выглядеть так, как показано на рис. 8.33.

Для контраста вернем нашему табличному представлению простой стиль и посмотрим, на что похожа простая таблица с несколькими разделами. Выберите файл `Main.storyboard`, чтобы отредактировать его в программе `Interface Builder`. Выберите табличное представление и воспользуйтесь инспектором атрибутов, чтобы изменить стиль табличного представления на `Plain`. Сохраните проект, а затем соберите и запустите его. Как видите, данные — те же, но внешний вид совсем другой (рис. 8.34).



Рис. 8.33. Сгруппированная таблица с несколькими разделами

Добавление индекса

Одной из проблем, связанных с нашей текущей таблицей, является само количество строк. В списке 2000 имен. Пока вы доберетесь до Zachariah или Zayne, не говоря уж о Zoie, ваш палец ужасно устанет.

Одним из решений этой проблемы является добавление индекса в правой части таблицы. Теперь, когда мы вновь установили стиль нашего табличного представления в Plain, это относительно легко сделать, как показано на рис. 8.35. Добавьте следующий метод в конец файла ViewController.swift:

```
func sectionIndexTitles(for tableView: UITableView) -> [String]? {
    return keys
}
```

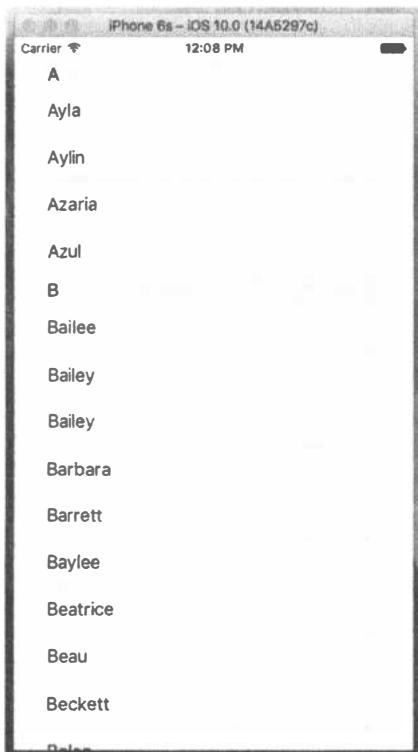


Рис. 8.34. Простая таблица с разделами и без индексов

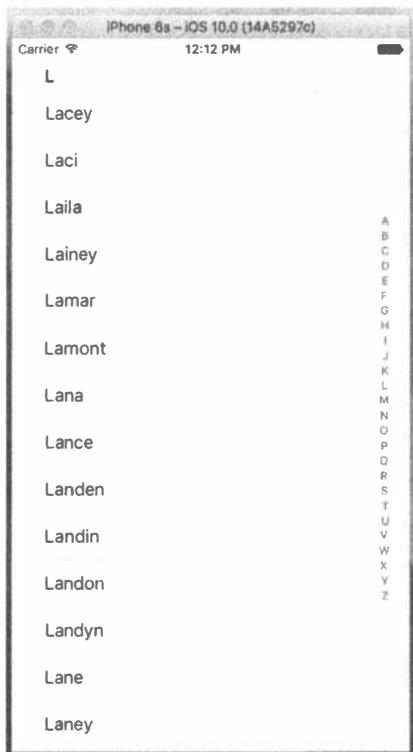


Рис. 8.35. Добавление индекса в табличное представление

Реализация строки поиска

Индекс полезен, но несмотря на это у нас все еще слишком много имен. Если, например, мы хотим узнать, имеется ли в списке имя Arabella, то должны будем выполнить некоторую прокрутку даже при применении индекса. Было бы неплохо, если бы мы могли дать пользователю возможность уменьшить список,

введя поисковый запрос. Это требует небольшой дополнительной работы, но результат того стоит. Мы собираемся реализовать стандартную строку поиска iOS, показанную на рис. 8.36, слева, с использованием контроллера поиска.



Рис. 8.36. Приложение с добавленной строкой поиска

По мере ввода пользователем информации в строку поиска список имен сокращается, и в нем остаются только те из них, которые содержат введенный текст в качестве подстроки. В качестве бонуса строка поиска также позволяет вам определить кнопки области видимости, которые вы можете использовать для уточнения поиска тем или иным образом. Мы добавим три такие кнопки в нашу строку поиска: `Short`, которая позволит ограничивать поиск именами менее шести символов длиной; `Long` — для имен длиной не менее шести символов; `All` — для поиска по всем именам. Эти кнопки появляются только тогда, когда пользователь вводит что-то в строку поиска (см. рис. 8.36, справа).

Добавление функции поиска выполняется достаточно просто. Для этого нужны три вещи.

- ⌘ Данные, среди которых выполняется поиск. В нашем случае это список имен.
- ⌘ Контроллер представления для вывода результатов. Временно заменяет контроллер, предоставляющий данные. Результаты для отображения могут

выбираться любым способом, но обычно исходные данные представлены в таблице, и контроллер использует другую таблицу, которая выглядит очень похоже на основную, создавая тем самым впечатление, будто поиск просто фильтрует исходную таблицу. Однако, как вы увидите, это не совсем то, что происходит на самом деле.

- Класс `UISearchController`, предоставляющий строку поиска и управляющий выводом результатов поиска в контроллере результирующего представления.

Начнем с создания скелета нашего контроллера представления. Мы будем отображать результаты поиска в таблице, так что наш контроллер представления результатов должен содержать таблицу. Мы можем перетащить контроллер представления на раскладовку и добавить к нему табличное представление, как мы делали в более ранних примерах в этой главе, но в этот раз мы пойдем другим путем. Мы будем использовать класс `UITableViewController`, который представляет собой контроллер представления со встроенным объектом класса `UITableView`, настроенным и как источник данных, и как делегат для своего табличного представления. В окне навигатора проекта щелкните правой кнопкой мыши на группе `Sections` и выполните команду `New File...` из всплывающего меню. В области выбора шаблона выберите пиктограмму `Cocoa Touch Class` из группы `iOS Source` и щелкните на кнопке `Next`. Назовите ваш новый класс `SearchResultsController` и сделайте его подклассом `UITableViewController`. Щелкните на кнопке `Next`, выберите местоположение для нового файла и позвольте программе Xcode создать его.

Выберите в окне навигатора проекта файл `SearchResultsController.swift` и внесите в него следующие изменения:

```
class SearchResultsController: UITableViewController,
    UISearchResultsUpdating {
```

Мы собираемся реализовать логику поиска в этом контроллере представления, поэтому делаем его соответствующим протоколу `UISearchResultsUpdating`, который позволяет нам назначить его в качестве делегата класса `UISearchController`. Как вы увидите позже, единственный метод, определенный этим протоколом, вызывается для обновления результатов поиска по мере ввода пользователем данных в строку поиска.

Поскольку этот контроллер будет реализовывать операции поиска, классу `SearchResultsController` необходим доступ к списку имен, которые отображает основной контроллер, так что мы должны дать ему свойства, которые сможем использовать для передачи словаря имен и списка ключей, используемых для вывода в основном контроллере представления. Добавим эти свойства в файл `SearchResultsController.swift`. Вы, наверное, заметили, что этот файл уже содержит некоторый незаконченный код, который обеспечивает

частичную реализацию протокола `UITableViewDataSource`, и некоторые блоки закомментированного кода других методов, которые подклассам класса `UITableViewController` часто необходимо реализовать. В данном примере мы не собираемся использовать большинство из них, так что не стесняйтесь удалить весь закомментированный код, а затем добавить следующий код в верхней части файла:

```
class SearchResultsController: UITableViewController,
UISearchResultsUpdating {
    let sectionsTableIdentifier = "SectionsTableIdentifier"
    var names: [String: [String]] = [String: [String]]()
    var keys: [String] = []
    var filteredNames: [String] = []
```

Мы добавили переменную `sectionsTableIdentifier` для хранения идентификатора ячеек таблицы в этом контроллере представления и используем тот же идентификатор, что и в контроллере главного представления, хотя мы могли бы использовать любое имя. Мы также добавили два свойства, которые будут хранить имена словаря и списка ключей, которые мы будем использовать при поиске, и еще одно для ссылки на массив, в котором будут храниться результаты поиска.

Далее добавим в метод `viewDidLoad()` строку кода для регистрации идентификатора ячейки таблицы с результатами встроенного в контроллер табличного представления:

```
override func viewDidLoad() {
    super.viewDidLoad()
    tableView.register(UITableViewCell.self,
                      forCellReuseIdentifier: sectionsTableIdentifier)
}
```

Это все, что нам пока что нужно было сделать в контроллере представления результатов, так что давайте ненадолго вернемся к нашему контроллеру главного представления и добавим в него строку поиска. Выберите `ViewController.swift` в окне навигатора проекта и добавьте в верхней части файла свойство для хранения ссылки на экземпляр `UISearchController`, который будет делать за нас большую часть тяжелой работы.

```
class ViewController: UIViewController, UITableViewDataSource {
    let sectionsTableIdentifier = "SectionsTableIdentifier"
    var names: [String: [String]]!
    var keys: [String]!
    @IBOutlet weak var tableView: UITableView!
    // Добавьте следующую строку
    var searchController: UISearchController!
```

Теперь добавим код, который создает контроллер поиска, в метод `viewDidLoad()`, как показано в листинге 8.12.

Листинг 8.12. Добавление контроллера поиска в метод viewDidLoad в файле ViewController.swift

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.
    tableView.register(UITableViewCell.self,
        forCellReuseIdentifier: sectionsTableIdentifier)
    let path = Bundle.main.pathForResource(
        "sortednames", ofType: "plist")
    let namesDict = NSDictionary(contentsOfFile: path!)
    names = namesDict as! [String: [String]]
    keys = (namesDict!.allKeys as! [String]).sorted()

    let resultsController = SearchResultsController()
    resultsController.names = names
    resultsController.keys = keys
    searchController =
        UISearchController(searchResultsController: resultsController)

    let searchBar = searchController.searchBar
    searchBar.scopeButtonTitles = ["All", "Short", "Long"]
    searchBar.placeholder = "Enter a search term"
    searchBar.sizeToFit()
    tableView.tableHeaderView = searchBar
    searchController.searchResultsUpdater = resultsController
}
```

Начнем с создания контроллера представления результатов и установим его свойства names и keys. Затем создадим объект класса UISearchController и передадим ему ссылку на контроллер результатов — именно этот объект будет играть роль контроллера представления результатов во время демонстрации.

```
let resultsController = SearchResultsController()
resultsController.names = names
resultsController.keys = keys
searchController =
    UISearchController(searchResultsController: resultsController)
```

Следующие три строки кода получают и настраивают объект класса UISearchBar, который создается контроллером класса UISearchController и который мы получаем из его свойства searchBar:

```
let searchBar = searchController.searchBar
searchBar.scopeButtonTitles = ["All", "Short", "Long"]
searchBar.placeholder = "Enter a search term"
```

Свойство строки поиска scopeButtonTitles содержит имена, назначаемые кнопкам области видимости. По умолчанию этих кнопок нет, но здесь мы устанавливаем имена трех кнопок, о которых говорили ранее в этом разделе. Мы также устанавливаем некоторый замещающий текст, чтобы дать пользователю понять, для чего эта строка поиска. Вы можете увидеть этот текст на рис. 8.36, слева.

Пока что мы создали класс `UISearchController`, но не подключили его к своему пользовательскому интерфейсу. Для того чтобы это сделать, мы получаем строку поиска и устанавливаем ее как представление заголовка таблицы в нашем контроллере основного представления.

```
searchBar.sizeToFit()
tableView.tableHeaderView = searchBar
```

Представление заголовка таблицы автоматически управляется табличным представлением. Заголовок всегда располагается перед первой строкой первого раздела таблицы. Обратите внимание: чтобы задать размер полосы поиска, подходящий для его содержимого, мы используем метод `sizeToFit()`. В результате получается верная высота заголовка — ширина, задаваемая этим методом, не важна, поскольку табличное представление гарантирует, что оно растягивается на всю ширину таблицы и будет автоматически изменять размер при изменении размера таблицы (обычно из-за вращения устройства).

Последнее изменение в методе `viewDidLoad()` присваивает значение свойству `searchResultsUpdater` контроллера класса `UISearchController`, которое имеет тип `UISearchResultsUpdating`:

```
searchController.searchResultsUpdater = resultsController
```

Каждый раз, когда пользователь вводит что-то в строке поиска, объект класса `UISearchController` для обновления результатов поиска использует объект, хранящийся в свойстве `searchResultsUpdater`. Как уже упоминалось, поиск будет выполняться в классе `SearchResultsController`, а потому нам необходимо, чтобы он соответствовал протоколу `UISearchResultsUpdating`.

Это все, что нам нужно сделать в нашем главном контроллере для добавления строки поиска и вывода результатов поиска. Далее мы должны вернуться к файлу `SearchResultsController.swift` и завершить в нем выполнение двух задач — добавить код, реализующий поиск, и методы класса `UITableViewDataSource` для встраиваемого табличного представления.

Начнем с кода для поиска. По мере ввода пользователя в строке поиска объект класса `UISearchController` вызывает метод `updateSearchResultsForSearchController()` для обновления результатов с помощью объекта `SearchResultsController`. В этом методе нам нужно получить текст поиска из строки поиска и использовать его для создания отфильтрованного списка имен в массиве `filteredNames`. Мы также будем использовать кнопки области видимости для ограничения имен, включаемых в поиск. Добавьте следующие определения констант в верхней части файла `SearchResultsController.swift`:

```
class SearchResultsController: UITableViewController,
UISearchResultsUpdating {
    private static let longNameSize = 6
    private static let shortNamesButtonIndex = 1
    private static let longNamesButtonIndex = 2
```

Добавьте в конец файла код, приведенный в листинге 8.13.

Листинг 8.13. Код для вывода результатов поиска

```
// MARK: UISearchResultsUpdating Conformance
func updateSearchResults(for searchController: UISearchController) {
    if let searchString = searchController.searchBar.text {
        let buttonIndex = searchController.searchBar.selectedScopeButtonIndex
        filteredNames.removeAll(keepingCapacity: true)

        if !searchString.isEmpty {
            let filter: (String) -> Bool = { name in
                // Фильтрация длинных или коротких имен в зависимости от кнопки
                let nameLength = name.characters.count
                if (buttonIndex == SearchResultsController.shortNamesButtonIndex
                    && nameLength >= SearchResultsController.longNameSize)
                    || (buttonIndex == SearchResultsController.longNamesButtonIndex
                        && nameLength < SearchResultsController.longNameSize) {
                    return false
                }

                let range = name.range(of: searchString, options: NSString.CompareOptions.caseInsensitive, range: nil, locale: nil)
                // let range = name.rangeOfString(searchString,
                // options: NSString.CompareOptions.CaseInsensitiveSearch)
                return range != nil
            }

            for key in keys {
                let namesForKey = names[key]!
                let matches = namesForKey.filter(filter)
                filteredNames += matches
            }
        }
    }
    tableView.reloadData()
}
```

Пройдемся по этому коду и посмотрим, что он делает. Сначала мы получаем строку поиска из панели поиска и индекс выбранной кнопки области видимости, а затем очищаем список отфильтрованных имен:

```
if let searchString = searchController.searchBar.text {
    let buttonIndex = searchController.searchBar.selectedScopeButtonIndex
    filteredNames.removeAll(keepingCapacity: true)
```

Затем мы проверяем, что строка поиска не пустая, — мы не выводим никакие результаты для пустой строки поиска:

```
if !searchString.isEmpty {
```

Далее определяем замыкание для имен, соответствующих строке поиска. Замыкание будет вызываться для каждого имени в каталоге имен, давать имя (в виде строки) и возвращать значение `true`, если значение имени соответствует строке

поиска, и `false`, если нет. Но сначала проверяем, соответствует ли длина имени выбранной кнопке области видимости, и возвращаем `false`, если это не так:

```
let filter: (String) -> Bool = { name in
    // Фильтрация длинных или коротких имен в зависимости от кнопки
    let nameLength = name.characters.count
    if (buttonIndex == SearchResultsController.shortNamesButtonIndex
        && nameLength >= SearchResultsController.longNameSize)
        || (buttonIndex == SearchResultsController.longNamesButtonIndex
            && nameLength < SearchResultsController.longNameSize) {
        return false
    }
}
```

Если имя проходит эту проверку, мы ищем строку поиска как подстроку в имени. Если находим ее, у нас имеется соответствие:

```
let range = name.range(of: searchString, options:
    NSString.CompareOptions.caseInsensitive,
    range: nil, locale: nil)
    return range != nil
}
```

Далее перебираем все ключи в словаре имен, каждый из которых соответствует массиву имен (ключ A отображается на имена, которые начинаются с буквы A, и т.д.). Для каждого ключа мы получаем его массив имен и фильтруем его с помощью нашего замыкания. Это дает нам (возможно, пустой) отфильтрованный массив имен, которые соответствуют строке поиска и который мы добавляем в массив `filteredNames`:

```
for key in keys {
    let namesForKey = names[key]!
    let matches = namesForKey.filter(filter)
    filteredNames += matches
}
```

В этом коде объект `namesForKey` имеет тип `[String]` и содержит имена, которые соответствуют обрабатываемому значению ключа. Мы используем метод `filter()` типа `Array` для применения нашего замыкания к каждому из элементов `namesToKey`. В результате получается другой массив, содержащий только те элементы, которые соответствуют фильтру, т.е. только те имена, которые соответствуют тексту поиска и кнопке выбранной области видимости. Затем мы добавляем эти имена в массив `filteredNames`.

После того как будут обработаны все массивы имен, у нас в массиве `filteredNames` будет полный набор имен, соответствующих строке поиска. Теперь все, что нам нужно сделать, — это организовать их для отображения в таблице в классе `SearchResultsController`. Начнем с того, что укажем таблице на необходимость отобразить ее содержимое:

```
}
tableView.reloadData()
}
```

Нам нужно, чтобы табличное представление отображало по одному имени из массива `filteredNames` в каждой строке. Для этого мы реализуем методы протокола `UITableViewDataSource` в классе `SearchResultsController`. Напомним, что класс `SearchResultsController` является подклассом `UITableViewController`, поэтому он автоматически действует в качестве источника данных его таблицы. Добавьте в файл `SearchResultsController.swift` код, приведенный в листинге 8.14.

Листинг 8.14. Методы источника данных нашего табличного представления

```
// MARK: Table View Data Source Methods
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return filteredNames.count
}

override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
sectionsTableIdentifier)
    cell.textLabel?.text = filteredNames[indexPath.row]
    return cell!
}
```

Теперь можете запустить приложение и попробовать отфильтровать список имен, как показано на рис. 8.37.

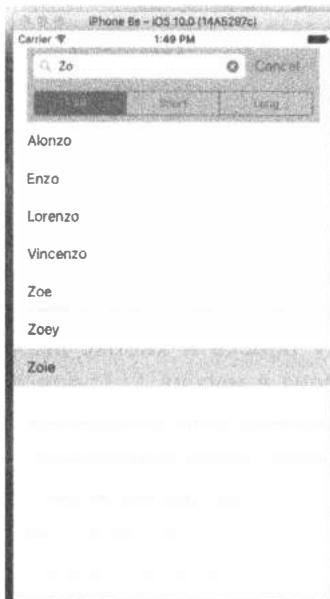


Рис. 8.37. Приложение с добавленной к таблице строкой поиска

Отладка представления

Класс UISearchController так хорошо выполняет работу по переключению между двумя таблицами в нашем последнем примере, что вы могли бы просто не поверить, что такое переключение выполняется вообще! Помимо того, что вы видели весь код, есть также несколько визуальных подсказок, свидетельствующих о переключении, — таблица поиска является простой однородной таблицей, поэтому имена в ней не сгруппированы так же, как в главной таблице, и у нее нет раздела индекса. Если вы хотите еще больше доказательств, то можете получить их с помощью новой возможности Xcode, которая называется отладкой представления (View Debugging) и позволяет делать снимки иерархии представлений запущенного приложения и изучать их в области редактора Xcode. Эта возможность работает как на симуляторе, так и в реальных устройствах, и вы, вероятно, полностью сможете оценить ее, когда попытаетесь выяснить, почему одно из ваших представлений кажется отсутствующим или появляется не там, где вы ожидаете его видеть.

Начнем с того, что посмотрим, что сделает функция View Debugging из нашего приложения, когда оно показывает полный список имен. Запустите приложение и в строке меню Xcode выполните команду **Debug**⇒**View Debugging**⇒**Capture View Hierarchy**. Программа Xcode получает иерархию представлений из симулятора или устройства и выводит ее так, как показано на рис. 8.38.



Рис. 8.38. Иерархия представлений приложения Sections

Вероятно, это выглядит не очень полезно — мы не видим ничего большего, чем могли бы видеть в симуляторе. Для того чтобы выявить иерархию представлений, необходимо повернуть изображение приложения так, чтобы вы могли посмотреть на него “со стороны”. Для этого нажмите кнопку мыши в области редактора, где-то слева от изображения, и перетащите указатель вправо. Когда вы это сделаете, наслойение представлений в приложении станет видимым. Если вы повернете представления на угол около 45° , то увидите что-то наподобие рис. 8.39.

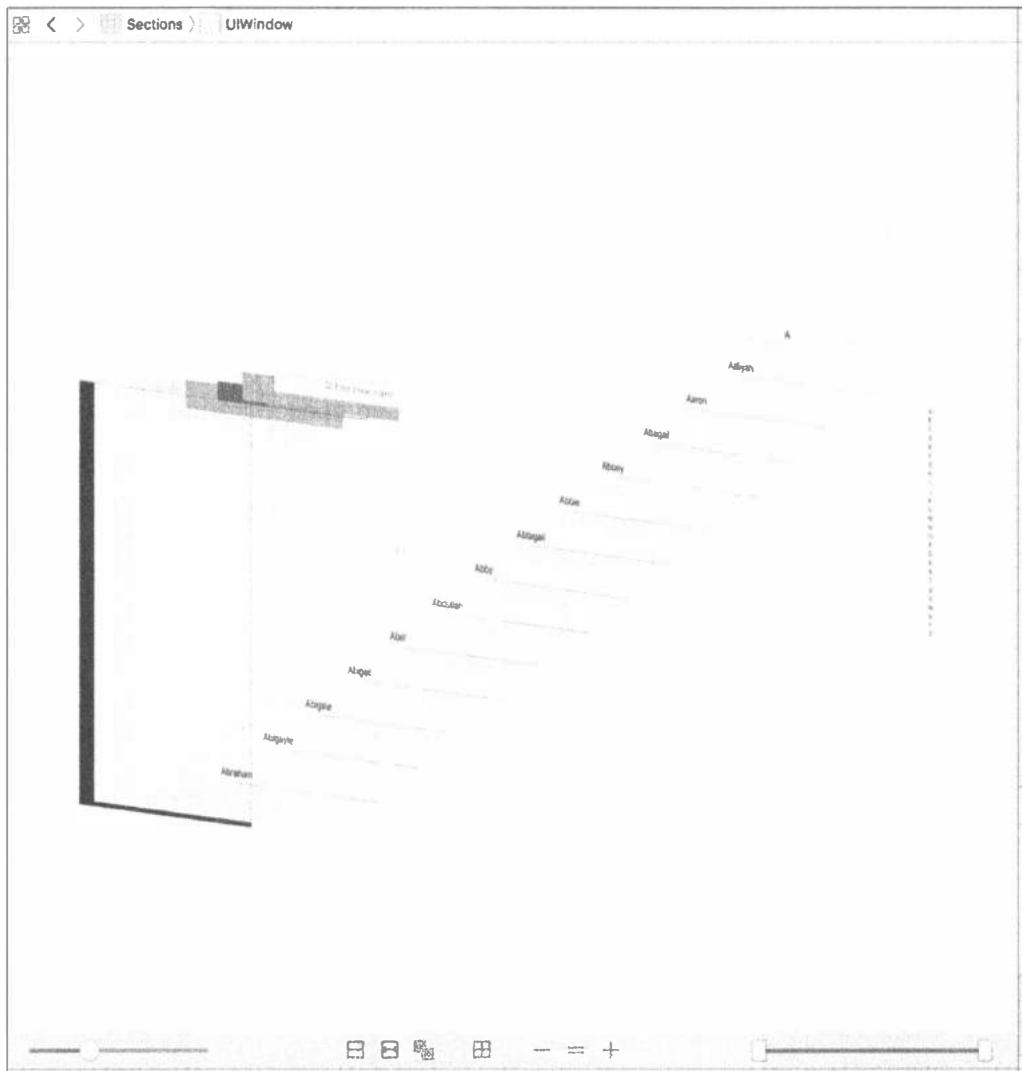


Рис. 8.39. Изучение иерархии представлений приложения

Щелкнув на различных представлениях в стеке, вы увидите, что панель переходов в верхней части изменится и покажет имя класса представления, на

котором был сделан щелчок, и имена классов всех родительских представлений. Щелкните на каждом из представлений с последнего до первого, чтобы ознакомиться с тем, как строится таблица. Вы сможете найти главное представление контроллера представления, табличные представления, некоторые представления ячеек, панель поиска, панель индекса и различные представления, которые являются частью реализации таблицы.

Теперь посмотрим, как выглядит иерархия представлений во время поиска. Программа Xcode приостанавливает работу приложения, позволяя изучить снимок представлений, поэтому сначала возобновим выполнение, выбрав команду *Debug*⇒*Continue*. Начните ввод в строке поиска и вновь выполните команду *Debug*⇒*View Debugging*⇒*Capture View Hierarchy*. Когда появится иерархия представлений, поверните ее немного, и вы увидите что-то наподобие рис. 8.40.

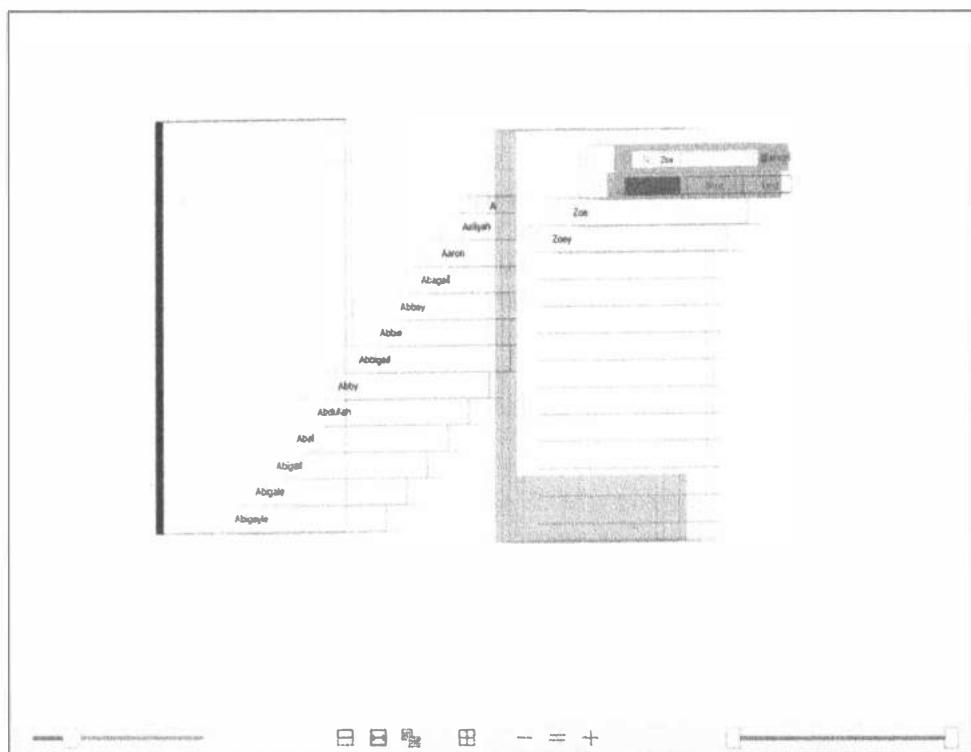


Рис. 8.40. Иерархия представлений при использовании строки поиска “Zoe”

Сейчас совершенно очевидно, что действительно используются две таблицы. Вы можете увидеть исходную таблицу примерно на полпути через стек представлений, а выше (т.е. справа) — табличное представление, которое принадлежит контроллеру представления результатов поиска. Сразу за ним есть полупрозрачное серое представление, покрывающее исходную таблицу, — это представление затемняет исходную таблицу в начале ввода в строке поиска.

Немного поэкспериментируйте с кнопками в нижней части области редактора — их можно использовать для включения и отключения отображения ограничений механизма Auto Layout, сброса представления вниз иерархии, а также увеличения и уменьшения масштаба. Можно также изменить расстояние между представлениями с помощью ползунка в левой части и использовать ползунок справа для удаления слоев в верхней или нижней части иерархии, так что вы сможете увидеть, что находится за ними. View Debugging — очень мощный инструмент.

Резюме

Это была солидная глава, и вы изучили массу новинок. Теперь вы должны очень ясно понимать, как работают таблицы, знать, как настроить таблицы и ячейки табличных представлений, и конфигурировать табличные представления. Вы также узнали, как реализовать панель поиска, которая является жизненно важным инструментом в любых iOS-приложениях, которые представляют пользователям большие объемы данных. Наконец вы познакомились с View Debugging — новым исключительно полезным инструментом среды Xcode. Убедитесь, что хорошо понимаете все, что мы делали в этой главе, потому что дальнейший материал будет опираться на эту главу.

Мы продолжим работу с табличными представлениями в следующей главе. Например, вы узнаете, как использовать их для представления иерархических данных; увидите, как создать представления, которые позволяют пользователю редактировать данные, выбранные в табличном представлении, а также как представлять в таблицах перечни, вставлять элементы управления в строки таблиц строк и удалять строки.

ГЛАВА 9



Контроллеры навигации и табличные представления

В предыдущей главе мы сосредоточились на основах работы с табличными представлениями, а в этой мы сделаем акцент на изучении **контроллеров навигации**.

Контроллеры навигации и табличные представления дополняют друг друга, но, строго говоря, контроллер навигации для выполнения своих функций может обойтись без табличного представления. Однако на практике при реализации контроллера навигации почти всегда используется хотя бы одна таблица, поскольку мощь контроллера навигации как раз и состоит в простоте, с которой он обрабатывает сложные иерархические данные. На небольшом экране устройства iPhone иерархические данные лучше всего представляются именно с помощью последовательности табличных представлений.

В этой главе мы шаг за шагом создадим новое приложение, подобно тому, как сделали приложение Pickers в главе 7. Мы создадим корневой контроллер навигации и, как только он заработает, будем добавлять другие контроллеры и наращивать уровни иерархии. Каждый создаваемый нами контроллер навигации будет усиливать тот или иной аспект использования таблицы или конфигурации, в результате чего вы получите ответы на следующие вопросы:

- как перейти от табличных представлений к дочерним табличным представлениям;
- как перейти от табличных представлений к представлениям содержания, которые позволяют не только отобразить детализированные данные, но и отредактировать их;
- как использовать несколько разделов в табличном представлении;
- как использовать режим редактирования, чтобы из табличного представления можно было удалять строки;
- как использовать режим редактирования, чтобы пользователь мог переупорядчивать файлы в табличном представлении.

Основы контроллеров навигации

Основным инструментом, который мы будем использовать для написания иерархических приложений, является класс `UINavigationController`, который похож на класс `UITabBarController` в том, как он управляет представлениями содержимого, загружает и выгружает их. Основное различие между этими классами состоит в том, что класс `UINavigationController` реализован как стек, что делает его весьма подходящим для работы с иерархиями.

Если у вас есть знания в этой области, вы можете просто перелистать эту главу. Если же термин *стек* вам незнаком, продолжайте читать. К счастью, понять, что такое стек, не так уж трудно.

Стеки

Стек (*stack*) — это часто используемая структура данных, которая работает по принципу “последним вошел — первым вышел”. Прекрасный пример стека — механическая игрушка-дозатор Pez (рис. 9.1). Вам когда-либо доводилось заправлять его конфетами? В соответствии с небольшой инструкцией, которая прилагается к каждой такой игрушке, вам необходимо выполнить несколько простых действий. Во-первых, разверните упаковку конфет. Во-вторых, откройте дозатор, опрокинув его верхушку назад. В-третьих, возьмите конфеты, крепко удерживая их между указательным и большим пальцами, и вставьте в открытый дозатор.



Рис. 9.1. Механическая игрушка-дозатор Pez — простой пример стека

Помните, мы сказали о стеке “последним вошел — первым вышел”? Вот именно так у нас и получается с этим дозатором. Первую конфету, которую вы опустите в дозатор, вы сможете достать из него только в последнюю очередь. В то же время последняя конфета, которую вы затолкнули в дозатор, останется верхней, а значит, ее вы достанете первой. Компьютерный стек работает по тем же правилам.

- ✿ Добавляя объект в стек, мы говорим, что **заталкиваем** (*push*) его в стек.
- ✿ Первый объект, который вы поместили в стек, называется **основанием стека** (*base*).

- Последний объект, который вы поместили в стек, называется **вершиной стека** (**top**) (он остается таковым до тех пор, пока вы не замените его следующим объектом, который поместите в стек).
- Удаляя объект из стека, мы говорим, что **выталкиваем** (**pop**) его из стека. Удаляемым из стека объектом всегда оказывается объект, который был помещен туда последним. И наоборот, первый объект, который был помещен в стек, всегда извлекается из него последним.

Стек контроллеров

Контроллер навигации обслуживает стек контроллеров представления. При разработке контроллера навигации вам необходимо указать первое представление, которое увидит пользователь. Как упоминалось в предыдущих главах, такое представление называется **контроллером корневого представления** (**root view controller**) или просто **корневым контроллером** (**root controller**) и служит основанием стека контроллеров представления в контроллере навигации. При выборе пользователем следующего представления для отображения новый контроллер представления помещается в стек, в результате чего и на экране появляется соответствующее представление. Мы называем эти новые контроллеры представлений **подконтроллерами** (**subcontroller**). Как будет показано ниже, приложение Fonts, которое мы создадим в этой главе, будет состоять из контроллера навигации и нескольких подконтроллеров.

Посмотрите на рис. 9.2. Обратите внимание на **заголовок** посередине навигационной панели и на **кнопку возврата** в левой части этой панели. Заголовок навигационной панели заполнен свойством **Title** контроллера представления, находящегося на вершине стека, а название кнопки навигации заполнено свойством **Title** предыдущего контроллера представления. Эта кнопка похожа на кнопку возврата в окне веб-браузера. Если пользователь нажмет эту кнопку, контроллер текущего представления будет извлечен из стека и текущим станет предыдущее представление.

Нам нравится этот шаблон проектирования, так как он позволяет итеративно создавать сложные иерархические приложения. Для того чтобы система постоянно находилась в состоянии готовности к работе, нам необязательно знать всю иерархию. Каждый контроллер должен знать только о существовании своих дочерних контроллеров, чтобы, когда пользователь делает выбор, контроллер мог поместить в стек соответствующий объект нового контроллера. Таким способом можно написать большое приложение из нескольких маленьких составляющих — именно этим мы и займемся в данной главе.

Контроллер навигации можно назвать ядром многих iPhone-приложений, но в приложениях для устройства iPad контроллер навигации играет менее значительную роль. В качестве типичного примера можно привести iPhone-приложение Mail, содержащее иерархический контроллер навигации, который позволяет пользователю легко перемещаться между почтовыми серверами, папками и

сообщениями. В iPad-версии приложения Mail контроллер навигации никогда не заполняет экран, но отображается в виде либо врезки, либо временного представления, закрывающего часть основного окна. Мы разберемся с этим в главе 11.

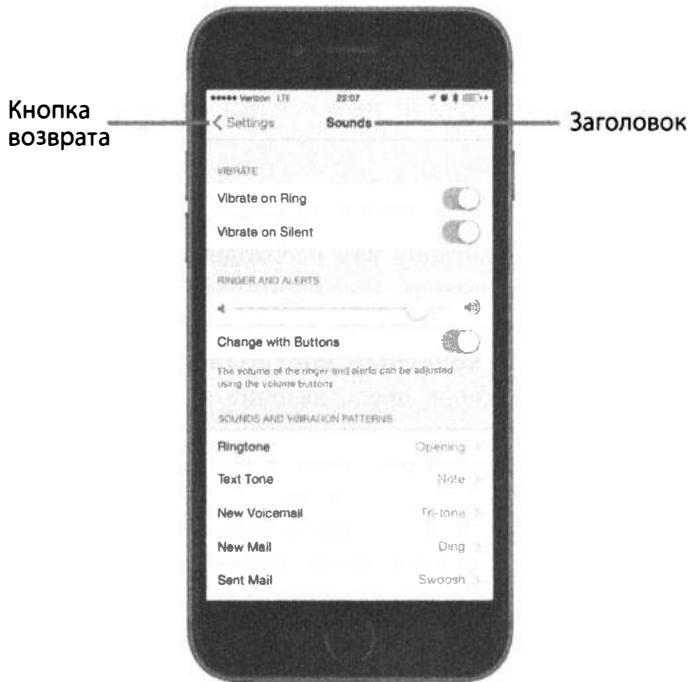


Рис. 9.2. Приложение *Settings* использует контроллер навигации. В левом верхнем углу находится кнопка возврата, которая используется для извлечения контроллера текущего представления из стека. Нажатие этой кнопки возвращает вас на предыдущий уровень иерархии. Справа отображается заголовок контроллера текущего представления содержимого

Font – простой браузер шрифтов

На примере приложения, к написанию которого мы приступаем, будет показано, как можно справиться с большинством распространенных задач, связанных с отображением иерархии данных. При запуске этого приложения вы увидите список всех **семейств шрифтов**, имеющихся в iOS, как показано на рис. 9.3. Семейство шрифтов представляет собой группу тесно связанных шрифтов, или шрифтов, являющихся стилистическими вариациями друг друга. Например, шрифты Helvetica, Helvetica-Bold, Helvetica-Oblique и прочие вариации входят в семейство шрифтов Helvetica.



Рис. 9.3. В нашем проекте контроллер корневого представления приложения выводит на экран вспомогательные пиктограммы в правой части каждой строки представления. Эти пиктограммы называются индикаторами раскрытия. Если коснуться строки с таким индикатором, то отобразится табличное представление более низкого уровня

Выбор любой строки в этом представлении верхнего уровня заталкивает контроллер представления в стек контроллеров навигации. Пиктограммы в правой части каждой строки называются **вспомогательными** (accessory icons). В частности, такая вспомогательная пиктограмма (в виде серой стрелки) называется **индикатором раскрытия** (disclosure indicator), поскольку уведомляет пользователя о том, что при выборе этой строки раскроется еще одно табличное представление.

Подконтроллеры приложения Fonts

Прежде чем приступить к написанию приложения Fonts, необходимо вкратце рассмотреть все его подконтроллеры.

Контроллер списка шрифтов

Коснувшись первой строки таблицы, показанной на рис. 9.2, можно открыть дочернее представление, показанное на рис. 9.4.



Рис. 9.4. Первый из подконтроллеров приложения Fonts реализует таблицу, в которой каждая строка содержит кнопку раскрытия детализированной информации

Вспомогательные пиктограммы в правом конце каждой строки (рис. 9.4) немногого отличаются от стрелок, которые мы видели раньше. Они называются **кнопками раскрытия детализированной информации** (detail disclosure button). В отличие от индикатора раскрытия, кнопка раскрытия детализированной информации — это

не просто пиктограмма, а управляющий элемент, который пользователь может нажимать. Это означает, что строка с такой кнопкой предоставляет пользователю два варианта действия: одно связывается с выбором строки, а другое — с нажатием кнопки раскрытия. Нажатие маленькой информационной кнопки должно позволить пользователю просматривать и, возможно, изменять более подробную информацию о текущей строке. Наличие же стрелки, направленной вправо, сообщает пользователю, что имеется возможность более глубокой навигации, которую можно найти путем нажатия в другом месте строки.

Контроллер представления размеров шрифтов

Прикосновение к любой строке таблицы, показанной на рис. 9.4, приводит к выводу дочернего представления, показанного на рис. 9.5.



Рис. 9.5. Контроллер представления размеров шрифтов, находящийся на один уровень глубже контроллера представления списка шрифтов, показывает разные размеры выбранного шрифта по одному в строке

Использование индикаторов и кнопок раскрытия

Сформулируем краткие правила применения индикаторов и кнопок раскрытия детализированной информации.

- Если вы хотите предложить единственный выбор при нажатии строки, не используйте вспомогательную пиктограмму, когда нажатие строки ведет *только* к более детальному представлению строки.
- Помечайте строку индикатором раскрытия (стрелка вправо), если нажатие строки ведет к новому представлению, содержащему больше элементов (*не* к детализированному представлению).
- Если вы хотите предложить два варианта выбора в строке, пометьте ее либо с помощью индикатора раскрытия детализированной информации, либо с помощью кнопки детализированной информации. Это позволит пользователю нажать строку для получения нового представления или кнопку раскрытия для получения подробной информации.

Контроллер представления информации о шрифте

Наш последний подконтроллер приложения — единственный, который не является табличным представлением, — показан на рис. 9.6. Это представление появляется при нажатии пиктограммы информации в любой строке в контроллере представления списка шрифтов, показанного на рис. 9.2.

Это представление позволяет пользователю перетащить ползунок, чтобы настроить размер отображаемого шрифта. Оно также включает переключатель, который позволяет указать, должен ли этот шрифт быть перечислен среди предпочтаемых шрифтов пользователя. Если некоторые шрифты установлены как предпочтаемые, они появляются в отдельной группе контроллера корневого представления.

Основа приложения для работы с шрифтами

Программа Xcode предлагает очень хороший шаблон для создания приложений на основе навигации, и вы, скорее всего, будете часто его использовать, когда потребуется создавать иерархические приложения. Однако сейчас мы не собираемся использовать этот шаблон — мы будем строить наше приложение на основе навигации “с нуля”, чтобы прочувствовать, как все соединяется в единое приложение. Мы пройдем через весь процесс создания приложения постепенно, так что он не должен быть тяжелым для вас.

В среде Xcode нажмите комбинацию клавиш **<⌘+Shift+N>** для создания нового проекта. Выберите элемент **Single View Application** из списка шаблонов iOS, а затем щелкните на кнопке **Next**. Установите значение **Fonts** для атрибута **Product Name**, выберите пункт **Swift** в списке **Language** и пункт **Universal** в списке **Devices**. Убедитесь, что флаг **Use Core Data** сброшен, щелкните на кнопке **Next** и выберите местоположение для сохранения вашего проекта.



Рис. 9.6. Последний контроллер представления в приложении Fonts позволяет просмотреть выбранный шрифт любого размера

Настройка контроллера навигации

Теперь нужно создать базовую структуру навигации для нашего приложения. Ее ядром будет класс `UINavigationController`, управляющий стеком контроллеров представлений, между которыми может перемещаться пользователь, и класс `UITableViewController`, показывающий список строк верхнего уровня, который мы собираемся выводить. Оказывается, Interface Builder делает это легко.

Выберите файл `Main.storyboard`. Шаблон создал для нас контроллер основного представления, но вместо него нам надо использовать класс `UINavigationController`, так что выберем контроллер представления либо в области редактора, либо в окне `Document Outline` и удалим его, чтобы раскладировка стала пустой. Затем воспользуемся библиотекой объектов, найдем в библиотеке класс `UINavigationController` и перетащим его экземпляр в редактируемую область. Вы увидите, что фактически получили вместо одной две сцены, подобно тому, что мы видели при создании контроллера представления вкладок в главе 7. Слева находится класс `UINavigationController`. Выберите этот контроллер, откройте окно инспектора атрибутов и установите флагок `Is Initial`

View Controller в разделе View Controller, чтобы этот контроллер появлялся при запуске приложения на выполнение.

Класс UINavigationController подключен ко второй сцене, которая содержит класс UITableView Controller. Вы увидите, что таблица имеет заголовок Root View Controller. Щелкните на нем, откройте окно инспектора атрибутов и установите в качестве заголовка Fonts.

Таким образом, мы получаем представление, созданное контроллером навигации, составное представление, которое содержит комбинацию, состоящую из панели навигации в верхней части экрана (обычно она содержит какой-то заголовок и зачастую некоторую кнопку возврата слева) и содержания того, что контроллер текущего представления контроллера навигации планирует выводить. В нашем случае нижняя часть дисплея будет заполнена табличным представлением, которое было создано вместе с контроллером навигации.

По мере изучения главы вы узнаете больше о том, как управлять тем, что контроллер навигации отображает на панели навигации. Вы также будете понимать, как контроллер навигации перемещает фокус с одного контроллера подчиненного представления на другой. Пока что вы заложили достаточный фундамент, чтобы можно было приступить к определению того, что собираются делать ваши контроллеры пользовательских представлений.

В данный момент основа приложения, по существу, завершена. Вы увидите предупреждение о настройке повторно используемого идентификатора для ячейки таблицы прототипа, но сейчас его можно игнорировать. Сохраните все ваши файлы, а затем постройте и запустите приложение. Если все прошло хорошо, приложение должно запуститься и должна появиться панель навигации с заголовком Fonts. Вы не предоставили табличному представлению никакой информации о том, что следует показывать, так что пока никакие строки в нем не отображаются (рис. 9.7).

Отслеживание предпочтаемых шрифтов

В нескольких точках данного приложения мы планируем дать пользователю возможность поддержки списка любимых шрифтов, позволяя ему добавлять в него выбранные шрифты, просматривать список уже выбранных предпочтаемых шрифтов и удалять из него шрифты. Для того чтобы согласованно управлять этим списком, мы собираемся создать новый класс, который будет работать с массивом предпочтаемых шрифтов и хранить их в пользовательских параметрах настройки данного приложения. Вы узнаете намного больше о пользовательских настройках в главе 12; здесь же мы коснемся только некоторых основ.

Начнем с создания нового класса. Выберите папку Fonts в окне навигатора проекта и нажмите комбинацию клавиш $<\text{⌘}+\text{N}>$ для вызова нового файлового помощника. Выберите элемент Swift File в разделе iOS Source и щелкните на кнопке Next. На следующем экране назовите новый файл FavoritesList.swift и щелкните на кнопке Create. Выберите новый файл в окне навигатора проекта и добавьте в него код, представленный в листинге 9.1.



Рис. 9.7. Основа приложения
без каких-либо данных

Листинг 9.1. Файл класса FavoritesList

```
import Foundation
import UIKit

class FavoritesList {
    static let sharedFavoritesList = FavoritesList()
    private(set) var favorites:[String]

    init() {
        let defaults = UserDefaults.standard
        let storedFavorites = defaults.object(forKey: "favorites") as?
        [String]
        favorites = storedFavorites != nil ? storedFavorites! : []
    }

    func addFavorite(fontName: String) {
        if !favorites.contains(fontName) {
            favorites.append(fontName)
            saveFavorites()
        }
    }

    func removeFavorite(fontName: String) {
        if let index = favorites.index(of: fontName)) {
            favorites.remove(at: index)
            saveFavorites()
        }
    }

    func removeAllFavorites() {
        favorites.removeAll()
        saveFavorites()
    }

    private func saveFavorites() {
        let defaults = UserDefaults.standard
        defaults.set(favorites, forKey: "favorites")
        defaults.synchronize()
    }
}
```

358 ГЛАВА 9 ■ КОНТРОЛЛЕРЫ НАВИГАЦИИ И ТАБЛИЧНЫЕ ПРЕДСТАВЛЕНИЯ

```
func removeFavorite(fontName: String) {
    if let index = favorites.index(of: fontName) {
        favorites.remove(at: index)
        saveFavorites()
    }
}

private func saveFavorites() {
    let defaults = UserDefaults.standard
    defaults.set(favorites, forKey: "favorites")
    defaults.synchronize()
}
}
```

В предыдущем фрагменте мы объявили интерфейс прикладного программирования для нашего нового класса. Для начала мы объявили свойство класса с именем `sharedFavoritesList`, которое возвращает экземпляр данного класса. Независимо от того, сколько раз вызывается этот метод, всегда будет возвращаться один и тот же экземпляр. Идея заключается в том, что `FavoritesList` должен быть синглтоном, т.е. во всем приложении может использоваться только один экземпляр этого класса.

Затем мы объявляем свойство для хранения имен наших предпочтаемых шрифтов. Обратим особое внимание на определение этого массива:

```
private(set) var favorites:[String]
```

Квалификатор `private(set)` означает, что массив может быть прочитан кодом, находящимся вне класса, но модифицировать его может только код внутри класса. Это именно то, что нам надо, поскольку мы хотим, чтобы пользователи класса были способны читать список предпочтаемых шрифтов:

```
let favorites = FavoritesList.sharedFavoriteList.favorites // Чтение  
разрешено
```

Однако мы не хотим, чтобы были позволены приведенные далее действия:

```
FavoritesList.sharedFavoriteList.favorites = [] // Не разрешено  
FavoritesList.sharedFavoriteList.favorites.append(  
    "Comic Sans MS") // Не разрешено
```

Инициализатор класса отвечает за установку начального содержимого массива `favorites`:

```
init() {
    let defaults = UserDefaults.standard
    let storedFavorites = defaults.object(forKey: "favorites") as? [String]
    favorites = storedFavorites != nil ? storedFavorites! : []
}
```

Как вы вскоре увидите, в любой момент, когда мы что-то добавляем в массив или удаляем из него, мы сохраняем его содержимое в настройках пользователя

(о чем мы поговорим более подробно в главе 12) так, чтобы содержимое списка сохранялось при перезапуске приложения. В инициализаторе мы проверяем, имеется ли у нас сохраненный список предпочтаемых шрифтов, и если да, то используем его для инициализации свойства `favorites`. Если нет, то просто делаем его пустым.

Оставшиеся три метода предназначены для добавления и удаления записей из массива предпочтаемых шрифтов. Реализации должны быть самоочевидны. Обратите внимание, что первые два метода вызывают `saveFavorites()`, который сохраняет обновленное значение в пользовательских настройках с тем же ключом (“`favorites`”), который инициализатор использует для чтения. О том, как это работает, вы узнаете подробнее в главе 12; а сейчас достаточно знать, что объект `NSUserDefaults`, который мы используем здесь, действует как своего рода сохраняемый словарь, и все, что мы поместим в него, будет доступно при следующем запуске, когда мы выполним запрос, — даже если при этом приложение было остановлено и перезапущено позже.

ЗАМЕЧАНИЕ. В версии Xcode 8 компания Apple сделала многие старые объекты, имя класса которых начинается с префикса `NS-`, более удобными для использования в языке Swift; в частности, класс `NSUserDefaults` теперь называется просто `UserDefault`.

Создание контроллера корневого представления

Теперь мы готовы начать работу над нашим первым контроллером представления. В предыдущей главе мы использовали простые массивы строк для заполнения таблицы. Мы собираемся сделать что-то подобное и здесь, но на этот раз будем использовать класс `UIFont` для получения списка семейств шрифтов, а затем использовать имена этих семейств шрифтов для заполнения каждой строки. Мы также будем использовать сами шрифты для отображения их названий так, чтобы каждая строка, по сути, содержала небольшой предварительный просмотр семейства шрифтов.

Настало время для создания первого класса контроллера данного приложения. Шаблон создает для нас контроллер представления, но его имя — `ViewController` — не слишком удачное, потому что в этом приложении будет несколько контроллеров представления. Так что сначала выберите файл `ViewController.swift` в окне навигатора проекта и нажмите клавишу `<Delete>`, чтобы удалить его и переместить в корзину. Далее выберите папку `FONTS` в окне навигатора проекта и нажмите комбинацию клавиш `<⌘+N>`, чтобы вызвать помощник по созданию новых файлов. Выберите элемент `Сocoa Touch Class` в разделе `iOS Source` и щелкните на кнопке `Next`. На следующем экране назовите новый класс `RootViewController` и введите имя `UITableViewController` в качестве значения атрибута `Subclass of`. Щелкните на кнопке `Next`, а затем на кнопке `Create` для создания нового класса. Выберите

файл `RootViewController.swift` в окне навигатора проекта и добавьте в него выделенные полужирным шрифтом строки из приведенного далее фрагмента кода.

```
class RootViewController: UITableViewController {
    private var familyNames: [String]!
    private var cellPointSize: CGFloat!
    private var favoritesList: FavoritesList!
    private static let familyCell = "FamilyName"
    private static let favoritesCell = "Favorites"
```

Мы будем с самого начала присваивать значения первым трем из этих свойств, а затем использовать их в различные моменты, когда используется этот класс. Массив `familyNames` будет содержать список всех семейств шрифтов, которые мы планируем отображать; свойство `cellPointSize` будет содержать размер шрифта, который мы будем использовать во всех ячейках нашего табличного представления; а `favoritesList` будет содержать указатель на singleton `FavoritesList`. Последние два свойства являются константами, которые представляют идентификаторы ячеек, которые мы будем использовать для ячеек табличного представления в этом контроллере.

Установите все свойства этого класса, добавив в метод `viewDidLoad()` код, представленный в листинге 9.2.

Листинг 9.2. Метод `viewDidLoad` для файла `RootViewController.swift`

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

    familyNames = (UIFont.familyNames() as [String]).sorted()
    let preferredTableViewFont =
        UIFont.preferredFont(forTextStyle: UIFontTextStyleHeadline)
    cellPointSize = preferredTableViewFont.pointSize
    favoritesList = FavoritesList.sharedFavoritesList
    tableView.estimatedRowHeight = cellPointSize
}
```

В листинге 9.2 мы заполнили массив `familyNames`, запрашивая у класса `UIFont` все известные семейства шрифтов, а затем сортируя получившийся в результате массив. После этого мы обратились к классу `UIFont` еще раз, запрашивая предпочтительный шрифт для заголовка. Мы сделали это с помощью функции, добавленной в систему iOS 7, которая использует размер шрифта, задаваемый пользователем в приложении `Settings`. Это динамическое изменение размера шрифта позволяет пользователю установить общий масштаб шрифта, применяемый во всей системе. Здесь мы использовали свойство `pointSize` для установки базового размера шрифта, который будет использоваться везде в этом контроллере представления. Наконец мы получили singleton с объектом списка предпочтительных шрифтов и задали свойство `estimatedRowHeight` табличного представления, чтобы указать примерную высоту строк таблицы. Если

свойство `estimatedRowHeight` задано, таблица самостоятельно вычислит правильную высоту строк для каждой ячейки на основе ее содержания. Оставим значение свойства `estimatedRowHeight` равным его значению по умолчанию `UITableViewAutomaticDimension` и будем использовать стандартные ячейки табличного представления (если хотите настроить ячейки табличного представления по-своему, используйте ограничения Auto Layout).

Прежде чем двигаться далее, удалим метод `didReceiveMemoryWarning()`, а также закомментированные методы делегата табличного представления и источника данных — мы не собираемся использовать их в этом классе.

Идея, лежащая в основе этого контроллера представления, заключается в том, чтобы показать два раздела. Первый раздел — список всех доступных семейств шрифтов, каждое из которых приводит к списку всех шрифтов семейства. Второй раздел — для предпочтаемых шрифтов — содержит только одну запись, которая ведет пользователя к списку предпочтаемых им шрифтов. Однако, если у пользователя нет предпочтаемых шрифтов (например, когда приложение запускается в первый раз), мы не показываем второй раздел вообще, так как будет выведен пустой список. Таким образом, мы должны выполнить несколько действий в остальной части класса, чтобы обработать этот случай. Первое из них заключается в реализации метода, который вызывается непосредственно перед тем, как представление корневого контроллера представления отображается на экране:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    tableView.reloadData()
}
```

Причина этого заключается в том, что могут быть ситуации, когда множество выводимых объектов меняется от одного просмотра к следующему. Например, пользователь может начать работу без предпочтаемых шрифтов, но затем раскрыть детальную информацию, выбрать шрифт, добавить его в список предпочтаемых, а потом вернуться к корневому представлению. В этот момент нам нужно повторно загрузить табличное представление, так как должен будет появиться второй раздел.

Далее мы реализуем некоторый служебный метод для использования в этом классе. В паре точек при настройке табличного представления с помощью методов источника данных мы должны быть в состоянии выяснить, какой шрифт должен отображаться в ячейке. Мы помещаем эту функциональную возможность в отдельный метод, показанный в листинге 9.3.

Листинг 9.3. Выяснение, какой шрифт мы хотим вывести на экран

```
func fontForDisplay(atIndexPath indexPath: NSIndexPath) -> UIFont? {
    if indexPath.section == 0 {
        let familyName = familyNames[indexPath.row]
        let fontName = UIFont.fontNames(forFamilyName: familyName).first
        return fontName != nil ?
```

```

        UIFont(name: fontName!, size: cellPointSize) : nil
    } else {
        return nil
    }
}
}

```

Этот метод использует класс `UIFont`, чтобы найти все имена шрифтов данного семейства, а потом получить первый шрифт семейства. Нам не обязательно знать, что первый из именованных шрифтов семейства является лучшим для представления всего семейства, но это предположение не хуже любого другого. Если в семействе нет имен шрифтов, возвращается значение `nil`.

Теперь перейдем к основному коду контроллера представления, а именно — к методам источника данных. Для начала определим количество разделов:

```

override func numberOfSections(in tableView: UITableView) -> Int {
    return favoritesList.favorites.isEmpty ? 1 : 2
}
}

```

Мы используем список предпочтаемых шрифтов, чтобы определить, нужно ли показывать второй раздел. Далее определяем количество строк в каждом разделе:

```

override func tableView(_ tableView: UITableView,
                      numberOfRowsInSection section: Int) -> Int {
    // Возвращает количество строк в разделе.
    return section == 0 ? familyNames.count : 1
}
}

```

Эта реализация тоже очень проста. Мы просто используем номер раздела для выяснения, показывает ли этот раздел все имена семейств шрифтов или только одну ячейку, связанную со списком предпочтаемых шрифтов. Теперь определим еще один метод — необязательный метод протокола `UITableViewDataSource`, который позволит нам указать название для каждого из разделов:

```

override func tableView(_ tableView: UITableView,
                      titleForHeaderInSection section: Int) -> String? {
    return section == 0 ? "All Font Families" : "My Favorite Fonts"
}
}

```

Это еще один простой метод, использующий номер раздела, чтобы определить его заголовок. Последним фундаментальным методом, который должен реализовать каждый источник данных табличного представления, является метод для настройки каждой ячейки, приведенный в листинге 9.4.

Листинг 9.4. Функция `cellForRow(atIndexPath:)`

```

override func tableView(_ tableView: UITableView,
                      cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    if indexPath.section == 0 {
        // Список имен шрифтов
}
}

```

```

let cell = tableView.dequeueReusableCell(withIdentifier:
RootViewController.
familyCell, for: indexPath)
cell.textLabel?.font = fontForDisplay(atIndexPath: indexPath)
cell.textLabel?.text = familyNames[indexPath.row]
cell.detailTextLabel?.text = familyNames[indexPath.row]
return cell
} else {
// Список предпочтаемых шрифтов
return tableView.dequeueReusableCell(withIdentifier:
RootViewController.
favoritesCell, for: indexPath)
}

```

Создавая этот класс, мы определили два различных идентификатора ячеек, которые используем для загрузки двух прототипов ячеек из раскладовки (так же, как мы загружали ячейку таблицы из nib-файла в главе 8). Мы еще не настраивали эти прототипы ячеек, но скоро этим займемся. Далее используем номер indexPath, чтобы определить, какие из этих ячеек хотим показать для текущего indexPath. Если ячейка должна содержать имя семейства шрифтов, помещаем имя семейства в свойства `TextLabel` и `detailTextLabel`. Мы также используем в текстовой метке один из шрифтов семейства (тот, который получаем с помощью метода `fontForDisplay(atIndexPath:)`), так что имя семейства шрифтов выводится как шрифтом этого семейства, так и (его уменьшенный вариант) стандартным системным шрифтом.

Начальная настройка раскладовки

Теперь, когда у нас есть контроллер представления (который, как мы думаем, должен что-то показывать), настроим раскладовку, чтобы заставить его работать. Выберите файл `Main.storyboard` в окне навигатора проекта. Вы увидите контроллер навигации и контроллер табличного представления, которые добавили ранее. Первое, что нужно настроить, — это контроллер табличного представления. Класс контроллера по умолчанию — `UITableViewController`. Мы должны заменить его классом корневого контроллера представления. В окне `Document Outline` выберите желтую пиктограмму с надписью `Root View Controller`, а затем используйте инспектор идентичности для изменения параметра `Class` контроллера представления на `RootViewController`.

Еще одно изменение конфигурации, которое необходимо внести прямо сейчас, — это создание пары ячеек-прототипов для сопоставления с идентификаторами ячеек, которые мы использовали в нашем коде. Изначально табличное представление имеет единственную ячейку-прототип. Выберите ее и нажмите комбинацию клавиш `<⌘+D>`, чтобы создать дубликат; вы увидите, что теперь у вас есть две ячейки. Выберите первую, а затем воспользуйтесь инспектором атрибутов, чтобы задать для нее атрибут `Style` равным `Subtitle`; `Identifier` — `FamilyName`; `Accessory` — `Disclosure Indicator`. Далее выберите вторую ячейку-прототип и задайте для нее атрибут `Style` равным `Basic`;

Identifier — Favorites; **Accessory** — Disclosure Indicator. Дважды щелкните на заголовке, отображаемом в ячейке, и замените слово Title словом Favorites.

ПОДСКАЗКА. Обе ячейки-прототипы, которые мы используем в данном примере, имеют стиль ячеек стандартного табличного представления. Если вы установите Style равным Custom, то сможете спроектировать схему ячейки прямо в ячейке-прототипе, так же, как вы создавали ячейку в nib-файле в главе 8.

Теперь постройте и запустите это приложение на вашем устройстве или симуляторе, и увидите красивый список шрифтов. Прокрутите его немного, и увидите, что не все шрифты дают текст одной и той же высоты (рис. 9.8). Тем не менее все ячейки имеют достаточную высоту, чтобы вмещать их содержание, даже несмотря на то, что для этого мы ничего специально не делали. Если вы забыли, почему это происходит, вернитесь к обсуждению метода viewDidLoad().

Первый подконтроллер: представление списка шрифтов

В настоящее время наше приложение отображает только список семейств шрифтов. Мы хотим добавить для пользователя возможность прикоснуться к семейству шрифтов и увидеть все шрифты, которые он содержит, так что давайте сделаем новый контроллер представления, который может управлять списком шрифтов. Воспользуемся новым файловым помощником в среде Xcode для создания нового класса Cocoa Touch под названием FontListViewController, являющегося подклассом UITableViewcontroller. Выберите файл FontListViewController.swift в окне навигатора проекта и добавьте в него следующие свойства:

```
class FontListViewController: UITableViewController {
    var fontNames: [String] = []
    var showsFavorites: Bool = false
    private var cellPointSize: CGFloat!
    private static let cellIdentifier = "FontName"
```

Свойство fontNames представляет собой именно то свойство, которое мы будем использовать для указания контроллеру представления, что требуется выводить. Мы также создали свойство showsFavorites, которое будем использовать для того, чтобы дать знать контроллеру представления, показывает ли он список предпочтаемых шрифтов или просто список шрифтов в семействе, поскольку это окажется полезным в дальнейшем. Мы будем использовать свойство cellPointSize для хранения предпочтительного размера для вывода каждого шрифта; этот размер мы получаем с помощью UIFont. Наконец cellIdentifier — это идентификатор, используемый для ячеек табличного представления данного контроллера.

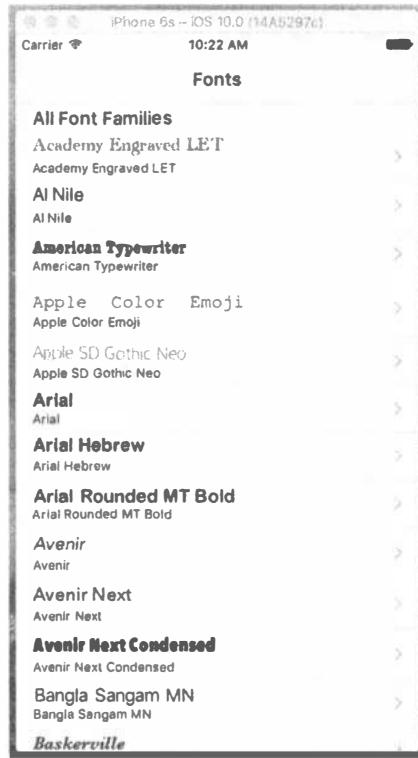


Рис. 9.8. Контроллер корневого представления выводит инсталлированные семейства шрифтов

Для инициализации свойства `cellPointSize` добавьте в метод `viewDidLoad()` код, представленный в листинге 9.5.

Листинг 9.5. Метод `viewDidLoad` для файла `RootViewController.swift`

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

    let preferredTableViewFont =
        UIFont.preferredFont(forTextStyle: UIFontTextStyleHeadline)
    cellPointSize = preferredTableViewFont.pointSize
    tableView.estimatedRowHeight = cellPointSize
}
```

Следующее, что нам надо сделать, — это создать небольшой вспомогательный метод для выбора шрифта для показа в каждой строке, похожий на тот, который используется у нас в классе `RootViewController`. Однако здесь код немного отличается. Вместо списка семейств шрифтов в этом контроллере представления мы храним список названий шрифтов и будем использовать класс `UIFont` для получения каждого имени шрифта следующим образом:

```
func fontForDisplay(atIndexPath indexPath: NSIndexPath) -> UIFont {
    let fontName = fontNames[indexPath.row]
    return UIFont(name: fontName, size: cellPointSize)!
}
```

Теперь создадим небольшое дополнение в виде реализации метода `viewWillAppear()`. Вспомните, как в классе `RootViewController` мы реализовали этот метод для случая, когда список предпочтаемых шрифтов может изменяться, требуя обновления. То же самое применимо и здесь. Этот контроллер представления может показывать список предпочтаемых шрифтов, и пользователь может переключиться на другой контроллер представления, изменить предпочтаемые шрифты (об этом — позже), а затем вернуться обратно. В таком случае нам нужно повторно загрузить табличное представление. Именно для этого предназначен метод, приведенный в листинге 9.6.

Листинг 9.6. Обновление представления в случае изменений

```
override func viewWillAppears(_ animated: Bool) {  
    super.viewWillAppears(animated)  
    if showsFavorites {  
        fontNames = FavoritesList.sharedFavoritesList.favorites  
        tableView.reloadData()  
    }  
}
```

Основная идея заключается в том, что этому контроллеру представления в нормальном режиме передается список названий шрифтов перед тем, как он будет выведен на экран, и этот список остается неизменным на все время вывода контроллера. В одном частном случае (который вы увидите позже) этот контроллер представления должен перезагрузить свой список шрифтов.

Продолжая работу, мы можем полностью удалить метод `numberOfSectionsInTableView()`. Здесь у нас будет только один раздел, так что его удаление эквивалентно его реализации, которая всегда возвращает единицу. Далее реализуем два других метода главного источника данных (листинг 9.7).

Листинг 9.7. Методы источника данных

```

cell.textLabel?.font = fontForDisplay(atIndexPath: indexPath)
cell.textLabel?.text = fontNames[indexPath.row]
cell.detailTextLabel?.text = fontNames[indexPath.row]

return cell
}

```

Ни один из этих методов не нуждается в каких-либо объяснениях, потому что они очень похожи на методы, которые мы использовали в `RootViewController`, и даже проще них.

Мы дополним этот класс позже, но сначала хотим увидеть его в действии. Для этого следует настроить раскадровку, а затем внести некоторые изменения в `RootViewController`. Перейдем для начала к файлу `Main.storyboard`.

Раскадровка списка шрифтов

Раскадровка в настоящее время содержит контроллер табличного представления, который отображает список семейств шрифтов, встроенный в контроллер навигации. Нам нужно добавить один новый слой глубины для включения контроллера представления, который будет отображать шрифты для данного семейства. Найдите элемент `Table View Controller` в библиотеке объектов и перетащите его в область редактирования, справа от существующего контроллера табличного представления. Выберите новый контроллер табличного представления и с помощью инспектора идентичности установите в качестве его класса `FontListViewController`. Выберите ячейку-прототип в табличном представлении и откройте окно инспектора атрибутов для внесения некоторых исправлений. Измените атрибут `Style` на `Subtitle`, `Identifier` — на `FontName`, а `Accessory` — на `Detail Disclosure`. Использование раскрытия детализации позволит строкам этого типа отвечать на два вида нажатий, так что пользователи смогут активизировать два разных действия в зависимости от того, нажимают ли они аксессуар или любую иную часть строки.

Одним из способов добиться того, чтобы одно действие пользователя в одном контроллере представления приводило к созданию экземпляра и отображению другого контроллера представления, является создание *перехода* (*segue*) между ними. Вероятно, это слово не знакомо большинству из вас, так что поясним: *segue*, по сути, означает “переход” и иногда используется писателями и кинематографистами для описания плавного перехода от одного абзаца (или сцены) к следующему. Компания Apple могла бы быть попроще и назвать его просто переходом, но, возможно, из-за того, что это слово уже появляется в интерфейсах API `UIKit`, разработчики решили использовать отдельный термин, чтобы избежать путаницы. Следует также заметить, что слово “*segue*” произносится точно так же, как название `Segway` для самоката (зато теперь вы знаете, почему он носит такое название).

Часто такие переходы создаются исключительно в пределах программы `Interface Builder`. Идея заключается в том, что действие в одной из сцен может

вызвать переход к загрузке и отображению другой сцены. Если вы используете контроллер навигации, такой переход может внести следующий контроллер в стек навигации автоматически. Мы будем использовать эту функциональность в нашем приложении и начнем прямо сейчас.

Для того чтобы ячейки контроллера корневого представления выводили контроллер представления списка шрифтов, необходимо создать пару переходов, соединяющих две сцены. Это делается путем проведения соединительной линии от первой из двух ячеек-прототипов в сцене Fonts к новой сцене; вы увидите, что вся сцена подсвечивается, когда указатель проходит над ней, а это указывает на то, что она готова к подключению (рис. 9.9).



Рис. 9.9. Создание перехода из контроллера списка шрифтов к контроллеру имен шрифтов

Отпустите кнопку мыши и выберите пункт **show** в разделе **Selection Segue** всплывающего меню. Потом сделайте то же самое для других ячеек-прототипов. Создание этих переходов означает, что, как только пользователь коснется какой-либо из этих клеток, будет выделен и подготовлен контроллер представления на другом конце соединения.

Подготовка контроллера корневого представления к переходам

Сохраните внесенные изменения и вернитесь к `RootViewController.swift`. Обратите внимание, что мы говорим не о нашем последнем классе, `FontListViewController`, а о его “родительском” контроллере. Это место, где

вы должны реагировать на прикосновения пользователя в корневом табличном представлении путем подготовки нового класса `FontListViewController` (определенного одним из только что созданных переходов) для отображения и передачи ему значений, которые требуется вывести.

Реальная подготовка нового контроллера представления выполняется с использованием метода `prepareForSegue(_:sender:)`. Добавьте реализацию этого метода, приведенную в листинге 9.8.

Листинг 9.8. Подготовка нового контроллера представления к выводу на экран

```
// MARK: - Navigation

override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    // Получает новый контроллер представления, используя
    // [segue destinationViewController].
    // Передает выбранный объект новому контроллеру представления.
    let indexPath = tableView.indexPath(for: sender as! UITableViewCell)!
    let listVC = segue.destinationViewController as! FontListViewController

    if indexPath.section == 0 {
        // Список имен шрифтов
        let familyName = familyNames[indexPath.row]
        listVC.fontNames = (UIFont.fontNames(forFamilyName: familyName)
            as [String]).sorted()
        listVC.navigationItem.title = familyName
        listVC.showsFavorites = false
    } else {
        // Список предпочтительных шрифтов
        listVC.fontNames = favoritesList.favorites
        listVC.navigationItem.title = "Favorites"
        listVC.showsFavorites = true
    }
}
```

Этот метод использует параметр `sender` (произошло прикосновение к объекту класса `UITableViewCell`) для определения, какой строки коснулся пользователь, и запрашивает переход к объекту `destinationViewController`, представляющему собой экземпляр класса `FontListViewController`, который будет выводиться. Затем мы передаем некоторые значения новому контроллеру представления, в зависимости от того, что нажал пользователь: семейство шрифтов (раздел 0) или ячейку предпочтаемых шрифтов (раздел 1). Так же, как и при установке пользовательских свойств целевого контроллера представления, мы обращаемся к свойству `navigationItem` контроллера для того, чтобы задать заголовок. Свойство `navigationItem` представляет собой экземпляра `UINavigationItem`, который является классом каркаса `UIKit`, содержащим сведения о том, что будет выводиться на панели навигации для любого заданного контроллера представления.

Теперь запустите приложение. Вы увидите, что после нажатия на имя любого шрифта семейства отобразится список всех отдельных шрифтов, которые оно содержит (рис. 9.10). Кроме того, вы можете нажать на метку `Fonts` в заголовке

списка шрифтов контроллера навигации списка шрифтов, чтобы вернуться к родительскому контроллеру и выбрать другой шрифт.

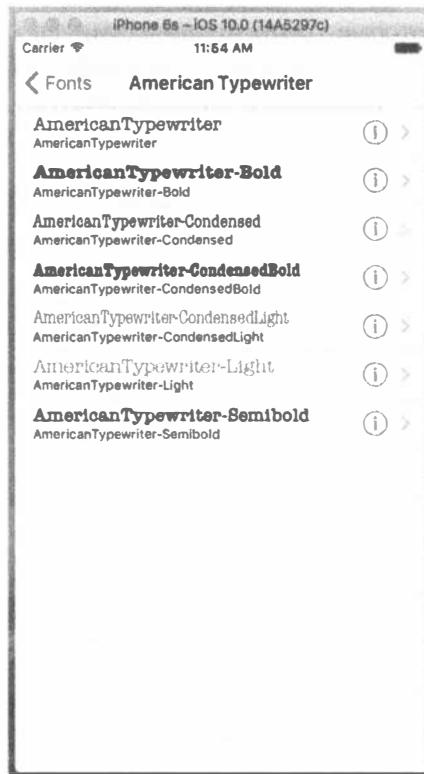


Рис. 9.10. Демонстрация отдельных шрифтов, содержащихся в семействе шрифтов

Создание контроллера представления размеров шрифтов

Однако вы должны заметить, что в настоящее время приложение не позволяет вам идти дальше. На рис. 9.4 и 9.5 показаны дополнительные экраны, которые позволяют просматривать выбранный шрифт различными способами и которые пока что отсутствуют. Но это ненадолго! Создадим показанное на рис. 9.4 представление, которое показывает сразу несколько размеров шрифта. Используя те же шаги, что и при создании класса `FontListViewController`, добавьте новый контроллер представления, являющийся подклассом класса `UITableViewController`, и назовите его `FontSizesViewController`. Единственным параметром, который этот класс будет получать от родительского контроллера, является шрифт. Кроме того, нам понадобится пара закрытых свойств.

Для начала перейдите к файлу `FontSizesViewController.swift` и удалите методы `didReceiveMemoryWarning` и `numberOfSectionsInTableView`, а также все закомментированные методы в нижней части — они нам не нужны. Затем добавьте следующие свойства в верхней части определения класса:

```
import UIKit
class FontSizesViewController: UITableViewController {
    var font: UIFont!
    private static let pointSizes: [CGFloat] = [
        9, 10, 11, 12, 13, 14, 18, 24, 36, 48, 64, 72, 96, 144
    ]
    private static let cellIdentifier = "FontNameAndSize"
```

Свойство `font` будет устанавливаться объектом класса `FontListViewController` перед тем, как он затолкнет этот контроллер представления в стек контроллеров навигации. Свойство `pointSizes` представляет собой массив размеров, с которыми будет отображаться шрифт. Нам также нужен следующий вспомогательный метод, который получает версию шрифта с заданным размером на основе индекса строки таблицы:

```
func fontForDisplay(atIndexPath indexPath: NSIndexPath) -> UIFont {
    let pointSize = FontSizesViewController.pointSizes[indexPath.row]
    return font.withSize(pointSize)
}
```

Мы также должны задать свойство табличного представления `estimatedRowHeight`, чтобы таблица могла автоматически корректировать высоту каждой строки в зависимости от ее содержания. Для этого добавим в метод `viewDidLoad()` следующий код.

```
tableView.estimatedRowHeight = FontSizesViewController.pointSizes[0]
```

Значение, которое присваивается этому свойству, на самом деле не играет никакой роли, мы произвольным образом выбрали самый маленький размер шрифта, который может отображаться в таблице.

Для этого контроллера представления мы собираемся игнорировать метод, который позволяет нам определить количество выводимых разделов, поскольку собираемся использовать только их число по умолчанию (1). Однако мы должны реализовать методы для указания числа строк и содержимого каждой ячейки. Эти два метода приведены в листинге 9.9.

Листинг 9.9. Методы источника данных для табличного представления `FontSizeViewController`

```
// MARK: - Table view data source
override func tableView(_ tableView: UITableView,
                      numberOfRowsInSection section: Int) -> Int {
    return FontSizesViewController.pointSizes.count
}
```

372 ГЛАВА 9 * КОНТРОЛЛЕРЫ НАВИГАЦИИ И ТАБЛИЧНЫЕ ПРЕДСТАВЛЕНИЯ

```
override func tableView(_ tableView: UITableView,
                      cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: FontSizesViewController.cellIdentifier,
        for: indexPath)

    cell.textLabel?.font = fontForDisplay(atIndexPath: indexPath)
    cell.textLabel?.text = font.fontName
    cell.detailTextLabel?.text =
        "\u{FontSizesViewController.pointSizes[indexPath.row]} point"
    return cell
}
```

В каждом из этих методов нет ничего того, чего мы не видели раньше, так что сразу перейдем к созданию графического интерфейса пользователя.

Создание раскладовки контроллера представления размеров шрифтов

Вернитесь к файлу Main.storyboard и перетащите еще один контроллер табличного представления в область редактирования. Воспользуйтесь инспектором идентичности, чтобы установить его класс FontSizesViewController. Вам нужно создать подключение перехода от родительского узла FontListViewController. Так что найдите контроллер, нажмите клавишу <Control> и перетащите указатель от его ячейки-прототипа в новейший контроллер представления, а затем выберите пункт show в разделе Selection Segue всплывающего меню. Далее выберите ячейку-прототип в только что добавленной новой сцене и используйте инспектор атрибутов, чтобы установить для атрибута Style значение Subtitle, а для Identifier — FontNameAndSize.

Подготовка контроллера представления списка шрифтов к переходам

Теперь, так же, как и в прошлый раз, когда мы расширяли иерархию навигации раскладовки, перейдем к родительскому контроллеру, чтобы он мог настроить свой дочерний контроллер. Это означает, что мы должны перейти к файлу FontListViewController.swift и реализовать метод prepareForSegue(_:sender:) так, как показано в листинге 9.10.

Листинг 9.10. Метод preparedForSegue для табличного представления FontSizeViewController

```
// MARK: - Navigation

override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    // Получает новый контроллер представления, используя
    // [segue destinationViewController].
    // Передает выбранный объект новому контроллеру представления.
    let tableViewCell = sender as! UITableViewCell
    let indexPath = tableView.indexPath(for: tableViewCell)!
    let font = fontForDisplay(atIndexPath: indexPath)
```

```

let sizesVC = segue.destinationViewController as!
FontSizesViewController
sizesVC.title = font.fontName
sizesVC.font = font
}

```

Вероятно, все выглядит довольно знакомо для вас, поэтому не будем останавливаться на этом коде.

Запустите приложение, выберите семейство шрифтов, шрифт (путем нажатия строки в любом месте, кроме ее правой части), и вы увидите список с разными размерами шрифтов, показанный на рис. 9.11.



Рис. 9.11. Список табличного представления с переменными размерами шрифтов

Создание контроллера представления информации о шрифте

Последний контроллер представления, который мы создадим, показан на рис. 9.5. Этот контроллер не основан на табличном представлении. Вместо этого он оснащен большой текстовой меткой, ползунком для настройки размера текста и переключателем для включения этого шрифта в список предпочтительных шрифтов. Создайте в вашем проекте новый класс Cocoa Touch, используя UIViewController как суперкласс, и назовите его FontInfoViewController. Как и большинство других контроллеров в этом приложении, данный контроллер должен иметь пару параметров, передаваемых ему родительским контроллером.

Эта проблема решается путем определения в файле `FontInfoViewController.swift` свойств и четырех выходов, которые мы будем использовать при построении пользовательского интерфейса:

```
class FontInfoViewController: UIViewController {
    var font: UIFont!
    var favorite: Bool = false
    @IBOutlet weak var fontSampleLabel: UILabel!
    @IBOutlet weak var fontSizeSlider: UISlider!
    @IBOutlet weak var fontSizeLabel: UILabel!
    @IBOutlet weak var favoriteSwitch: UISwitch!
```

Затем реализуйте `viewDidLoad()` и пару методов действий, которые будут запускаться соответственно ползунком и переключателем (листинг 9.11).

Листинг 9.11. Методы `viewDidLoad()`, ползунка и переключателей

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

    // Дополнительная настройка после загрузки представления.
    fontSampleLabel.font = font
    fontSampleLabel.text =
        "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVv"
        + "WwXxYyZz 0123456789"
    fontSizeSlider.value = Float(font.pointSize)
    fontSizeLabel.text = "\(Int(font.pointSize))"
    favoriteSwitch.isOn = favorite
}

@IBAction func slideFontSize(slider: UISlider) {
    let newSize = roundf(slider.value)
    fontSampleLabel.font = font.withSize(CGFloat(newSize))
    fontSizeLabel.text = "\(Int(newSize))"
}

@IBAction func toggleFavorite(sender: UISwitch) {
    let favoritesList = FavoritesList.sharedFavoritesList
    if sender.isOn {
        favoritesList.addFavorite(fontName: font.fontName)
    } else {
        favoritesList.removeFavorite(fontName: font.fontName)
    }
}
```

Эти методы довольно простые. Метод `viewDidLoad()` устанавливает отображение на основе выбранного шрифта; `slideFontSize()` изменяет размер шрифта в метке `fontSampleLabel` на основе значения ползунка; а `toggleFavorite()` добавляет текущий шрифт в список предпочтаемых шрифтов или удаляет его оттуда в зависимости от значения переключателя.

Раскадровка контроллера представления информации о шрифте

Теперь вернемся к файлу Main.storyboard, чтобы создать графический интерфейс пользователя для этого последнего контроллера представления в приложении. Найдите в библиотеке объектов стандартный элемент View Controller. Перетащите его в область редактирования и воспользуйтесь инспектором идентичности, чтобы задать его класс FontInfoViewController. Затем найдите в библиотеке объектов остальные объекты и перетащите их в новую сцену. Вам нужны три метки, переключатель и ползунок. Расположите их примерно так, как показано на рис. 9.12.



Рис. 9.12. Макет метки, переключателя и ползунка

Обратите внимание, что мы оставили некоторое пространство выше верхней метки, так как собираемся в конечном итоге разместить там панель навигации. Кроме того, мы хотим, чтобы верхняя метка могла отображать длинные фрагменты текста в несколько строк, но по умолчанию метка выводит только одну строку. Для того чтобы изменить это поведение, выберите метку, откройте окно инспектора атрибутов и установите в поле Lines значение 0.

На рис. 9.12 также показан измененный текст в двух нижних метках. Внесите и здесь такие же изменения. На рисунке не показано, что для выравнивания обеих меток по правой стороне был использован инспектор атрибутов. Вы должны поступить так же, так как обе они должны быть привязаны к правому краю. Кроме того, установите ползунок в нижней части и воспользуйтесь инспектором атрибутов, чтобы установить значение **Minimum** равным 1, а **Maximum** — 200.

Теперь пора подключать все соединения для этого графического интерфейса. Начните с выбора контроллера представления и открытия инспектора связей. Когда требуется сделать много соединений, этот инспектор предоставляет возможность неплохого их обзора. Перетаскивая указатель, свяжите каждый выход (маленькие кружки рядом с именами `favoriteSwitch`, `fontSampleLabel`, `fontSizeLabel` и `fontSizeSlider`) с соответствующими объектами в сцене. Выход `fontSampleLabel` должен быть связан с меткой в верхней части; выход `fontSizeLabel` — с меткой в правом нижнем углу; выходы `favoriteSwitch` и `fontSizeSlider` — с любыми местами, где они могут располагаться. Для связывания действий с элементами управления можно продолжать использовать инспектор связей. Находясь в разделе **Received Actions** для контроллера представления, перетащите указатель от маленького кружка рядом с `slideFontSize`: к ползунку, отпустите кнопку мыши и выберите из всплывающего меню пункт **Value Changed**. Затем перетащите указатель от маленького кружка рядом с `toggleFavorite`: к переключателю и вновь выберите из всплывающего меню пункт **Value Changed**. Связи должны выглядеть так, как показано на рис. 9.13.

Кроме того, нам нужно создать переход для демонстрации представления. Напомним, что это представление будет отображаться всякий раз, когда пользователь нажимает пиктограмму просмотра детальной информации (маленькая синяя буква “i” в круге) в контроллере представления списка шрифтов. Итак, найдите этот контроллер, нажмите клавишу `<Control>`, перетащите указатель от прототипа ячейки к новому контроллеру представления информации о шрифте и выберите пункт **show** из раздела **Accessory Action** всплывающего меню. Обратите внимание, что мы говорим **Accessory Action**, а не **Selection Segue** (рис. 9.14). Действие аксессуара является переходом, который запускается, когда пользователь касается пиктограммы детальной информации, тогда как выбор перехода является переходом, который запускается нажатием в любом ином месте строки. Мы уже установили, что выбор перехода этой ячейки открывает объект `FontSizesViewController`.

Теперь у нас есть два разных перехода, которые могут запускаться нажатиями в разных частях строки. Поскольку они представляют разные контроллеры представления с разными свойствами, нам нужно иметь возможность их различать. К счастью, класс `UIStoryboardSegue`, который представляет переход, позволяет это сделать: мы можем использовать идентификатор так же, как мы делаем это с ячейками табличного представления.

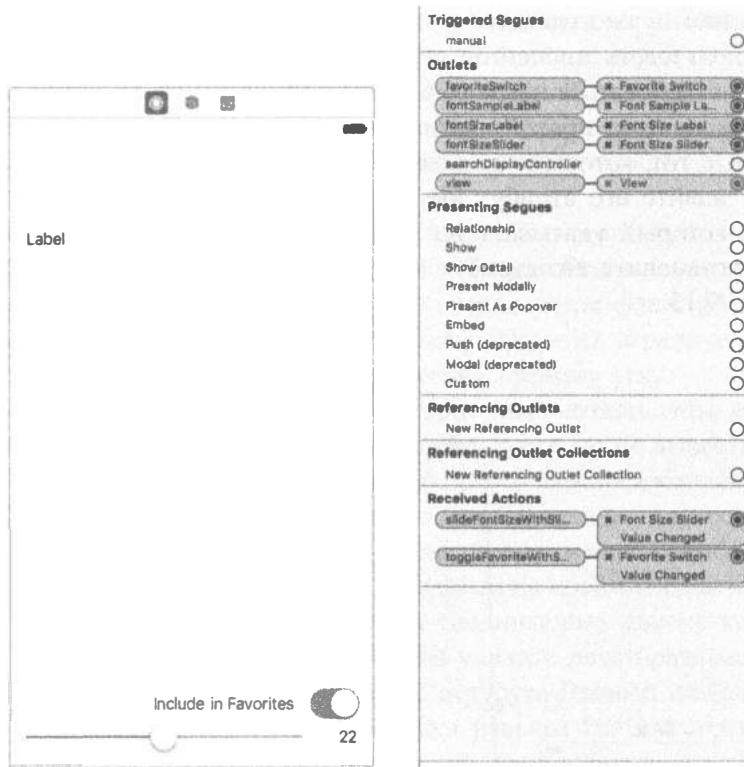


Рис. 9.13. Связи в раскладовке контроллера FontInfoViewController

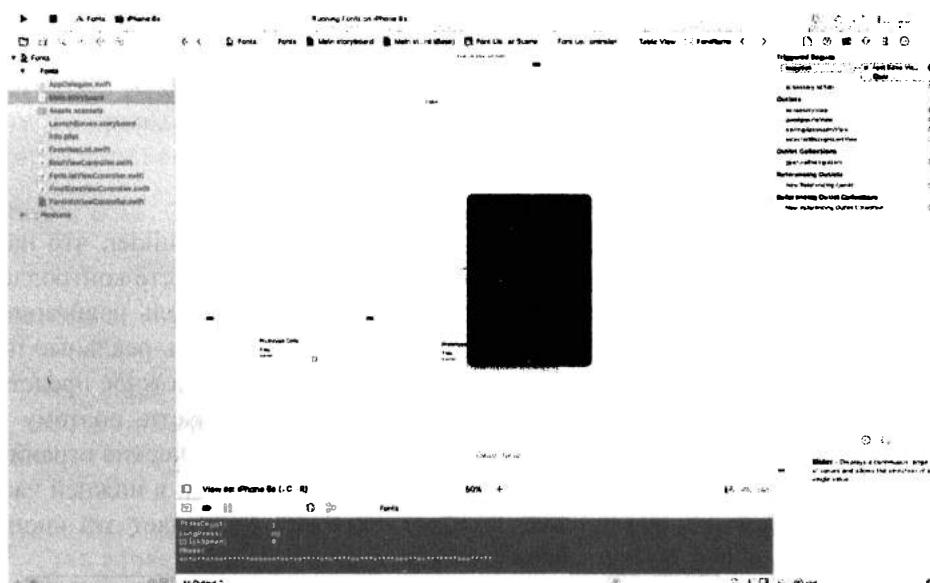


Рис. 9.14. Настройка переходов для действия Accessory Action

Все, что вам нужно сделать, — это выбрать переход в области редактирования и использовать инспектор атрибутов для задания его идентификатора. Вам может потребоваться немного сдвинуть ваши сцены, чтобы видеть оба перехода, выходящие из правой части контроллера представления списка шрифтов. Выберите тот, который указывает на контроллер представления размеров шрифта, и задайте его атрибут Identifier равным ShowFontSizes. Далее выберите тот, который указывает на контроллер представления информации о шрифте, и установите его атрибут Identifier равным ShowFontInfo, как показано на рис. 9.15.

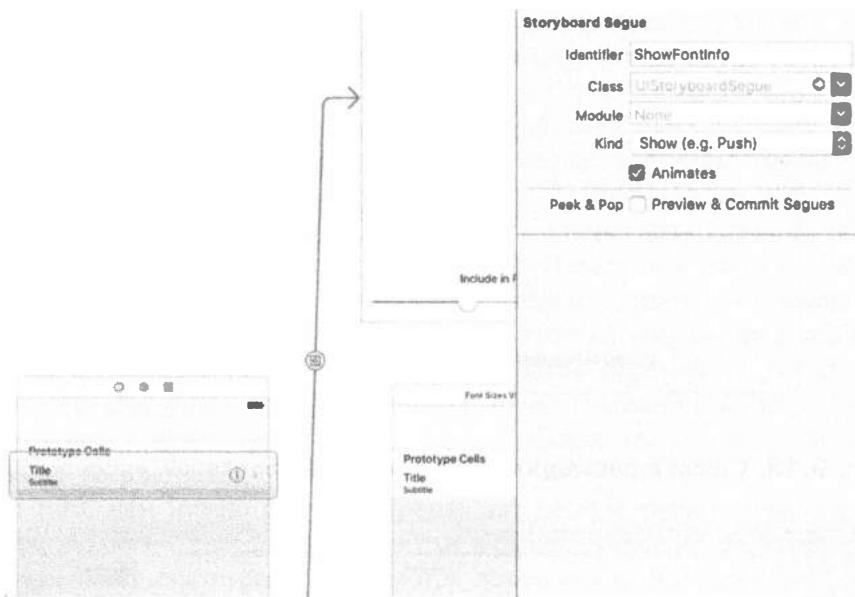


Рис. 9.15. Идентификация переходов

Настройка ограничений

Настройка этого перехода дает знать программе Interface Builder, что наша новая сцена будет использоваться, как и все остальное, в контексте контроллера навигации, так что сцена автоматически получает пустую панель навигации в верхней части. Теперь, когда у нашего представления появились реальные пределы, самое время для настройки ограничений. Это довольно сложное представление с несколькими представлениями, особенно в нижней части, поэтому мы не можем полностью полагаться на то, что системные автоматические ограничения сделают все вместо нас. Мы будем использовать кнопку Pin в нижней части области редактирования и всплывающее окно, которое открывает эта кнопка, чтобы создать большую часть необходимых нам ограничений.

Начнем с самой верхней метки. Щелкните на кнопке Pin, а затем во всплывающем окне выберите небольшие красные полоски выше, слева и справа от

небольшого квадрата, но не под ним. Затем щелкните на кнопке Add 3 Constraints в нижней части.

Выберите ползунок в нижней части и щелкните на кнопке Pin. На этот раз выберите красные полоски ниже, слева и справа от маленького квадрата, но не над ним. Вновь щелкните на кнопке Add 3 Constraints, чтобы поместить их на место.

Для каждой из двух оставшихся меток и переключателя выполните следующие действия: выберите объект, щелкните на кнопке Pin, выберите красные полоски ниже и справа от маленького квадрата, установите флагки Width и Height и щелкните на кнопке Add 4 Constraints. Установка этих ограничений для всех трех объектов будет привязывать их к нижнему правому углу.

Есть еще только одно ограничение, которое надо создать. Мы хотим, чтобы верхняя метка увеличивалась, чтобы содержать текст, но не вырастала настолько, чтобы перекрывать представления снизу. Этого можно добиться с помощью единственного ограничения. Проведите соединительную линию от верхней метки к метке `Include in favorites`, отпустите кнопку мыши и выберите `Vertical Spacing` из всплывающего меню. Затем щелкните на новом ограничении для его выбора (это голубая вертикальная полоса, соединяющая две метки) и откройте окно инспектора атрибутов, в котором вы увидите некоторые настраиваемые атрибуты ограничения. Измените значение атрибута `Relation` на `Greater Than or Equal`, а затем задайте значение `Constant` равным 10. Это гарантирует, что расширение верхней метки не затронет другие представления в нижней части.

Адаптация контроллера представления списка шрифтов для нескольких переходов

Вернемся вновь к файлу `FontListViewController.swift`. Поскольку этот класс теперь будет иметь возможность запускать переходы к двум различным дочерним контроллерам представлений, необходимо адаптировать метод `prepareForSegue(_:sender:)` так, как показано в листинге 9.12.

Листинг 9.12. Обработка нескольких переходов

```
// MARK: - Navigation

override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    // Получает новый контроллер представления, используя
    // [segue destinationViewController].
    // Передает выбранный объект новому контроллеру представления.
    let tableViewCell = sender as! UITableViewCell
    let indexPath = tableView.indexPath(for: tableViewCell)!
    let font = fontForDisplay(atIndexPath: indexPath)

    if segue.identifier == "ShowFontSizes" {
        let sizesVC = segue.destinationViewController as!
        FontSizesViewController
        sizesVC.title = font.fontName
        sizesVC.font = font
    }
}
```

```

} else {
    let infoVC = segue.destinationViewController as!
FontInfoViewController
    infoVC.title = font.fontName
    infoVC.font = font
    infoVC.favorite =
        FavoritesList.sharedFavoritesList.favorites.contains(font.
fontName)
}
}
}

```

Запустите приложение и посмотрите, что у вас получилось. Выберите семейство, содержащее много шрифтов (например, Gill Sans), а затем коснитесь значка посередине строки любого шрифта. Отобразится список, который вы видели ранее, с шрифтами различных размеров. Нажмите кнопку навигации в левом верхнем углу (она помечена как Gill Sans), чтобы вернуться обратно, а затем нажмите другую строку (на этот раз нажмите справа, на пиктограмме детальной информации). Это нажатие должно вызвать последний контроллер представления, который отображает образец шрифта с ползунком в нижней части, позволяющим выбрать любой размер шрифта.

Кроме того, теперь можно воспользоваться переключателем `Include in favorites` для отметки этого шрифта как избранного. Сделав это, нажмите кнопку навигации в левом верхнем углу несколько раз, чтобы вернуться к контроллеру корневого представления.

Предпочитаемые шрифты

Прокрутите экран вниз до нижней части корневого контроллера представления, и увидите нечто новое: второй раздел, показанный на рис. 9.16.

Тонкости табличного представления

Итак, основные функции нашего приложения готовы. Тем не менее, прежде чем мы сможем действительно назвать его готовым, реализуем в нем еще несколько возможностей. Если вы уже некоторое время используете систему iOS, то, вероятно, вам известно, что можно удалить строку из табличного представления, проведя пальцем справа налево. Например, в приложении Mail можно использовать этот метод для удаления сообщения из списка. Этот жест выводит небольшой графический интерфейс в строке табличного представления. Он запрашивает подтверждение удаления, после чего строка исчезает, а оставшиеся строки поднимаются вверх, чтобы заполнить пробел. Обо всем этом взаимодействии — включая обработку жеста, запрос подтверждения и анимации затронутых строк — заботится само табличное представление. Все, что вам нужно сделать, — это реализовать два метода в своем контроллере.

Кроме того, табличное представление позволяет пользователю легко изменить порядок строк в этом табличном представлении, перетаскивая их вверх и вниз. Как и в случае жеста удаления, табличное представление само заботится

обо всем. Все, что вы должны сделать, — это добавить одну строку установки (чтобы создать кнопку, которая активизирует переупорядочение графического интерфейса), а затем реализовать один метод, который вызывается, когда пользователь заканчивает перетаскивание.



Рис. 9.16. Список ранее выбранных предпочтаемых шрифтов

Реализация жеста удаления

В этом приложении класс `FontListViewController` является типичным примером использования этой возможности. Всякий раз, когда приложение отображает список предпочтаемых шрифтов, мы должны позволить пользователю удалить шрифт из списка предпочтаемых, сделав жест пальцем, коротким путем, т.е. не нажимая пиктограмму детальной информации и переключатель. Выберите для начала файл `FontListViewController.swift` в среде Xcode. Начните с добавления реализации метода `tableView(_:canEditRowAt:IndexPath:)`:

```
override func tableView(_ tableView: UITableView,
    canEditRowAt indexPath: IndexPath) -> Bool {
    return showsFavorites
}
```

Этот метод будет возвращать значение `true`, если выводится список предпочтаемых шрифтов, и `false` в противном случае. Это означает, что функция редактирования, которая позволяет удалять строки, включается только при отображении списка предпочтаемых шрифтов. Если вы попытаетесь запустить приложение и удалить строки с помощью только что внесенного изменения, то не увидите никакой разницы. Табличное представление никак не отреагирует на жест удаления, потому что еще не реализован метод, который требуется для завершения удаления. Так что добавьте реализацию метода `tableView(_:commitEditingStyle:forRowAt indexPath:)`, как показано в листинге 9.13.

Листинг 9.13. Метод, обеспечивающий возможность удаления строк из списка предпочтаемых шрифтов

```
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    if !showsFavorites {
        return
    }

    if editingStyle == UITableViewCellEditingStyle.delete {
        // Удаляет строку из источника данных
        let favorite = fontNames[indexPath.row]
        FavoritesList.sharedFavoritesList.removeFavorite(fontName: favorite)
        fontNames = FavoritesList.sharedFavoritesList.favorites

        tableView.deleteRows(at: [indexPath],
                            with: UITableViewRowAnimation.fade)
    }
}
```

Этот метод вызывается после завершения редактирования таблицы. Он достаточно прост, но в нем есть некоторые тонкости. Первое, что мы делаем, — убеждаемся, что показываем список предпочтаемых шрифтов; если нет, то просто ничего не делаем. Вообще-то, это никогда не должно произойти, так как мы указали в предыдущем методе, что редактируемым может быть только список предпочтаемых шрифтов. Тем не менее на всякий случай мы немножко перестраховываемся. После этого проверяем стиль редактирования, чтобы убедиться, что конкретная операция, которую мы собираемся выполнить, — действительно удаление. Можно добавить вставку в табличное представление, но не без дополнительной настройки, которую мы здесь не делаем, а потому нам не следует беспокоиться о других действиях, кроме удаления. Далее мы определяем, какой шрифт следует исключить из списка, удаляем его из синглтона `FavoritesList` и обновляем локальную копию списка предпочтаемых шрифтов.

Наконец мы требуем от табличного представления удалить строку и заставить ее исчезнуть с визуальным затуханием. Важно понимать, что именно происходит,

когда вы сообщаете табличному представлению о том, что нужно удалить строку. Интуитивно вы можете подумать, что вызов этого метода будет удалять некоторые данные, но это не так. На самом деле мы уже удалили данные! Последний вызов метода — на самом деле просто наш способ сказать табличному представлению “Я внес изменения и хочу, чтобы ты анимировало удаление этой строки. Если тебе нужно что-то еще, обращайся ко мне”. Когда это произойдет, табличное представление начнет анимацию всех строк, находящихся ниже удаленной, перемещая их вверх, так что вполне возможно, что одна или несколько строк, которые были ранее не видны, теперь отобразятся на экране, и произойдет запрос контроллера о данных ячейки обычными методами. Поэтому важно, чтобы наша реализация метода `tableView(_:commitEditingStyle:forRowAtIndexPath:)` вносила необходимые изменения в модель данных (в данном случае — в синглтон `FavoritesList`) перед тем, как требовать от табличного представления удалить строку.

Снова запустите приложение, убедитесь, что у вас есть некоторые предпочтаемые шрифты, а затем перейдите к их списку и удалите строку, проводя по ней пальцем справа налево. Стока частично сдвинется влево, а справа появится кнопка `Delete` (рис. 9.17). Нажмите ее, и строка будет удалена.



Рис. 9.17. Стока предпочтаемого шрифта с кнопкой `Delete`

Реализация переупорядочения перетаскиванием

Последняя возможность, которую мы собираемся добавить в список шрифтов, позволяет пользователям изменять список предпочтаемых шрифтов, перетаскивая их вверх и вниз. Для того чтобы добиться этого, добавим в класс `FavoritesList` один метод, который позволит нам переупорядочивать элементы. Откройте файл `FavoritesList.swift` и добавьте в него следующий метод:

```
func moveItem(fromIndex from: Int, toIndex to: Int) {
    let item = favorites[from]
    favorites.remove(at: from)
    favorites.insert(item, at: to)
    saveFavorites()
}
```

Этот новый метод обеспечивает основу для того, что мы собираемся делать. Выберите файл `FontListViewController.swift` и добавьте следующие строчки в конце метода `viewDidLoad`:

```
if showsFavorites {
    navigationItem.rightBarButtonItem = editButtonItem()
}
```

Мы уже упоминали об этом элементе навигации. Это объект, содержащий информацию о том, что должно появиться на панели навигации контроллера представления. Он имеет свойство `rightBarButtonItem`, которое может содержать экземпляр `UIBarButtonItem`, особый вид кнопки, предназначенный только для панелей навигации и инструментов. Здесь этому свойству присваивается `editButtonItem`, свойство `UIViewController`, которое дает нам специальную кнопку, настроенную для активизации графического интерфейса редактирования или переупорядочения табличного представления.

Попробуйте запустить приложение и перейти к списку предпочтаемых шрифтов. Вы увидите, что теперь в правом верхнем углу имеется кнопка `Edit`. Нажатие этой кнопки включает графический интерфейс редактирования табличного представления, который означает, что каждая строка получает кнопку удаления слева, а содержание сдвигается немного вправо, чтобы освободить место для кнопки (рис. 9.18). Это еще один способ, которым пользователи могут удалять строки, используя уже реализованные нами методы.

Однако наша главная цель — добавление функциональности переупорядочения. Все, что для этого нужно сделать, — это добавить в файл `FontListViewController.swift` следующий метод:

```
override func tableView(_ tableView: UITableView,
                      moveRowAt sourceIndexPath: IndexPath,
                      to destinationIndexPath: IndexPath) {
    FavoritesList.sharedFavoritesList.moveItem(
        fromIndex: sourceIndexPath.row,
        toIndex: destinationIndexPath.row)
    fontNames = FavoritesList.sharedFavoritesList.favorites:
}
```

Этот метод вызывается, как только пользователь завершает перетаскивание строки. Мы сообщаем синглтону `FavoritesList` о том, чтобы он выполнил переупорядочение, а затем обновляем наш список названий шрифтов так же, как после удаления элемента. Чтобы увидеть эту возможность в действии, запустите приложение, перейдите в список предпочтаемых шрифтов и нажмите кнопку `Edit`. Вы увидите, что в режиме редактирования строки включают в правой части пиктограммы для перетаскивания, которые можно использовать для изменения порядка элементов.

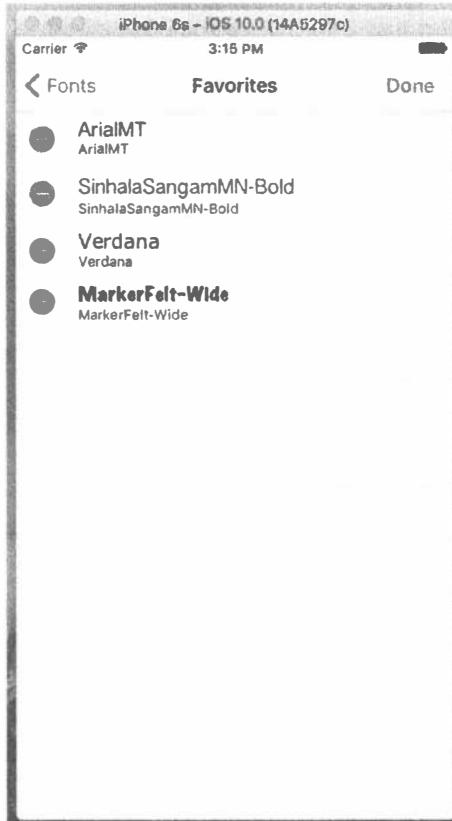


Рис. 9.18. Таблица предпочтаемых шрифтов с добавленной функцией редактирования

Резюме

Несмотря на то что в этой главе мы много работали с табличными представлениями, наше основное внимание было сосредоточено на контроллерах навигации и способах просмотра иерархически организованного содержания в условиях ограниченного пространства, которые характерны для большинства устройств iPhone, особенно в книжной ориентации.

Мы создали приложение для просмотра списка шрифтов, которое демонстрирует не только способы перехода к детализированным представлениям, но и методы обработки многочисленных переходов из отдельной ячейки табличного представления, например при просмотре размеров или информации о шрифтах.

В заключение мы показали, как можно откорректировать табличное представление, чтобы включить в него возможности удаления или перемещения строк.

ГЛАВА 10



Представление коллекции

Многие годы разработчики программ для системы iOS использовали компонент UITableView для создания самых разнообразных интерфейсов. Благодаря тому что класс UITableView предоставляет программисту возможности определять несколько типов ячеек, создавать их “на лету” и прокручивать в вертикальном направлении, он стал ключевым компонентом для тысяч приложений. На протяжении долгого времени компания Apple основывала свой подход к созданию интерфейсов прикладного программирования на классе табличного представления, постоянно добавляя новые, более эффективные возможности для его наполнения в новых выпусках системы iOS. Однако во многих областях обработки данных табличное представление не является окончательным решением. Например, если необходимо представить данные из нескольких столбцов, приходится объединять все столбцы в каждой строке данных в одну ячейку. Кроме того, компонент UITableView не позволяет прокручивать свое содержимое в горизонтальном направлении. В целом большая часть методов класса UITableView является результатом определенного компромисса: разработчики не имеют контроля над общей схемой табличного представления. Программист может как угодно определить внешний вид любой отдельной ячейки по своему желанию, но в конце дня они просто нагромождаются одна над другой в одном большом списке прокрутки.

В системе iOS 6 был введен новый класс UICollectionView, который должен был устранить эти недостатки. Как и табличное представление, этот класс позволяет выводить на экран совокупность ячеек, заполненных данными, и работать с ними, располагая неиспользованные ячейки в очереди для использования в будущем. Однако, в отличие от табличного представления, класс UICollectionView не создает из этих ячеек вертикальный стек. На самом деле компонент UICollectionView вообще их не представляет на экране. Вместо этого он поручает эту работу вспомогательному классу.

Создание проекта DialogViewer

Для того чтобы продемонстрировать некоторые возможности класса UICollectionView, мы будем использовать его для представления нескольких абзацев текста. Каждое слово будет помещаться в отдельной ячейке, и все ячейки каждого абзаца будут объединяться в разделы. Каждый раздел также будет иметь свой заголовок. Все это, может быть, не слишком захватывающе, особенно если учесть, что библиотека UIKit уже содержит другие превосходные способы для представления текста, но их использование было бы непедагогичным, поскольку мы хотим объяснить вам, насколько гибким является новый класс. На практике вы вряд ли станете делать что-нибудь подобное тому, что изображено на рис. 10.1, с помощью табличного представления.



Рис. 10.1. Каждое слово находится в отдельной ячейке за исключением заголовков. Все они выводятся на экран с помощью компонента UICollectionView и не требуют от программиста явных геометрических вычислений

Для того чтобы решить поставленную задачу, необходимо определить несколько пользовательских классов и воспользоваться классом `UICollectionViewFlowLayout` (единственным вспомогательным классом для вывода данных на экран, включенных в библиотеку `UIKit` в настоящее время), а затем, как обычно, использовать контроллер представления для того, чтобы связать все это в одно целое.

Откройте в среде Xcode новый проект `Single View Application`, как мы делали уже много раз. Назовите свой проект `DialogViewer` и используйте стандартные настройки, применяемые повсеместно в этой книге (выберите пункт `Swift` в списке `Language` и пункт `Universal` в списке `Devices`). Откройте файл `ViewController.swift` и измените его суперкласс на `UICollectionView`:

```
class ViewController: UICollectionViewController {
```

Откройте файл `Main.storyboard`. Нам нужно настроить контроллер представления, чтобы он соответствовал тому, что мы только что указали в файле `ViewController.swift`. Выберите элемент `View Controller` в окне `Document Outline` и удалите его, оставив раскладовку пустой. Найдите в библиотеке объектов элемент `Collection View Controller` и перетащите его в область редактирования. Вы увидите, что объект `Collection View` в окне `Document Outline` имеет вложенное представление коллекции. Его отношение с представлением коллекции очень похоже на отношение между классами `UITableViewController` и вложенным в него классом `UITableView`. Выберите пиктограмму контроллера представления коллекции и с помощью инспектора идентичности измените его класс на `ViewController` точно так же, как мы это сделали в подклассе `UICollectionViewController`. Перейдите в окно инспектора атрибутов и установите флажок `Is Initial View Controller`. Затем выберите представление коллекции в окне `Document Outline` и с помощью инспектора атрибутов измените цвет его фона на белый. В заключение вы увидите, что представление коллекции имеет дочернее представление с именем `Collection View Cell`. Это ячейка-прототип, которую можно использовать для разработки макета ваших реальных ячеек в программе `Interface Builder`. Мы не собираемся делать это в данной главе, так что выделите эту ячейку и удалите ее.

Определение пользовательских ячеек

Теперь определим несколько классов для ячеек. Как показано на рис. 10.1, мы выводим на экран два основных вида ячеек: “обычную”, которая содержит слово, и “необычную”, которая используется как заголовок. Любая ячейка, которую вы будете использовать совместно с классом `UICollectionView`, должна быть подклассом системного класса `UICollectionViewCell`, обеспечивающего основные функциональные возможности, аналогичные возможностям класса `UITableViewCell`. Эта функциональная возможность включает `backgroundView`, `contentView` и т.д. Поскольку две наши ячейки имеют

сходные функциональные возможности, мы сделаем одну из них подклассом другой и будем замещать в подклассе некоторые методы.

Начнем с создания нового класса Cocoa Touch в среде Xcode. Назовите новый класс ContentCell и сделайте его подклассом класса UICollectionViewCell. Выберите исходный файл нового класса и добавьте в него объявления трех свойств и заготовку метода класса, как показано в листинге 10.1.

Листинг 10.1. Определение класса ContentCell

```
class ContentCell: UICollectionViewCell {
    var label: UILabel!
    var text: String!
    var maxWidth: CGFloat!

    class func sizeForContentString(s: String,
                                    forMaxWidth maxWidth: CGFloat) -> CGSize {
        return CGSize.zero
    }
}
```

Свойство label будет указывать на объект UILabel, используемый для вывода метки на экран. Свойство text будет сообщать ячейке, что именно в ней должно отображаться, свойство maxWidth — управлять максимальной шириной ячейки, а метод sizeForContentString(_:_forMaxWidth:) , который мы вскоре реализуем, будет использоваться для запроса размера ячейки, чтобы она могла уместить заданную строку на экране. Это удобно при создании и настройке экземпляров классов, описывающих наши ячейки.

Добавьте переопределения методов init(frame:) и init(coder:) в UIView, как показано в листинге 10.2.

Листинг 10.2. Переопределения методов в классе ContentCell

```
override init(frame: CGRect) {
    super.init(frame: frame)
    label = UILabel(frame: self.contentView.bounds)
    label.isOpaque = false
    label.backgroundColor =
        UIColor(red: 0.8, green: 0.9, blue: 1.0, alpha: 1.0)
    label.textColor = UIColor.black()
    label.textAlignment = .center
    label.font = self.dynamicType.defaultFont()
    contentView.addSubview(label)
}

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

Код, приведенный в листинге 10.2, довольно простой. Он создает метку, задает ее свойства для отображения на экране и добавляет ее в объект contentView. Единственная тонкость состоит в том, что этот код определяет шрифт для метки

с помощью метода `defaultFont()`. Идея заключается в том, что этот класс должен определять, каким шрифтом отображать содержимое, позволяя своим подклассам объявлять собственные шрифты, переопределяя метод `defaultFont()`. Поскольку мы пока не создали метод `defaultFont()`, настал момент это сделать.

```
class func defaultFont() -> UIFont {
    return UIFont.preferredFontForTextStyle(UIFontTextStyleBody)
}
```

Все довольно просто. Здесь, чтобы получить предпочтительный пользовательский шрифт для основного текста, используется метод `preferredFontForTextStyle()` класса `UIFont`. Пользователь может использовать приложение `Settings`, чтобы изменить размер данного шрифта. С помощью этого метода (вместо жесткого кодирования размера шрифта) мы делаем наши приложения более удобными для пользователей. Обратите внимание на то, как вызывается этот метод.

```
label.font = self.dynamicType.defaultFont()
```

Метод `defaultFont()` является методом типа класса `ContentCell`. Для того чтобы вызвать его, должно было бы использоваться имя класса:

```
ContentCell.defaultFont()
```

В данном случае это не сработает. Если вызов производится от имени подкласса `ContentCell` (такого, как класс `HeaderCell`, который мы вскоре создадим), то на самом деле мы хотим вызвать замещенный метод `defaultFont()` из подкласса. Для того чтобы сделать это, нам нужна ссылка на объект подкласса. Это именно то, что дает нам выражение `self.dynamicType`. Если это выражение выполняется из экземпляра класса `ContentCell`, оно возвращает объект типа `ContentCell`, и мы будем вызывать метод `defaultFont()` этого класса; но в подклассе `HeaderCell` оно возвращает объект `HeaderCell`, так что будет вызываться метод `defaultFont()` класса `HeaderCell`, а это именно то, что нам нужно. Для того чтобы закончить этот класс, реализуем метод, заготовку которого добавили ранее и который вычисляет подходящий размер ячейки, как показано в листинге 10.3.

Листинг 10.3. Вычисление приблизительного размера ячейки

```
class` func sizeForContentString(s: String,
                                forMaxWidth maxWidth: CGFloat) -> CGSize {
    let maxSize = CGSize(width: maxWidth, height: 1000.0)
    let opts = NSStringDrawingOptions.usesLineFragmentOrigin

    let style = NSMutableParagraphStyle()
    style.lineBreakMode = NSLineBreakMode.byCharWrapping
    let attributes = [ NSFontAttributeName: defaultFont(),
                      NSParagraphStyleAttributeName: style]
    let string = s as NSString
```

```

let rect = string.boundingRect(with: maxSize, options: opts,
                                attributes: attributes, context: nil)
return rect.size
}

```

Метод, приведенный в листинге 10.3, делает много полезного, так что его стоит рассмотреть подробнее. Сначала мы объявляем максимальный размер так, что не допускаются слова, которые могут быть шире, чем значение аргумента maxWidth, которое получается из ширины UICollectionView. Мы также создаем стиль абзаца, который позволяет переносить слишком большие строки текста на следующие строки абзаца. Мы еще создаем словарь атрибутов, который содержит определенный нами для данного класса шрифт по умолчанию и стиль абзаца, который мы только что создали. Наконец используем функции класса NSString, предоставляемого библиотекой UIKit, который позволяет нам рассчитать размеры строки. Мы передаем максимальный размер и другие настроенные нами параметры и атрибуты и получаем размер ячейки.

Осталось правильно задать свойство text. Вместо обычной практики, когда мы используем неявную переменную экземпляра, определим get- и set-методы, которые будут извлекать и задавать значение на основе объекта класса UILabel, созданного ранее, используя в качестве хранилища для отображаемого значения объект класса UILabel. Благодаря этому мы можем также использовать set-метод для повторного вычисления геометрических размеров ячейки при изменении текста. Заменим определение свойства text в файле ContentCell.swift кодом, приведенным в листинге 10.4.

Листинг 10.4. Определение текстового свойства в файле ContentCell.swift

```

var label: UILabel!
var text: String! {
    get {
        return label.text
    }
    set(newText) {
        label.text = newText
        var newLabelFrame = label.frame
        var newContentFrame = contentView.frame
        let textSize = self.dynamicType.sizeForContentString(
            s: newText,
            forMaxWidth: maxWidth)
        newLabelFrame.size = textSize
        newContentFrame.size = textSize
        label.frame = newLabelFrame
        contentView.frame = newContentFrame
    }
}

```

В get-методе нет ничего особенного, но set-метод выполняет некоторую дополнительную работу. В основном он модифицирует рамку для метки и представления содержимого, ориентируясь на размеры, требуемые для вывода текущей строки.

Это все, что требуется для базового класса ячейки. Теперь создадим класс ячейки для заголовка. Создайте в среде Xcode еще один новый класс на языке Сосоа Touch, назовите его HeaderCell и сделайте его подклассом класса ContentCell. Заголовочный файл трогать не надо, поэтому перейдите к файлу HeaderCell.swift и внесите в него несколько изменений. В этом классе мы заменим несколько методов класса ContentCell для изменения внешнего представления ячейки, чтобы оно отличалось от обычного, как показано в листинге 10.5.

Листинг 10.5. Класс HeaderCell

```
class HeaderCell: ContentCell {

    override init(frame: CGRect) {
        super.init(frame: frame)
        label.backgroundColor = UIColor(red: 0.9, green: 0.9,
                                         blue: 0.8, alpha: 1.0)
        label.textColor = UIColor.black()
    }

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

    override class func defaultFont() -> UIFont {
        return UIFont.preferredFont(forTextStyle: UIFontTextStyleHeadline)
    }
}
```

Теперь ячейка, содержащая заголовок, будет выглядеть иначе, потому что у нее свои цвет и шрифт.

Настройка контроллера представления

Выберите файл ViewController.swift и начните с объявления массива для содержимого, которое мы собираемся выводить (листинг 10.6).

Листинг 10.6. Содержимое класса ViewController.swift

```
private var sections = [
    ["header": "First Witch",
     "content" : "Hey, when will the three of us meet up later?"],
    ["header" : "Second Witch",
     "content" : "When everything's straightened out."],
    ["header" : "Third Witch",
     "content" : "That'll be just before sunset."],
    ["header" : "First Witch",
     "content" : "Where?"],
    ["header" : "Second Witch",
     "content" : "The dirt patch."],
    ["header" : "Third Witch",
     "content" : "I guess we'll see Mac there."]
]
```

Массив `sections` содержит список словарей, каждый из которых имеет два ключа: `header` и `content`. Значения, ассоциированные с этими ключами, будут использованы для определения содержимого экрана.

Аналогично классу `UITableView`, класс `UICollectionView` позволяет зарегистрировать класс повторно используемой ячейки, используя идентификатор. Это позволяет в дальнейшем вызвать метод извлечения из очереди, в которой будет находиться ячейка, и, если ячейка недоступна, представление коллекции создаст ее автоматически, как и класс `UITableView`. Добавьте следующую строку в конец метода `viewDidLoad()`, чтобы реализовать эту возможность:

```
self.collectionView?.register(ContentCell.self,
    forCellReuseIdentifier: "CONTENT")
```

Так как это приложение не имеет панели навигации, основное представление будет перекрываться со строкой состояния. Для того чтобы предотвратить это явление, добавьте следующие строки в конец `viewDidLoad()`:

```
var contentInset = collectionView!.contentInset
contentInset.top = 20
collectionView!.contentInset = contentInset
```

Указанной конфигурации метода `viewDidLoad()` пока достаточно. Прежде чем код станет заполнять представление коллекции, необходимо написать небольшой вспомогательный метод. Все наше содержимое экрана содержится в длинных строках, а мы собираемся выводить слова одно за другим, помещая их в отдельные ячейки. Следовательно, нам нужен внутренний метод для разделения строк на слова. Этот метод будет получать номер раздела, извлекать соответствующие строки из данных указанного раздела и разделять их на слова.

```
func wordsInSection(section: Int) -> [String] {
    let content = sections[section]["content"]
    let spaces = NSCharacterSet.whitespacesAndNewlines
    let words = content?.components(separatedBy: spaces)
    return words!
}
```

Представление содержимого ячеек

Пришло время для группы методов, которые будут заполнять представление коллекции реальными данными. Следующие три метода очень похожи на свои аналоги из класса `UITableView`. Для начала нам нужен метод, сообщающий коллекции, сколько разделов надо вывести на экран.

```
override func numberOfSections(in collectionView: UICollectionView) -> Int {
    return sections.count
}
```

Далее приведен метод, сообщающий коллекции, сколько элементов содержится в каждом разделе. Эту задачу решает метод `wordsInSection:`, определенный ранее.

```

override func collectionView(_ collectionView: UICollectionView,
    numberOfItemsInSection section: Int) -> Int {
    let words = wordsInSection(section: section)
    return words.count
}

```

В листинге 10.7 метод возвращает отдельную ячейку, предназначенную для хранения одного слова. Этот метод использует метод `wordsInSection()`. Он применяет метод извлечения из очереди к объекту класса `UICollectionView` аналогично классу `UITableView`. Поскольку мы зарегистрировали класс ячейки для идентификатора, мы знаем, что метод извлечения из очереди всегда возвращает экземпляр.

Листинг 10.7. Содержимое класса `ViewController.swift`

```

override func collectionView(_ collectionView: UICollectionView,
    cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
    let words = wordsInSection(section: indexPath.section)
    let cell = collectionView.dequeueReusableCell(withIdentifier: "CONTENT", for: indexPath) as! ContentCell
    cell.maxWidth = collectionView.bounds.size.width
    cell.text = words[indexPath.row]
    return cell
}

```

Зная, как работает класс `UITableView`, вы можете предположить, что в этот момент приложение уже способно что-то делать. Скомпилируйте и запустите приложение, и увидите, что это не так (рис. 10.2).

Мы видим отдельные слова, а не поток слов. Все ячейки имеют одинаковые размеры, и все они прижаты одна к другой. Причина заключается в том, что нам нужно предпринять меры для дополнительной ответственности делегатов.

Создание потока

До сих пор мы работали с классом `UICollectionView`, но, как уже указывалось, у этого класса есть ассистент, который на самом деле реализует макет. Класс `UICollectionViewFlowLayout`, по умолчанию выполняющий работу по созданию макета для класса `UICollectionView`, имеет несколько методов делегата, с помощью которого он выдает нам информацию. Один из этих методов реализуем прямо



Рис. 10.2. Это выглядит не слишком работоспособным...

сейчас. Объект макета применяет этот метод к каждой ячейке, чтобы определить ее требуемый размер. Здесь мы снова используем метод `wordsInSection()` для получения доступа к требуемому слову, а затем метод, определенный ранее в классе `ContentCell` и определяющий требуемый размер ячейки.

При инициализации `UICollectionViewController` он становится делегатом своего класса `UICollectionView`. `UICollectionViewFlowLayout` представления коллекции будет рассматривать контроллер представления как собственный делегат, если он объявит о его соответствии протоколу `UICollectionViewDelegateFlowLayout`. Первое, что нам нужно сделать, — это изменить объявление нашего контроллера представления в файле `ViewController.swift` так, чтобы он объявил о соответствии этому протоколу:

```
class ViewController: UICollectionViewController,
    UICollectionViewDelegateFlowLayout {
```

Все методы протокола `UICollectionViewDelegateFlowLayout` являются необязательными, и нам нужно реализовать только один из них. Добавьте следующий метод в файл `ViewController.swift` (листинг 10.8).

Листинг 10.8. Изменение размеров ячейки с помощью протокола `UICollectionViewDelegateFlowLayout`

```
func collectionView(collectionView: UICollectionView,
    layout collectionViewLayout: UICollectionViewLayout,
    sizeForItemAtIndexPath indexPath: NSIndexPath) -> CGSize {
    let words = wordsInSection(indexPath.section)
    let size = ContentCell.sizeForContentString(words[indexPath.row],
        forMaxWidth: collectionView.bounds.size.width)
    return size
}
```

Скомпилируйте и запустите приложение снова, и увидите, что оно стало намного лучше (рис. 10.3).

Теперь ячейки образуют поток и текст стал более удобочитаемым, причем каждый раздел немного смешен вниз. И все же разделы слишком прижаты один к другому. Это не слишком красиво. Исправим это, добавив немного параметров конфигурации. Добавьте следующие строки в конец метода `viewDidLoad()`:

```
let layout = collectionView!.collectionViewLayout
let flow = layout as! UICollectionViewFlowLayout
flow.sectionInset = UIEdgeInsetsMake(10, 20, 30, 20)
```

Здесь мы извлекаем объект макета из представления коллекции. Сначала присваиваем его временной переменной, тип которой выводится как указатель на объект класса `UICollectionViewLayout`. Мы делаем это в основном для того, чтобы подчеркнуть, что только этот объект знает о существовании обобщенного класса макета, но в действительности он использует экземпляр класса `UICollectionViewFlowLayout`, который является подклассом класса `UICollectionViewLayout`. Зная истинный

типа объекта макета, мы можем использовать операторы приведения типа для присвоения его другой переменной корректного типа и для доступа к методам, которые содержатся только в этом подклассе, — в данном случае нам нужен `set`-метод для свойства `sectionInset`. Еще раз скомпилируйте и выполните приложение, и увидите, что ячейки более равномерно заполняют пространство (рис. 10.4).



Рис. 10.3. Теперь абзацы выделяются



Рис. 10.4. Стало куда свободнее

Представления заголовка

Осталось вывести на экран объекты заголовков. Класс `UITableView` имеет представления для заголовков и сносок. Их можно было бы использовать для вывода каждого раздела. Класс `UICollectionView` использует эту концепцию в несколько более обобщенном виде, обеспечивая большую гибкость макета. Наряду с системой доступа к обычным ячейкам из делегатов существует параллельная система для доступа к дополнительным представлениям, которые можно использовать для заголовков, сносок и других элементов. Добавьте следующий код в конец метода `viewDidLoad()`, чтобы представление коллекции знало о существовании класса ячеек для заголовка:

```
self.collectionView?.register(HeaderCell.self,
    forSupplementaryViewOfKind: UICollectionViewElementKindSectionHeader,
    withReuseIdentifier: "HEADER")
```

Как видим, в данном случае мы не только указываем класс ячейки и ее идентификатор, но и вид. Идея заключается в том, что разные макеты могут определять разные виды дополнительных представлений и запрашивать эти представления у делегатов. Класс UICollectionViewLayout будет запрашивать раздел заголовка для каждого раздела представления коллекции (листинг 10.9).

Листинг 10.9. Раздел представления коллекции

```
override func collectionView(_ collectionView: UICollectionView,
    viewForSupplementaryElementOfKind kind: String,
    at indexPath: IndexPath) ->
    UICollectionViewReusableView {
    if (kind == UICollectionViewElementKindSectionHeader) {
        let cell =
            collectionView.dequeueReusableCellSupplementaryView(
                ofKind: kind, withReuseIdentifier: "HEADER",
                for: indexPath) as! HeaderCell
        cell.maxWidth = collectionView.bounds.size.width
        cell.text = sections[indexPath.section]["header"]
        return cell
    }
    abort()
}
```

Обратите внимание на вызов `abort()` в конце данного метода. Эта функция заставляет приложение немедленно прекратить работу. Это не та вещь, к которой следует часто прибегать в рабочем коде. Здесь мы ожидаем только вызова для создания ячейки заголовка и ничего не сможем сделать, если будет запрошен другой вид ячеек, — мы даже не сможем вернуть `nil`, потому что возвращаемый тип метода это не допускает. Если этот метод вызван для создания другой разновидности заголовка, то это серьезная ошибка нашей программы или ошибка в библиотеке UIKit.

Скомпилируйте и выполните приложение, и увидите... постойте! А где же заголовки? Оказывается, класс UICollectionViewFlowLayout не выделяет для них места, если ему точно не указать их размеры. Вернемся к методу `viewDidLoad()` и добавим в него следующую строку:

```
flow.headerReferenceSize = CGSize(width: 100, height: 25)
```

Снова скомпилируйте и выполните приложение, и теперь все заголовки окажутся на своих местах, как показано на рис. 10.1 и 10.5.



Рис. 10.5. Окончательный вид приложения DialogViewer

Резюме

В этой главе мы только слегка коснулись класса UICollectionView и того, что может быть достигнуто с помощью класса по умолчанию UICollectionViewFlowLayout. С его помощью путем определения собственных классов макетов можно получить еще более красивые приложения, но это тема для другой книги.

Рассматривая приложения, созданные с помощью представлений коллекций, следует не забывать о представлениях стека. Эта альтернативная возможность может сэкономить время. Поскольку по мере появления новых версий языка Swift, среди Xcode и iOS книга становится все больше и больше, представления стека мы оставляем читателю в качестве упражнения.

ГЛАВА 11



Разделенные представления и всплывающие меню

В главе 9 мы уделили много внимания навигации приложений, основанной на выбранных элементах в табличных представлениях, где выбор каждого элемента вызывает представление верхнего уровня, которое заполняет весь экран с перемещением влево и отображением следующего в иерархии представления, возможно, уже из другой таблицы. Так работают многие приложения для устройств iPhone и iPod touch, например приложение Mail, которое позволяет вам пробираться через учетные записи и папки до тех пор, пока вы, наконец, не найдете сообщение. Несмотря на то что этот метод работы подходит и для устройства iPad, он может затруднить взаимодействие с пользователем.

Размеры экранов iPhone и iPod touch позволяют плавно заменять одно полноэкранное представление другим. На устройстве iPad аналогичная операция выполняется менее гладко, а иногда даже становится совершенно неприемлемой. Кроме того, в большинстве случаев использование такого большого дисплея для одного табличного представления неэффективно. По этой причине во встроенные iPad-приложения заложено совсем другое поведение. Любые навигационные действия, подобные тем, которые используются в приложении Mail, низводятся до использования одного узкого столбца, содержимое которого смещается влево или вправо по мере того, как пользователь погружается вглубь своих изысканий или “выныривает” назад. При использовании устройства iPad в альбомном режиме навигационный столбец фиксируется слева, а содержимое выбранного элемента отображается справа. Это называется **разделенным представлением** (*split view*) (рис. 11.1), а приложения, созданные таким образом, именуются приложениями **master-detail** (*master–detail application* — главное представление — детализированное представление).

Разделенное представление идеально подходит для разработки таких приложений **master-detail**, как Mail. До появления версии iOS 8 класс разделенного представления `UISplitViewController` был доступен только на iPad, а это

означает, что если бы вы захотели построить универсальное приложение master-detail, вам пришлось бы делать это одним способом на iPad и другим — на iPhone. Теперь же класс `UISplitViewController` доступен везде, т.е. вам больше не нужно писать специальный код для работы с устройством iPhone.

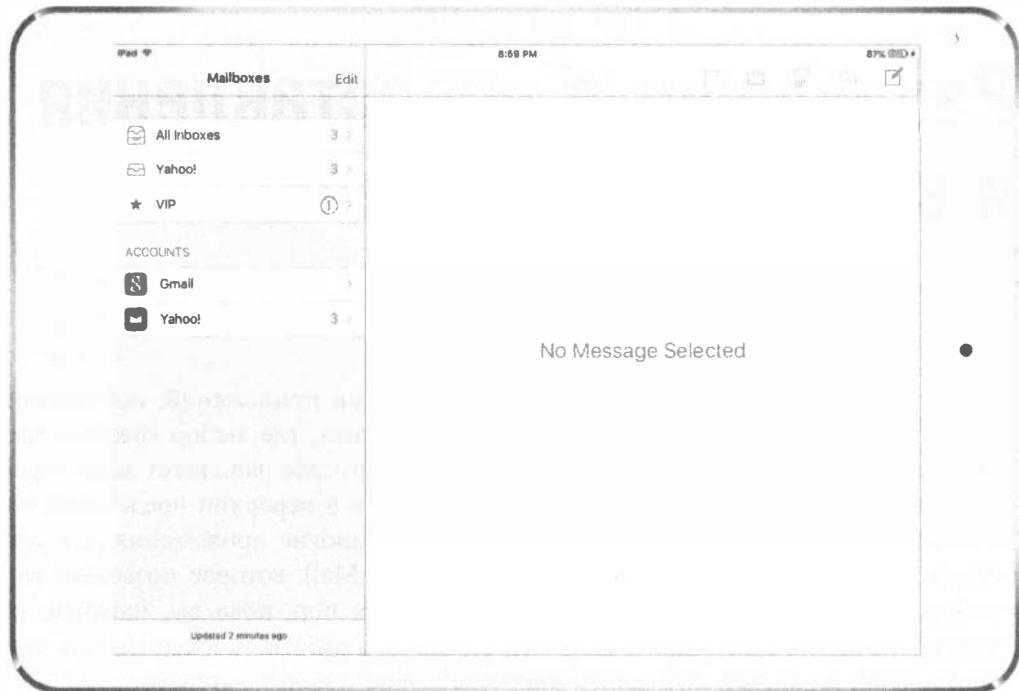


Рис. 11.1. Пример разделенного представления в альбомном режиме на экране iPad. Столбец навигации расположен слева. Выберите в нем элемент, в данном случае — конкретную учетную запись почты, и содержимое этого элемента отобразится в области справа

При использовании iPad ширина левой области разделенного представления составляет по умолчанию 320 точек, а само разделенное представление со смежными областями навигации и содержания обычно работает только в альбомном режиме. Если переключить устройство в книжную ориентацию, разделенное представление, не теряя функциональных возможностей, переходит в неявную форму. Область навигации становится плавающей и может быть активизирована только нажатием кнопки панели инструментов, которое заставит область навигации “всплыть” и отобразиться поверх всего остального изображения на экране (рис. 11.2).

Однако некоторые приложения не следуют строго этому правилу. Приложение `Settings` для устройства iPad, например, использует разделенное представление, которое видимо все время, и его левая часть никогда не исчезает и не перекрывает представление содержимого справа. Но в этой главе мы будем придерживаться шаблона стандартного использования разделенного представления.



Рис. 11.2. Разделенное представление на экране iPad в книжной ориентации. Информация из левой части прежнего разделенного представления в альбомном режиме появляется, только когда пользователь выполняет жест скольжения или нажимает кнопку на панели инструментов

На примере проекта в данной главе вы увидите, как создать приложение *master-detail*, которое использует контроллер разделенного представления. Первоначально мы будем тестировать приложения на симуляторе iPad, но когда оно будет закончено, вы увидите, что тот же код работает и на устройстве iPhone, хотя выглядит при этом иначе. Вы также узнаете, как настраивать внешний вид и поведение разделенного представления и как создавать и выводить всплывающие меню (*popover*), как те, которые вы видели в главе 4, когда мы рассматривали представления предупреждений и списки действий. В отличие от всплывающего меню на рис. 4.28, которое служит “оберткой” для списка действий, в данном случае всплывающее меню будет иметь содержимое, специфичное для данного примера приложения, в частности — список языков (рис. 11.3).

Построение приложения *master-detail* с помощью класса *UISplitViewController*

Начнем с простой задачи и воспользуемся преимуществами одного из предопределенных в программе Xcode шаблонов для создания нашего проекта. Наша цель — написать приложение, которое перечисляет всех президентов США и показывает статью из Википедии о выбранном президенте.

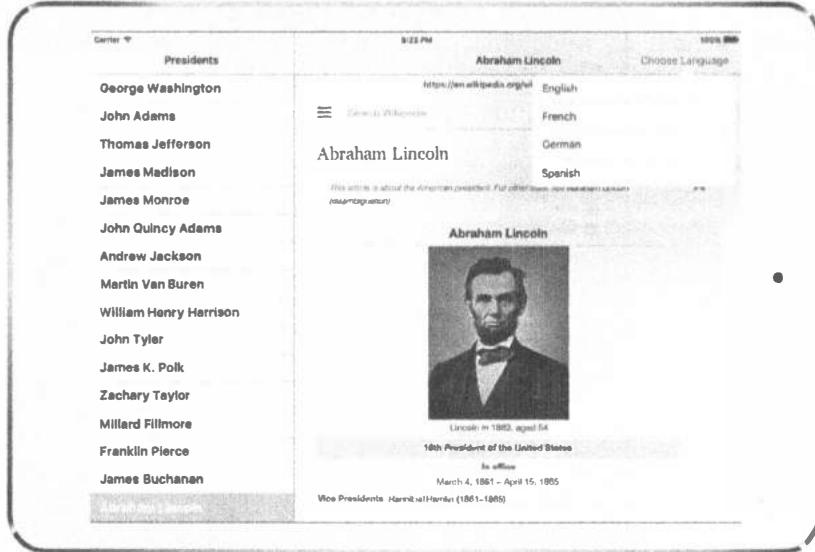


Рис. 11.3. Вспыльвающее меню, визуально выглядящее как растущее из кнопки, которая его активизирует

Перейдите в среду Xcode и выполните команду **File⇒New⇒Project...**. В разделе iOS Application выберите вариант **Master-Detail Application** и щелкните на кнопке **Next**. На следующей странице экрана назовите новый проект **Presidents**, выберите пункт **Swift** в списке **Language** и пункт **Universal** в списке **Devices**. Убедитесь, что все флагки сброшены. Щелкните на кнопке **Next**, выберите место хранения своего проекта и щелкните на кнопке **Create**. Среда Xcode сделает свою обычную работу, создав вместо вас классы и файл раскладовки, а затем покажет проект. Раскройте папку **Presidents** и рассмотрите ее содержимое.

Изначально новый проект содержит класс делегата приложения (как обычно), классы **MasterViewController** и **DetailViewController**. Эти два контроллера обеспечивают представления, которые будут отображаться соответственно в левой и правой частях разделенного представления в альбомной ориентации. Класс **MasterViewController** определяет верхний уровень навигационной структуры, а класс **DetailViewController** задает, что именно отображается в большей области экрана при выборе элемента навигации. При запуске приложения оба контроллера содержатся в разделенном представлении, которое, как вы помните, выполняет небольшую трансформацию при вращении устройства.

Для того чтобы понять, какие функции предоставляет этот конкретный шаблон приложения, скомпонуем его и выполним в симуляторе iPad. При запуске приложения в книжной ориентации вы увидите только контроллер детализированного представления (рис. 11.4, слева). Нажатие на кнопку **Master** панели инструментов или жест слева направо от левого края экрана приводит к появлению контроллера главного представления поверх контроллера детализированного представления (рис. 11.4, справа).

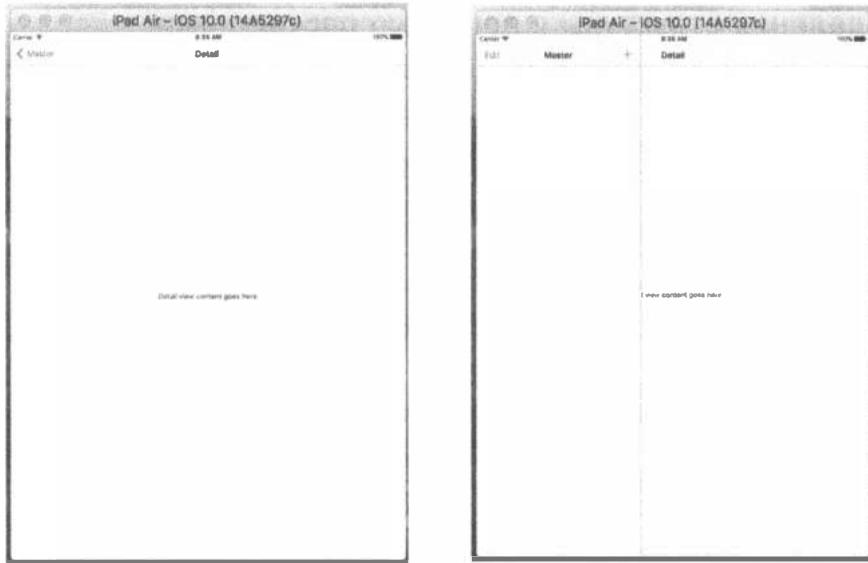


Рис. 11.4. Поведение по умолчанию приложения master-detail в книжной ориентации. Размещение справа подобно приведенному на рис. 11.2

Поверните симулятор влево или вправо в альбомный режим. При этом разделенное представление показывает представление навигации слева и детализированное представление справа (рис. 11.5).



Рис. 11.5. Поведение по умолчанию приложения master-detail в альбомной ориентации. Размещение справа подобно приведенному на рис. 11.1

Определение структуры с помощью раскладовки

Итак, у вас действует довольно сложный набор контроллеров представлений.

- ❖ Контроллер разделенного представления, который содержит все элементы.
- ❖ Контроллер навигации для обработки действий в левой области.
- ❖ Контроллер главного представления (выводит главный список элементов) в контроллере навигации.
- ❖ Контроллер детализированного представления в правой области.
- ❖ Контроллер навигации, играющий роль контейнера для детализированного представления в правой области.

В стандартном шаблоне приложения master-detail, который мы использовали, эти контроллеры представления устанавливаются и связываются между собой в основном файле раскладовки, а не в коде. Редактор интерфейса Interface Builder не только предоставляет инструментарий для быстрого построения графического интерфейса пользователя, но и позволяет связывать различные компоненты без написания кода. Для того чтобы понять, как это происходит, необходимо подробнее рассмотреть файл раскладовки нашего проекта.

Выберите файл Main.storyboard, чтобы открыть его в программе Interface Builder. Этот файл раскладовки содержит множество элементов. Для того чтобы достичь наилучших результатов, следует открыть окно Document Outline (рис. 11.6). Уменьшение масштаба (щелчком правой кнопки мыши в редакторе раскладовки и выбором величины увеличения во всплывающем меню) также может помочь вам увидеть общую картину.

Для того чтобы лучше разобраться во взаимоотношениях контроллеров, откройте инспектор связей и уделите некоторое время изучению контроллеров представлений, щелкая на каждом из них, чтобы понять, как они взаимосвязаны. Ниже кратко описано то, что вы увидите.

- ❖ Объект класса UISplitViewController имеет отношения переходов (segues) к двум контроллерам класса UINavigationController — **контроллерам главного и детализированного представлений**. Они используются для того, чтобы указать объекту класса UISplitViewController, что именно он должен использовать для узкой полосы в левой части (контроллер главного представления) и что должно находиться в большей области отображения (контроллер детализированного представления).
- ❖ Объект класса UINavigationController, связанный переходом с **контроллером главного представления**, имеет отношение **контроллера корневого представления** со своим контроллером корневого представления, который является объектом класса MasterViewController, сгенерированным шаблоном. Контроллер главного представления является подклассом UITableViewcontroller, с которым вы ознакомились в главе 9.

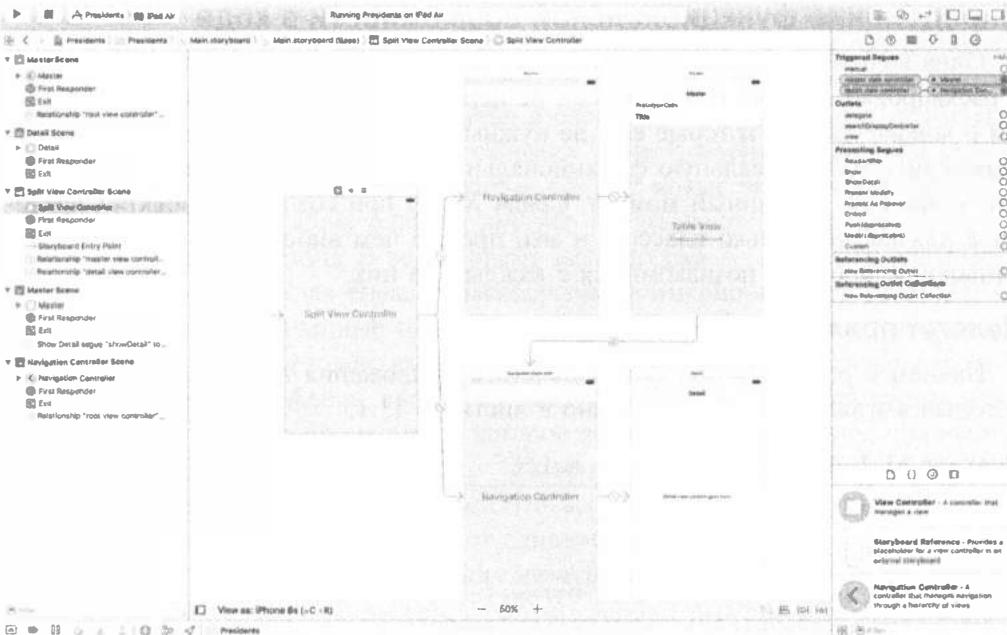


Рис. 11.6. Файл MainStoryboard.storyboard, открытый в программе Interface Builder. Эта сложная иерархия объектов лучше всего видна в окне Document Outline

- ❖ Аналогично другой объект класса UINavigationController имеет отношение контроллера корневого представления с **контроллером детализированного представления**, который является объектом класса шаблона DetailViewController. Контроллер детализированного представления, сгенерированный шаблоном, является обычным объектом подкласса класса UIViewController, но вы вправе использовать здесь любой контроллер представления, который отвечает требованиям вашего приложения.
- ❖ Существует переход раскладовки из ячеек контроллера главного представления к контроллеру детализированного представления типа showDetail. Этот переход приводит к выводу элемента из ячейки в детализированном представлении, на которой пользователь щелкнул мышью. Больше об этом вы узнаете позже, когда мы будем подробно рассматривать контроллер главного представления.

В данный момент файл Main.storyboard содержит определение того, как взаимосвязаны различные контроллеры приложения. Как и в большинстве случаев, использование раскладовок позволяет обойтись без написания объемного кода, что само по себе приятно. По желанию можете изучить все аспекты конфигурации в коде, но, что касается данного примера, мы ограничимся лишь теми строками кода, которые были предоставлены средой Xcode.

Определение функциональной возможности в коде

Одна из основных причин хранить взаимосвязи контроллеров представлений в раскладовке состоит в том, что он не перегружает наш исходный код данными о конфигурации, которые ему не нужны. Следовательно, в коде мы должны определить только реальную функциональную возможность. Итак, рассмотрим, что у нас есть на данный момент. Среда Xcode при создании проекта определила для нас несколько классов, и мы, прежде чем вносить в них какие-либо изменения, поближе познакомимся с каждым из них.

Делегат приложения

Начнем с рассмотрения файла делегата приложения `AppDelegate.swift`, который выглядит так, как показано в листинге 11.1.

Листинг 11.1. Файл `AppDelegate.swift`

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate,
    UISplitViewControllerDelegate {

    var window: UIWindow?

    func application(_ application: UIApplication,
                    didFinishLaunchingWithOptions
        launchOptions: [NSObject: AnyObject]?) -> Bool {
        // Точка замещения для настройки после загрузки представления.
        let splitViewController = self.window!.rootViewController as!
    UISplitViewController
        let navigationController = splitViewController.viewControllers[splitVi
ewController.
            viewControllers.count-1] as! UINavigationController
        navigationController.topViewController!.navigationItem.
        leftBarButtonItem =
        splitViewController.displayModeButtonItem()
        splitViewController.delegate = self
        return true
    }
}
```

Рассмотрим сначала последнюю часть этого кода:

```
splitViewController.delegate = self
```

Эта строка настраивает свойство `delegate` класса `UISplitViewController`, которое в результате указывает на самого себя. Почему мы создали это соединение в коде, а не прямо в раскладовке? Ведь совсем недавно мы говорили об исключении утомительного кодирования (“соедините это с тем”) — одном из реальных преимуществ XIB-файлов и раскладовок. Мы много раз видели, как делегаты включаются в Interface Builder, так почему же не можем сделать это сейчас?

Для того чтобы понять, почему механизм раскадровок для создания соединений не может сработать в данной ситуации, надо вспомнить, чем раскадровка отличается от XIB-файла. XIB-файл — это, по существу, “замороженный” граф объектов. Когда мы погружаем XIB-файл в работающее приложение, объекты, которые он содержит, “оттывают” и “оживают”. Это относится и ко всем взаимосвязям, определенным в файле. Система создает новые экземпляры каждого отдельного объекта в файле, один за другим, и связывает все выходы и подключения между объектами. В то же время раскадровка — это нечто большее. Можно сказать, что каждая сцена в раскадровке примерно соответствует XIB-файлу. Добавляя в метаданные описание того, как сцены связываются одна с другой посредством переходов, вы имеете дело с раскадровкой. Однако в отличие от одиночного XIB-файла, сложная раскадровка обычно не загружается вся сразу. Вместо этого каждое действие, которое вызывает новую сцену, приводит к загрузке из раскадровки конкретного “замороженного” графа объектов, соответствующего этой сцене. Это означает, что все объекты, которые вы видите в раскадровке, необязательно существуют одновременно.

Поскольку программа Interface Builder ничего не знает о взаимосвязях между сценами, она фактически запрещает создавать любые выходы или соединения “цель/действие” от объекта в одной сцене к объекту в другой сцене. Фактически единственными разрешенными связями между сценами являются переходы.

Можете попробовать сами. Сначала выберите в раскадровке элемент Split View Controller (вы найдете его в окне Document Outline в разделе Split View Controller Scene). Вызовите инспектор связей и попробуйте перетащить соединение от выхода делегата к другому контроллеру представления или объекту. Вы можете перетаскивать все в пределах области макетирования и представления списка и не найдете ни одной подсказки, которая указывала бы на то, что в данном месте объект можно отпустить. Единственный способ создать такое соединение — написать соответствующий код. Эти небольшие дополнительные фрагменты кода — не слишком большая цена, учитывая, сколько кода устраниется из-за нашего использования раскадровок.

Давайте вернемся к началу и посмотрим, что происходит в начале метода application(_:didFinishLaunchingWithOptions:):

```
let splitViewController =
    self.window!.rootViewController as UISplitViewController
```

Эта инструкция позволяет нам получить свойство окна rootViewController, которое указывается в раскадровке плавающей стрелкой. Если вернуться к рис. 11.6, вы увидите, что стрелка указывает на наш экземпляр класса UISplitViewController. Далее идет код

```
let navigationController = splitViewController.viewControllers[
    splitViewController.viewControllers.count-1]
    as UINavigationController
```

В этой строке мы “погружаемся” в массив `viewControllers` объектов класса `UISplitViewController`. Когда разделенное представление загружается из раскладовки, этот массив содержит ссылки на контроллеры навигации, обернутые вокруг контроллеров главного и детализированного представлений. Мы получаем последний элемент в этом массиве, который указывает на объект класса `UINavigationController` для нашего детализированного представления. Наконец мы видим следующий код:

```
navigationController.topViewController.navigationItem.leftBarButtonItem =
    splitViewController.displayModeButtonItem()
```

Он присваивает свойство `displayModeButtonItem` контроллера разделенного представления навигационной панели контроллера детализированного представления. Свойство `displayModeButtonItem` представляет собой кнопку полосы, которая создается и управляется самим разделенным представлением. Этот код фактически является добавлением кнопки `Master`, которую вы видели на панели навигации на рис. 11.4, слева. В устройстве iPad разделенное представление показывает эту кнопку, когда устройство находится в книжной ориентации, и контроллер главного представления не виден. Когда устройство поворачивается в альбомную ориентацию или пользователь нажимает кнопку, чтобы сделать видимым контроллер главного представления, кнопка скрывается.

Контроллер главного представления

Рассмотрим класс `MasterViewController`, управляющий настройкой табличного представления, содержащего панель навигации приложения. Вот как выглядит код в начале файла `MasterViewController.swift` (листинг 11.2).

Листинг 11.2. Файл `MasterViewController.swift`

```
import UIKit

class MasterViewController: UITableViewController {

    var detailViewController: DetailViewController? = nil
    var objects = [AnyObject]()

    override func viewDidLoad() {
        super.viewDidLoad()
        // Дополнительная настройка после загрузки представления,
        // обычно из nib-файла.
        self.navigationItem.leftBarButtonItem = self.editButtonItem()

        let addButton = UIBarButtonItem(barButtonSystemItem: .add,
                                       target: self, action: #selector(insertNewObject(_:)))
        self.navigationItem.rightBarButtonItem = addButton
        if let split = self.splitViewController {
            let controllers = split.viewControllers
            self.detailViewController = (controllers[controllers.count-1] as!
                UINavigationController).topViewController as? DetailViewController
        }
    }
```

Основной интерес здесь представляет метод `viewDidLoad()`. В предыдущих главах при реализации контроллеров табличного представления, которые реагируют на выбор пользователем строки в таблице, на этом этапе вы обычно создавали новый контроллер представления и помещали его в стек контроллера навигации. В данном приложении контроллер приложения, который мы хотим отобразить, уже существует: это экземпляр класса `DetailViewController`, содержащийся в файле раскладовки. Поэтому здесь мы получаем экземпляр класса `DetailViewController` и сохраняем его в виде свойства, так как планируем использовать его позже. Однако в оставшейся части шаблонного кода это свойство не используется.

Метод `viewDidLoad()` также добавляет кнопку на панель инструментов. Это кнопка `+`, которая показана в правой части панели навигации контроллера главного представления на рис. 11.4 и 11.5. Шаблонное приложение использует эту кнопку для создания и добавления новой записи к табличному представлению контроллера главного представления. Поскольку в нашем “президентском” приложении эта кнопка не нужна, этот код вскоре будет удален.

Имеется еще несколько методов, включенных в шаблон для этого класса, но не будем беспокоиться о них сейчас. Мы собираемся удалить одни из них и переписать другие, но только после того, как познакомимся с контроллером детализированного представления.

Контроллер детализированного представления

Последним классом, созданным для нас средой Xcode, является класс `DetailViewController`, который отвечает за отображение элемента, выбранного пользователем. Приведем содержимое файла `DetailViewController.swift` (листинг 11.3).

Листинг 11.3. Файл `DetailViewController.swift`

```
import UIKit
class DetailViewController: UIViewController {
    @IBOutlet weak var detailDescriptionLabel: UILabel!

    func configureView() {
        // Обновление пользовательского интерфейса
        // для элемента детализированного представления
        if let detail = self.detailItem {
            if let label = self.detailDescriptionLabel {
                label.text = detail.description
            }
        }
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Дополнительная настройка после загрузки представления,
        // обычно из nib-файла.
        self.configureView()
    }
}
```

412 ГЛАВА 11 ■ РАЗДЕЛЕННЫЕ ПРЕДСТАВЛЕНИЯ И ВСПЛЫВАЮЩИЕ МЕНЮ

```
override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Освобождение любых возобновляемых ресурсов.
}

var detailItem: NSDate? {
    didSet {
        // Обновление представления.
        self.configureView()
    }
}
}
```

Метод `detailDescriptionLabel` представляет собой выход, который подключен к метке в раскладовке. В шаблоне приложения метка просто выводит описание объекта в свойстве `detailItem`. Само свойство `detailItem` предназначено для хранения контроллером представления ссылки на объект, который пользователь выбрал в контроллере главного представления. Его наблюдатель свойств (код в блоке `didSet`), который вызывается после того, как его значение было изменено, вызывает еще один созданный для нас метод — `configureView()`. Все, что он делает, — это вызывает метод описания объекта с детализированной информацией и использует его результат для настройки свойства `text` метки в раскладовке:

```
func configureView() {
    // Обновление пользовательского интерфейса
    // для элемента детализированного представления
    if let detail = self.detailItem {
        if let label = self.detailDescriptionLabel {
            label.text = detail.description
        }
    }
}
```

Метод `description` реализован в каждом подклассе `NSObject`. Если ваш класс не переопределяет его, то он возвращает (вероятно, не очень полезное) значение по умолчанию. Однако в нашем примере все объекты с детализированной информацией являются экземплярами класса `NSDate`, а реализация метода описания в классе `NSDate` возвращает дату и время, сформированные в обобщенном виде.

Как работает приложение master-detail

Итак, вы видели все части шаблонного приложения, но, вероятно, все еще не очень ясно представляете себе, как оно работает. Так что давайте запустим его и посмотрим, что оно собой представляет. Запустите приложение на симулятор устройства iPad и поверните устройство в альбомную ориентацию, чтобы появился контроллер главного представления. Вы можете увидеть, что метки в контроллере детализированного представления в настоящее время имеют текст по умолчанию, присвоенный им в раскладовке. В этом разделе нас интересует,

как выбор элемента в контроллере главного представления приводит к изменению этого текста. В настоящее время в контроллере главного представления нет элементов. Чтобы исправить ситуацию, нажмите несколько раз кнопку + в верхней части панели навигации. Каждый раз, после того, как вы это сделаете, в табличное представление контроллера добавляется новый элемент (рис. 11.7).

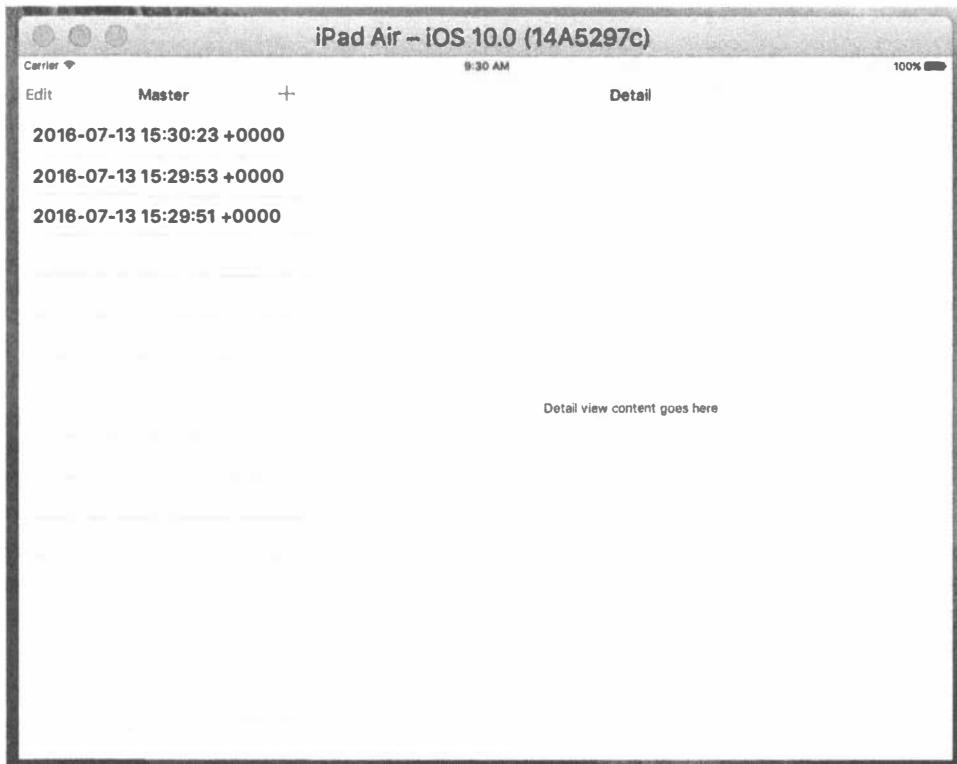


Рис. 11.7. Шаблонное приложение с выбранным в контроллере главного представления элементом и информацией в контроллере детализированного представления

Все элементы в таблице контроллера главного представления являются данными. Выберите один из них, и метка в детализированном представлении обновится, показывая ту же дату. Вы уже видели метод `configureView` в файле `DetailViewController.swift`, который вызывается, когда новое значение сохраняется в свойстве `detailItem` контроллера детализированного представления. Что же заставляет устанавливать новое значение свойства? Взгляните еще раз на раскладовку на рис. 11.6. Имеется переход, который связывает табличную ячейку-прототип в ячейке таблицы контроллера главного представления с контроллером детализированного представления. Если вы щелкнете на этом переходе и откроете инспектор атрибутов, то увидите, что это переход `Show Detail` с идентификатором `showDetail` (рис. 11.8).

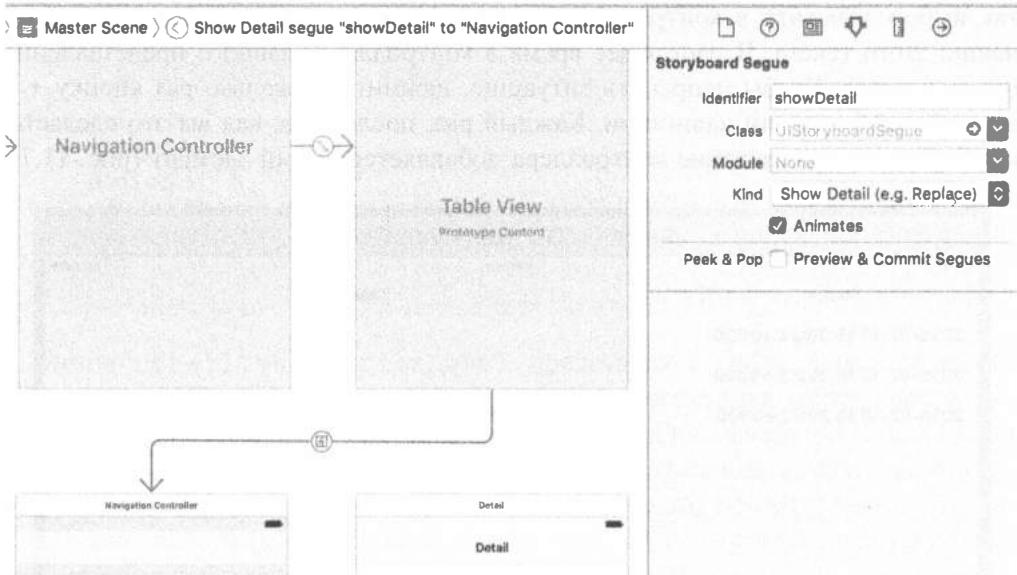


Рис. 11.8. Переход Show Detail связывает контроллеры главного и детализированного представлений

Как было показано в главе 9, при выборе ячейки табличного представления срабатывает связанный с ней переход. Следовательно, при выборе строки в табличном представлении контроллер главного представления система iOS выполнит переход Show Detail. При этом контроллер навигации служит “оберткой” для контроллера детализированного представления, являющегося целью перехода. Это приводит к выполнению двух действий.

- ※ Создается новый экземпляр контроллера детализированного представления, и его представление добавляется в иерархию представлений.
- ※ Вызывается метод `prepareForSegue(_:sender:)` в контроллере главного представления.

Первый шаг гарантирует видимость контроллера детализированного представления. На втором шаге контроллер главного представления должен вывести объект, выбранный в контроллере главного представления тем или иным способом. Вот как это делает код шаблона в файле `MasterViewController.swift` (листинг 11.4).

Листинг 11.4. Метод `prepare(forSegue:)` в файле `MasterViewController.swift`

```
// MARK: - Segues

override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showDetail" {
        if let indexPath = self.tableView.indexPathForSelectedRow {
            let object = objects[indexPath.row] as! NSDate
```

```
        let controller = ( segue.destinationViewController as!
                            UINavigationController).
                                topViewController as! DetailViewController
                controller.detailItem = object
                controller.navigationItem.leftBarButtonItem =
                    self.splitViewController?.
                        displayModeButtonItem()
                controller.navigationItem.leftItemsSupplementBackButton = true
            }
        }
    }
}
```

Сначала выполняется проверка идентификатора перехода, чтобы убедиться, что это тот переход, который ожидался, и что получен объект NSDate из выбранного объекта таблицы контроллера представления. Далее контроллер главного представления находит экземпляр DetailViewController из свойства topViewController целевого контроллера представления перехода, приведшего к вызову этого метода. Теперь у нас есть как выбранный объект, так и контроллер детализированного представления, и все, что мы должны сделать, — это установить свойство detailItem контроллера детализированного представления, чтобы обновить детализированное представление. Последние две строки метода prepare(forSegue:) добавляют кнопку режима отображения на панель навигации контроллера детализированного представления. Когда устройство находится в альбомной ориентации, ничего не происходит, поскольку кнопка режима отображения невидима, но если вы повернете устройство в книжную ориентацию, то увидите, как появится кнопка Master.

Так что теперь вы знаете, как выбранный в контроллере главного представления элемент отображается в контроллере детализированного представления. Хотя внешне это выглядит не слишком впечатляющим, на самом деле при этом выполняется много действий за сценой, обеспечивающих корректную работу как на iPad, так и на iPhone, как в книжной, так и в альбомной ориентации. Преимущество контроллера разделенного представления в том, что он заботится обо всех деталях и не требует от вас задумываться о реализации своих контроллеров главного и детализированного представлений.

На этом мы закончим обзор того, что нам дает шаблон Master-Detail Application в среде Xcode. На первый взгляд, это слишком большой кусок информации, чтобы справиться с ним с первого раза, но в идеале разделение приложения на части поможет вам понять, как эти части взаимодействуют и в результате дают единое приложение.

Добавление данных о президентах

Теперь, когда мы разобрались в базовой структуре рассматриваемого приложения, самое время заполнить пробелы и превратить этот автоматически сгенерированный “каркас” в авторский продукт. Начнем с архива исходного кода для нашей книги: папка 11 – Presidents Data должна содержать файл PresidentList.plist. Перетащите этот файл в папку Presidents своего

проекта в среде Xcode, чтобы добавить его в проект. При этом убедитесь, что флажок, предлагающий среде Xcode скопировать файл, установлен. Этот файл .plist содержит информацию обо всех президентах США до настоящего времени, которая включает только имя и URL статьи из Википедии (Wikipedia) для каждого из них.

Рассмотрим контроллер главного представления и подумаем, как нужно модифицировать его для надлежащей обработки данных о президентах. Должно быть достаточно лишь загрузить список президентов, представив его в виде таблицы, и передать URL в детализированное представление для отображения. Начнем с добавления в файл MasterViewController.swift строки, выделенной в следующем коде полужирным шрифтом, и удаления зачеркнутой строки:

```
class MasterViewController: UITableViewController {
    var detailViewController: DetailViewController? = nil
    var objects = [AnyObject]()
    var presidents: [[String: String]]!
```

Обратите внимание на метод viewDidLoad(), в котором изменения носят немного более сложный характер (но до сих пор все шло не так уж плохо). Вам придется добавить несколько строк, чтобы загрузить список президентов, а затем удалить несколько строк, которые создают кнопки редактирования и вставки на панели инструментов, как показано в листинге 11.5.

Листинг 11.5. Метод viewDidLoad в файле MasterViewController.swift

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // как правило, из nib-файла.
    let path = Bundle.main.path(forResource: "PresidentList",
                                ofType: "plist")
    let presidentInfo = NSDictionary(contentsOfFile: path)!
    presidents = presidentInfo["presidents"]! as! [[String: String]]

    if let split = self.splitViewController {
        let controllers = split.viewControllers
        self.detailViewController = (controllers[controllers.count-1] as!
                                    UINavigationController).topViewController as?
        DetailViewController
    }
}
```

На первый взгляд, этот код может показаться несколько запутанным:

```
let path = Bundle.main.path(forResource:"PresidentList", ofType: "plist")!
let presidentInfo = NSDictionary(contentsOfFile: path)!
presidents = presidentInfo["presidents"]! as! [[String: String]]
```

Метод pathForResource(_:ofType:) из файла Bundle.main получает путь к файлу PresidentList.plist, содержание которого затем загружается в NSDictionary. Этот словарь содержит одну запись с ключом "presidents".

Значением этой записи является массив, в котором каждый элемент представляет собой словарь класса `NSDictionary`. Массив имеет по одному словарю для каждого президента; этот словарь содержит пары “ключ–значение”, где и ключ, и значение являются строками. Мы приводим массив кциальному типу языка `Swift` `[[String: String]]` и присваиваем переменной `presidents`.

Этот созданный шаблоном класс включает также метод `insertNewObject()` для добавления элементов в массив объектов. У нас больше нет этого массива, так что удаляем весь метод.

Кроме того, у нас есть пара методов источника данных, которые позволяют пользователям редактировать строки в табличном представлении. Мы не собираемся разрешать какие-либо изменения строк в данном приложении, так что просто удалим методы `canEditRowAt` и `commitEditingStyle`.

Теперь самое время адаптировать методы источника данных главного табличного представления для наших целей. Начнем с изменения метода, который сообщает табличному представлению, сколько строк следует выводить.

```
override func tableView(_ tableView: UITableView,
                      numberOfRowsInSection section: Int) -> Int {
    return presidents.count
}
```

После этого отредактируем метод `tableView(:cellForRowIndexPath:)`, чтобы сделать каждую ячейку отображающей имя президента:

```
override func tableView(_ tableView: UITableView,
                      cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: "Cell", for: indexPath)

    let president = presidents[indexPath.row]
    cell.textLabel!.text = president["name"]
    return cell
}
```

Наконец отредактируем метод `prepare(forSegue:)` для передачи данных для выбранного президента (которые, как было сказано ранее, представляют собой словарь типа `[String:String]`) контроллеру детализированного представления (листинг 11.6).

Листинг 11.6. Метод `prepare(forSegue:)`

```
// MARK: - Segues

override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showDetail" {
        if let indexPath = self.tableView.indexPathForSelectedRow {
            let object = presidents[indexPath.row]
            let controller = (segue.destinationViewController
                as! UINavigationController).topViewController as!
```

418 ГЛАВА 11 ■ РАЗДЕЛЕННЫЕ ПРЕДСТАВЛЕНИЯ И ВСПЛЫВАЮЩИЕ МЕНЮ

```
DetailViewController
    controller.detailItem = object
    controller.navigationItem.leftBarButtonItem =
        self.splitViewController?.displayModeButtonItem()
    controller.navigationItem.leftItemsSupplementBackButton = true
}
}
}
```

ЗАМЕЧАНИЕ. Если переменная `detailItem` в шаблонном файле `DetailView Controller.swift` имеет значение `NSDate` или подобное ему, то из его текста необходимо убрать строку '`AnyObject?`', чтобы избежать потенциальных ошибок.

Это все, что нам нужно сделать в контроллере главного представления.

Далее выберите файл `Main.storyboard` и щелкните на пиктограмме **Master** в разделе **Master Scene** окна **Document Outline**, чтобы выбрать контроллер главного представления, а затем дважды щелкните на его заголовке, замените **Master** на **Presidents** и сохраните раскладовку.

Теперь можно построить и запустить приложение. Переключитесь в альбомную ориентацию или щелкните на кнопке **Master** в верхнем левом углу, чтобы вызвать контроллер главного представления, показывающий список президентов (рис. 11.9). Щелкните на имени президента для отображения в детализированном представлении пока что не слишком полезной строки.

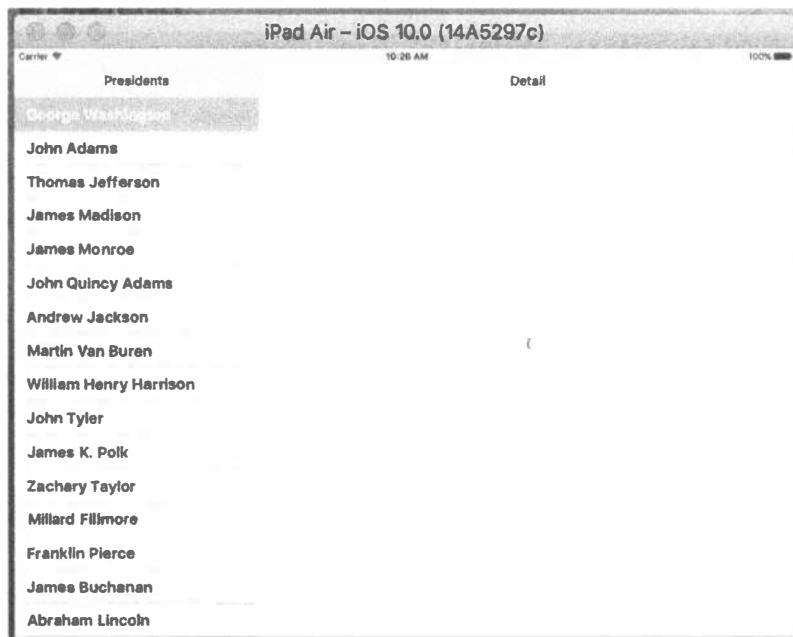


Рис. 11.9. Первый запуск нашего приложения, демонстрирующий список президентов в контроллере главного представления, но пока нет ничего полезного в контроллере детализированного представления

Закончим наш пример, заставляя детализированное представление делать с передаваемыми данными что-то немного более полезное. Добавьте следующую выделенную полужирным шрифтом строку в файл `DetailViewController.swift`, чтобы создать выход для веб-представления, отображающего страницу Википедии для выбранного президента:

```
class DetailViewController: UIViewController {
    @IBOutlet weak var detailDescriptionLabel: UILabel!
    @IBOutlet weak var webView: UIWebView!
```

Прокрутите файл до метода `configureView()` и замените его следующим кодом (листинг 11.7).

Листинг 11.7. Метод `configureView`

```
func configureView() {
    // Обновляем пользовательский интерфейс для элемента
    // детализированного представления
    if let detail = self.detailItem {
        if let label = self.detailDescriptionLabel {
            let dict = detail as! [String: String]
            let urlString = dict["url"]!
            label.text = urlString

            let url = NSURL(string: urlString)!
            let request = URLRequest(url: url as URL)
            webView.loadRequest(request)
            let name = dict["name"]!
            title = name
        }
    }
}
```

Свойство `detailItem`, которое было установлено контроллером главного представления, — это словарь, содержащий две пары “ключ–значение”: одну с именем ключа, который хранит имя президента, а другую — с адресом, который указывает URL страницы Википедии президента. Мы используем URL-адрес, чтобы задать текст метки детализированного описания и создать `NSURLRequest`, который будет использоваться `UIWebView` для загрузки страницы. Для установки заголовка контроллера детализированного представления мы используем имя президента. Когда контроллер представления представляет собой контейнер в классе `UINavigationController`, значение его свойства `title` отображается на панели навигации контроллера навигации. Это все, что нам нужно, чтобы наше веб-представление загрузило запрошенную страницу.

Последние изменения коснутся файла `Main.storyboard`. Откроем его для редактирования и найдем детализированное представление справа внизу. Сначала позаботимся о метке в графическом интерфейсе (текст которой имеет вид “*Detail view content goes here*” (Здесь находится содержимое детализированного представления)). Сначала выберите метку. Возможно, проще всего найти метку

в разделе Detail Scene окна Document Outline. Выбрав метку, перетащите ее в верхнюю часть окна. Обратите внимание на то, что метка должна привести в движение голубую линию разметки (слева направо) и удобно разместиться под панелью навигации (чтобы добиться этого, вы можете изменить ее размер). Назначение этой метки — отображать текущий URL. Но при запуске приложения, еще до того, как пользователь выберет президента, хотелось бы, чтобы в этом текстовом поле содержалась подсказка о том, что должен делать пользователь.

Дважды щелкните на метке и измените ее значение Select на President. Вы должны также использовать инспектор размеров, чтобы убедиться, что метка прикреплена к левому и правому краям, а также к верхнему краю и что она сможет менять свои размеры в горизонтальном направлении при переходе из альбомной ориентации в книжную или обратно (рис. 11.10). Если необходимо настроить эти ограничения, воспользуйтесь описанным ранее методом. Вероятно, вы получите то, что хотите, выбрав метку, а затем выполнив команду Editor⇒Resolve Auto Layout Issues⇒Reset to Suggested Constraints.

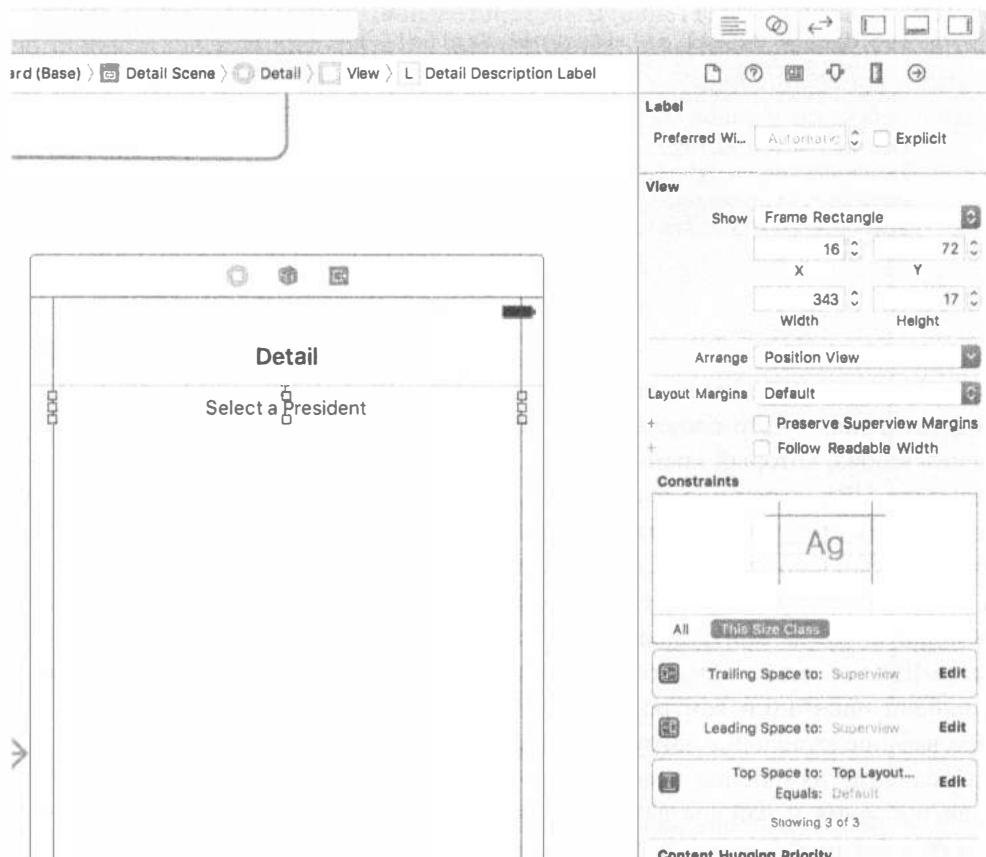


Рис. 11.10. Инспектор размеров, демонстрирующий ограничения, наложенные на метку Select a President в нижней части экрана

Затем используйте библиотеку, чтобы найти элемент типа UIWebView, и перетащите его под надпись (которую мы только что “поставили на место”). С помощью манипуляторов размеров заставьте этот элемент заполнить оставшую часть представления под надписью: от левого края до правого и от синей направляющей линии (как раз под нижней границей надписи) до нижней границы окна. Используя инспектор размеров, прикрепите наше веб-представление ко всем четырем краям окна и разрешите ему изменять размеры как по горизонтали, так и по вертикали (рис. 11.11). Как и ранее, вероятно, вы получите то, что хотите, выбрав метку, а затем выполнив команду Editor⇒Resolve Auto Layout Issues⇒Reset to Suggested Constraints.

Выберите в окне Document Outline контроллер представления Master и откройте окно инспектора атрибутов. В разделе View Controller измените значение поля Title с Master на Presidents. Таким образом, название кнопки навигации в верхней части контроллера детализированного представления станет более полезным.

Осталось выполнить еще один шаг. Для того чтобы подключить выход только что созданного веб-представления, нажмите клавишу <Control> и перетащите указатель от пиктограммы Detail (в разделе Detail Scene окна Document Outline) к новому веб-представлению (в том же разделе прямо под меткой в окне Document Outline или в раскладовке) и подключите выход webView. Сохраните изменения.

Теперь можете скомпоновать и запустить приложение, которое позволит просматривать страницы Википедии для каждого президента США (см. рис. 11.11). Поверните дисплей, чтобы поменять ориентацию изображения, и увидите, как контроллер разделенного представления обрабатывает вместо вас изменение ситуации при небольшой помощи контроллера детализированного представления.

Создание пользовательского всплывающего меню

В главе 4 вы видели, что можно вывести список действий в том виде, в котором в мультфильмах передают речь персонажей (см. рис. 4.29). Это визуальное представление контроллера всплывающего меню, или просто всплывающее меню (popover) для краткости. Всплывающее меню, которое вы получаете из списка действий, создается для вас, если список действий представлен классом

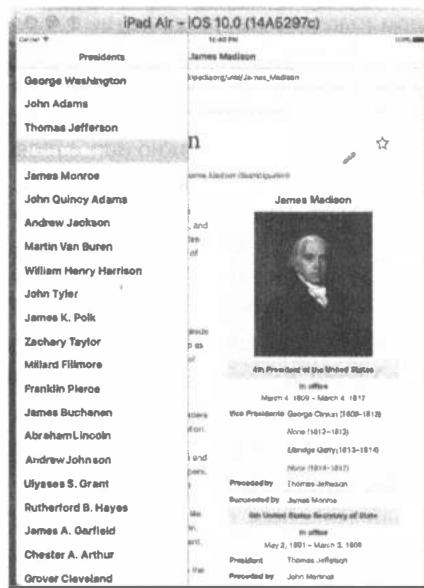


Рис. 11.11. Приложение Presidents, демонстрирующее страницу Википедии, посвященную Джеймсу Мэдисону (James Madison)

`UIPopoverPresentationController`, над которым у вас имеется очень небольшой контроль. Оказывается, что с помощью класса `UIPopoverController` можно создать собственное всплывающее меню.

Для того чтобы узнать, как это можно сделать, попробуем добавить меню, которое будет активизироваться постоянно присутствующим на панели инструментов элементом (а не элементом класса `UISplitView`). Наше всплывающее меню должно отображать табличное представление, содержащее список языков. При выборе пользователем нового языка из списка веб-представление загрузит уже отображаемую страницу Википедии на заданном языке. Это несложная задача, поскольку для перехода с одного языка на другой в Википедии достаточно изменить небольшую часть URL-адреса, содержащего встроенный код страны. На рис. 11.3 показано, к чему мы стремимся. Одно важно заметить, что класс `UIPopoverController` доступен только на устройстве iPad, так что при работе приложения на устройстве iPhone выбор языка будет отсутствовать.

ЗАМЕЧАНИЕ. В этом примере мы используем класс `UIPopoverPresentationController` меню для демонстрации таблицы, но это не должно вводить вас в заблуждение: его можно использовать для обработки вывода любого содержимого контроллера представления! Почему мы используем в этом примере именно табличные представления? Во-первых, потому, что это самый распространенный случай, во-вторых, это просто сделать с помощью небольшого по объему кода и, в-третьих, этот вариант представления вам уже достаточно хорошо знаком.

Итак, начнем. Щелкните правой кнопкой мыши на папке `Presidents` в среде Xcode и выберите в контекстном меню команду `New File...`. В появившемся окне выберите пункт `Cocoa Touch Class` из раздела `iOS Source`, а затем щелкните на кнопке `Next`. На следующей странице экрана назовите новый класс `LanguageListController` и выберите пункт `UITableViewController` в поле `Subclass of`. Щелкните на кнопке `Next`, дважды щелкните на каталоге, в котором сохраняете файл, и щелкните на кнопке `Create`.

Наш класс `LanguageListController` — стандартный класс контроллера табличного представления. Его назначение — отображать список элементов и уведомлять контроллер детализированного представления (с помощью соответствующего указателя) о том, что выбор сделан. Отредактируйте файл `LanguageListController.swift`, добавив в него следующие выделенные полужирным шрифтом строки:

```
class LanguageListController: UITableViewController {
    weak var detailViewController: DetailViewController? = nil
    private let languageNames: [String] =
        ["English", "French", "German", "Spanish"]
    private let languageCodes: [String] = ["en", "fr", "de", "es"]
```

Эти добавления определяют указатель на контроллер детализированного представления (который мы установим из кода в самом контроллере

детализированного представления, когда будем готовы отображать список языков), а также два массива для хранения отображаемых значений (“English”, “French” и т.д.) и тех значений, которые будут использованы для создания URL, взятых из выбранного языка (“en”, “fr” и т.д.).

Если бы вы скопировали этот код из архива исходного кода данной книги (или электронной книги) и вставили в свой проект или ввели его сами, то, скорее всего, не заметили бы важное отличие объявления свойства `detailViewController` от его предыдущего варианта. В отличие от большинства свойств, которые ссылаются на объектный указатель, мы объявили это свойство с атрибутом `weak`, а не `strong`. Это вызвано необходимостью избежать **цикла удержания** (*retain cycle*).

Что такое цикл удержания? Это ситуация, при которой несколько объектов (два или больше) циклически ссылаются друг на друга. Каждый из объектов не позволяет освободить память другого объекта. Большинства потенциальных циклов удержания можно избежать, если внимательно подходить к созданию своих объектов, стараясь выяснить, “кто кем владеет”. В этом смысле экземпляр контроллера типа `DetailViewController` “владеет” экземпляром контроллера типа `LanguageListController`, поскольку в действительности именно контроллер `DetailViewController` создает экземпляр типа `LanguageListController`, чтобы обеспечить выполнение некоторой порции работы. Когда у вас есть пара объектов, каждому из которых нужно ссылаться на другой, вполне естественно, что вы захотите, чтобы объект-“владелец” удерживал другой объект, в то время как этот другой объект не должен удерживать своего владельца. Поскольку мы используем функциональное свойство ARC, введенное Apple в Xcode 4.2, компилятор выполнит большую часть работы вместо нас. Для того чтобы не заниматься удалением объектов из памяти и их сохранением в памяти, мы должны всего лишь объявить свойство, ссылающееся на объект, которым мы не владеем, с использованием ключевого слова `weak` вместо `strong`. ARC сделает все остальное вместо нас.

Теперь перейдем к методу `viewDidLoad()` и добавим немного кода настройки.

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

    clearsSelectionOnViewWillAppear = false
    preferredContentSize = CGSize(width: 320,
        height: (languageCodes.count * 44))
    tableView.register(UITableViewCell.self,
        forCellReuseIdentifier: "Cell")
}
```

Здесь мы определяем размер, который будет использовать представление контроллера представления в случае его отображения во всплывающем меню. Без этого определения всплывающее меню будет растягиваться по вертикали, чтобы заполнить почти весь экран, даже если оно может разместиться в гораздо

меньшем представлении. В заключение зарегистрируем класс ячейки табличного представления, как было показано в главе 8.

Идем дальше. У нас есть два метода, сгенерированных Xcode-шаблоном, в которых вместо кода использован текст-заполнитель. Поэтому этот шаблонный текст необходимо заменить реальным кодом.

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}
override func tableView(_ tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return languageCodes.count
}
```

Теперь добавим метод `tableView(_:cellForRowIndexPath:)`; для получения объекта ячейки и размещения в ней названия языка (листинг 11.8).

Листинг 11.8. Ячейка для табличного представления

```
override func tableView(_ tableView: UITableView,
    cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(
        withIdentifier: "Cell",
        for: indexPath)
    // Настройка ячейки...
    cell.textLabel!.text = languageNames[indexPath.row]

    return cell
}
```

Далее реализуем `tableView(_:didSelectRowAt IndexPath:)` так, чтобы можно было ответить на нажатие пользователя передачей выбранного языка в контроллер детализированного представления.

```
override func tableView(_ tableView: UITableView,
    didSelectRowAt indexPath: IndexPath) {
    detailViewController?.languageString = languageCodes[indexPath.row]
    dismiss(animated: true, completion: nil)
}
```

ЗАМЕЧАНИЕ. Обратите внимание на то, что в классе `DetailViewController` свойство `languageString` не определено. Это вызовет ошибку компиляции. Скоро мы ее исправим.

Теперь пора модифицировать класс `DetailViewController`, чтобы он мог обрабатывать всплывающее меню и генерировать корректный URL, когда пользователь либо изменит язык отображения, либо выберет другого президента. Начнем с внесения следующих изменений в файл `DetailViewController.swift` ниже объявления класса `UIWebView`.

```
private var languageListController: LanguageListController?
private var languageButton: UIBarButtonItem?
var languageString = ""
```

Здесь мы добавили некоторые свойства для отслеживания компонентов графического интерфейса, необходимых для всплывающего меню и выбранного пользователем языка. Все, что нам осталось сделать, — это исправить файл `DetailViewController.swift`, чтобы обрабатывать всплывающее меню языка и создавать URL.

Для начала добавим функцию, которая принимает в качестве аргументов указатель URL на страницу Википедии и двухбуквенный код языка, а возвращает новое значение URL, в котором учтены все входные данные. Позже мы используем эту функцию в соответствующих местах нашего кода контроллера (листинг 11.9).

Листинг 11.9. Функция для создания специального URL с учетом выбранного языка

```
private func modifyUrlForLanguage(url: String, language lang: String?) -> String {
    var newUrl = url
    // Здесь мы полагаемся на конкретный формат URL Википедии.
    // Это не очень надежно!
    if let langStr = lang {
        // URL имеет вид https://en.wikipedia...
        let range = NSMakeRange(8, 2)
        if !langStr.isEmpty && (url as NSString).substring(with: range) != langStr {
            newUrl = (url as NSString).
                replacingCharacters(in: range,
                                   with: langStr)
        }
    }
    return newUrl
}
```

Наш следующий ход — обновление метода `configureView()`. Данный метод будет использовать функцию, которую мы только что определили, чтобы объединить переданную функции URL-строку с выбранным значением `languageString` и сгенерировать корректный URL-адрес, как показано в листинге 11.10.

Листинг 11.10. Настройка метода `configureView` на указатель URL с учетом выбранного языка

```
func configureView() {
    // Обновление пользовательского интерфейса
    // для элемента детализированного представления.
    if let detail = self.detailItem {
        if let label = self.detailDescriptionLabel {
            let dict = detail as! [String: String]
            // let urlString = dict["url"]!
```

```
        let urlString = modifyUrlForLanguage(url: dict["url"]!,  
                                              language: languageString)  
        label.text = urlString  
  
        let url = URL(string: urlString)!  
        let request = URLRequest(url: url )  
        webView.loadRequest(request)  
        let name = dict["name"]!  
        title = name  
    }  
}
```

Теперь необходимо обновить метод `viewDidLoad()`. Создадим объект класса `UIBarButtonItem` и поместим его в объект класса `UINavigationItem` в верхней части экрана, но только если мы работаем на устройстве iPad, как показано в листинге 11.11.

Листинг 11.11. Модифицированный метод viewDidLoad

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.
    self.configureView()
    languageButton = UIBarButtonItem(title: "Choose Language",
                                    style: .plain,
                                    target: self, action:
            #selector(DetailViewController.showLanguagePopover))
    navigationItem.rightBarButtonItem = languageButton
}
```

Далее реализуем наблюдателя свойств для свойства `languageString`, который вызывается, когда изменяется значение свойства. Наблюдатель свойства вызывает `configureView()`, так что URL-адрес может быть регенерирован (а новая страница загружена) немедленно, и убирает всплывающее меню выбора языка, если оно видимо:

```
var languageString = "" {  
    didSet {  
        if languageString != oldValue {  
            configureView()  
        }  
    }  
}
```

Теперь реализуем метод, который вызывается, когда пользователь нажимает кнопку `Choose Language`. В этом случае мы выводим на экран объект класса `LanguageListController`, создавая его при первом выводе. Затем мы настраиваем свойства его контроллера всплывающего представления. Поместите приведенный метод после метода `viewDidLoad()`, как показано в листинге 11.12.

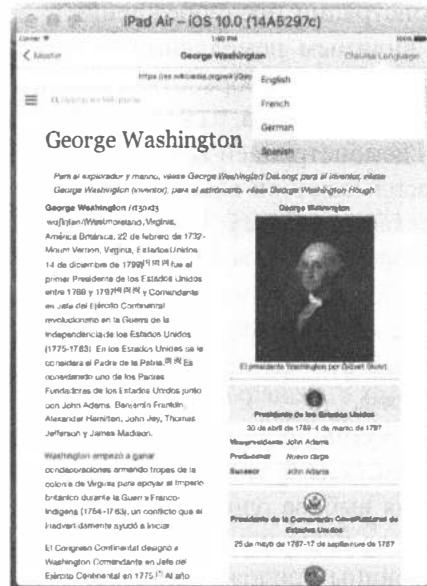


Рис. 11.12. Выбор языка при загрузке страницы

Листинг 11.12. Метод showLanguagePopover

```
func showLanguagePopover() {
    if languageListController == nil {
        // Отложенное создание при использовании в первый раз
        languageListController = LanguageListController()
        languageListController!.detailViewController = self
        languageListController!.modalPresentationStyle = .popover
    }
    present(languageListController!, animated: true, completion: nil)
    if let ppc = languageListController?.popoverPresentationController {
        ppc.barButtonItem = languageButton
    }
}
```

В первой части этого метода мы проверяем, существует ли объект класса `LanguageListController`. Если нет, мы создаем экземпляр этого класса и задаем его свойство `detailViewController` так, чтобы объект ссылался на самого себя. Мы также задаем его свойство `modalPresentationStyle` равным `.popover`. Это свойство определяет, как будет демонстрироваться контроллер. Существует несколько возможных значений, которые можно найти в документации, описывающей класс `UIViewController`. Не удивительно, что значение `.popover` используется, если контроллер должен демонстрироваться как всплывающий объект.

Затем мы используем метод `presentViewController(_:animated:completion:)` для того, чтобы объект класса `LanguageListController` был видимым,

как мы делали при демонстрации предупреждения в главе 4. Вызов этого метода не сделает объект видимым немедленно — библиотека UIKit выполнит эту функцию, когда будет завершена обработка события, связанного с нажатием кнопки, но она создаст объект класса UIPopoverPresentationController, который будет управлять демонстрацией всплывающего контроллера. Перед тем как на экране появится всплывающий объект, необходимо сообщить библиотеке UIKit, где именно он должен появиться. В главе 4 мы использовали этот прием для размещения всплывающего окна возле конкретного представления, задавая свойства sourceRect и sourceView класса UIPopoverPresentationController. В этом примере мы хотим, чтобы всплывающий объект появился возле кнопки выбора языка. Для этого мы присвоим ссылку на эту кнопку свойству контроллера barButtonItem.

Теперь запустите приложение на симуляторе iPad и нажмите кнопку Choose Language. Во всплывающем окне будет выведен контроллер списка языков, как показано на рис. 11.12. Для выбора одного из четырех доступных языков выберите одну из команд, которая позволит вывести на экран веб-представление для демонстрации версии страницы приложения President на выбранном языке.

Переключение с одного языка на другой не должно влиять на выбранного президента. Аналогично выбор другого президента не должно влиять на выбор языка, однако на деле это не так. Попробуйте: выберите президента, измените язык, например на испанский, а затем выберите другого президента. К сожалению, язык страницы больше не испанский.

Почему это происходит? При каждом вызове переход Show Detail создает новый экземпляр контроллера детализированного представления. Это значит, что выбор языка, который сохраняется как свойство контроллера детализированного представления, теряется каждый раз при выборе нового президента. Для того чтобы исправить этот недостаток, необходимо добавить несколько строк к код контроллера главного представления. Откройте файл MasterViewController.swift и внесите в метод prepare изменения, показанные в листинге 11.13.

Листинг 11.13. Модифицированный метод prepare

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showDetail" {
        if let indexPath = self.tableView.indexPathForSelectedRow {
            let object = presidents[indexPath.row]
            let controller = (segue.destinationViewController as!
                UINavigationController).topViewController as!
                DetailViewController
            if let oldController = detailViewController {
                controller.languageString = oldController.languageString
            }
            controller.detailItem = object
            controller.navigationItem.leftBarButtonItem =
                self.splitViewController?.displayModeButtonItem()
            controller.navigationItem.leftItemsSupplementBackButton = true
        }
    }
}
```

```
    detailViewController = controller
}
}
}
```

Резюме

В этой главе вы ознакомились с контроллером разделенного представления и его ролью в создании приложений Master-Detail. Вы также увидели, как может быть настроено полностью в пределах программы Interface Builder сложное приложение с несколькими взаимосвязанными контроллерами. Хотя разделенные представления теперь доступны на всех устройствах, пожалуй, все же наиболее полезны они в устройствах с большим экраном, а именно — в iPhone 6/6s Plus и iPad.

ГЛАВА 12



Настройки приложений и пользовательские настройки по умолчанию

Практически во всех современных компьютерных программах, за исключением, быть может, самых простых, предусмотрено окно, в котором пользователь может устанавливать глобальные параметры, отражающие особенности работы конкретного приложения. В операционной системе macOS практически каждое приложение имеет команду Preferences.... После ее выбора открывается окно, в котором пользователь может ввести значения различных параметров и изменить их. В iPhone и других мобильных устройствах под управлением системы iOS имеется специальное приложение Settings, которым вы, без сомнения, пользовались неоднократно. В этой главе будет показано, как добавить пользовательские настройки в стандартное приложение Settings и получить доступ к ним из своего приложения.

Знакомство с пакетом настроек

Приложение Settings дает пользователю возможность вводить и изменять глобальные параметры любого приложения, имеющего пакет настроек. *Пакет настроек* (setting bundle) представляет собой группу встроенных в приложение файлов, из которых приложение Settings может выяснить, какие именно глобальные параметры настройки данное приложение должно получить от пользователя (рис. 12.1).

В приложении iOS функции стандартных пользовательских настроек реализуют класс `NSUserDefaults`. Если вам приходилось программировать в среде Сосоа под управлением системы macOS, то вы, вероятно, уже знакомы с классом `NSUserDefaults`, поскольку именно этот класс используется для записи и считывания данных глобальных параметров настройки. В своих приложениях вы можете пользоваться классом `NSUserDefaults` для доступа к данным глобальных

параметров настройки с помощью пар ключей и значений аналогично доступу к данным по ключу в словаре. Отличие состоит лишь в том, что данные типа `NSUserDefaults` хранятся в файловой системе, а не в экземпляре объекта, находящегося в оперативной памяти. В этой главе мы разработаем приложение, сформируем пакет настроек, а затем отредактируем глобальные параметры настройки, обратившись к ним в приложении `Settings` непосредственно из данного приложения.

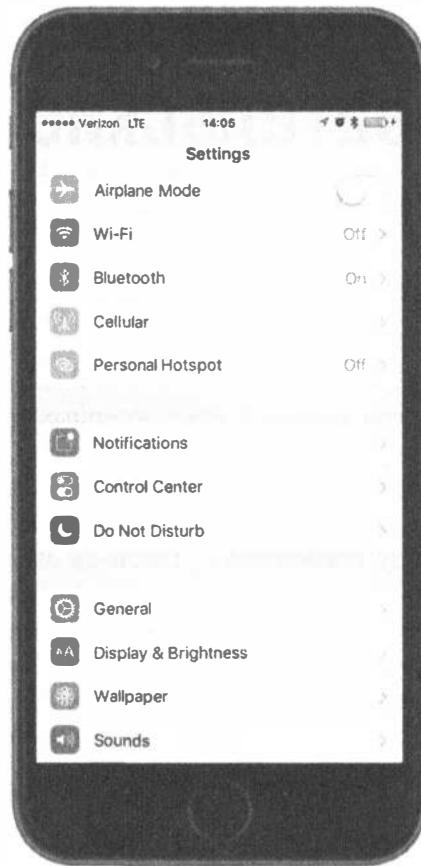


Рис. 12.1. Приложение `Settings` на типичном устройстве iPhone

Поскольку приложение `Settings` предоставляет стандартный интерфейс, у вас нет необходимости разрабатывать собственный пользовательский интерфейс для настройки глобальных параметров приложения. Нужно лишь составить список свойств, определяющих доступные параметры настройки приложения, а соответствующий интерфейс приложение `Settings` построит автоматически. В приложениях с эффектом присутствия, таких как игры, должно быть предусмотрено собственное представление глобальных параметров настройки, чтобы пользователю не нужно было выходить из приложения для внесения изменений. Даже

некоторые утилиты и приложения, повышающие производительность труда, могут иметь глобальные параметры настройки, к которым пользователь должен иметь доступ, а значит, и возможность изменить их, не выходя из приложения. В этой главе мы также покажем, как получить глобальные параметры настройки от пользователя непосредственно в своем приложении, а затем сохранить их в механизме User Defaults системы iOS.

Пользователь может перейти к приложению *Settings*, изменить значение параметра, а затем вернуться в свое приложение. В конце главы мы покажем, как это сделать.

Приложение Bridge Control

В этой главе нам предстоит написать простое приложение *Bridge Control*, отслеживающее некоторые особенности управления мостиком космического корабля. Сначала мы создадим пакет настроек, чтобы предоставить пользователю доступ к приложению *Bridge Control* из приложения *Settings* (рис. 12.2).

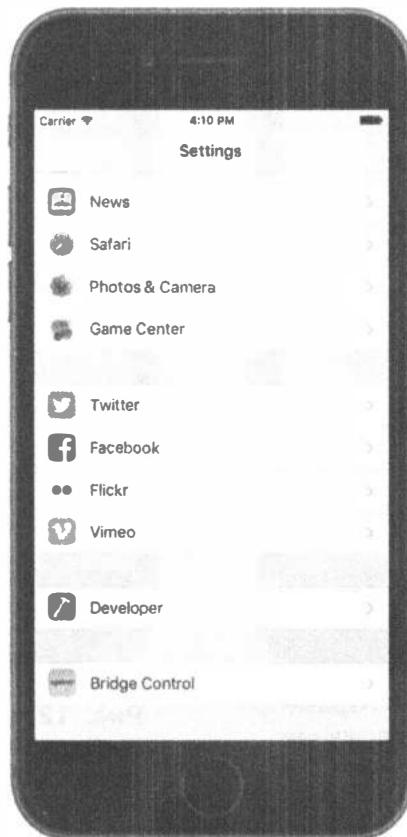


Рис. 12.2. Доступ к приложению *Bridge Control* из приложения *Settings* на симуляторе

Если пользователь выберет приложение Bridge Control, в приложении Settings откроется представление с глобальными параметрами настройки, относящимися к данному приложению. Для получения от пользователя значений этих параметров в приложении Settings используются поля редактирования, защищенные поля редактирования, переключатели и ползунки (рис. 12.3).

Обратите в этом представлении внимание на два элемента с индикаторами продолжения, раскрывающими следующее представление. Так, при выборе элемента Rank появится другое табличное представление, отображающее доступные для этого элемента параметры. Из этого табличного представления пользователь может выбрать только одно значение (рис. 12.4).

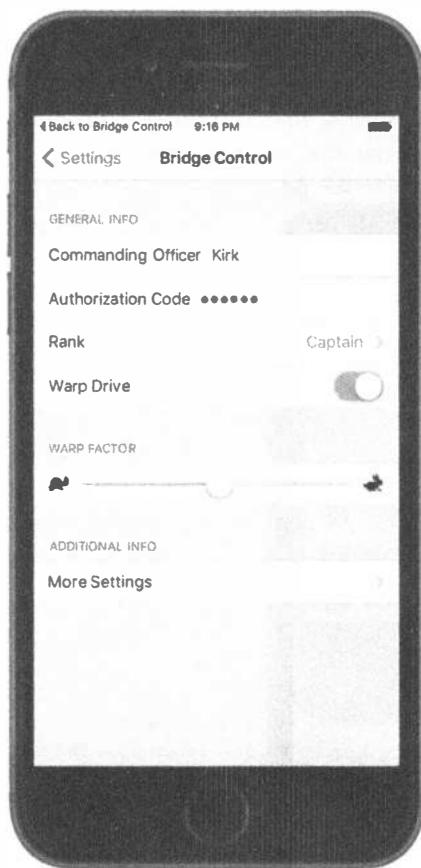


Рис. 12.3. Глобальные параметры настройки нашего приложения

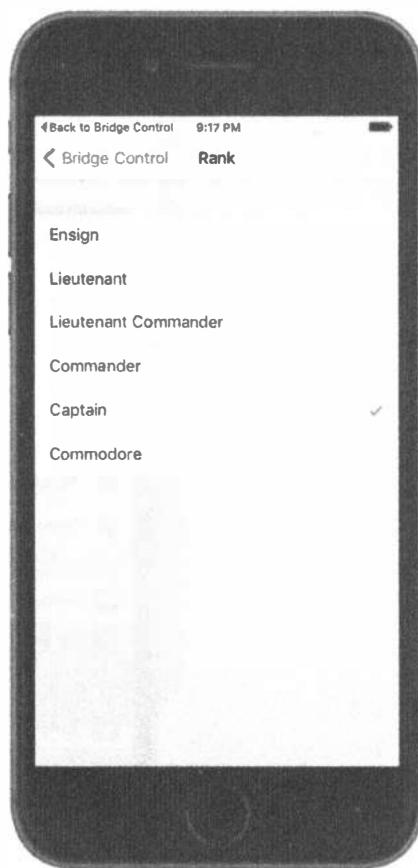


Рис. 12.4. Выбор одного отдельного параметра настройки из списка

Элемент с индикатором раскрытия More Settings дает пользователю возможность перейти еще к одному набору глобальных параметров настройки

(рис. 12.5). Это дочернее представление может иметь такие же элементы управления, как и родительское, а также собственное дочернее представление. В приложении *Settings* используется навигационный контроллер, поддерживающий построение иерархических представлений глобальных параметров настройки.

После запуска данного приложения пользователь увидит список глобальных параметров настройки, собранный в приложении *Settings* (рис. 12.6).



Рис. 12.5. Дочернее представление глобальных параметров настройки в приложении *Bridge Control*

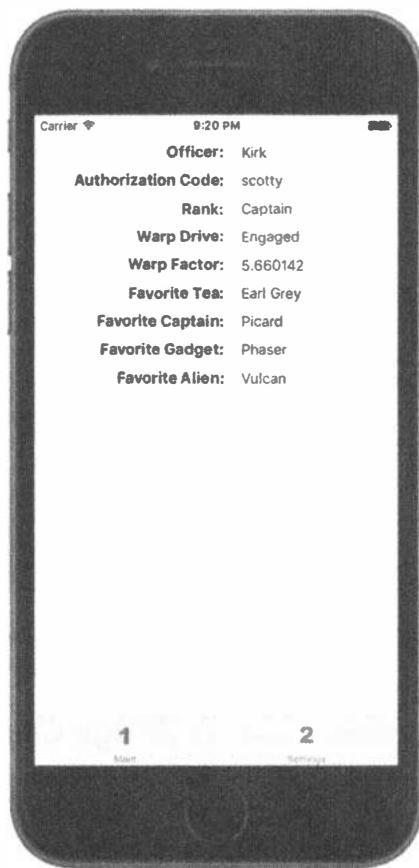


Рис. 12.6. Главное представление приложения *Bridge Control*

Для того чтобы продемонстрировать процесс обновления глобальных параметров настройки в данном приложении, мы предоставим второе представление, в котором пользователи смогут изменять дополнительные параметры настройки, не покидая приложение (рис. 12.7).

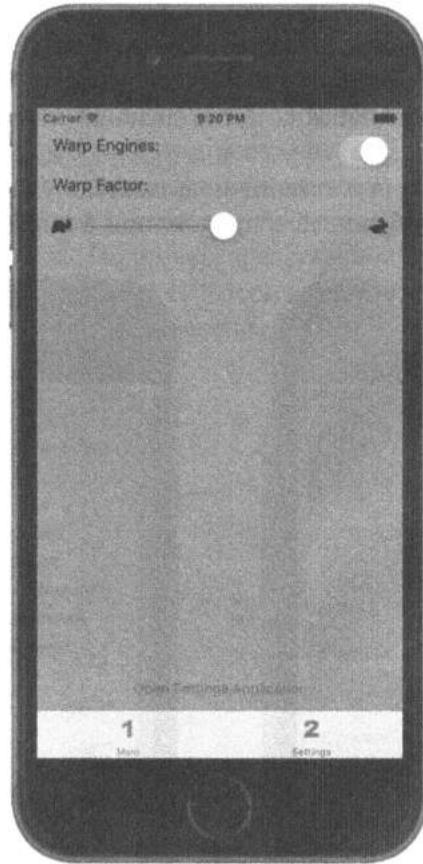


Рис. 12.7. Настройка некоторых глобальных параметров в самом приложении

Создание проекта Bridge Control

Находясь в среде Xcode, нажмите комбинацию клавиш `<Shift+⌘+N>` или выполните команду `File⇒New⇒Project....`. На левой панели в окне помощника нового проекта выберите элемент `Application` под заголовком `iOS`, щелкните на пиктограмме `Tabbed Application`, а затем на кнопке `Next`. В следующем окне присвойте новому проекту имя `Bridge Control` и выберите пункт `Universal` в списке `Devices`, а затем щелкните на кнопке `Next`. В заключение выберите место для хранения данного проекта и щелкните на кнопке `Create`.

Приложение `Bridge Control` разрабатывается на основе класса `UITabBarController`, упоминавшегося в главе 7. По выбранному шаблону создаются две вкладки. Каждую вкладку необходимо снабдить пиктограммой. Файлы изображений этих пиктограмм можно найти в папке 12 – `Images` с исходным кодом примеров, прилагаемым к этой книге. Выберите в среде Xcode папку `Assets.xcassets` и удалите из нее первое и второе изображения, добавленные шабло-

ном Xcode. Затем перетащите папки `singleicon.imageset` и `doubleicon.imageset` из папки 12 – `Images` в область редактирования, чтобы добавить новые изображения.

Далее нужно назначить пиктограммы для элементов вкладок. Выберите файл раскадровки `Main.storyboard`. На экране появятся контроллеры панели вкладок и два дочерних контроллера отдельных вкладок, обозначенные метками `First View` и `Second View`. Выберите первый контроллер, а затем щелкните на его панели вкладок, содержащей квадрат и заглавие `First`. Введите в поле `Title` из раздела `Bar Item` инспектора атрибутов строку `Main` и выберите значение поля `Image` равным `singleicon`, как показано на рис. 12.8. Затем выберите панель вкладок для второго дочернего контроллера, измените в поле `Title` строку `Second` на `Settings` и выберите значение поля `Image` равным `doubleicon`. Этого пока достаточно. Для того чтобы расширить свои возможности, в дальнейшем мы создадим пакет настроек.

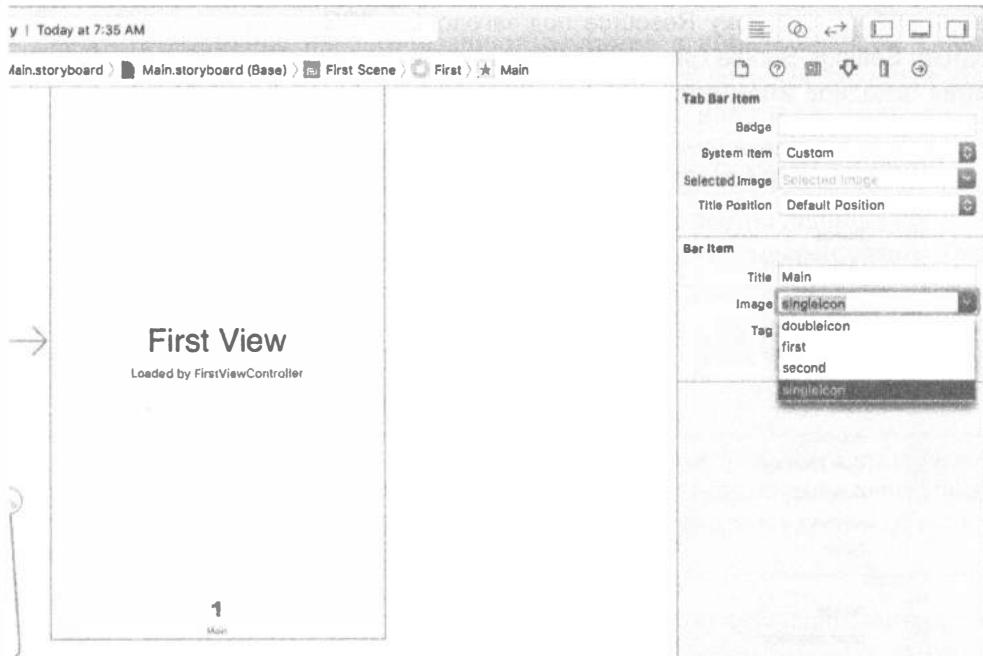


Рис. 12.8. Назначение пиктограммы для первого элемента панели вкладок

Подготовка пакета настроек

Содержимое пакета настроек каждого приложения используется в приложении `Settings` для построения представления, предназначенного для настройки отдельного приложения. Если же у приложения отсутствует пакет настроек, то последние в приложении `Settings` вообще не отображаются. У каждого пакета настроек должен быть список свойств, хранимых в файле `Root.plist`, который

определяет представление глобальных параметров настройки корневого уровня. Этот список свойств должен очень точно соответствовать формату, который мы обсудим при создании списка свойств для пакета настроек рассматриваемого здесь приложения.

После запуска приложения *Settings* каждое приложение проверяется в нем на наличие подготовленного пакета настроек, и если таковой существует, то добавляется группа параметров для отдельного приложения. Если требуется, чтобы представление глобальных параметров настройки включало в себя какие-нибудь дочерние представления, то в пакет настроек нужно ввести списки свойств и отдельную запись в файл *Root.plist* для каждого дочернего представления. Именно этим мы и займемся далее.

Ввод пакета настроек в проект

Щелкните на папке *Bridge Control* в окне навигатора проекта и выполните команду меню *File*⇒*New*⇒*File...* или нажмите комбинацию клавиш *<Shift+⌘+N>*. Затем выберите строку *Resource* под заголовком *iOS* слева, а справа — пиктограмму *Settings Bundle* (рис. 12.9). Щелкните на кнопке *Next*, оставьте без изменения исходное имя *Settings.bundle* и щелкните на кнопке *Create*.

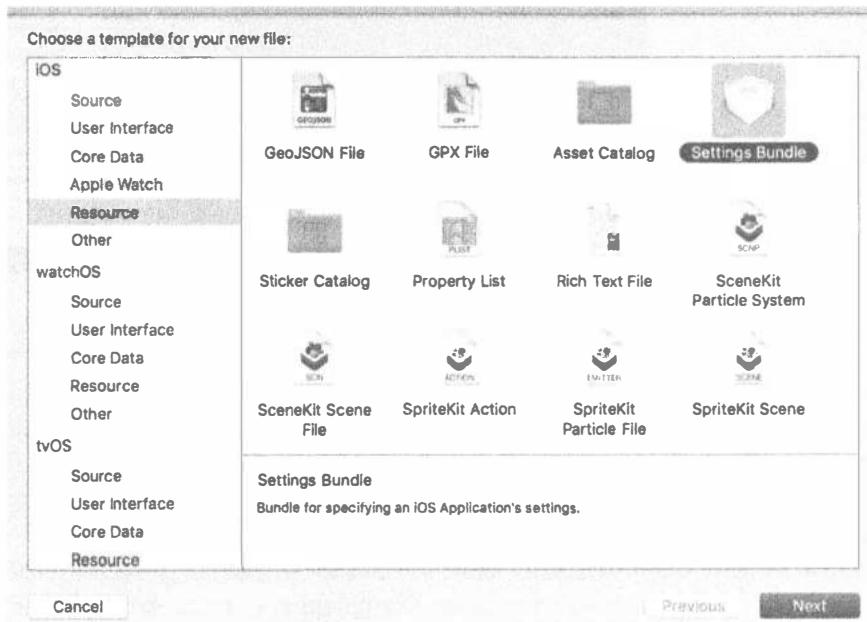


Рис. 12.9. Создание пакета настроек в среде Xcode

В окне проекта должен появиться новый элемент под именем *Settings.bundle*. Раскрыв его, вы увидите два элемента: папку *en.lproj*, содержащую файлы *Root.strings*, и файл *Root.plist*. Папку *en.lproj* мы рассмотрим позже, когда будем обсуждать локализацию приложения, т.е. перевода

их пользовательского интерфейса на другие языки. Здесь мы остановимся на файле Root.plist.

Выберите файл Root.plist и посмотрите на панель редактора. Вы увидите доступный в среде Xcode редактор списка свойств (рис. 12.10).

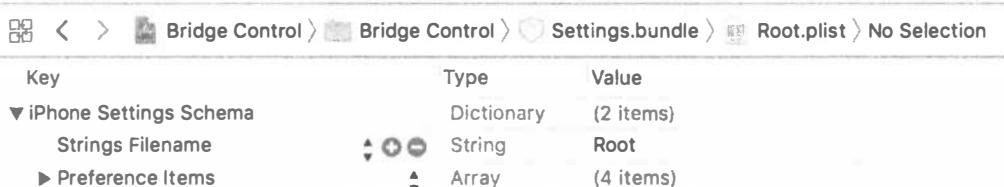


Рис. 12.10. Список из файла Root.plist в окне редактора списка свойств. Если вид окна редактора отличается от приведенного на этом рисунке, нажмите клавишу <Control> в этом окне и выполните команду Show Raw Keys/Values из контекстного меню

Обратите внимание на организацию элементов в списке свойств. Списки свойств, по существу, представляют собой словари, в которых хранятся типы и значения элементов списка, а для доступа к этим значениям используются ключи, подобно тому, как это делается в словаре типа Dictionary. В список свойств можно ввести разнотипные узлы. Узлы типа Boolean, Data, Date, Number и String служат для хранения отдельных элементов данных, но, помимо этого, имеется возможность оперировать целыми коллекциями узлов. Кроме узлов типа Dictionary, в которых можно хранить дополнительные словари, имеются узлы типа Array, в которых можно хранить упорядоченный список дополнительных узлов. Только узлы типа Dictionary и Array могут содержать дополнительные узлы в списке свойств.

ЗАМЕЧАНИЕ. Несмотря на то что в качестве ключей в словаре типа Dictionary можно использовать большинство типов объектов, ключами к словарю, образующему список свойств, должны служить только символьные строки. Однако для значений можно использовать любой тип узла.

Создавая список свойств для настроек, нужно строго соблюдать конкретный формат. К счастью, список свойств Root.plist, сопровождающий пакет настроек, только что добавленный в проект, точно соответствует этому формату. Рассмотрим его подробнее.

Имена ключей на панели редактора списка свойств Root.plist могут выводиться в исходном или более понятном для людей виде. Мы считаем, что вещи следует воспринимать такими, какими они являются, поэтому щелкните правой кнопкой мыши на любом участке данного редактора и установите флагок слева от пункта Show Raw Keys/Values всплывающего меню (рис. 12.11). В дальнейшем мы будем пользоваться настоящими именами всех ключей, поэтому очень важно установить данный режим работы.

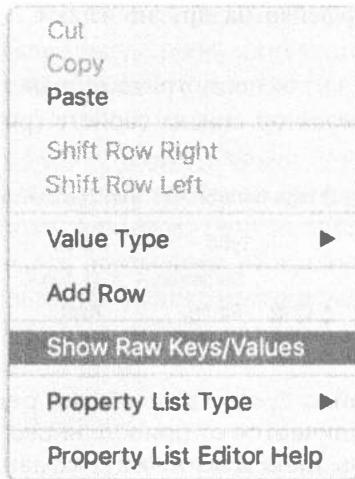


Рис. 12.11. Щелкните мышью на любом участке редактора списка свойств, удерживая нажатой клавишу <Control>. Установите флагок слева от пункта Show Raw Keys/Values. Благодаря этому в редакторе списка свойств будут использоваться настоящие имена ключей, обеспечивая более точное редактирование

ПРЕДУПРЕЖДЕНИЕ. Если выйти из списка свойств, чтобы отредактировать другой файл или покинуть среду Xcode, флагок слева от пункта Show Raw Keys/Values всплывающего меню будет сброшен. Если текст неожиданно изменился, проверьте меню и убедитесь, что данный флагок установлен.

Одним из элементов словаря является таблица символьных строк типа `StringsTable`, которая используется при переводе пользовательского интерфейса приложения на другой язык, как поясняется в главе 22, посвященной локализации приложений. В данном случае таблица символьных строк не применяется, поэтому ее можно удалить, щелкнув на ней и нажав клавишу <Delete>. Но ее можно и не удалять, поскольку она ничем не повредит. Кроме `StringsTable`, список свойств содержит узел `PreferenceSpecifiers`. Он представляет собой массив, предназначенный для хранения ряда узлов словаря, где каждый узел представляет один параметр, который может ввести пользователь, или одно дочернее представление, к которому может перейти пользователь.

Щелкните на треугольнике раскрытия слева от элемента `PreferenceSpecifiers`, чтобы раскрыть этот узел. Как видите, шаблон Xcode любезно предоставил нам четыре дочерних узла (рис. 12.12). Эти узлы не отражают параметры настройки, необходимые в данном примере, поэтому удалите элементы `Item 1`, `Item 2` и `Item 3`, щелкнув по очереди на каждом из них и нажав клавишу <Delete>, но оставив на месте элемент `Item 0`.

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	(4 items)
► Item 0 (Group - Group)	Dictionary	(2 items)
► Item 1 (Text Field - Name)	Dictionary	(8 items)
► Item 2 (Toggle Switch - Enabled)	Dictionary	(4 items)
► Item 3 (Slider)	Dictionary	(7 items)

Рис. 12.12. Вид списка Root.plist на панели редактора списка свойств — на этот раз с раскрытым элементом PreferenceSpecifiers

ЗАМЕЧАНИЕ. Чтобы выбрать элемент из списка свойств, лучше всего щелкнуть на одной или другой стороне столбца Key и тем самым избежать появления меню, всплывающего в этом столбце.

Щелкните на элементе Item 0, но не раскрывайте его. Редактор списка свойств в среде Xcode позволяет добавлять строки нажатием клавиши <Return>. Место вставки новой строки определяется тем, какая строка выбрана и раскрыта ли она. Так, если выбран нераскрытый массив или словарь, то нажатием клавиши <Return> ниже выбранной строки добавляется еще один узел на том же самом уровне иерархии. Если вы нажмете клавишу <Return> (пока что не делайте этого), ниже элемента Item 0 сразу же появится новая строка с элементом Item 1. На рис. 12.13 приведен пример вставки новой строки нажатием клавиши <Return>. Обратите внимание на выпадающее меню, где можно выбрать вид спецификатора глобальных параметров настройки, которые представляет данный элемент, как поясняется далее.

Раскройте элемент Item 0 и посмотрите на его содержимое (рис. 12.14). Редактор готов добавить дочерние узлы к выбранному элементу. Если вы нажмете клавишу <Return> (пока что не делайте этого), ниже элемента Item 0 сразу же появится новая строка с первым дочерним элементом.

Один из элементов из иерархии элемента Item 0 имеет ключ Type. Каждый узел списка свойств в массиве PreferenceSpecifiers должен иметь элемент с таким ключом. Ключ Type сообщает приложению Settings тип данных, связанный с элементом. Ключ Type в элементе Item 0 имеет значение PSGroupSpecifier. Это означает, что рассматриваемый элемент представляет начало новой группы. Каждый последующий элемент будет оставаться частью данной группы вплоть до очередного ключа Type со значением PSGroupSpecifier. Если вернуться к рис. 12.13, то можно заметить, что в приложении Settings настройки приложения представлены в виде сгруппированной таблицы. Элемент Item 0 из массива PreferenceSpecifiers списка свойств, подготавливаемого для пакета

442 ГЛАВА 12 ■ НАСТРОЙКИ ПРИЛОЖЕНИЙ И ПОЛЬЗОВАТЕЛЬСКИЕ НАСТРОЙКИ ПО УМОЛЧАНИЮ

настроек, должен всегда иметь значение `PSGroupSpecifier` своего ключа `Type`, чтобы настройки начинались с новой группы. Это важно потому, что в каждой таблице настроек должна быть хотя бы одна группа.

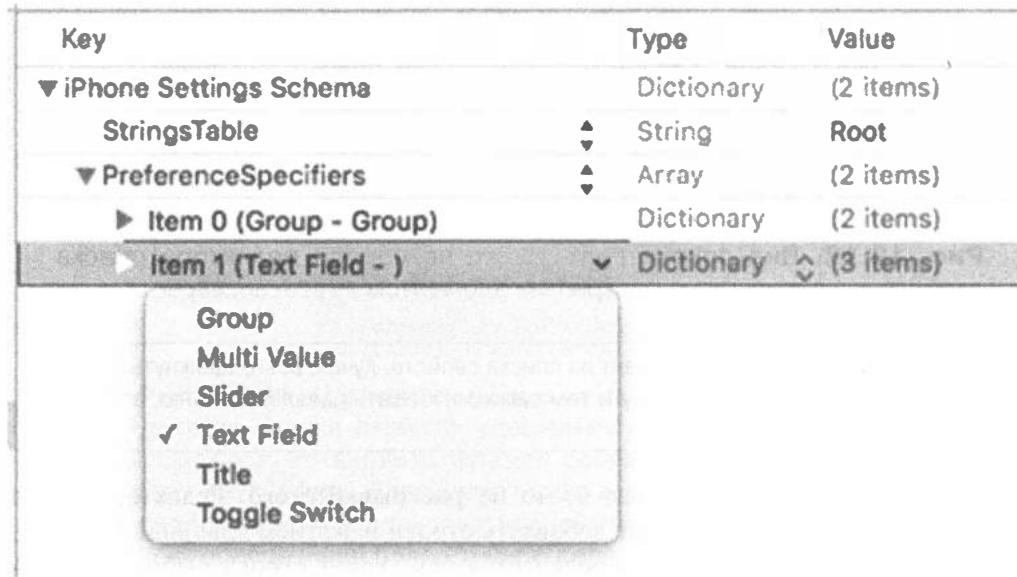


Рис. 12.13. На этом рисунке показано, что мы выбрали элемент `Item 0` и нажали клавишу `<Return>`, чтобы добавить новую строку того же уровня. Обратите внимание на выпадающее меню, в котором можно задать вид спецификатора глобальных параметров настройки, которые представляет данный элемент

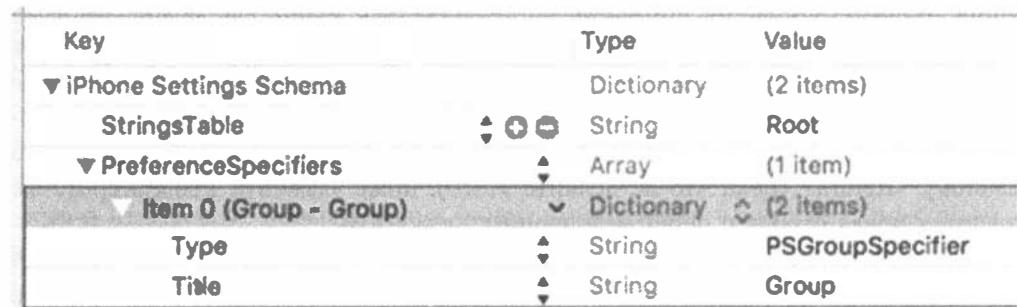


Рис. 12.14. При расширении элемента `Item 0` появляется одна строка с ключом `Type` и другая строка с ключом `Title`. Они представляют группу под заглавием `Group`

Еще только один элемент из иерархии элемента `Item 0` имеет ключ `Title`, который служит для установки дополнительного, хотя и не обязательного заголовка в начале группы. Присмотревшись внимательнее к самой строке `Item 0`, вы заметите, что на самом деле она обозначается как `Item 0 (Group - Group)`.

Обозначения в скобках представляют элемент Type (первое обозначение Group) и элемент Title (второе обозначение Group). Такое сокращение делает более удобным просмотр пакетов настроек.

Как следует из рис. 12.3, первая группа глобальных параметров настройки получила название General Info. Дважды щелкните в столбце Value на значении, соответствующем ключу Title, а затем измените его с Group на General Info (рис. 12.15). Как только вы введете новый заголовок, вы обнаружите небольшое изменение в элементе Item 0. Теперь он обозначается как Item0 (Group – General Info). Заглавие группы отображается в приложении Settings прописными буквами, и поэтому пользователь увидит обозначение GENERAL INFO, как и показано на рис. 12.3.

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	(1 item)
▼ Item 0 (Group - General Info)	Dictionary	(2 items)
Type	String	PSGroupSpecifier
Title	String	General Info

Рис. 12.15. Заглавие группы элемента Item 0 изменено с Group на General Info

Добавление поля редактирования для настройки

Теперь в данный массив нужно ввести второй элемент, который будет представлять первое конкретное поле глобальных параметров настройки. Это будет простое поле редактирования. Если щелкнуть сначала на строке PreferenceSpecifiers непосредственно на панели редактора (пока что не дейлайт этого), а затем нажать клавишу <Return>, чтобы ввести дочерний элемент, то в начале списка будет вставлена новая строка, которая нам не нужна. Нам же нужно добавить строку в конце массива.

Для этого щелкните на раскрывающем треугольнике слева от элемента Item 0, чтобы свернуть его, а затем выберите элемент Item 0 и нажмите клавишу <Return>, вставив ниже текущей строки новую строку того же уровня (рис. 12.16). Как обычно, при добавлении элемента появляется всплывающее меню со значением атрибута Text Field по умолчанию.

Щелкните сначала на любом участке за пределами выпадающего меню, чтобы оно исчезло, а затем на раскрывающем треугольнике рядом с элементом Item 1, чтобы раскрыть его иерархию. Как видите, он содержит атрибут Type row с установленным значением PSFieldSpecifier. Это значение сообщает приложению Settings, что пользователю следует разрешить редактировать

данного параметр настройки в поле редактирования. Элемент Item 1 содержит также две пустые строки для атрибутов Title и Key (рис. 12.17).

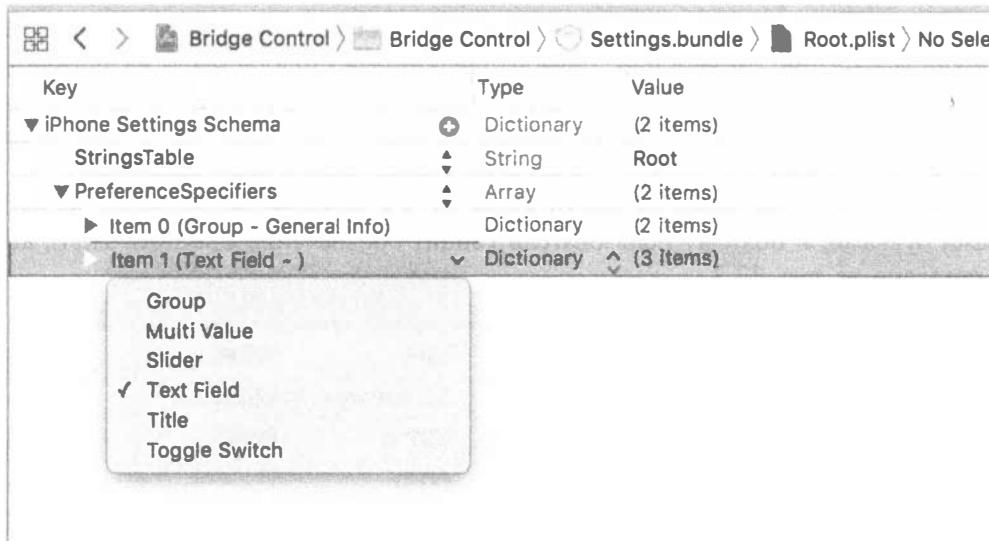


Рис. 12.16. Добавление новой строки того же уровня в иерархии элемента Item 0

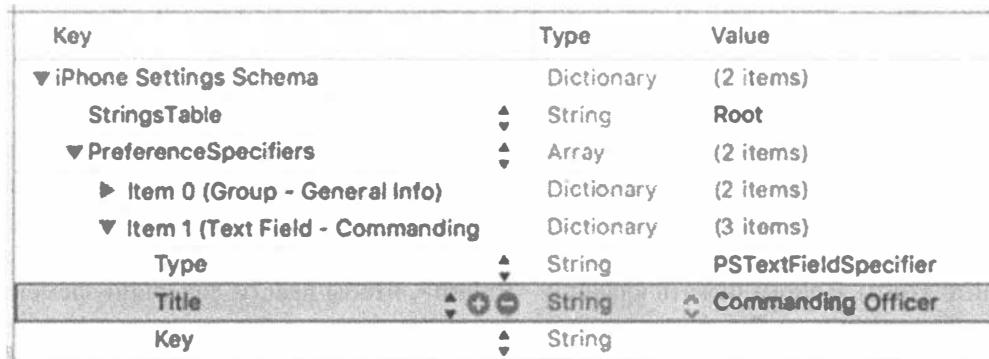


Рис. 12.17. Раскрыта иерархия элемента поля редактирования, демонстрирующая тип, заголовок и ключ

Выберите строку Title, а затем дважды щелкните на столбце Value. Введите значение Commanding Officer атрибута Title. Именно этот текст и появится в приложении Settings.

Сделайте то же самое в строке Key, но на этот раз введите значение officer строчными буквами в столбце Value. В этой строке приложению Settings сообщается, каким именном ключом следует воспользоваться, чтобы сохранить значение, введенное в данном поле редактирования.

Помните, что мы говорили о классе `NSUserDefaults`? Он позволяет сохранять значения по некоторому ключу, подобно тому, как это делается в классе `Dictionary`. То же самое будет сделано в приложении `Settings` с каждым глобальным параметром настройки, сохраняемым от вашего имени. Так, если задать для ключа значение `foo`, а затем запросить в приложении конкретное значение по ключу `foo`, оно возвратит значение, введенное пользователем для данного глобального параметра настройки. Мы воспользуемся этим же значением ключа в дальнейшем, чтобы извлечь соответствующую настройку из пользовательских настроек по умолчанию в рассматриваемом здесь приложении.

ЗАМЕЧАНИЕ. Обратите внимание на то, что ключу `Title` соответствует значение `Commanding Officer`, а ключу `Key` — значение `officer`. Такое отличие в написании прописными и строчными буквами встречается очень часто, а в данном случае оно становится еще более заметным благодаря двум словам в отображаемом заглавии и одному слову в значении ключа. Значение ключа `Title` отображается на экране, поэтому имеет смысл использовать в нем прописные буквы С и О. А значение ключа `Key` представлено текстовой строкой, которой мы будем пользоваться для извлечения глобальных параметров настройки из пользовательских настроек по умолчанию, поэтому в данном случае логичным будет написание только строчными буквами. Могли бы мы написать значение ключа `Title` только строчными буквами? Конечно. Можно ли написать значение ключа `Key` только прописными буквами? Безусловно! Дело в том, что при сохранении и извлечении глобальных параметров настройки пишутся прописными буквами, поэтому не имеет значения, каким образом обозначаются ключи.

Выберите последнюю из трех строк в иерархии элемента `Item 1` (с ключом `Key` в столбце `Key`). Нажмите клавишу `<Return>`, чтобы добавить в словарь `Item 1` еще одну запись, задав ключ `AutocapitalizationType`. Как только вы начнете вводить ключ `AutocapitalizationType`, в среде Xcode будет представлен список подходящих вариантов, так что вы можете просто выбрать его из списка, а не набирать полностью. Введя ключ `AutocapitalizationType`, нажмите клавишу `<Tab>` или щелкните на маленькой пиктограмме со стрелками вниз и вверх справа от столбца `Value`, чтобы раскрыть список, из которого можно выбрать имеющиеся варианты. Выберите вариант `Words`, который означает, что каждое слово, введенное пользователем в поле редактирования, должно быть автоматически написано прописными буквами.

Создайте еще одну, последнюю строку и задайте в ней ключ `AutocorrectionType` и значение `No`. Тем самым приложению `Settings` дается команда не корректировать автоматически значения, вводимые в данном поле редактирования. В тех случаях, когда для поля редактирования потребуется автоматическая коррекция, измените значение в этой строке на `Yes`. Опять же, как только вы начнете вводить слово `AutocorrectionType`, в среде Xcode будет представлен список подходящих вариантов, поэтому вы можете просто выбрать его из списка, а не набирать полностью.

По завершении описанных выше действий список свойств должен выглядеть так, как показано на рис. 12.18.

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	(9 items)
► Item 0 (Group - General Info)	Dictionary	(2 items)
▼ Item 1 (Text Field - Commanding	Dictionary	(5 items)
Type	String	PSTextFieldSpecifier
Title	String	Commanding Officer
Key	String	officer
AutocapitalizationType	String	Words
AutocorrectionType	String	No

Рис. 12.18. Завершенное поле редактирования в списке Root.plist

Добавление пиктограммы приложения

Прежде чем перейти к следующему параметру настройки, добавьте в проект пиктограмму приложения. Вам уже приходилось делать это раньше.

Сохраните только что отредактированный файл свойств Root.plist. Перейдите к навигатору проекта и выберите папку Assets.xcassets, а в ней — элемент AppIcon. Там вы найдете пару целей, предназначенных для размещения пиктограмм.

Откройте в окне Finder сначала архив с исходным кодом примеров, прилагаемым к этой книге, а затем папку 12 – Images. Проходя по элементам окна редактирования папки Assets.xcassets сверху вниз, перетащите из папки 12 – Images следующие папки.

- Файлы Settings-iPhone@2x.png и Settings-iPhone@3x.png — на позиции 2x и 3x в левой верхней группе.
- Файлы Spotlight-iPhone@2x.png и Spotlight-iPhone@3x.png — на позиции 2x и 3x в правой верхней группе.
- Файлы AppIcon-iPhone@2x.png и AppIcon-iPhone@3x.png — на позиции 2x и 3x в группе во второй строке.
- Файлы Settings-iPad.png и Settings-iPad@2x.png — на позиции 1x и 2x в левой группе в третьей строке.
- Файлы Spotlight-iPad.png и Spotlight-iPad@2x.png — на позиции 1x и 2x в правой группе в третьей строке.
- Файлы AppIcon-iPad.png и AppIcon-iPad@2x.png — на позиции 1x и 2x в группе в нижней строке.
- Файлы AppIcon-iPadPro.png — на позицию iPad Pro в правом нижнем углу.

Обратите внимание на представление *Activity*. Если вы перетащите файл в неправильную позицию, на экране появится треугольник предупреждения. Если это произойдет, исправьте ошибку и продолжайте работу. По завершении редактор ресурсов должен выглядеть так, как показано на рис. 12.19.

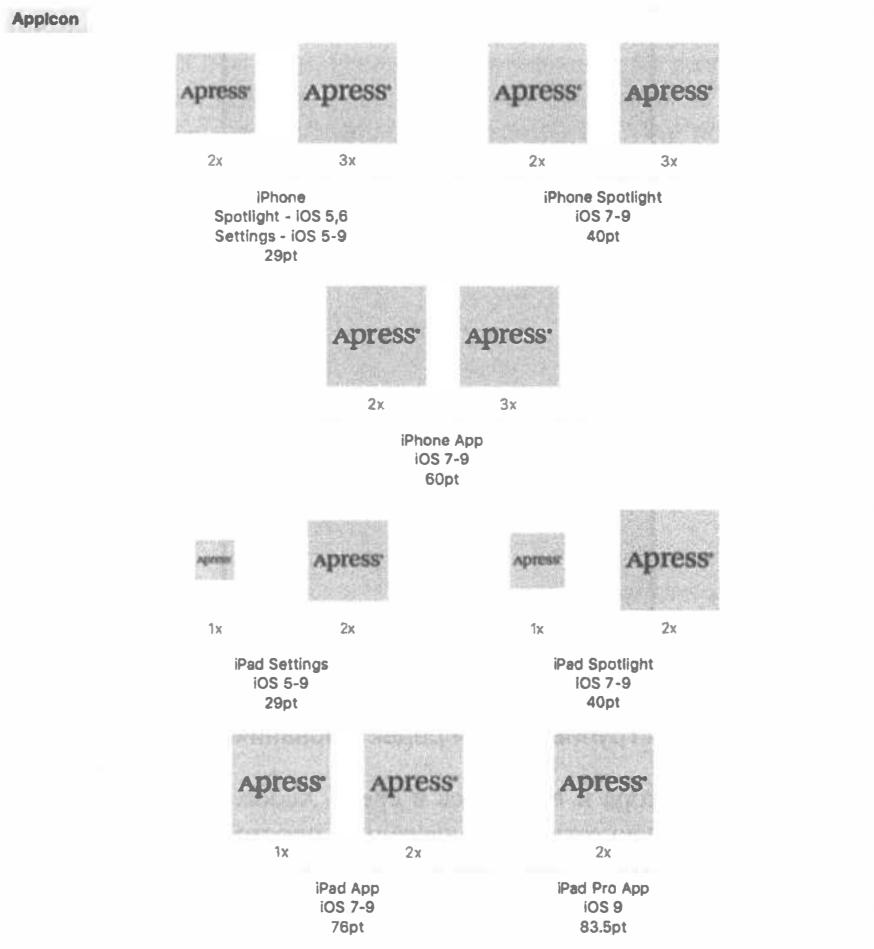


Рис. 12.19. Ввод в приложение пиктограмм настроек и самого приложения

Вот и все. Осталось лишь скомпилировать и запустить приложение на выполнение по команде *Product⇒Run*. Мы еще не создали никакого графического пользовательского интерфейса для данного приложения, поэтому вы увидите только первую вкладку из панели вкладок. Щелкните сначала на кнопке *Home*, а затем на пиктограмме приложения *Settings*. Найдите элемент, соответствующий рассматриваемому здесь приложению с добавленной раньше пиктограммой (см. рис. 12.2). Щелкните на строке *Bridge Control*, чтобы увидеть простое представление настроек с одним полем редактирования (рис. 12.20).

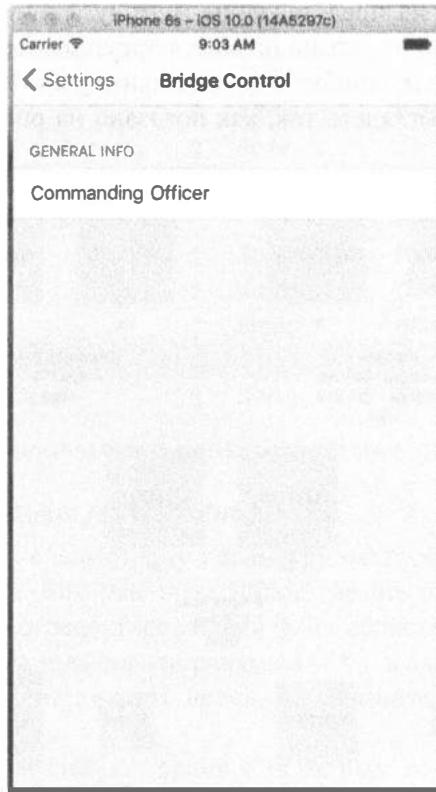


Рис. 12.20. Корневое представление настроек в приложении *Settings* после добавления группы и поля редактирования

Выходите из симулятора и вернитесь в среду Xcode. Мы еще не закончили, но вы, вероятно, уже почувствовали, насколько просто можно ввести глобальные параметры настройки в свое приложение. Теперь добавим остальные поля в корневое представление настроек, начав с защищенного поля редактирования, предназначенного для ввода кода полномочий пользователя.

Добавление защищенного поля редактирования

Откройте среду Xcode и щелкните на элементе *Root.plist*, чтобы вернуться к спецификаторам настроек (не забудьте установить флагок *Show Raw Keys/Values*). Сверните элементы *Item 0* и *Item 1*. Выберите элемент *Item 1*. Нажмите комбинацию клавиш *<⌘+C>*, чтобы скопировать выбранный элемент в буфер обмена, а затем комбинацию клавиш *<⌘+V>*, чтобы вставить его. В итоге будет создан новый элемент *Item 2*, идентичный элементу *Item 1*. Раскройте новый элемент и измените значение ключа *Title* на *Authorization Code*, а значение ключа *Key* — на *authorizationCode*. Напомним, что значение ключа *Title* отображается на экране в виде метки, а ключ *Key* служит для сохранения значения.

Затем добавьте к новому элементу Item 2 еще один дочерний элемент. Напомним, что порядок следования элементов не имеет значения, поэтому дочерний элемент можно разместить непосредственно под только что отредактированным элементом Key. Для этого выберите строку Key/authorizationCode и нажмите клавишу <Return>.

Замените ключ в столбце Key для нового элемента на IsSecure, а тип в столбце Type — на Boolean. Измените его значение NO в столбце Value на YES. Этим приложению Settings сообщается, что данное поле будет служить для скрытого ввода аналогично паролю, а не открытого, как обычно, текста. В заключение измените для ключа AutocapitalizationType значение в столбце Value на None. В завершенном виде элемент Item 2 показан на рис. 12.21.

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	{2 items}
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	{3 items}
► Item 0 (Group - General Info)	Dictionary	{2 items}
► Item 1 (Text Field - Commanding)	Dictionary	{5 items}
▼ Item 2 (Text Field - Authorization)	Dictionary	{6 items}
Type	String	PSTextFieldSpecifier
Title	String	Authorization Code
Key	String	authorizationCode
IsSecure	Boolean	YES
AutocapitalizationType	String	None
AutocorrectionType	String	No

Рис. 12.21. Завершенный вид элемента Item 2, представляющего поле редактирования для ввода кода полномочий пользователя

Добавление многозначного поля

Следующий элемент, который нам предстоит ввести, называется *многозначным полем*. Этот тип поля позволяет автоматически формировать строку с индикатором раскрытия. Щелкнув на нем, пользователь перейдет к еще одной таблице, в которой он сможет выбрать одну из нескольких строк.

Сверните элемент Item 2, выберите строку этого элемента, а затем нажмите клавишу <Return>, чтобы добавить элемент Item 3. Затем выберите из раскрывающегося списка, связанного с полем Key, вариант Multi Value и разверните элемент Item 3, щелкнув на раскрывающем треугольнике.

Развернутый элемент Item 3 уже содержит несколько строк. В одной из них (с ключом Type) находится значение PSMultiValueSpecifier. Найдите строку

с ключом `Title` и задайте в ней значение `Rank`. Затем найдите строку с ключом `Key` и задайте в ней значение `rank`. Следующий этап несколько сложнее предыдущих, поэтому стоит остановиться на нем подробнее.

Нам требуется добавить еще две дочерние строки к элементу `Item 3`, но это будут узлы типа `Array`, а не `String`.

- Первый массив под названием `Titles` должен содержать список значений, из которого пользователь может выбрать нужный вариант.
- Второй массив под названием `Values` должен содержать список значений, которые фактически хранятся в пользовательских настройках по умолчанию.

Если пользователь выберет из списка первый элемент, соответствующий первому элементу в массиве `Titles`, приложение `Settings` фактически сохранит первое значение из массива `Values`. Такое спаренное применение массивов `Titles` и `Values` позволяет предоставить пользователю данные в удобочитаемом текстовом виде, но в то же время сохранить совсем другое содержимое, например число, дату или другую символьную строку. Оба эти массива обязательны. Если требуется, чтобы эти массивы были одинаковы, создайте один массив, скопируйте его и вставьте обратно, но измените ключ, чтобы получить в конечном итоге два массива с одинаковым содержимым, но с разными ключами. Именно так мы теперь и поступим.

Выберите элемент `Item 3` (оставив его раскрытым) и нажмите клавишу `<Return>`, чтобы добавить новую дочернюю строку к этому элементу. При этом вы сможете еще раз убедиться, что в среде Xcode заранее известен тип файла, который вы редактируете, и поэтому, предвидя ваши действия, в поле `Key` новой дочерней строки уже установлено значение `Titles`, а в поле `Type` — тип `Array`, т.е. именно то, что нам и нужно! Нажмите клавишу `<Return>`, чтобы прекратить редактирование поля `Key`, а затем раскройте строку `Titles` и нажмите клавишу `<Return>`, чтобы добавить дочерний узел. Повторите эту процедуру еще пять раз, чтобы в итоге добавить шесть дочерних узлов. Все они должны быть узлами типа `String` и содержать следующие значения: `Ensign`, `Lieutenant`, `Lieutenant Commander`, `Commander`, `Captain` и `Commodore`.

Введя все эти узлы, сверните элемент `Titles` и выберите его. Нажмите комбинацию клавиш `<⌘+C>`, чтобы скопировать выбранный элемент в буфер обмена, а затем комбинацию клавиш `<⌘+V>`, чтобы вставить его. В итоге будет создан новый элемент с ключом `Titles - 2`. Дважды щелкните на ключе `Titles - 2` и измените его на `Values`.

Мы почти завершили создание многозначного поля. В словаре есть еще одно обязательное значение, которое предоставляется по умолчанию. Многозначные поля должны иметь одну и только одну выбранную строку. Поэтому нужно указать такое значение, которое будет использоваться по умолчанию, если ни одно

из предложенных для выбора значений не будет выбрано, причем это значение должно быть связано с одним из элементов массива `Values` (но не массива `Titles`, если эти массивы разные). При создании этого элемента средствами среды Xcode к нему была автоматически добавлена строка `DefaultValue`, поэтому вам остается только задать в ней значение `Ensign`. Сделайте это теперь. В завершенном виде элемент `Item 3` показан на рис. 12.22.

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
Strings Filename	String	Root
▼ Preference Items	Array	(4 items)
► Item 0 (Group - General Info)	Dictionary	(2 items)
► Item 1 (Text Field - Commanding)	Dictionary	(5 items)
► Item 2 (Text Field - Authorization)	Dictionary	(6 items)
▼ Item 3 (Multi Value - Rank)	Dictionary	(6 items)
▼ Titles	Array	(6 items)
Item 0	String	Ensign
Item 1	String	Lieutenant
Item 2	String	Lieutenant Commander
Item 3	String	Commander
Item 4	String	Captain
Item 5	String	Commodore
▼ Values	Array	(6 items)
Item 0	String	Ensign
Item 1	String	Lieutenant
Item 2	String	Lieutenant Commander
Item 3	String	Commander
Item 4	String	Captain
Item 5	String	Commodore
Type	String	Multi Value
Title	String	Rank
Identifier	String	rank
Default Value	String	Ensign

Рис. 12.22. Завершенный вид элемента `Item 3`, представляющего многозначное поле для выбора одного из шести возможных значений

Проверим результаты наших трудов. Сохраните список свойств, а затем постройте и запустите данное приложение на выполнение. После этого нажмите главную кнопку и запустите приложение `Settings` на выполнение. Выбрав элемент `Bridge Control` в корневом представлении, вы должны увидеть три поля (рис. 12.23). Поэкспериментируйте с ними, и двинемся дальше.

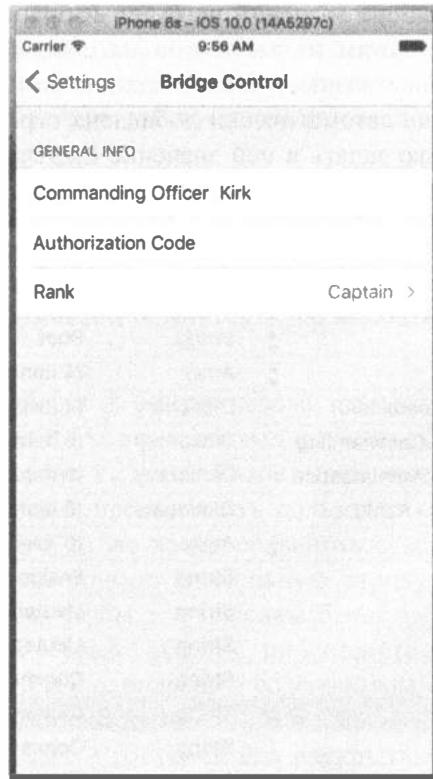


Рис. 12.23. Три готовых к употреблению поля. Совсем неплохо!

Добавление переключателя

Следующий элемент для получения данных от пользователя предоставляет логическое значение типа Boolean, которое означает, подключены ли двигатели искривления пространства. Для того чтобы зафиксировать логическое значение типа Boolean в глобальных параметрах настройки, дадим приложению `Settings` команду воспользоваться классом `UISwitch`, добавив в массив `PreferenceSpecifiers` еще один элемент типа `PSToggleSwitchSpecifier`.

Сверните элемент `Item 3`, если он все еще развернут, а затем щелкните на нем, чтобы выбрать его. Нажмите клавишу `<Return>`, чтобы создать элемент `Item 4`. Выберите из раскрывающегося списка вариант `Toggle Switch`, а затем щелкните на раскрывающем треугольнике, чтобы развернуть элемент `Item 4`. В итоге вы увидите в нем дочернюю строку, в которой поле `Key` содержит значение `Type`, а поле `Value` — значение `PPSToggleSwitchSpecifier`. Добавьте еще одну дочернюю строку с ключом `Title` и значением `Warp Drive`, а затем третью дочернюю строку с ключом `Key` и значением `warp`.

В этом словаре должен быть еще один обязательный элемент, предоставляющий значение параметра по умолчанию. Как и при настройке элемента `Multi Value`,

в Xcode уже создана строка с ключом `DefaultValue`. Пусть двигатели искривления пространства будут по умолчанию включены; присвойте в строке `DefaultValue` значение `YES`. В завершенном виде элемент `Item 4` показан на рис. 12.24.

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
Strings Filename	String	Root
▼ Preference Items	Array	(5 items)
► Item 0 (Group - General Info)	Dictionary	(2 items)
► Item 1 (Text Field - Commanding)	Dictionary	(5 items)
► Item 2 (Text Field - Authorization)	Dictionary	(6 items)
► Item 3 (Multi Value - Rank)	Dictionary	(6 items)
Item 4 (Toggle Switch - Warp)	Dictionary	(4 items)
Type	String	Toggle Switch
Title	String	Warp Drive
Identifier	String	warp
Default Value	Boolean	YES

Рис. 12.24. Завершенный вид элемента `Item 4`, представляющего переключатель, управляемый двигателями искривления пространства

Добавление ползунка

Теперь нужно добавить ползунок. В приложении `Settings` ползунок может иметь небольшие изображения на концах, но не может иметь метку. Разместим ползунок в отдельной группе с заголовком, чтобы пользователь знал назначение ползунка. Для начала сверните элемент `Item 4`. Щелкните на элементе `Item 4` и нажмите клавишу `<Return>`, чтобы вставить новый элемент в группу. Выберите из раскрывающегося списка вариант `Group`, чтобы превратить новый элемент в группу, а затем раскройте его. Как видите, в поле `Type` уже установлено значение `PSGroupSpecifier`. Этим приложению `Settings` предписывается начать здесь новую группу. Дважды щелкните на значении в строке `Title` и измените его на `Warp Factor`.

Сверните элемент `Item 5` и выберите его. Нажмите клавишу `<Return>`, чтобы добавить новую строку того же уровня. Выберите из раскрывающегося списка элемент `Slider`, указав приложению `Settings` воспользоваться классом `UISlider`, чтобы получить данные от пользователя. Раскройте элемент `Item 6` и задайте в строке `Key` значение `warpFactor`, чтобы приложению `Settings` стало известно, какой именно ключ следует использовать для хранения этого значения.

Нам требуется разрешить пользователю регулировать коэффициент искривления пространства в пределах от 1 до 10, а по умолчанию установить его равным 5. Ползунки должны иметь минимальное значение, максимальное и начальное (по умолчанию) значение, и все эти значения нужно сохранить в списке свойств как числа, но не как символьные строки. Правда, в среде Xcode уже

созданы строки для установки всех этих значений. Поэтому установите в строке `DefaultValue` значение 5, в строке `MinimumValue` — значение 1, а в строке `MaximumValue` — значение 10.

При желании можете прямо сейчас проверить правильность функционирования ползунка. Тем временем мы сделаем еще кое-что для его настройки.

Как отмечалось ранее, ползунки могут иметь изображения на концах: воспользуемся мелкими пиктограммами, чтобы показать, что смещение ползунка влево дает эффект замедления, а смещение вправо — эффект ускорения.

Добавление пиктограмм в пакет настроек

В папке 12 – `Images` с исходным кодом примеров, прилагаемым к этой книге, вы найдете две пиктограммы в файлах изображений `rabbit.png` и `turtle.png`. Нам нужно добавить их в пакет настроек рассматриваемого здесь приложения. Эти изображения будут использоваться в приложении `Settings`, и поэтому нельзя просто переместить их в папку `Bridge Control`, но придется ввести в пакет настроек, чтобы обеспечить доступ к ним из приложения `Settings`. С этой целью найдите в навигаторе проекта папку `Settings.bundle` и откройте ее в окне `Finder`. Щелкните правой кнопкой мыши на пиктограмме `Settings.bundle` в окне навигатора проекта. Выполните команду `Show in Finder` из всплывающего меню (рис. 12.25).

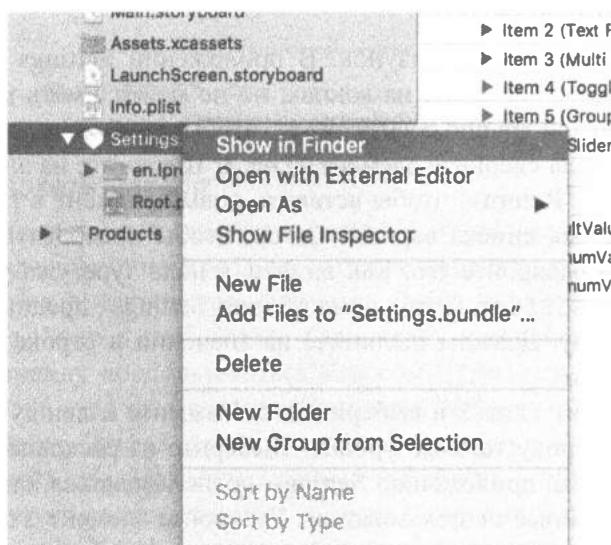


Рис. 12.25. Вид контекстного меню элемента `Settings.bundle`

Напомним, что пакеты на самом деле являются папками, хотя в окне `Finder` они выглядят, как файлы. Чтобы посмотреть содержимое папки `Settings.bundle` в окне `Finder`, щелкните на ней правой кнопкой мыши и выберите пункт `Show Package Contents` из всплывающего меню. В итоге пакет настроек

откроется в новом окне Finder, в котором вы должны обнаружить те же два элемента, которые видели и в Xcode, раскрыв папку Settings.bundle. Скопируйте два файла пиктограмм rabbit.png и turtle.png из папки 12 – Images в папку Settings.bundle непосредственно в окне Finder, расположив их рядом с файлами en.proj и Root.plist. Можете оставить это окно Finder открытым, поскольку нам придется вскоре скопировать в нем еще один файл. Тем временем вернемся в среду Xcode, чтобы использовать эти два изображения на концах ползунка.

Итак, вернувшись в Xcode, обратитесь снова к списку Root.plist и введите в нем еще две дочерние строки под элементом Item 6: одну — с ключом **MinimumValueImage** и значением turtle.png, а другую — с ключом **MaximumValueImage** и значением rabbit.png. В завершенном виде элемент Item 6 показан на рис. 12.26.

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	(7 items)
► Item 0 (Group - General Info)	Dictionary	(2 items)
► Item 1 (Text Field - Commanding)	Dictionary	(5 items)
► Item 2 (Text Field - Authorization)	Dictionary	(6 items)
► Item 3 (Multi Value - Rank)	Dictionary	(6 items)
► Item 4 (Toggle Switch - Warp)	Dictionary	(4 items)
► Item 5 (Group - Warp Factor)	Dictionary	(2 items)
▼ Item 6 (Slider)	Dictionary	(7 items)
Type	String	PSSliderSpecifier
Key	String	warpFactor
DefaultValue	Number	5
MinimumValue	Number	1
MaximumValue	Number	10
MinimumValueImage	String	turtle
MaximumValueImage	String	rabbit

Рис. 12.26. Завершенный вид элемента Item 6, представляющий ползунок с пиктограммами черепахи и кролика на концах для обозначения замедления и ускорения соответственно

Сохраните список свойств, а затем постройте и запустите данное приложение на выполнение, чтобы убедиться, что оно функционирует должным образом. В таком случае вы сможете войти в приложение Settings и найти ползунок со спящей черепахой и счастливым кроликом на разных его концах (рис. 12.27).

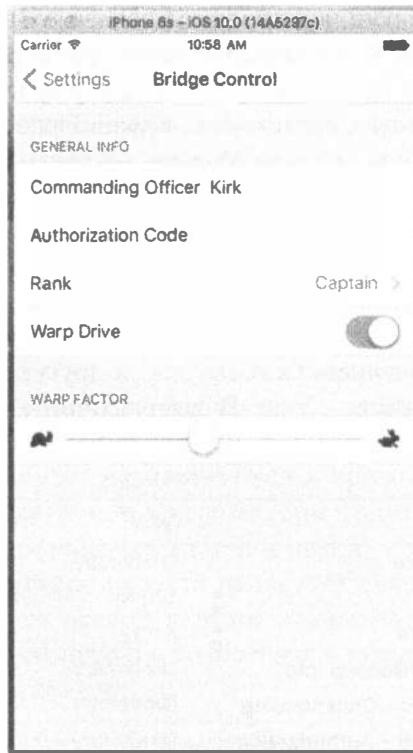


Рис. 12.27. Теперь у нас есть поля редактирования и многозначные поля, переключатель и ползунок, и мы близки к завершению

Добавление дочернего представления настроек

Добавим еще один спецификатор глобальных параметров настройки, чтобы дать приложению Settings команду отобразить дочернее представление настроек. Этот спецификатор будет представлять строку с раскрывающим индикатором продолжения, после касания которого откроется совершенно новое представление с глобальными параметрами настройки.

Поскольку этот новый глобальный параметр настройки не должен находиться в одной группе с ползунком, скопируем сначала групповой спецификатор из элемента Item 0 и вставим его в конце массива PreferenceSpecifiers, чтобы создать новую группу для дочернего представления настроек. Сверните все открытые элементы в списке Root.plist, а затем щелкните на элементе Item 0, чтобы выбрать его. Нажмите комбинацию клавиш **<⌘+C>**, чтобы скопировать его в буфер обмена. Выберите элемент Item 6 и нажмите комбинацию клавиш **<⌘+V>**, чтобы вставить новый элемент Item 7. Раскройте элемент Item 7 и дважды щелкните на поле в столбце Value рядом с ключом Title, изменив в этом поле значение General Info на Additional Info.

Сверните элемент Item 7. Выберите его и нажмите клавишу <Return>, чтобы добавить элемент Item 8, являющийся дочерним представлением. Разверните элемент Item 8, щелкнув на раскрывающем треугольнике. Найдите строку с ключом Type и задайте в ней значение PSChildPaneSpecifier, а в строке с ключом Title — значение More Settings. Осталось добавить в элементе Item 8 последнюю строку, чтобы сообщить приложению Settings, какой именно список свойств следует загрузить для представления More Settings. Итак, добавьте еще одну дочернюю строку с ключом File и значением More (для этого можно изменить последнюю в группе строку с Key на File; рис. 12.28). Для файла со списком свойств предполагается расширение .plist, и поэтому указывать его не нужно (если вы укажете расширение, приложение Settings не найдет файл со списком свойств).

Key	Type	Value
▼ iPhone Settings Schema	Dictionary	(2 items)
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	(9 items)
► Item 0 (Group - General Info)	Dictionary	(2 items)
► Item 1 (Text Field - Commanding	Dictionary	(5 items)
► Item 2 (Text Field - Authorization	Dictionary	(6 items)
► Item 3 (Multi Value - Rank)	Dictionary	(6 items)
► Item 4 (Toggle Switch - Warp	Dictionary	(4 items)
► Item 5 (Group - Warp Factor)	Dictionary	(2 items)
► Item 6 (Slider)	Dictionary	(7 items)
► Item 7 (Group - Additional Info)	Dictionary	(2 items)
▼ Item 8 (Child Pane - More	Dictionary	(4 items)
Type	String	PSChildPaneSpecifier
Title	String	More Settings
Key	String	
File	String	More

Рис. 12.28. Завершенный вид элементов Item 7 и Item 8, представляющих группу параметров настройки Additional Info и ссылку на дочернее окно, связанное с файлом More.plist соответственно

Добавим дочернее представление к главному представлению глобальных параметров настройки. Параметры настройки в этом дочернем представлении определяются в файле More.plist, поэтому мы должны скопировать его в пакет настроек. Но мы не можем добавить в пакет новые файлы в среде Xcode, а диалоговое окно Save редактора списка свойств не позволяет сохранить файл в пакете. Поэтому придется создать новый список свойств, сохранить его в какой-нибудь папке, а затем перетащить ее в папку Settings.bundle непосредственно в окне Finder. Чтобы упростить создание дочернего представления, скопируйте

файл `Root.plist`, присвойте ему новое имя, удалите все существующие настройки, за исключением первой, и добавьте в него те настройки, которые необходимы для вашего приложения. Для того чтобы сэкономить усилия, можете просто перетащить файл `More.plist` из папки 12 – `Images` с исходным кодом примеров, в папку `Settings.bundle` непосредственно в окне `Finder`, которое было оставлено открытым, и оставить его рядом с файлом `Root.plist`.

Мы завершили формирование пакета настроек. Скомпилируйте, запустите на выполнение данное приложение и проверьте доступность его глобальных параметров настройки в приложении `Settings`. В конечном итоге мы должны добраться до дочернего представления и установить значения для всех остальных полей. Поэкспериментируйте с этим представлением и по желанию внесите изменения в список свойств.

ПОДСКАЗКА. Мы рассмотрели почти все возможные варианты полей. Полное описание формата списка свойств для хранения настроек можно найти в руководстве *Settings Application Schema Reference*, доступном на сайте iOS Dev Center. Получить этот документ (и немало других полезных справочных документов) можно по адресу <http://developer.apple.com/library/ios/navigation/>.

Прежде чем продолжить работу, выберите папку `Assets.xcassets` в окне навигатора проекта и скопируйте файлы изображений `rabbit.png` и `turtle.png` из папки 12 – `Images` с исходным кодом примеров, прилагаемым к этой книге, в левую часть области редактирования. В итоге эти изображения пиктограммы будут введены в данный проект как новые, готовые к употреблению ресурсы. Мы еще воспользуемся ими в рассматриваемом здесь приложении, чтобы отобразить значения текущих настроек.

Вероятно, вы заметили, что две пиктограммы, которые вы только что добавили, точно такие же, как и те, которые были добавлены в пакет настроек раньше. Возникает вопрос: а зачем их дублировать? Напомним, что приложения под управлением системы iOS не могут считывать файлы из “песочниц” других приложений. Дело в том, что пакет настроек становится частью “песочницы” приложения `Settings`, а не рассматриваемого здесь приложения. Но поскольку эти пиктограммы требуется использовать и в данном приложении, их нужно отдельно добавить в папку `Bridge Control`, чтобы скопировать их и в “песочницу” данного приложения.

Чтение настроек из приложения

Для чтения пользовательских настроек воспользуемся классом `UserDefaults` (`NSUserDefaults`), реализованным как одноэлементный класс. Это означает, что в данном приложении будет существовать только один экземпляр класса `UserDefaults`. Чтобы получить доступ к этому единственному экземпляру, необходимо вызвать из этого класса метод `standard()` следующим образом:

```
let defaults = UserDefaults.standard()
```

Имея указатель на стандартные пользовательские настройки, воспользуемся им таким же образом, как это делается в словаре типа `Dictionary`. Чтобы прочитать из него значение, достаточно вызвать метод `object(forKey:)`, возвращающий объект типа `String` или `Foundation`, например `Date` (`NSDate`) или `Number` (`NSNumber`). Если же требуется извлечь скалярное значение наподобие `int`, `float` или `Boolean`, то достаточно вызвать соответствующий метод: `int(forKey:)`, `float(forKey:)` или `bool(forKey:)`.

При составлении списка свойств для данного приложения в файл с расширением `.plist` был введен массив спецификаторов типа `PreferenceSpecifiers`. Одни спецификаторы были использованы в приложении `Settings` для составления групп, а другие — для создания интерфейсных объектов для организации взаимодействия с пользователем. Нас интересуют именно эти последние спецификаторы, поскольку именно они обеспечивают хранение реальных данных настроек. Каждый спецификатор, который был связан с каким-нибудь параметром пользователя, имеет ключ под названием `Key`. Например, ключ `Key` для ползунка имеет значение `warpfactor`, ключ `Key` для поля `Authorization Code` — `authorizationCode`. Мы воспользуемся этими ключами для извлечения пользовательских настроек.

Вместо того чтобы воспользоваться символьными строками для каждого ключа в рассматриваемых далее методах, мы определим для их значений ряд констант. Этими константами мы можем воспользоваться в коде данного приложения вместо встраиваемых символьных строк, чтобы исключить опечатки при их вводе. Эти константы должны быть установлены в отдельном исходном коде на языке `Swift`, поскольку мы собираемся воспользоваться ими еще не в одном классе. Перейдите в среду `Xcode`, нажмите комбинацию клавиш `<⌘+N>` и выберите в окне создания файлов сначала подраздел `Source` в разделе `iOS`, а затем шаблон `Swift File` и щелкните на кнопке `Next`. Присвойте новому файлу имя `Constants.swift` и щелкните на кнопке `Create`. Откройте вновь созданный исходный файл и введите в нем строки кода, приведенные в листинге 12.1.

Листинг 12.1. Константы

```
let officerKey = "officer"
let authorizationCodeKey = "authorizationCode"
let rankKey = "rank"
let warpDriveKey = "warp"
let warpFactorKey = "warpFactor"
let favoriteTeaKey = "favoriteTea"
let favoriteCaptainKey = "favoriteCaptain"
let favoriteGadgetKey = "favoriteGadget"
let favoriteAlienKey = "favoriteAlien"
```

Эти константы служат ключами для доступа к различным полям глобальных параметров настройки в файле с расширением `.plist`. Теперь, когда есть место для отображения настроек, быстро установим главное представление с рядом методов. Прежде чем переходить в `Interface Builder`, создадим выходы для всех нужных

нам меток. С этой целью щелкните на исходном файле `FirstViewController.swift` и внесите в него изменения, показанные в листинге 12.2.

Листинг 12.2. Добавление выходов в файл `FirstViewController.swift`

```
class FirstViewController: UIViewController {
    @IBOutlet var officerLabel:UILabel!
    @IBOutlet var authorizationCodeLabel:UILabel!
    @IBOutlet var rankLabel:UILabel!
    @IBOutlet var warpDriveLabel:UILabel!
    @IBOutlet var warpFactorLabel:UILabel!
    @IBOutlet var favoriteTeaLabel:UILabel!
    @IBOutlet var favoriteCaptainLabel:UILabel!
    @IBOutlet var favoriteGadgetLabel:UILabel!
    @IBOutlet var favoriteAlienLabel:UILabel!
```

Как видите, в этом коде нет для нас ничего нового. Мы лишь объявили девять свойств с помощью ключевого слова `@IBOutlet`, чтобы связать их метками в Interface Builder. Сохраните внесенные изменения. Теперь, когда выходы объявлены, можно перейти к файлу раскладовки для создания графического пользовательского интерфейса данного приложения.

Создание главного представления

Выберите файл раскладовки `MainStoryboard.storyboard`, чтобы отредактировать его в Interface Builder. Таким образом, слева появится контроллер представления панели вкладок, а справа один за другим — контроллеры представлений двух вкладок. Верхний контроллер служит для представления первой вкладки и реализуется в классе `FirstViewController`, а нижний контроллер — для представления второй вкладки и реализуется в классе `SecondViewController`.

Введем сначала ряд меток в представление типа `FirstViewController`, как показано на рис. 12.29. В общем, их должно быть 18. Первая их половина слева на экране должна быть выровнена по правому краю и выделена полужирным шрифтом, а вторая половина справа на экране должна служить для отображения конкретных значений, извлекаемых из пользовательских настроек, и поэтому они должны быть связаны с соответствующими входами. Все вносимые здесь изменения будут сделаны в контроллере представления первой вкладки, расположенному справа вверху на раскладовке.

Разверните сначала узел главной сцены (Main Scene) в окне `Document Outline`, а затем элемент `View`. В итоге вы обнаружите три дочерних представления, уже находящихся на своих местах. Удалите все эти представления. Затем переименуйте элемент `View` на `Main View`. Далее перетащите объект `Label` из библиотеки объектов, опустив его у левого верхнего края данного представления. Перетащите метку к левому краю окна (или хотя бы к голубой направляющей линии слева), а затем расширьте ее правый край по направлению к центру представления, как показано на примере метки `Officer` на рис. 12.29. Перейдите к инспектору атрибутов, выровняйте текст метки по правому краю и выберите для него шрифт

System Bold 15. Нажмите клавишу <Option> и перетащите метку вниз, чтобы создать еще восемь ее копий. Аккуратно выровняйте метки, чтобы образовать из них левый столбец. Измените текст меток в соответствии с рис. 12.29.

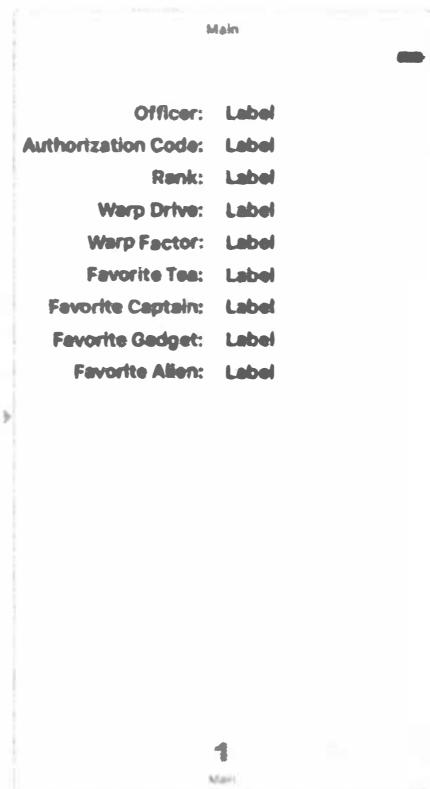


Рис. 12.29. Вид контроллера представления первой вкладки в окне программы Interface Builder с 18 введенными метками

Построить правый столбец немного проще. Перетащите еще одну метку из библиотеки объектов в текущее представление, расположив ее справа от метки Officer и оставив между ними небольшой промежуток. Выберите для текста этой метки шрифт System 15 в инспекторе атрибутов. Нажмите клавишу <Option> и перетащите метку вниз, чтобы создать еще восемь ее копий. Аккуратно выровняйте их по меткам в левом столбце.

Теперь нам нужно задать ограничения Auto Layout. С этой целью свяжите вместе две верхние метки, нажав клавишу <Control> и перетащив метку Officer к метке справа. Отпустите кнопку мыши и нажмите клавишу <Shift>. Выберите команды Horizontal Spacing и Baseline из всплывающего меню, а затем щелкните за пределами этого меню. Сделайте то же самое в остальных восьми рядах меток, попарно связав их вместе.

Далее нужно зафиксировать расположение меток в левом столбце относительно левого и верхнего краев представления. С этой целью перейдите в окно Document Outline, нажмите клавишу <Control> и перетащите указатель от метки Officer к представлению Main View. Отпустите кнопку мыши, нажмите клавишу <Shift> и выберите команды Leading Space to Container Margin и Vertical Spacing to Top Layout Guide из всплывающего меню, а затем нажмите клавишу <Return>. Сделайте то же самое с остальными восемью метками в левом столбце.

В заключение нужно зафиксировать ширину меток в левом столбце. Для этого выберите метку Officer и щелкните на кнопке Pin, расположенной ниже редактора раскадровки. Установите флагок Width в открывшемся окне и щелкните на кнопке Add 1 Constraint. Повторите эту же процедуру для всех остальных меток в левом столбце.

Теперь ограничения должны быть правильно наложены на все метки. Выберите контроллер представления Main в окне Document Outline и выполните команду Editor⇒Resolve Auto Layout Issues⇒Update Frames из меню Xcode (если этот режим не активизирован, значит, все метки уже находятся на своих местах в раскадровке). Если все пройдет нормально, то все метки займут свое окончательное положение.

Далее метки в правом столбце нужно связать с их выходами. С этой целью откройте исходный файл FirstViewController.swift в окне помощника редактора, нажмите клавишу <Control> и перетащите указатель от верхней метки в правом столбце к выходу officerLabel, чтобы связать их вместе. Снова нажмите клавишу <Control> и перетащите указатель от второй метки в правом столбце к выходу authorizationLabel, повторяя эти действия до тех пор, пока все метки в правом столбце не окажутся связанными со своими выходами. Сохраните внесенные изменения в файле раскадровки Main.storyboard.

Обновление контроллера представления первой вкладки

Выберите в среде Xcode файл FirstViewController.swift и введите в конце его класса код из листинга 12.3.

Листинг 12.3. Демонстрация данных в полях меток

```
func refreshFields() {
    let defaults = UserDefaults.standard
    officerLabel.text = defaults.string(forKey: officerKey)
    authorizationCodeLabel.text = defaults.string(forKey:
authorizationCodeKey)
    rankLabel.text = defaults.string(forKey: rankKey)
    warpDriveLabel.text = defaults.bool(forKey: warpDriveKey)
        ? "Engaged" : "Disabled"
    warpFactorLabel.text = defaults.object(forKey: warpFactorKey)?.
stringValue
    favoriteTeaLabel.text = defaults.string(forKey: favoriteTeaKey)
    favoriteCaptainLabel.text = defaults.string(forKey: favoriteCaptainKey)
    favoriteGadgetLabel.text = defaults.string(forKey: favoriteGadgetKey)
    favoriteAlienLabel.text = defaults.string(forKey: favoriteAlienKey)
}
```

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

Вряд ли в этом коде есть что-нибудь для вас непонятное. Новый метод, `refreshFields()`, лишь собирает пользовательские настройки по умолчанию и устанавливает в свойстве `text` всех меток соответствующий объект из пользовательских настроек по умолчанию, используя значения ключей, введенные нами ранее в файл списка свойств с расширением `.plist`. Обратите внимание, что для ссылки `warpFactorLabel` вызывается метод `stringValue()` из возвращаемого объекта. Все остальные глобальные параметры настройки представлены символьными строками, которые возвращаются из пользовательских настроек по умолчанию в виде объектов типа `String`. Но глобальный параметр настройки, сохраняемый ползунком, возвращается в виде объекта типа `NSNumber`, поэтому для него вызывается метод `stringValue()`, чтобы получить строковое представление числового значения, которое он содержит.

После этого переопределяется метод `viewDidAppear()` из суперкласса, а в нем вызывается метод `refreshFields()`. Благодаря этому значения, отображаемые для пользователя, обновляются всякий раз, когда появляется представление. Это происходит при запуске данного приложения на выполнение и возврате пользователя из второй вкладки к первой.

Запустив теперь данное приложения на выполнение, вы должны увидеть пользовательский интерфейс, созданный вами для первой вкладки, хотя некоторые или все поля в нем окажутся пустыми. Не волнуйтесь, это не программная ошибка, а правильное поведение. Причины и порядок устранения этого недостатка обсуждаются ниже, в разделе “Регистрация значений по умолчанию”.

Изменение пользовательских настроек по умолчанию непосредственно из приложения

Поскольку главное представление находится в рабочем состоянии, займемся построением пользовательского интерфейса для второй вкладки. Как показано на рис. 12.30, на второй вкладке находятся переключатель `Warp Engines` и ползунок `Warp Factor`. Воспользуемся теми же самыми элементами управления, что и в приложении `Settings`, для построения переключателя и ползунка. Помимо входов, мы должны объявить метод `refreshFields()` таким же образом, как это уже было сделано в классе `FirstViewController`, а также два метода, которые будут вызываться при касании этих элементов пользовательского интерфейса.

Выберите исходный файл `SecondViewController.swift` и внесите в него изменения, выделенные ниже полужирным шрифтом.

```
class SecondViewController: UIViewController {
    @IBOutlet var engineSwitch: UISwitch!
    @IBOutlet var warpFactorSlider: UISlider!
```

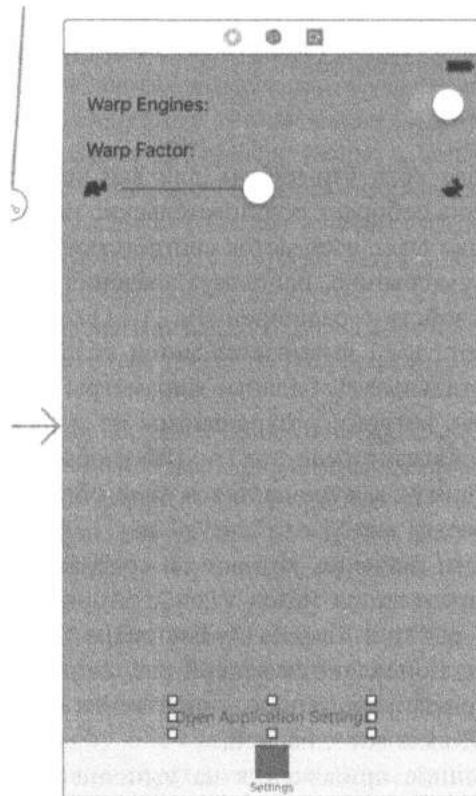


Рис. 12.30. Построение контроллера представления второй вкладки в программе Interface Builder

Сохраните внесенные вами изменения и выберите файл раскадровки Main Storyboard.storyboard, чтобы отредактировать графический пользовательский интерфейс в программе Interface Builder. На этот раз мы уделим основное внимание сцене настроек (Settings Scene) в окне Document Outline. Нажмите клавишу **<Option>** и щелкните на раскрывающем треугольнике, чтобы развернуть элемент **Settings Scene** и все расположенные ниже элементы. Найдите элемент **View** и удалите все его дочерние узлы. Выберите элемент **View** и откройте окно инспектора атрибутов. Измените цвет фона, выбрав вариант **Light Gray Color** из раскрывающегося списка **Background**.

Далее перетащите две метки из библиотеки стандартных объектов в раскадровку. Перетащите их на контроллер сцены **Settings Scene**, расположенный справа внизу на раскадровке. Дважды щелкните сначала на одной из меток, изменив ее текст на **Warp Engines:**, а затем на другой метке, изменив ее текст на **Warp Factor:**. В качестве образца можете пользоваться рис. 12.30.

Перетащите переключатель из библиотеки стандартных объектов в правую часть текущего представления, разместив его напротив метки **Warp Engines:**.

Нажав клавишу <Control>, перетащите указатель от пиктограммы View Controller в верхней части сцены Settings Scene к новому переключателю, а затем соедините его с выходом engineSwitch. После этого откройте исходный файл SecondViewController.swift в помощнике редактора, нажмите клавишу <Control> и перетащите указатель от переключателя в точку, расположенную чуть выше строки кода @end в конце данного файла. Отпустите кнопку мыши и создайте действие engineSwitchTapped, оставив без изменения все остальные значения, выбираемые в открывшемся окне по умолчанию.

Перетащите ползунок из библиотеки стандартных объектов, расположив его ниже метки Warp Factor:. Измените размеры ползунка таким образом, чтобы он занял весь промежуток между двумя голубыми направляющими линиями слева и справа. Нажав клавишу <Control>, перетащите указатель от пиктограммы View Controller в верхней части сцены Scene приложения Settings к ползунку, а затем соедините его с выходом warpFactorSlider. Снова нажмите клавишу <Control> и перетащите указатель от ползунка в конец исходного файла SecondViewController.swift. Отпустите кнопку мыши и создайте действие warpSliderTouched, оставив без изменения все остальные значения, выбираемые в открывшемся окне по умолчанию.

Щелкните на ползунке, если он еще не выбран, и откройте окно инспектора атрибутов. Установите в поле Minimum значение 1.00, в поле Maximum — значение 10.00, а в поле Current — значение 5.00. Затем выберите вариант turtle из раскрывающегося списка Min Image и вариант rabbit из раскрывающегося списка Max Image. Если эти варианты отсутствуют в соответствующих раскрывающихся списках, перетащите файлы изображений в папку Assets.xcassets.

Для того чтобы завершить пользовательский интерфейс второй вкладки, перетащите кнопку из библиотеки стандартных объектов, опустив ее у нижнего края текущего представления, а затем переименуйте ее на Open Settings Application. Нажмите клавишу <Control>, перетащите указатель от новой кнопки к строке кода, следующей сразу за определением метода onWarpSliderTouched в исходном файле SecondViewController.swift, а затем создайте действие onSettingButtonTapped. Мы еще вернемся к данной кнопке в конце этой главы.

Настало время установить ограничения Auto Layout. Сначала выберите файл раскладовки Main.storyboard, затем перейдите в окно Document Outline, нажмите клавишу <Control>, перетащите указатель от метки Warp Engines к ее родительскому представлению и отпустите кнопку мыши. Нажмите клавишу <Shift> и выберите команды Leading Space to Container Margin и Vertical Spacing to Top Layout Guide из всплывающего меню, а затем нажмите клавишу <Return>, чтобы наложить выбранные ограничения. Повторите эту же процедуру для метки Warp Factor.

Нажмите клавишу <Control>, перетащите указатель от переключателя к представлению Main View и отпустите кнопку мыши. Нажмите клавишу <Shift>

и выберите команды **Trailing Space to Container Margin** и **Vertical Spacing to Top Layout Guide** из всплывающего меню, а затем нажмите клавишу <Return>. Еще раз нажмите клавишу <Control>, перетащите указатель от ползунка к представлению Main View и отпустите кнопку мыши. Нажмите клавишу <Shift> и выберите команды **Leading Space to Container Margin**, **Trailing Space to Container Margin** и **Vertical Spacing to Top Layout Guide**, а затем нажмите клавишу <Return>.

Наконец нужно зафиксировать положение кнопки в нижней части представления. С этой целью нажмите клавишу <Control>, перетащите указатель от кнопки к представлению Main View и отпустите кнопку мыши. Нажмите клавишу <Shift> и выберите команды **Vertical Spacing to Bottom Layout Guide** и **Center Horizontally in Container**, а затем нажмите клавишу <Return>. На этом процесс создания ограничений Auto Layout заканчивается. Теперь выберите контроллер представления в окне Document Outline и выполните команду **Editor⇒Resolve Auto Layout Issues⇒Update Frames** в меню Xcode. Убедитесь, что представление содержит все необходимые элементы.

Теперь завершим создание контроллера представления настроек. Откройте файл `SecondViewController.swift` и введите в нем код представленный в листинге 12.4.

Листинг 12.4. Методы `refreshFields` и `viewWillAppear` для контроллера `SecondViewController`

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

func refreshFields() {
    let defaults = UserDefaults.standard
    engineSwitch.isOn = defaults.bool(forKey: warpDriveKey)
    warpFactorSlider.value = defaults.float(forKey: warpFactorKey)
}
```

Далее введите следующие строки кода в методы `onEngineSwitchTapped()` и `onWarpSliderTouched()`.

```
@IBAction func onEngineSwitchTapped(_ sender: AnyObject) {
    let defaults = UserDefaults.standard
    defaults.set(engineSwitch.isOn, forKey: warpDriveKey)
}
@IBAction func onWarpSliderDragged(_ sender: AnyObject) {
    let defaults = UserDefaults.standard
    defaults.set(warpFactorSlider.value, forKey: warpFactorKey)
}
```

Как только появляется представление данного контроллера (например, после выбора вкладки), вызывается метод `refreshFields()`. В трех строках кода из тела этого метода сначала получается ссылка на пользовательские настройки по умолчанию, а затем выходы к переключателю и ползунку используются для

отображения в этих элементах управления значений, сохраняемых в пользовательских настройках по умолчанию. Мы также реализовали методы действия `onEngineSwitchTapped()` и `onWarpSliderTouched()`, чтобы заполнить пользовательские настройки значениями из этих элементов управления после того, как пользователь изменит их.

Запустив теперь данное приложение на выполнение, перейдите на вторую вкладку, откорректируйте значения представленных на ней параметров настройки и понаблюдайте за тем, как внесенные вами коррективы отразятся на содержимом первой вкладки после возврата к ней.

Регистрация значений по умолчанию

Мы создали ранее пакет настроек, включая настройки по умолчанию для некоторых значений, чтобы предоставить приложению `Settings` доступ к глобальным параметрам настройки рассматриваемого здесь приложения. Мы также настроили приложение `Bridge Control` на доступ к этой же информации через графический пользовательский интерфейс, чтобы пользователь мог ее видеть и править. Но мы упустили из виду следующее обстоятельство: данному приложению ничего неизвестно о значениях, задаваемых в пакете настроек по умолчанию. В этом можно убедиться, удалив приложение `Bridge Control` из симулятора или мобильного устройства iOS, на котором оно выполнялось, а следовательно, и хранящиеся там глобальные параметры настройки данного приложения, после чего запустив его снова в среде Xcode. При повторном запуске для всех настроек приложение выведет на экран пустые поля. Исчезнут даже значения, задаваемые по умолчанию для двигателя искривления пространства, которые мы определили в пакете настроек. Перейдя к приложению `Settings`, вы обнаружите значения по умолчанию, но если вы не измените эти значения в приложении `Settings`, то никогда не увидите их в приложении `Bridge Control`.

Причина, по которой настройки по умолчанию исчезают, заключается в том, что данному приложению ничего неизвестно о пакете настроек, который их содержит. Именно поэтому, когда приложение пытается прочитать значение по ключу `warpFactor` из объекта типа `UserDefaults` и не находит по этому ключу никакой сохраненной информации, оно ничего не выводит на экран. Правда, в классе `UserDefaults` имеется метод `register()`, позволяющий указать значения по умолчанию, которые должны быть обнаружены, если пара “ключ–значение” не была задана. С этой целью данный метод лучше всего вызывать при запуске приложения на выполнение. Итак, выберите исходный файл `AppDelegate.swift` и внесите в тело метода `application(_:didFinishLaunchingWithOptions:)` изменения, выделенные ниже полужирным шрифтом.

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) ->
Bool {
    // Точка замещения для настройки приложения после запуска
    let defaults = [warpDriveKey: true, warpFactorKey: 5,
favoriteAlienKey: "Vulcan"]
```

```
UserDefaults.standard.register(defaults)
return true
}
```

Сначала в данном методе создается словарь, содержащий три пары “ключ–значение”: по одной на каждый ключ, доступный в приложении `Settings` и требующий значение по умолчанию. Определенные раньше имена ключей используются здесь для того, чтобы исключить опечатки. Затем весь словарь передается методу `register()` стандартного экземпляра класса `UserDefault`s. С этого момента данный экземпляр будет предоставлять значения, заданные здесь по умолчанию, при условии, что другие значения не были заданы ни в данном приложении, ни в приложении `Settings`.

На этом разработка рассматриваемого здесь приложения завершается. Скомпилируйте и запустите его на выполнение. Оно должно выглядеть так, как показано на рис. 12.6, за исключением, разумеется, значений, которые вы ввели в приложении `Settings`.

ЗАМЕЧАНИЕ. Для того чтобы удалить приложение, щелкните на его пиктограмме на главном экране устройства и удерживайте, пока пиктограмма не начнет покачиваться. Затем отпустите кнопку мыши и щелкните на крестике в левом верхнем углу пиктограммы приложения.

Суровая действительность

Запустите данное приложение на выполнение, просмотрите его настройки, а затем нажмите главную кнопку и откройте приложение `Settings`, чтобы настроить значения некоторых параметров. Еще раз нажмите главную кнопку, снова запустите данное приложение на выполнение, и здесь, возможно, вас ждет неприятный сюрприз. Вернувшись в данное приложение, вы не увидите измененных настроек. Они остались такими, какими и были, отражая прежние значения.

Дело в том, что в системе iOS нажатие кнопки `Home` при активном приложении не означает завершение этого приложения. Операционная система в таком случае переводит приложение в фоновый режим, оставляя его готовым к новой активизации. Эта особенность системы iOS отлично подходит для переключения между приложениями, поскольку время, требующееся для активизации приостановленного приложения, намного короче по сравнению с тем временем, которое понадобилось бы для его запуска с самого начала. Но в данном случае нам нужно выполнить доработки, чтобы после “пробуждения” приложение получало “встряску”, перезагружало пользовательские настройки и отображало содержащиеся в них значения.

Подробнее о приложениях, работающих в фоновом режиме, речь пойдет в главе 15, а пока рассмотрим лишь самые основы того, как уведомить приложение, что оно снова активизировано. Для этого зарегистрируем все классы контроллеров на получение уведомления, которое посыпается приложением при его выходе из состояния приостановленного выполнения.

Уведомление представляет собой упрощенный механизм, которым могут пользоваться объекты для общения. Любой объект может определить одно или несколько уведомлений, которые он будет передавать в центр уведомлений данного приложения. Этот центр представляет собой объект, существующий лишь в одном экземпляре и только для того, чтобы передавать уведомления между объектами. Уведомление можно рассматривать как признак того, что произошло некоторое событие. Объекты, публикующие уведомления, включают в свою документацию список возможных уведомлений. Например, объект типа `UIApplication` публикует разные уведомления (их можно найти внизу страницы из документации на Xcode, где описывается класс `UIApplication`). Назначение большинства уведомлений можно, как правило, определить по их именам, а за дополнительными сведениями можно обратиться к соответствующей документации.

Рассматриваемое здесь приложение должно обновлять изображение перед своей активизацией, поэтому нас интересует уведомление `UIApplicationWillEnterForegroundNotification`. Мы изменим метод `viewWillAppear()`, чтобы зарегистрироваться на получение данного уведомления и указать центру уведомлений на необходимость вызвать другой метод, когда данное уведомление поступит. Итак, введите в исходные файлы `FirstViewController.swift` и `SecondViewController.swift` следующий код.

```
func applicationWillEnterForeground(notification:NSNotification) {
    let defaults = UserDefaults.standard
    defaults.synchronize()
    refreshFields()
}
```

Сам по себе этот метод довольно прост. Он получает ссылку на объект пользовательских настроек по умолчанию и вызывает метод `synchronize()`, вынуждающий механизм `User Defaults` сохранять любые несохраненные изменения и перезагружать любые неизмененные настройки из оперативной памяти. По существу, мы вынуждаем эту систему повторно считывать сохраненные настройки, чтобы иметь возможность собрать все изменения, внесенные в приложении `Settings`. Затем вызывается метод `refreshFields()`, используемый в каждом классе для обновления изображения.

Теперь нам нужно организовать регистрацию каждого контроллера для получения интересующего нас уведомления. С этой целью введите выделенные ниже полужирным строки кода в теле метода `viewWillAppear`: в файлах `FirstViewController.swift` и `SecondViewController.swift`.

```
let app = UIApplication.shared()
NotificationCenter.default.addObserver(self, selector:
#selector(self.applicationWillEnterForeground(notification:)),
name: Notification.Name.UIApplicationWillEnterForeground,
object: app)
```

Сначала мы получаем ссылку на экземпляр рассматриваемого здесь приложения, а затем используем ее для регистрации на получение уведомления

`UIApplicationWillEnterForegroundNotification` с помощью стандартного экземпляра класса `NSNotificationCenter` и метода `addObserver(_ :selector:name:object:)`. Этому методу передаются следующие данные.

- В качестве первого аргумента (`observer`) передается ссылка `self`, обозначающая, что класс контроллера сам является объектом, который должен быть уведомлен (для каждого класса контроллера это делается в отдельности).
- В качестве второго аргумента (`selector`) передается селектор только что написанного метода `applicationWillEnterForeground()`, предписывая центру уведомлений вызвать этот метод, когда уведомление будет отправлено.
- В качестве третьего параметра (`applicationWillEnterForeground`) передается имя уведомления, в получении которого мы заинтересованы.
- В заключение в качестве последнего аргумента (`app`), передается объект, от которого ожидается уведомление. В качестве этого аргумента передается ссылка на данное приложение. Если же в качестве четвертого аргумента передать пустое значение `nil`, то уведомление `UIApplicationWillEnterForegroundNotification` будет поступать всякий раз, когда любой метод отправляет его.

Итак, мы позаботились об обновлении изображения, но нам нужно подумать и о значениях, которыми заполняются пользовательские настройки по умолчанию, когда пользователь манипулирует элементами управления в данном приложении. Мы должны обеспечить надежное сохранение этих значений в хранилище, прежде чем управление будет передано другому приложению. Для этого проще всего вызвать метод `synchronize`, как только настройки изменятся, введя выделенную ниже полужирным шрифтом строку кода в каждом из новых методов действия в исходном файле `SecondViewController.swift`.

```
@IBAction func onEngineSwitchTapped(_ sender: AnyObject) {
    let defaults = UserDefaults.standard
    defaults.set(engineSwitch.isOn, forKey: warpDriveKey)
    defaults.synchronize()
}
@IBAction func onWarpSliderDragged(_ sender: AnyObject) {
    let defaults = UserDefaults.standard
    defaults.set(warpFactorSlider.value, forKey: warpFactorKey)
    defaults.synchronize()
}
```

ЗАМЕЧАНИЕ. Вызов метода `synchronize()` — потенциально затратная операция, поскольку полное содержимое пользовательских настроек в оперативной памяти подлежит сравнению с тем, что записано в хранилище. Когда приходится иметь дело сразу с большим количеством параметров и требуется гарантировать, что их изменение и запоминание будут происходить синхронно, то лучше всего постараться свести к минимуму обращения к методу `synchronize()`, чтобы полное сравнение выполнялось снова и снова. Но вызов этого метода в ответ на каждое действие пользователя, как это делается в данном случае, заметно не скажется на производительности.

Нужно обратить внимание на еще один момент, чтобы данное приложение работало как следует. Как вам должно быть уже известно, следует освободить оперативную память, выделяемую для переменных, если они больше не нужны, а также выполнить ряд других служебных операций по очистке. Система уведомлений — еще одна область, в которой нужно “после себя убрать”. Для этого достаточно сообщить стандартному экземпляру класса `NotificationCenter`, что получать уведомления больше не требуется. Если в данном случае мы зарегистрировали каждый контроллер представления, чтобы наблюдать за конкретным уведомлением в его методе `viewWillAppear()`, то должны снять контроллер с регистрации в соответствующем методе `viewDidDisappear()`. Поэтому введите в обоих исходных файлах, `FirstViewController.swift` и `SecondViewController.swift`, следующий метод:

```
override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    NotificationCenter.default.removeObserver(self)
}
```

Обратите внимание на то, что снять контроллер с регистрации для получения конкретных уведомлений можно с помощью метода `removeObserver(_:name:object:)`, передав ему те же значения, которые использовались для регистрации наблюдателя за уведомлениями. И все же приведенный выше метод удобен: он может гарантировать, что центр уведомлений совершенно забудет о наблюдателе, причем независимо от того, для получения скольких уведомлений он был зарегистрирован.

Сберите и запустите данное приложение на выполнение и понаблюдайте за тем, что произойдет при переходе из него в приложение `Settings` и обратно. Изменения, внесенные вами в приложении `Settings`, должны теперь сразу отразиться в данном приложении, как только вы к нему вернетесь.

Переключение в приложение `Settings`

Для того чтобы перейти из приложения `Bridge Control` к его настройкам в приложении `Settings`, нужно вернуться к начальному экрану, запустить приложение `Settings` на выполнение, найти элемент `Bridge Control` и выбрать его. Это настолько длительная и неудобная процедура, что во многих приложениях предоставляется свой экран настроек, чтобы пользователь не проходил все эти этапы. Не лучше ли направить пользователя сразу на экран настроек в самом приложении `Settings`? Оказывается, такая возможность существует. Помните о кнопке `Open Settings Application`, введенной нами в контроллере представления типа `SecondViewController` и показанной на рис. 12.30? Мы связали ее с методом `onSettingButtonClicked()` в данном контроллере представления, но не ввели никакого кода в тело этого метода. Добавьте в метод `settingButtonClicked()` следующий код.

```
@IBAction func onSettingsButtonTapped(_ sender: AnyObject) {
    let application = UIApplication.shared()
    let url = URL(string: UIApplicationOpenSettingsURLString)! as URL
    if application.canOpenURL(url) {
        application.open(url, options:["":""], completionHandler: nil)
    }
}
```

В этом коде используется системный URL, хранящийся в константе UIApplicationOpenSettings urlString (она фактически содержит значение app-settings:) для запуска приложения Settings на выполнение непосредственно из контроллера представления. Запустите рассматриваемое здесь приложение на выполнение, перейдите на вторую вкладку и щелкните на кнопке Open Settings Application. Произойдет непосредственный переход к экрану настроек данного приложения, как показано на рис. 12.3. Это значительное усовершенствование. В системе iOS 9 существует еще более удобная возможность. На экране Settings можно увидеть маленькую кнопку, позволяющую вернуться непосредственно в ваше приложение (рис. 12.31). Теперь можно без проблем переходить из приложения Bridge Control в приложение Settings и обратно, изменяя значения и следя за их воздействием на пользовательский интерфейс приложения.

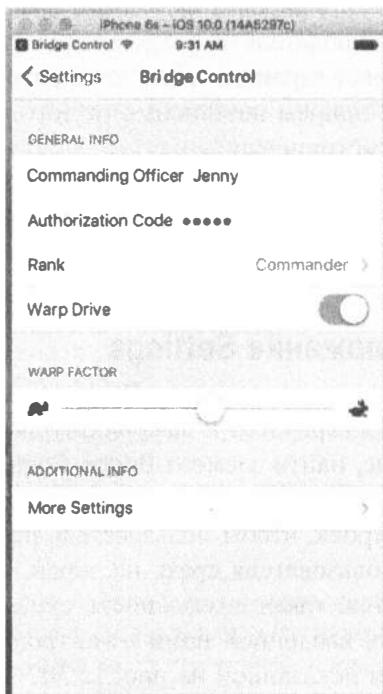


Рис. 12.31. Когда приложение Settings открывается из приложения Bridge Control, на панели состояния появляется кнопка, позволяющая вернуться прямо в ваше приложение

Резюме

Надеемся, вы хорошо освоили приложение `Settings` и механизм `User Defaults`. Теперь вы знаете, как добавить пакет настроек в свое приложение и как создать иерархию представлений. Из этой главы вы также узнали, как считывать и записывать настройки средствами класса `UserDefault`s и как разрешить пользователю изменять значения параметров из своего приложения. Вы даже освоили шаблон нового проекта в среде Xcode. Словом, с таким багажом знаний и навыков вы должны справиться практически с любой задачей, связанной с настройками приложения.

ГЛАВА 13



Основы долговременного хранения данных

До сих пор мы больше уделяли внимания таким аспектам парадигмы MVC (Model-View-Controller), как контроллер и представление. Несмотря на то что некоторые наши приложения считывали данные из специального пакета, только приложение Bridge Control из главы 12 сохраняло данные в постоянном хранилище. Остальные наши приложения при каждом очередном запуске использовали одни и те же данные. До поры до времени такой подход нас устраивал, но в реальном мире приложения должны сохранять данные. Если пользователь внес изменения, то, как правило, он хотел бы, чтобы они не исчезали при повторном запуске программы.

Для сохранения данных на iOS-устройствах существуют различные механизмы. Если вы программировали в объектно-ориентированной среде разработки приложений Соса для системы macOS, то, вероятно, вам приходилось использовать некоторые из этих методов. В этой главе мы рассмотрим четыре механизма, обеспечивающих долговременное хранение данных в файловой системе iOS:

- списки свойств;
- архивы объектов (или архивирование);
- SQLite3 (встроенная реляционная база данных iOS);

Core Data (инструмент для обеспечения долговременного хранения данных, разработанный компанией Apple).

Мы напишем приложения, которые реализуют все четыре подхода.

ЗАМЕЧАНИЕ. Списки свойств, архивы объектов, SQLite3 и Core Data — не единственные возможности сохранения данных в iOS-устройствах. Просто они самые популярные и простые. У вас всегда есть возможность использовать для считывания и записи данных такие традиционные С-функции ввода-вывода, как `fopen()`. Можете также прибегнуть к низкоуровневым инструментам управления файлами, предоставляемым технологией Соса. Однако почти во всех случаях это приведет к увеличению объемов кодирования, которое редко удается оправдать.

“Песочница” приложения

Все четыре механизма обеспечения долговременного хранения данных, рассматриваемые в этой главе, используют один важный общий элемент: папку Documents вашего приложения. У каждого приложения есть собственная папка Documents, которую приложениям разрешается использовать для чтения и записи.

Для более предметного разговора рассмотрим, как приложения организованы в системе iOS, изучив макет папки в симуляторе iPhone. Откройте каталог Library, содержащийся в вашей домашней папке. В системе OS X 10.6 и более ранних версиях это не было проблемой, но в версии OS X 10.7 компания Apple решила сделать папку Library скрытой по умолчанию, поэтому, для того чтобы ее увидеть, придется приложить немного дополнительных усилий. Откройте окно Finder и перейдите к домашней папке. Если вы видите папку Library — отлично. Если же нет, нажмите клавишу <Option> и выполните команду Go⇒Library. Команда Library остается скрытой, пока не нажата клавиша <Option>.

Откройте в папке Library подкаталог Developer/CoreSimulator/Devices. В этом каталоге вы увидите подкаталоги для каждой версии системы iOS, поддерживаемой текущей инсталляцией среды Xcode. Имена этих подкаталогов представляют собой глобально уникальные идентификаторы (GUID), автоматически генерируемые системой Xcode, поэтому по ним невозможно догадаться, какому симулятору они соответствуют. Найдите в каждом подкаталоге файл device.plist и откройте его. В нем вы найдете ключ, соответствующий названию симулируемого устройства. На рис. 13.1 показано содержимое файла device.plist, соответствующего симулятору iPad Pro.

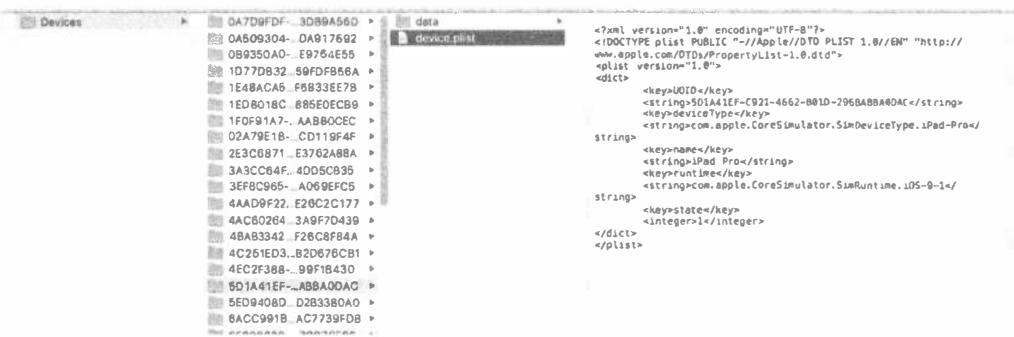


Рис. 13.1. Содержимое файла device.plist, устанавливающее соответствие между каталогом и симулятором

Выберите устройство и откройте каталог data, чтобы найти подкаталог data/Containers/Data/Application. Здесь вы снова обнаружите подкаталоги с глобально уникальными идентификаторами GUID. В этом случае каждое из них соответствует либо заранее инсталлированному приложению, либо

приложению, которое выполнялось на симуляторе. Выберите один из подкаталогов и откройте его. Вы увидите то, что показано на рис. 13.2.

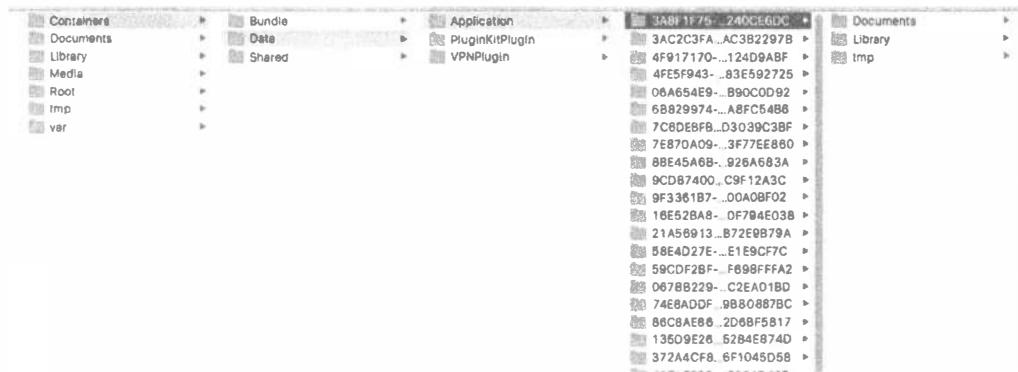


Рис. 13.2. “Песочница” для приложения на симуляторе

ЗАМЕЧАНИЕ. Поиск вложенной папки Containers, начиная с папки Devices, показанной на рис. 13.1, может занять некоторое время. Если вы не видите ее сразу, продолжайте просматривать список подкаталогов с именами GUID сверху вниз, и в конце концов вы найдете папку Containers.

Несмотря на то что этот список представляет симулятор, файловая структура подобна той, которая реализована в реальном устройстве. Для того чтобы увидеть песочницу для приложения на устройстве, подключите его к вашему компьютеру Mac и откройте окно Devices в среде Xcode (*Window*⇒*Devices*). На боковой панели окна вы увидите свое устройство. Выберите его и приложение в таблице Installed Apps. Под таблицей вы увидите пиктограмму в виде шестеренки. Щелкните на ней и выполните команду Show Container из всплывающего меню, чтобы увидеть содержимое песочницы для приложения (рис. 13.3). Все содержимое песочницы можно загрузить на свой компьютер Mac. На рис. 13.4 показана песочница для приложения townslot2 из книги Молли Маккри (*Molly Maskrey*) *App Development Recipes for iOS and watchOS* (Apress, 2016).

“Песочница” каждого приложения содержит три вспомогательные папки.

- Documents. Ваше приложение может сохранять свои данные в папке Documents. Если вы позволяете своему приложению использовать файл медиааплеера iTunes, то пользователь сможет увидеть содержимое этой папки (а также вложенных папок, созданных вашим приложением) в медиаплеере iTunes и загружать туда свои файлы.

ПОДСКАЗКА. Для того чтобы сделать свое приложение совместно используемым, откройте его файл Info.plist и добавьте ключ Application supports iTunes file sharing со значением YES.



Рис. 13.3. “Песочница” для приложения на реальном устройстве

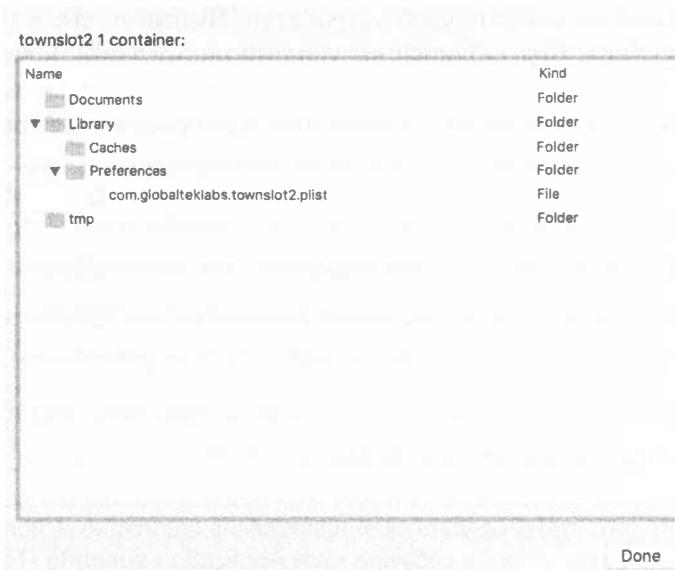


Рис. 13.4. “Песочница” для приложения townslot2 на iPhone 6s

- Library. Это еще одно место, в котором ваше приложение может хранить свои файлы. Здесь целесообразно хранить файлы, которые вы не хотите использовать совместно с пользователем. Как показано на рис. 13.4, система создает подкаталоги Cash и Preferences. В последнем каталоге хранится файл .plist, содержащий настройки приложения, заданные с помощью класса UserDefaults, который мы обсуждали в главе 12.
- tmp. Каталог tmp — это место, в котором ваше приложение может хранить временные файлы. Файлы, которые содержатся в папке tmp, не будут подлежать резервному копированию медиаплеером iTunes в режиме синхронизации, но ваше приложение должно нести ответственность за удаление уже ненужных файлов в каталоге tmp, чтобы избежать перегрузки файловой системы.

Определение местоположения каталогов Documents и Library

Несмотря на то что наше приложение находится в папке, имеющей, казалось бы, произвольно выбранное имя, используя метод urls(for:in:) класса FileManager, можно довольно легко найти полный путь к каталогу Documents для чтения и записи своих файлов. Этот класс принадлежит каркасу Foundation, поэтому его можно использовать в среде Сосоа для системы OS X. Существуют его варианты, предназначенные для системы macOS. Некоторые из них возвращают значения, которые не несут полезной информации для системы iOS, потому что ваше приложение не имеет прав доступа к каталогу из-за механизма обеспечения безопасности, именуемого *песочницей*. В листинге 13.1 показан пример кода на языке Swift 3, обеспечивающего доступ к каталогу Documents.

Листинг 13.1. Код для получения указателя NSURL, ссылающегося на каталог Documents

```
let urls = FileManager.default.urls(for:
    .documentDirectory, ins: .userDomainMask)
if let documentUrl = urls.first {
    print(documentUrl)
```

Первый аргумент метода urlsForDirectory(_:in:) указывает, какой каталог мы ищем. Все возможные значения определяются перечислением searchPathDirectory; в данном случае используется значение SearchPathDirectory.documentDirectory (сокращенно — .documentDirectory). Это значит, что мы ищем каталог Documents. Второй аргумент определяет домен или домены (в документации компании Apple они называются domainMask), которые используются в поиске. Все возможные значения доменов определяются в перечислении SearchPathDomainMask. В данном случае мы используем значение .userDomainMask. В системе iOS этот домен соответствует песочнице запущенного приложения. Метод urls(for: in:) возвращает массив, содержащий один или несколько указателей URL, соответствующих искомым каталогам в указанном домене. В системе iOS каждому приложению назначается только

один каталог Documents, поэтому разумно предположить, что метод вернет только один объект класса NSURL, но для подстраховки мы будем обращаться к нему как к первому элементу массива указателей класса NSURL, чтобы распознать случай, если этот массив окажется пустым. На реальном устройстве iOS указатель URL каталога Documents может выглядеть примерно так: file:///var/mobile/Containers/Data/Application/69BFDB0-E4A8-4359-8382-F6DDDF031481/Documents/.

Мы можем создать имя файла в каталоге Directory, присоединив другую строку к концу пути, который только что получили. Воспользуемся методом appendingPathComponent() из класса NSURL, разработанным как раз для таких целей.

```
let fileUrl = try documentUrl.appendingPathComponent("theFile.txt")
```

ЗАМЕЧАНИЕ. Обработка ошибок в языке Swift 3 осуществляется так же, как и в других языках, использующих ключевые слова try, catch и throw.

После этого вызова переменная fileURL будет содержать полный путь (см. листинг 13.2) к файлу theFile.txt, расположенному в каталоге Documents нашего приложения, и мы сможем использовать этот указатель для создания файла и выполнения операций чтения и записи. Следует подчеркнуть, что для получения объекта класса NSURL, соответствующего этому файлу, существование самого файла не обязательно.

Листинг 13.2. Предыдущий метод с первым аргументом

.libraryDirectory размещает приложение в каталоге Library

```
let urls = FileManager.default.urls(for:
    .libraryDirectory, in: .userDomainMask)
if let libraryUrl = urls.first {
    print(libraryUrl)
}
```

Этот код возвращает указатель URL, имеющий примерно такой вид:

```
file:///var/mobile/Containers/Data/Application/69BFDB0-E4A8-4359-8382-F6DDDF031481/Library/.
```

Можно также задать не один, а несколько доменов поиска. В этом случае класс FileManager ищет каталог во всех указанных доменах и может возвращать несколько объектов класса NSURL. По причинам, которые мы уже объясняли, в системе iOS это не очень полезная возможность, но для полноты картины целесообразно рассмотреть пример, приведенный в листинге 13.3.

Листинг 13.3. Получение нескольких указателей URL

```
let urls = FileManager.default.urls(for:
    .libraryDirectory, in: [.userDomainMask, .systemDomainMask])
print(urls)
```

В этом примере мы просим класс `FileManager` найти каталог `Library` в пользовательском и системном доменах, а результат вернуть в виде массива, содержащего два объекта класса `NSURL`:

```
file:///var/mobile/Containers/Data/Application/69BFDD0-E4A8-4359-8382-F6DDDF031481/Library/;
file:///System/Library/.
```

Второй указатель URL ссылается на системный каталог `Library`, к которому мы, конечно, не имеем доступа. Если метод возвращает несколько указателей URL, порядок их следования в возвращаемом массиве не определен.

Обратите внимание на то, как мы записали значение аргумента `inDomains` в листинге 13.3.

```
[.userDomainMask, .systemDomainMask]
```

Эта конструкция выглядит как инициализация массива, но на самом деле мы создаем множество — синтаксические конструкции для инициализации массива и множества в языке Swift совпадают.

Определение местоположения каталога `tmp`

Получить ссылку на каталог `tmp` вашего приложения даже проще, чем найти ссылку на каталог `Documents`. Для этого воспользуемся методом `NSTemporaryDirectory()` из каркаса `Foundation`, который возвращает строку, содержащую полный путь к каталогу, предназначенному для хранения временных файлов в вашем приложении. Для того чтобы создать полное имя файла, намеченного для хранения во временном каталоге, сначала отыщем путь к каталогу `tmp`:

```
let tempDirPath = NSTemporaryDirectory()
```

Затем преобразуем строку в URL и создадим путь к файлу во временном каталоге, присоединив имя файла к концу найденного пути, как показано в листинге 13.4.

Листинг 13.4. Добавление пути к URL

```
let tempDirUrl = NSURL(fileURLWithPath: tempDirPath)
let tempFileUrl = tempDirUrl.appendingPathComponent("tempFile.txt")
```

Получившийся указатель URL будет выглядеть примерно так:

```
file:///private/var/mobile/Containers/Data/Application/29233884-23EB-4267-8CC9-86DCD507D84C/tmp/tempFile.txt
```

Стратегии сохранения файлов

Все четыре подхода, которые мы рассмотрим в этой главе, используют файловую систему iOS. При использовании механизма SQLite3 создается один файл базы данных SQLite3, которой поручается хранение и поиск ваших данных.

В самой простой форме механизм Core Data берет на себя все функции управления файловой системой. Что касается двух остальных механизмов — списков свойств и архивирования, — то в этом случае вам сначала придется решить, как именно вы собираетесь хранить данные: в одном файле или в нескольких.

Долговременное хранение одного файла

Использование одного файла — самый простой вариант, и для многих приложений он вполне приемлем. Вы начинаете с создания корневого объекта — обычно это объект типа `Array` или `Dictionary` (при использовании архивирования ваш корневой объект может также базироваться на некотором пользовательском классе). Затем наполняете свой корневой объект всеми данными, для которых необходимо обеспечить долговременное хранение. Всякий раз, когда вам нужно что-либо сохранить, ваш код перезаписывает все содержимое этого корневого объекта в один-единственный файл. При запуске приложение считывает полное содержимое этого файла в память, а по завершении работы записывает из памяти в файл. Такой подход мы и будем использовать в данной главе.

Недостаток использования одного файла состоит в том, что вам приходится загружать все данные своего приложения в память и записывать все эти данные в файловую систему даже при небольших изменениях. Однако если ваше приложение собирается управлять данными в объеме, не превышающем несколько мегабайтов, то этот вариант прекрасно работает, а его простота, определенно, облегчит вам жизнь.

Долговременное хранение нескольких файлов

Альтернативный подход состоит в использовании нескольких файлов для хранения данных. Например, приложение электронной почты может сохранять каждое электронное сообщение в отдельном файле.

Этот вариант обладает явными преимуществами. Он позволяет приложению загружать только те данные, которые затребует пользователь (еще одна форма “ленивой” загрузки), и, если пользователь внесет изменение, приложению придется сохранять только те файлы, содержимое которых было изменено. Этот метод также дает вам возможность освобождать память при получении уведомления о нехватке памяти. Любой объем памяти, используемый для хранения данных, с которыми пользователь в данный момент не работает, можно выгрузить, а затем снова загрузить из файловой системы, когда появится такая необходимость. К недостаткам механизма долговременного хранения нескольких файлов можно отнести существенное повышение уровня сложности приложения. Поэтому пока ограничимся однофайловым вариантом.

В дальнейшем мы обязательно рассмотрим особенности всех упомянутых выше методов обеспечения долговременного хранения: списков свойств, архивирования объектов, а также механизмов `SQLite3` и `Core Data`. При изучении каждого варианта будем создавать приложение, использующее соответствующий механизм сохранения данных в файловой системе устройства. Начнем со списков свойств.

Использование списков свойств

В некоторых наших примерах приложений уже использовались списки свойств, например для определения настроек пользователя в главе 12. С такими списками свойств очень удобно работать. Их можно отредактировать вручную в среде Xcode или с помощью приложения Property List Editor. Для записи в списки свойств и считывания их оттуда можно использовать экземпляры классов Dictionary и Array, поскольку словарь или массив содержит только *специальным образом* сериализуемые объекты.

Сериализация списка свойств

Сериализованный объект (*serialized object*) — это объект, преобразованный в поток байтов, чтобы его можно было сохранить в файле или передать по сети. Несмотря на то что любой объект можно сделать сериализованным, только определенные объекты можно поместить в такие классы коллекций, как NSDictionary или NSArray, а затем сохранить в списке свойств с помощью методов `writeToURL(_:atomically:)` или `writeToFile(_:atomically:)`. Таким способом можно сериализовать объекты следующих классов:

- Array или NSArray;
- NSMutableArray;
- Dictionary или NSDictionary;
- NSMutableDictionary;
- NSData;
- NSMutableData;
- String или NSString;
- NSMutableString;
- NSNumber;
- Date или NSDate.

Если вы построите свою модель данных только из таких объектов, то для сохранения и загрузки своих данных сможете использовать списки свойств.

ЗАМЕЧАНИЕ. Методы `writeToURL(_:atomically:)` и `writeToFile(_:atomically:)` делают одно и то же, только первый метод требует, чтобы местоположение файла задавалось объектом класса NSURL, в второй — String. Раньше местоположение файлов всегда задавалось в виде строки, но недавно компания Apple стала отдавать предпочтение классу NSURL, за исключением ситуаций, в которых интерфейс прикладного программирования требует указания пути. В нашей книге мы так и поступаем. Путь к файлу, местоположение которого задано объектом класса NSURL, можно легко получить, обратившись к его свойству `path` (см. первый пример в этой главе).

Если для сохранения своих данных вы собираетесь использовать списки свойств, то выбирайте между классами `Array` и `Dictionary`. Предположим, все объекты, которые вы помещаете в массив типа `Array` или `Dictionary`, являются сериализованными объектами из приведенного выше списка, тогда вы можете написать список свойств, вызвав метод `write(to url:URL, atomically:Bool) -> Bool` из экземпляра словаря или массива, как показано в листинге 13.5.

Листинг 13.5. Добавление пути к URL

```
let array: NSArray = [1,2,3]
let tempDirPath = NSTemporaryDirectory()
let tempDirUrl = NSURL(fileURLWithPath: tempDirPath)
let tempFileUrl = tempDirUrl.appendingPathComponent("tempFile.txt")
array.write(to: tempFileUrl!, atomically:true)
```

ЗАМЕЧАНИЕ. Параметр `atomically` предписывает этому методу записывать данные во вспомогательный файл, а не в заданный. Если запись в этот файл выполнилась успешно, вспомогательный файл будет скопирован по адресу, заданному первым параметром. Это более безопасный способ записи данных в файл, поскольку в случае, если приложение аварийно завершится во время сохранения, существующий файл (если таковой был создан) не будет поврежден. Конечно, такой подход увеличивает непроизводительные расходы, но в большинстве ситуаций такая игра стоит свеч.

При использовании списка свойств возникает одна проблема, связанная с тем, что *пользовательские объекты* невозможно сериализовать для сохранения в списке свойств. Вы также не сможете использовать другие классы из Соса Touch, которые не указаны в приведенном выше списке сериализованных объектов, а это означает, что такие классы, как `NSURL`, `UIImage` и `UIColor`, нельзя использовать напрямую.

Помимо проблемы с сериализацией, хранение всей модели данных в форме списков свойств означает, что вы не сможете легко создавать производные или вычисляемые свойства (как, например, свойство, представляющее собой сумму двух других свойств), а некоторые части кода, которые вы предполагали включить в классы моделей, необходимо перенести в ваши классы контроллеров. С этими ограничениями можно было бы смириться при построении простых моделей данных и простых приложений. Однако в большинстве случаев вашим приложением будет легче управлять, если вы создадите специализированные классы моделей.

Простые списки свойств могут быть полезными и в сложных приложениях, поскольку это прекрасный способ включить статические данные в свое приложение. Например, если ваше приложение содержит селектор, то зачастую наилучший способ включения для него списка элементов — создать файл списка свойств и поместить его в папку `Resources` своего проекта, а это означает, что файл будет скомпилирован в составе вашего приложения.

Теперь создадим простое приложение, которое для хранения данных использует списки свойств.

Первая версия приложения Persistence

Мы хотим написать программу, которая позволяет вводить данные в четыре поля редактирования, по завершении приложения сохраняет эти поля в файле .plist, а затем перезагружает эти данные из файла свойств при следующем запуске приложения (рис. 13.5).



Рис. 13.5. Приложение Persistence

ЗАМЕЧАНИЕ. В приложениях этой главы мы не будем тратить время на установку всех мелочей пользовательского интерфейса, как делали это раньше. Нажатие кнопки Return, например, не освободит клавиатуру и не переведет вас к следующему полю. Если же вы хотите добавить такой "шик" в свое приложение (а это будет для вас хорошей практикой), предлагаем сделать это самостоятельно.

В среде Xcode создайте новый проект с помощью шаблона Single View Application и сохраните его под именем Persistence. Этот проект содержит все файлы, которые нам понадобятся для написания приложения. Прежде чем создавать представление с четырьмя полями редактирования, необходимо

создать один выход. Откройте окно навигатора проекта, щелкните на файле ViewController.swift и внесите в него следующие изменения:

```
class ViewController: UIViewController {
    @IBOutlet var lineFields: [UITextField]!
```

Разработка представления для приложения Persistence

Откройте файл Main.storyboard, чтобы отредактировать графический пользовательский интерфейс. В окне редактора Interface Builder вы увидите в окне редактора сцену View Controller. Перетащите из библиотеки элемент Text Field и расположите его у верхней и правой голубых линий разметки. Откройте инспектор атрибутов. Убедитесь, что флајжок Clear When Editing Begins сброшен.

Затем перетащите в окно объект Label и разместите его слева от поля редактирования, используя левую голубую линию разметки, а затем с помощью горизонтальной голубой линии разметки выровняйте его по полю редактирования. Дважды щелкните на метке и измените ее надпись, например, на Line 1:. В заключение измените размер поля редактирования, используя левый маркер изменения размера, чтобы приблизить его к метке. Ориентируйтесь на рис. 13.6. Выберите метку и поле редактирования, нажмите клавишу <Option> и перетащите их вниз, чтобы сделать их копии. Выберите обе метки и оба поля редактирования, нажмите клавишу <Option> и перетащите их снова вниз. У вас должно получиться четыре метки и четыре поля редактирования. Дважды щелкните на оставшихся метках и измените их надписи на Line 2:, Line 3: и Line 4:. Сравните результат с рис. 13.6.

Разместив все четыре поля редактирования и надписи, перетащите указатель мыши при нажатой клавише <Control> с пиктограммы View Controller на каждое из четырех полей редактирования. Связав все четыре поля редактирования с соответствующими выходами, сохраните изменения, внесенные в файл Main.storyboard.

Теперь добавим ограничения Auto Layout, чтобы наш интерфейс работал одинаково на всех устройствах. Проведите соединительную линию от метки Line 1 к правому краю поля редактирования, а затем отпустите кнопку мыши. Нажмите клавишу <Shift> и выберите пункт Horizontal Spacing and Baseline, а затем нажмите клавишу <Return>. Сделайте то же самое для остальных трех меток и полей редактирования.

Затем мы зафиксируем позиции полей редактирования. Перейдите в окно Document Outline, нажмите клавишу <Control>, перетащите указатель от верхнего поля редактирования к ее родительской пиктограмме View, отпустите кнопку мыши, нажмите клавишу <Shift> и выполните команды Trailing Space to Container Margin и Vertical Spacing to Top Layout Guide. Сделайте то же самое для остальных трех полей редактирования.

Нам необходимо зафиксировать ширину меток, чтобы они не изменялись при вводе длинного текста. Выберите верхнюю метку и щелкните на кнопке Pin

под окном редактора раскладовок. Во всплывающем окне установите флажок Width и щелкните на кнопке Add 1 Constraint. Сделайте то же самое для остальных меток.

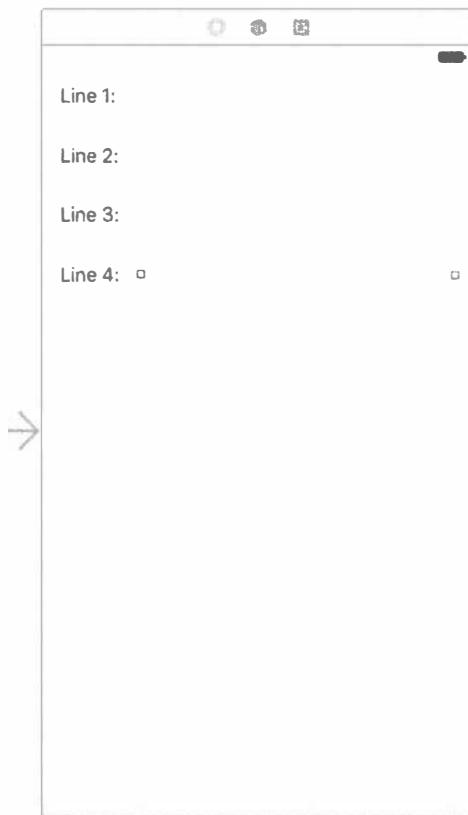


Рис. 13.6. Разработка представления для приложения Persistence

В заключение вернитесь в окно Document Outline, нажмите клавишу <Control>, перетащите указатель от метки Line 1 к ее родительской пиктограмме View, отпустите кнопку мыши и выполните команду Leading Space to Container Margin. Сделайте то же самое для всех меток. Вот и все — требуемые ограничения Auto Layout установлены. Выберите пиктограмму контроллера представления в окне Document Outline и выполните команду Editor⇒Resolve Auto Layout Issues⇒Update Frames в меню Xcode, чтобы удалить предупреждения из представления Activity. Соберите и выполните приложение, а затем сравните результаты с рис. 13.6.

Редактирование классов приложения Persistence

Выберите в навигаторе проекта файл ViewController.swift и добавьте в него код из листинга 13.6.

Листинг 13.6. Получение URL для файла data.plist

```
func dataFileURL() -> NSURL {
    let urls = FileManager.default.urls(for:
        .documentDirectory, in: .userDomainMask)
    var url: NSURL?
    url = URL(fileURLWithPath: "") // создаем пустой путь
    do {
        try url = urls.first!.appendingPathComponent("data.plist")
    } catch {
        print("Error is \(error)")
    }
    return url!
}
```

Метод `dataFileURL()` возвращает полный путь к нашему файлу данных, определяя местоположение каталога `Documents` и присоединяя к нему имя файла. Этот метод должен вызываться из любого кода, который обеспечивает загрузку и сохранение данных. В нем есть небольшой недостаток, связанный с указателем URL. Обратите внимание на то, что мы поместили метод `appendingPathComponent` в блок `do-catch`. Мы сделали это потому, что этот метод может вызвать ошибку, которую необходимо перехватить. Однако, поскольку нам известно, что наш пакет приложения должен содержать каталог `Document`, и к тому же мы сами создаем файл `data.plist`, мы не планируем обработку этой ошибки, потому что код написан правильно. В обычных условиях необходимо уделять вопросам безопасности немного больше внимания, чтобы приложение не потерпело крах, но для краткости мы не будем обрабатывать ошибки, поскольку они не относятся к теме нашего обсуждения.

ЗАМЕЧАНИЕ. В языке Swift для обработки исключений (ошибок) используется блок `do-catch` (его можно найти в библиотеке сниппетов Xcode), который следит за выполнением метода, генерирующего исключение с помощью оператора `throw` и перехватывает исключение с помощью оператора `catch`, чтобы предотвратить крах приложения.

Найдите метод `viewDidLoad()` и добавьте в него следующий код, а также новый метод, получающий уведомление `applicationWillResignActive()`, как показано в листинге 13.7.

Листинг 13.7. Методы `viewDidLoad` и `applicationWillResignActive`

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.
    let fileURL = self.dataFileURL()
    if (FileManager.default.fileExists(atPath: fileURL.path!)) {
        if let array = NSArray(contentsOf: fileURL as URL) as? [String] {
            for i in 0..

```

```

        }
    }

let app = UIApplication.shared()
NotificationCenter.default.addObserver(self, selector:
#selector(self.applicationWillResignActive(notification:)),
name: Notification.Name.UIApplicationWillResignActive,
object: app)
}

func applicationWillResignActive(notification:NSNotification) {
    let fileURL = self.dataFileURL()
    let array = (self.lineFields as NSArray).value(forKey: "text") as!
NSArray
    array.write(to: fileURL as URL, atomically: true)
}

```

В методе `viewDidLoad()` мы выполняем несколько операций. Сначала с помощью метода `fileExists(atPath:)` класса `FileManager` мы проверяем, существует ли заданный файл данных, поскольку приложение ранее могло уже выполняться. Метод запрашивает имя пути к файлу, который можно извлечь из свойства `path` его `URL` (к сожалению, для метода, требующего аргумент класса `URL`, этот вариант не подходит). Если файла нет, то мы не пытаемся его загрузить. Если же файл существует, то мы создаем экземпляр массива, заполняя его содержимым этого файла, а затем копируем объекты из этого массива в наши четыре поля редактирования. Поскольку массивы — это упорядоченные списки, мы копируем их в том же порядке, в каком сохраняли.

Для того чтобы прочитать этот файл, мы используем инициализатор объектов класса `Array`, который создает объект класса `NSArray` на основе содержания файла. Инициализатор объектов класса `Array` требует, чтобы содержание файла имело формат списка свойств. Это хорошо, потому что именно в этом виде мы и будем его хранить.

Наше приложение должно сохранять свои данные до того, как оно завершится, или до перехода в фоновый режим, поэтому мы заинтересованы в уведомлении типа `applicationWillResignActive`. Это уведомление посыпается тогда, когда пользователь больше не собирается взаимодействовать с приложением, например если пользователь нажал кнопку `Home` или приложение перешло в фоновый режим с последующей потенциальной возможностью возврата в активное состояние, например при поступлении входящего звонка. Именно это происходит при регистрации уведомлений, поступающих от центра уведомлений iOS. Этот центр посылает уведомление, вызывая метод, который зарегистрирован для его получения и передавая ему аргумент типа `Notification`, содержащий подробное описание события, вызывавшего появление уведомления. Для того чтобы зарегистрировать это уведомление, мы получаем ссылку на экземпляр нашего приложения и используем ее для подписки на объект класса `UIApplicationWillResignActive`, используя стандартный экземпляр класса `NotificationCenter` и метод `addObserver(_:selector:name:)`.

`object:`). В качестве первого аргумента передается указатель `self`, который означает, что наш экземпляр класса `ViewController` является наблюдателем, ожидающим уведомление. Второй параметр является селектором метода `applicationWillResignActive()`, который приказывает центру уведомлений вызвать данный метод, когда будет опубликовано уведомление. Третий параметр, объект класса `UIApplicationWillResignActive`, является именем уведомления, которое мы хотим получить. Это строковая константа, определенная в классе `UIApplication`.

В заключение мы добавили реализацию метода `applicationWillResignActive()`, который будет вызываться центром уведомлений.

```
func applicationWillResignActive(notification:NSNotification) {
    let fileURL = self.dataFileURL()
    let array = (self.lineFields as NSArray).value(forKey: "text") as! NSArray
    array.write(to: fileURL as URL, atomically: true)
}
```

Этот метод довольно простой, но выполняет большую работу с помощью всего нескольких вызовов методов. Мы создаем массив строк, вызывая метод `text` для полей редактирования, содержащихся в массиве `lineFields`. Для этого мы используем остроумный прием: вместо явного перебора полей редактирования в массиве, запроса их значений и добавления этих значений в новый массив мы приводим массив `lineFields` языка Swift (состоящий из объектов класса `UITextField`) к типу `NSArray` и вызываем его метод `value(forKey:)`, передавая в качестве параметра строку `"text"`. Реализация метода `value(forKey:)` в классе `NSArray` автоматически выполняет перебор элементов массива, запрашивает у объектов класса `UITextField` содержащееся в них значение и возвращает новый объект класса `NSArray`, состоящий из всех этих значений. После этого записываем содержимое этого массива в файл с расширением `.plist`. Осталось только сохранить содержание массива в формате списка свойств, используя метод `write(_ to:atomically)`. На этом процедура сохранения данных в формате списка свойств заканчивается.

Подведем итоги. После загрузки основного представления мы пытаемся найти файл списка свойств. Если он существует, то копируем из него данные в поля редактирования. Затем регистрируемся на получение уведомления о переходе приложения в неактивное состояние (в результате либо завершения работы, либо перевода в фоновый режим). Когда это происходит, собираем значения из четырех полей редактирования, сохраняя их в изменяемом массиве, и записываем этот массив в список свойств.

Скомпилируйте и выполните приложение в симуляторе. После запуска вы должны иметь возможность вводить данные в любое из четырех полей редактирования. После ввода тестовой информации нажмите клавиши `<⌘+Shift+N>`, чтобы вернуться на главный экран. Это очень важно. Если просто выйти из симулятора, это будет эквивалентно принудительному выходу из приложения. В этом

случае вы никогда не получите уведомление о том, что приложение завершилось, и ваши данные не будут сохранены. После возвращения на главный экран можно выйти из симулятора или остановить приложение из среды Xcode и снова запустить его. При повторном запуске приложения текст будет восстановлен.

ЗАМЕЧАНИЕ. Важно помнить, что возвращение на главный экран обычно не завершает приложение, по крайней мере не сразу. Приложение переводится в фоновое состояние и готово немедленно повторно активизироваться в случае, если пользователь снова вернется к нему. Мы углубимся в детали этих состояний и их импликаций для выполнения и завершения приложений в главе 15.

Упорядоченные списки свойств легко использовать, но в них может храниться только ограниченный круг объектов, поэтому стоит рассмотреть более универсальный подход.

Архивирование объектов моделей

В среде Сосоа термин **архивирование** (archiving) относится к другой форме сериализации, но это уже более обобщенный тип, который может реализовать любой объект. Каждый объект модели, специально написанный для хранения данных, должен поддерживать архивирование. Метод архивирования объектов модели позволяет легко записывать сложные объекты в файл, а затем читать их оттуда. Поскольку каждое свойство, которое вы реализуете в своем классе, является либо скаляром (как `Int` или `Float`), либо экземпляром класса, который соответствует протоколу `NSCoding`, можете архивировать все свои объекты. Так как большинство классов из каркасов Foundation и Сосоа Touch, допускающих возможность сохранения данных, действительно соответствует протоколу `NSCoding` (хотя и не без таких заслуживающих внимания исключений, как класс `UIImage`), архивирование и вправду относительно просто реализовать для большинства классов.

Несмотря на отсутствие строгого требования на выполнение архивирования, вместе с протоколом `NSCoding` должен быть реализован другой протокол, а именно — протокол `NSCopying`, который позволяет скопировать ваш объект. Возможность копирования объекта дает намного больше гибкости при использовании объектов моделей данных.

Поддержка протокола `NSCoding`

Протокол `NSCoding` объявляет два метода, и оба они обязательны. Один зашифровывает ваш объект в виде архива, другой создает новый объект путем дешифрования архива. Обоим методам передается экземпляр типа `NSCoder`, который требует практически такой же обработки, как и экземпляр класса `NSUserDefaults`, описанный в предыдущей главе. Вы можете шифровать и расшифровывать как объекты, так и скаляры (переменные типа `Int` и `Float`), используя **кодирование “ключ–значение”**.

Для поддержки архивирования в вашем объекте необходимо, чтобы он относился к подклассу класса `NSObject` (или другого класса, производного от класса `NSObject`), а также следует зашифровывать каждую переменную экземпляра в методе `encoder` с использованием соответствующего метода шифрования.

Допустим, мы создали простой контейнерный класс:

```
class MyObject : NSObject, NSCoding, NSCopying {
    var number = 0;
    var string = "";
    var child: MyObject?

    override init() {
    }
}
```

Этот класс содержит целочисленное и строковое свойства и ссылку на другой экземпляр того же самого класса. Он является производным от класса `NSObject` и поддерживает протоколы `NSCoding` и `NSCopying`. Метод шифрования может выглядеть примерно так:

```
func encode(with aCoder: NSCoder) {
    aCoder.encode(string, forKey: "stringKey")
    aCoder.encode(32, forKey: "intKey")
    if let myChild = child {
        aCoder.encode(myChild, forKey: "childKey")
    }
}
```

Если бы класс `MyObject` был подклассом класса, который также поддерживает протокол `NSCoding`, нам пришлось бы вызвать метод `encode(with aCoder:)` из его суперкласса, чтобы шифрование данных выполнял суперкласс. В этом случае метод выглядел бы следующим образом:

```
func encode(with aCoder: NSCoder) {
    super.encode(with aCoder: NSCoder)
    aCoder.encode(string, forKey: "stringKey")
    aCoder.encode(32, forKey: "intKey")
    if let myChild = child {
        aCoder.encode(myChild, forKey: "childKey")
    }
}
```

Мы также должны создать метод инициализации объекта класса `NSObject`, чтобы можно было восстановить ранее заархивированный объект. Этот метод очень похож на метод `encode(with aCoder:)`. Если вы создаете подкласс типа `NSObject` напрямую или создаете подкласс от другого класса, который не соответствует протоколу `NSCoding`, ваш метод должен выглядеть так:

```
required init?(coder aDecoder: NSCoder) {
    string = aDecoder.decodeObject(forKey: "stringKey") as! String
    number = aDecoder.decodeInteger(forKey: "intKey")
    child = aDecoder.decodeObject(forKey: "childKey") as? MyObject
}
```

Этот метод устанавливает свойства путем дешифрирования значений переданного методу экземпляра типа NSCoder. Поскольку свойство child исходного объекта равно nil, мы должны выполнить дополнительное приведение типа по время присваивания свойства child расшифрованному объекту, поскольку в заархивированном объекте может не оказаться дочернего объекта.

При реализации протокола NSCoding для класса, производного от класса, который тоже поддерживает этот протокол, мы должны добавить одну дополнительную строку.

```
required init?(coder aDecoder: NSCoder) {
    string = aDecoder.decodeObject(forKey: "stringKey") as! String
    number = aDecoder.decodeInteger(forKey: "intKey")
    child = aDecoder.decodeObject(forKey: "childKey") as? MyObject
    super.init(code: aDecoder)
}
```

Вот в целом и все. Если вы реализуете эти два метода для шифрования и дешифрования всех свойств своего объекта, ваш объект будет поддерживать архивирование и его можно будет записывать в архивы и считывать оттуда.

Реализация протокола NSCopying

Как упоминалось выше, соответствие протоколу NSCopying — прекрасная идея для любых объектов моделей данных. Протокол NSCopying содержит метод copy(with zone:), который позволяет копировать объекты. Реализация протокола NSCopying очень напоминает реализацию метода init(coder:). Вам просто необходимо создать новый экземпляр того же самого класса, а затем установить все свойства нового экземпляра равными значениям свойств копируемого объекта. Даже если вы реализуете метод copy(with zone:), код приложения на самом деле будет вызывать метод copy(), который переадресует операцию методу copy(with zone:).

```
let anObject = MyObject()
let objectCopy = anObject.copy() as! MyObject
```

Ниже показано, как может выглядеть метод copy(with zone:) в классе MyObject.

```
func copy(with zone: NSZone? = nil) -> AnyObject {
    let copy = MyObject()
    copy.number = number
    copy.string = string
    copy.child = child?.copy() as? MyObject
    return copy
}
```

Обратите внимание на то, что если бы существовала ссылка на дочерний объект, то новый объект содержал бы копию дочернего, а не базового объекта. Если бы дочерний объект имел неизменяемый тип или нам требовалась только

поверхностная копия объекта, то мы могли просто присвоить новому объекту ссылку на дочерний объект.

ЗАМЕЧАНИЕ. Не стоит слишком беспокоиться о параметре `NSZone`. Он указывает на структуру, которая используется системой для управления памятью. Только в редких случаях разработчики действительно должны были заботиться о зонах или создавать собственные зоны, а сейчас практически никто и не слышал о наличии нескольких зон. Вызов метода `copy` для объекта эквивалентен вызову метода `copy (with zone:)` с использованием зоны по умолчанию, которая почти всегда именно та, которая вам нужна. Факт, что класс `NSCopying` использует зоны, является историческим курьезом, сохраненным для обеспечения обратной совместимости.

Архивирование и разархивирование объектов данных

Создать архив из объекта или объектов, которые соответствуют протоколу `NSCoding`, относительно несложно. Сначала создаем экземпляр типа `NSMutableData` из каркаса Foundation для хранения зашифрованных данных, а затем экземпляр `NSKeyedArchiver`, чтобы архивировать объекты в этот экземпляр типа `NSMutableData`.

```
let data = NSMutableData()
let archiver = NSKeyedArchiver(forWritingWith: data)
```

После создания обоих экземпляров используем кодирование по принципу “ключ–значение” для архивирования любых объектов, которые хотим включить в архив.

```
archiver.encode(anObject, forKey: "keyValueString")
```

После шифрования всех нужных объектов достаточно уведомить архиватор о завершении этой операции и записать экземпляр типа `NSMutableData` в файловую систему.

```
archiver.finishEncoding()
let success = data.write(to: archiveUrl as URL, atomically: true)
```

Если вы чувствуете некоторый дискомфорт от архивирования, не беспокойтесь. В действительности здесь нет ничего сложного. В следующем разделе мы переориентируем приложение `Persistence` на использование метода архивирования данных, и вы увидите, как работает эта теория на практике. После нескольких упражнений архивирование войдет у вас в привычку и станет вашей “второй натурой”, поскольку на самом деле вы всего лишь сохраняете и извлекаете свойства своего объекта с использованием кодирования в стиле “ключ–значение”.

Приложение Archiving

Модифицируем приложение `Persistence` так, чтобы оно использовало вместо списков свойств архивирование. Для этого необходимо внести в исходный код

приложения Persistence довольно существенные изменения, поэтому, прежде чем продолжить, имеет смысл сделать копию проекта, например заархивировав его с помощью метода создания списка свойств в файл PersistencePL.zip.

Реализация класса *FourLines*

Если вы готовы продолжить работу, откройте копию проекта Persistence в среде Xcode, выберите папку Persistence и нажмите комбинацию клавиш **<⌘+N>** или выполните команду **File⇒New⇒File....** В открывшемся окне помощника для создания новых файлов выберите пиктограмму **Swift File** в разделе iOS и щелкните на кнопке **Next**. На следующей странице экрана назовите класс *FourLines.swift*, выберите для сохранения файлов папку Persistence и щелкните на кнопке **Create**. Этот класс будет представлять нашу модель данных. Он будет хранить данные, которые пока содержатся в словаре в списке свойств приложения.

Щелкните на файле *FourLines.swift* и внесите в него код, представленный в листинге 13.8.

Листинг 13.8. Класс *FourLines*

```
class FourLines : NSObject, NSCoding, NSCopying {
    private static let linesKey = "linesKey"
    var lines:[String]?

    override init() {
    }

    required init?(coder aDecoder: NSCoder) {
        lines = aDecoder.decodeObject(forKey: FourLines.linesKey) as?
    [String]
    }

    func encode(with aCoder: NSCoder) {
        if let saveLines = lines {
            aCoder.encode(saveLines, forKey: FourLines.linesKey)
        }
    }

    func copy(with zone: NSZone? = nil) -> AnyObject {
        let copy = FourLines()
        if let linesToCopy = lines {
            var newLines = Array<String>()
            for line in linesToCopy {
                newLines.append(line)
            }
            copy.lines = newLines
        }
        return copy
    }
}
```

Мы реализовали все методы, необходимые для согласования с протоколами NSCoder и NSCopying. Мы шифруем все четыре свойства в методе encode(with aCoder:) и расшифровываем их с использованием тех же самых четырех ключевых значений в методе init(with aCoder:). В методе copy(with zone:) создаем новый объект типа FourLines и копируем в него все четыре строки с помощью глубокого копирования, чтобы изменения оригинала не коснулись нового объекта. Как видите, это несложно. Просто будьте внимательны, выполняя многочисленные операции копирования и вставки.

Реализация класса ViewController

Теперь, когда у нас есть объект архивируемых данных, воспользуемся им для сохранения данных нашего приложения. Выберите файл ViewController.swift и внесите в него изменения, указанные в листинге 13.9.

Листинг 13.9. Код для сохранения и извлечения заархивированного объекта в файле ViewController.swift

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.
    let fileURL = self.dataFileURL()
    if (FileManager.default.fileExists(atPath: fileURL.path!)) {
        if let array = NSArray(contentsOf: fileURL as URL) as? [String] {
            for i in 0..<array.count {
                lineFields[i].text = array[i]
            }
        }
        let data = NSMutableData(contentsOf: fileURL as URL)
        let unarchiver = NSKeyedUnarchiver(forReadingWith: data as! Data)
        let fourLines = unarchiver.decodeObject(forKey: ViewController.
rootKey)
            as! FourLines
        unarchiver.finishDecoding()
        if let newLines = fourLines.lines {
            for i in 0..<newLines.count {
                lineFields[i].text = newLines[i]
            }
        }
    }
    let app = UIApplication.shared()
    NotificationCenter.default.addObserver(self, selector:
        #selector(self.applicationWillResignActive(notification:)),
        name: Notification.Name.UIApplicationWillResignActive,
        object: app)
}

func applicationWillResignActive(notification:NSNotification) {
    let fileURL = self.dataFileURL()
    let fourLines = FourLines()
    let array = (self.lineFields as NSArray).value(forKey: "text")
        as! [String]
    fourLines.lines = array
```

```

let data = NSMutableData()
let archiver = NSKeyedArchiver(forWritingWith: data)
archiver.encode(fourLines, forKey: ViewController.rootKey)
archiver.finishEncoding()
data.write(to: fileURL as URL, atomically: true)
}

func dataFileURL() -> NSURL {
    let urls = FileManager.default.urls(for:
        .documentDirectory, in: .userDomainMask)
    var url: NSURL?
    url = URL(fileURLWithPath: "") // Создаем пустой путь
    do {
        try url = urls.first!.appendingPathComponent("data.archive")
    } catch {
        print("Error is \(error)")
    }
    return url!
}

```

Сохраните изменения и запустите эту версию приложения Persistence. Сначала мы определили новое имя файла в методе `dataFileURL()`, чтобы программа не пыталась загружать старый список свойств как архив. Кроме того, мы определили новую константу, которая играет роль значения ключа при шифровании и расшифровке объекта. Затем мы переопределили методы загрузки и сохранения данных в классе `FourLines` и использовали методы протокола `NSCoding` для реальной загрузки и сохранения данных. Пользовательский интерфейс идентичен предыдущей версии.

Для этой новой версии потребовалось больше строк кода, чем для предыдущей (со списками свойств), поэтому у вас может возникнуть сомнение: а есть ли какое-то преимущество в использовании архивирования перед списками свойств? Что касается этого приложения, то ответ простой: нет, в данном случае действительно нет никакого преимущества. С другой стороны, если бы у нас был массив архивируемых объектов, например класса `FourLines` (который мы только что построили), то мы могли бы заархивировать весь массив, архивируя сам экземпляр массива. При архивировании такие классы коллекций, как `Array`, архивируют все объекты, которые они содержат. Поскольку каждый объект, который мы поместили в массив или словарь, соответствует протоколу `NSCoding`, можем архивировать этот массив или словарь и восстанавливать его так, чтобы все объекты, которые в нем содержались при архивировании, находились и в восстановленном массиве или словаре.

Иначе говоря, этот подход намного предпочтительнее (по крайней мере, если говорить о размере кода). Независимо от того, сколько объектов вы добавите, объем работы по записи этих объектов на диск (в предположении, что вы используете однофайловый вариант сохранения данных) останется тем же. И наоборот, объем работы при использовании списков свойств увеличивается с каждым добавляемым вами объектом.

Использование встроенной в iOS базы данных SQLite3

Третий вариант обеспечения долговременного хранения данных, который мы рассмотрим, состоит в использовании встроенной в iOS SQL-базы данных SQLite3. База данных SQLite3 отличается высокой эффективностью при сохранении и извлечении больших объемов данных. Она также может выполнять сложные операции по агрегированию данных, причем с гораздо большим быстродействием по сравнению с использованием с той же целью объектов. Например, если ваше приложение должно вычислить сумму значений конкретного поля по всем объектам приложения или если вам нужна была сумма значений только из тех объектов, которые отвечают определенному критерию, база данных SQLite3 справилась бы с этой задачей без загрузки каждого объекта в память. Получение результатов агрегирования из базы SQLite3 происходит на несколько порядков быстрее, чем загрузка всех объектов в память с последующим суммированием их значений. Как полноценная встроенная база данных SQLite3 содержит инструменты, позволяющие повысить ее быстродействие путем создания, например, индексов таблиц, что, безусловно, позволяет ускорить обработку ваших запросов.

В базе данных SQLite3 используется язык структурированных запросов (Structured Query Language — SQL). SQL — стандартный язык, используемый для взаимодействия с реляционными базами данных. Синтаксису SQL (как, впрочем, и SQLite) посвящено множество книг (даже не десятки, а сотни). Поэтому, если вы не знакомы с языком SQL, но хотите использовать базу SQLite3 в своем приложении, вам придется усердно поработать в этом направлении. Мы покажем, как установить базу данных SQLite и взаимодействовать с ней из своих iOS-приложений, и вы найдете в этой главе основы синтаксиса. Но для того чтобы действительно выжать все возможное из базы SQLite3, вы должны приложить дополнительные усилия. Например, можете начать освоение SQL с таких источников, как “An Introduction to the SQLite3 C/C++ Interface” (<http://www.sqlite.org/cintro.html>) и “SQL As Understood by SQLite” (<http://www.sqlite.org/lang.html>).

Реляционные базы данных, включая SQLite3, и языки объектно-ориентированного программирования используют фундаментально различные подходы к хранению и организации данных. Поэтому для преобразования данных между ними разработано множество методов, библиотек и прочих инструментов, которые все вместе получили “коллективное” название **объектно-реляционное отображение** (object-relational mapping — ORM). В настоящее время существует несколько технологических ORM-инструментов, предназначенных для Cocoa Touch. Одно из таких ORM-решений, предоставленное компанией Apple, — механизм Core Data — мы рассмотрим ниже.

Здесь мы остановимся на основных возможностях работы с базой SQLite3, а именно: на ее установке, создании таблицы для хранения данных и использовании базы данных в приложении. Безусловно, в реальности такое простое приложение, как наше, совсем не оправдало бы затраты на поддержание базы SQLite3. Однако именно простота этого приложения позволяет использовать его в качестве хорошего обучающего примера.

Создание или открытие базы данных

Прежде чем вы сможете работать с механизмом SQLite3, вы должны открыть базу данных. Функция, которая используется для этого, `sqlite3_open()`, откроет существующую базу данных, а если в указанном месте ее не окажется, то создаст новую. Код для открытия новой базы данных приведен ниже.

```
var database:OpaquePointer? = nil
let result = sqlite3_open("/path/to/database/file", &database)
```

Если результат окажется равным константе `SQLITE_OK`, значит, база данных была успешно открыта. Обратите внимание на то, что путь к файлу базы данных должен быть передан в виде переменной `database`. В интерфейсе прикладного программирования SQLite3 API эта переменная представляет собой структуру из языка С типа `sqlite3`. Когда в язык Swift был импортирован интерфейс прикладного программирования на языке С, эта переменная превратилась в объект типа `UnsafeMutablePointer<CopaquePointer>`, который в текущей версии языка Swift выражает указатель `void*` из языка С. Это значит, что мы должны работать с ним, как с непрозрачным указателем (opaque pointer). В этом нет никакой проблемы, потому что нам не нужен доступ в внутреннему содержанию этой структуры из кода Swift — нам всего лишь нужно передать этот указатель другим функциям SQLite3, таким как `sqlite3_close()`.

```
sqlite3_close(database)
```

Базы данных хранят все данные в таблицах. Можно создать новую таблицу путем “конструирования” SQL-оператора `CREATE` и передачи его открытой базе данных с помощью функции `sqlite3_exec`:

```
let createSQL = "CREATE TABLE IF NOT EXISTS PEOPLE" +
    "(ID INTEGER PRIMARY KEY AUTOINCREMENT, FIELD_DATA TEXT)"
var errMsg:UnsafeMutablePointer<Int8> = nil
result = sqlite3_exec(database, createSQL, nil, nil, &errMsg)
```

Функция `sqlite3_exec` используется для выполнения любой связанной с SQLite3 команды, которая не возвращает данные, т.е. реализует обновление, вставку и удаление данных. Извлечение информации из базы данных выглядит несколько сложнее. Сначала необходимо подготовить оператор на основе SQL-команды `SELECT`.

```
let createSQL = "SELECT ID, FIELD_DATA FROM FIELDS ORDER BY ROW"
var statement: OpaquePointer? = nil
result = sqlite3_prepare_v2(database, createSQL, -1, &statement, nil)
```

Если результат выполнения функции равен константе `SQLITE_OK`, значит, ваш оператор был успешно подготовлен, и вы можете переходить к обработке результирующего множества. Рассмотрим пример обработки структуры SQLite3 как непрозрачного указателя — в SQLite3 API переменная `statement` имеет тип `sqlite3_stmt`.

Рассмотрим пример обхода результирующего множества и извлечения из базы данных переменных типа `Int` и `String`.

```
while sqlite3_step(statement) == SQLITE_ROW {
    let row = Int(sqlite3_column_int(statement, 0))
    let rowData = sqlite3_column_text(statement, 1)
    let fieldValue = String.init(cString: UnsafePointer<CChar>(rowData!))
    lineFields[row].text = fieldValue!
}
sqlite3_finalize(statement)
```

Здесь мы снова должны установить мост между требованиями интерфейса прикладного программирования, написанного на языке C, и механизмами поддержки языка Swift. В этом случае функция `sqlite3_column_text()` возвращает значение типа `const unsigned char *`, которое компилятор языка Swift переводит в тип `UnsafePointer<UInt8>`. Нам необходимо создать объект типа `String` из возвращаемых символьных данных, и мы можем это сделать с помощью метода `String.init(cString:) (UnsafePointer<CChar>)`. Вместо объекта типа `UnsafePointer<CChar>` у нас есть объект типа `UnsafePointer<UInt8>`, но, к счастью, существует инициализатор, позволяющий создать второе из первого. Создав объект типа `String`, мы присваиваем его текстовому свойству `UITextField`.

Использование связанных переменных

Несмотря на то что для вставки значений можно конструировать SQL-строки, обычной практикой для этой цели все же является использование так называемых **связанных переменных** (*bind variables*). Корректная обработка строк (т.е. гарантирование отсутствия в их составе недопустимых символов и надлежащая обработка кавычек) иногда вызывает определенные трудности. Однако с использованием связанных переменных эти проблемы решаются практически сами собой.

Для того чтобы вставить некоторое значение с помощью связанной переменной, создайте обычный SQL-оператор, но поместите в SQL-строку знак вопроса (?). Каждый вопросительный знак представляет одну переменную, которая должна быть связана до выполнения этого оператора. Затем подготовьте SQL-оператор, свяжите значение с каждой переменной и выполните команду.

Рассмотрим пример подготовки SQL-оператора с двумя связанными переменными, связывания целочисленного значения с первой переменной и строки — со второй и, наконец, выполнения и завершения оператора:

```

var statement:OpaquePointer? = nil
let sql = "INSERT INTO FOO VALUES (?, ?);"
if sqlite3_prepare_v2(database, sql, -1, &statement, nil)
    == SQLITE_OK {
    sqlite3_bind_int(statement, 1, 235)
    sqlite3_bind_text(statement, 2, "Bar", -1, nil)
}
if sqlite3_step(statement) != SQLITE_DONE {
    print("This should be real error checking!")
}
sqlite3_finalize(statement);

```

Существует несколько операторов связывания, которые выбираются в зависимости от используемого типа данных. Большинство функций связывания принимает только три параметра.

- ❖ Первый параметр передается любой функции связывания независимо от типа данных. Он представляет собой указатель на переменную типа `sqlite3_stmt`, используемую ранее в вызове `sqlite3_prepare_v2()`.
- ❖ Второй параметр — это индекс переменной, с которой выполняется связывание. Это одноиндексное значение, которое интерпретируется так: первый вопросительный знак в SQL-операторе имеет индекс 1, а каждый последующий на единицу больше предыдущего.
- ❖ Третий параметр всегда содержит значение, которое должно заменить вопросительный знак.

Некоторые функции связывания (используемые, например, для связывания текстовых и двоичных данных) имеют еще два параметра.

- ❖ Первый дополнительный параметр представляет собой длину данных, передаваемых в третьем параметре. В случае использования С-строк можно вместо длины строки передать число `-1`, и тогда функция будет использовать всю строку. Во всех остальных случаях необходимо указывать длину передаваемых данных.
- ❖ Последний дополнительный параметр — необязательный обратный вызов функции, который передается в случае, когда необходимо позаботиться об освобождении памяти после выполнения вашего оператора. Обычно такая функция используется для освобождения памяти, занятой при вызове функции `malloc()`.

Синтаксис строк кода, следующих за операторами связывания, может показаться несколько странным. Дело в том, что они обеспечивают выполнение вставки. При использовании связанных переменных тот же самый синтаксис используется как для запросов, так и для обновлений. Если бы SQL-строка имела SQL-запрос, а не обновление, необходимо было бы многократно вызывать функцию `sqlite3_step()` — до тех пор, пока она не вернет значение

`SQLITE_DONE`. Поскольку в данном примере выполнялось именно обновление, мы вызвали эту функцию только один раз.

Приложение SQLite3

Используя среду Xcode, создайте новый проект по шаблону `Single View Application` и назовите его `SQLite Persistence`. Проект станет идентичным предыдущему, поэтому откройте файл `ViewController.swift` и внесите в него следующие изменения:

```
class ViewController: UIViewController {
    @IBOutlet var lineFields:[UITextField]!
```

Затем откройте файл `Main.storyboard`. Создайте представление и соедините коллекцию выходов, следуя инструкциям из раздела “Разработка представления для приложения Persistence”. Закончив проектирование, сохраните файл раскладовки.

Уделив немного внимания теории, можно переходить к практике. Давайте снова модифицируем наше приложение `Persistence`, на этот раз сохраняя данные с использованием механизма `SQLite3`. Мы ограничимся одной таблицей и будем сохранять значения полей в четырех различных ее строках. Каждой строке присвоим номер, который будет связан с полем, и поэтому в нашем примере значение из поля `field1` будет сохранено в таблице с номером строки, равным 1. Итак, начнем.

Компоновка с библиотекой базы данных `SQLite3`

Доступ к базе данных `SQLite 3` реализуется через процедурный интерфейс прикладного программирования, который предоставляет интерфейсы с различными вызовами С-функций. Для использования этого интерфейса нам необходимо скомпоновать приложение с динамической библиотекой `libsqlite3.dylib`. Выберите пункт `SQLite Persistence` в самом верху списка навигатора проекта (крайняя слева панель), а затем пункт `SQLite Persistence` в разделе `TARGETS` (средняя панель; рис. 13.7). Будьте внимательны: выбрать надо пункт `Persistence` в разделе `TARGETS`, а не в разделе `PROJECT`.

Выбрав цель `SQLite Persistence`, щелкните на вкладке `Build Phases` на крайней справа панели. Вы увидите список, состоящий из свернутых элементов, представляющих этапы, которые программа Xcode проходит при сборке приложения. Раскройте элемент с меткой `Link Binary With Libraries`. Вы увидите стандартные каркасы, которые подключаются к вашему приложению. По умолчанию этот список пуст, потому что компилятор автоматически устанавливает связи с любыми каркасами iOS, которые используются в вашем приложении, но компилятор ничего не знает о библиотеке `SQLite3`, поэтому мы должны добавить ее сюда.

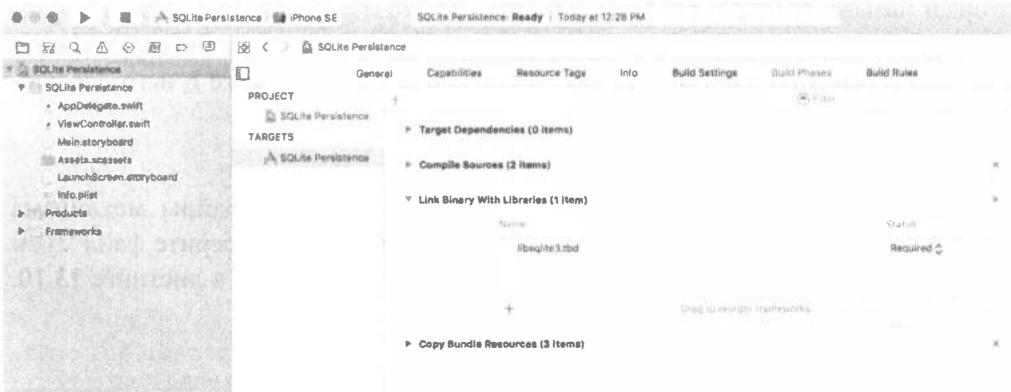


Рис. 13.7. Выбор проекта SQLite Persistence в навигаторе проекта, а также выбор цели SQLite Persistence и вкладки Build Phases

Щелкните на кнопке +, расположенной внизу списка подключенных каркасов, и вы увидите список доступных каркасов и библиотек. Найдите в этом списке библиотеку `libsqLite3.tbd` (или воспользуйтесь полем поиска) и щелкните на кнопке Add. Обратите внимание на то, что в каталоге может быть несколько элементов, имя которых начинается на `libsqLite3`. Необходимо выбрать именно библиотеку `libsqLite3.tbd`. Это псевдоним, который всегда ссылается на последнюю версию библиотеки SQLite3.

Модификация контроллера представления для приложения Persistence

Теперь необходимо импортировать заголовочные файлы для библиотеки SQLite3 в контроллер представления, чтобы компилятор мог видеть функции и другие определения из интерфейса прикладного программирования. Прямого способа импортировать заголовочный файл в код на языке Swift не существует, потому что библиотека SQLite3 не упакована в виде каркаса. Проще всего добавить в проект **мостовой заголовочный файл** (bridging header). С его помощью мы сможем добавлять другие заголовочные файлы, которые будет читать компилятор языка Swift. Есть несколько способов добавления мостового файла. Мы применим наиболее простой — временно добавим в проект класс на языке Objective-C.

Нажмите комбинацию клавиш <⌘+N> или выполните команду **File**⇒**New**⇒**File...**. В разделе iOS диалогового окна выберите пиктограмму **Cocoa Touch Class** и щелкните на кнопке **Next**. Присвойте классу имя `Temporary`, сделайте его подклассом класса `NSObject`, измените язык на Objective-C и щелкните на кнопке **Next**. На следующем экране щелкните на кнопке **Create**. После этого среда Xcode откроет всплывающее окно, в котором спросит вас, хотите ли вы создать мостовой заголовочный файл. Щелкните на кнопке **Create Bridging Header**. Перейдите в окно навигатора проекта, где вы увидите файлы для нового класса (`Temporary.m` и `Temporary.h`) и мостовой заголовочный файл,

который называется `SQLite Persistence-Bridging-Header.h`. Удалите файлы `Temporary.m` и `Temporary.h` — они вам больше не понадобятся. Откройте мостовой заголовочный файл в окне редактирования и добавьте в него следующую строку:

```
#import <sqlite3.h>
```

Теперь компилятор видит библиотеку и заголовочные файлы механизма `SQLite3`, и мы можем написать более сложную программу. Выберите файл `View Controller.swift` и внесите в него изменения, приведенные в листинге 13.10.

Листинг 13.10. Использование механизма `SQLite3` для сохранения и считывания информации

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из пив-файла.
    var database:OpaquePointer? = nil
    var result = sqlite3_open(dataFilePath(), &database)
    if result != SQLITE_OK {
        sqlite3_close(database)
        print("Failed to open database")
        return
    }
    let createSQL = "CREATE TABLE IF NOT EXISTS FIELDS "
        " (ROW INTEGER PRIMARY KEY, FIELD_DATA TEXT);"
    var errMsg:UnsafeMutablePointer<Int8>? = nil
    result = sqlite3_exec(database, createSQL, nil, nil, &errMsg)
    if (result != SQLITE_OK) {
        sqlite3_close(database)
        print("Failed to create table")
        return
    }

    let query = "SELECT ROW, FIELD_DATA FROM FIELDS ORDER BY ROW"
    var statement:OpaquePointer? = nil
    if sqlite3_prepare_v2(database, query, -1, &statement, nil) == SQLITE_
    OK {
        while sqlite3_step(statement) == SQLITE_ROW {
            let row = Int(sqlite3_column_int(statement, 0))
            let rowData = sqlite3_column_text(statement, 1)
            let fieldValue = String.init(cString:
                UnsafePointer<CChar>(rowData!))
            lineFields[row].text = fieldValue
        }
        sqlite3_finalize(statement)
    }
    sqlite3_close(database)
    let app = UIApplication.shared()
    NotificationCenter.default.addObserver(self, selector:
        #selector(self.applicationWillResignActive(notification:)),
        name: Notification.Name.UIApplicationWillResignActive,
        object: app)
}
```

```

func dataFilePath() -> String {
    let urls = FileManager.default.urls(for:
        .documentDirectory, in: .userDomainMask)
    var url:String?
    url = "" // создаем пустой путь
    do {
        try url = urls.first?.appendingPathComponent("data.plist").path!
    } catch {
        print("Error is \(error)")
    }
    return url!
}

func applicationWillResignActive(notification:NSNotification) {
    var database:OpaquePointer? = nil
    let result = sqlite3_open(dataFilePath(), &database)
    if result != SQLITE_OK {
        sqlite3_close(database)
        print("Failed to open database")
        return
    }
    for i in 0..<lineFields.count {
        let field = lineFields[i]
        let update = "INSERT OR REPLACE INTO FIELDS (ROW, FIELD_DATA) " +
            "VALUES (?, ?);"
        var statement:OpaquePointer? = nil
        if sqlite3_prepare_v2(database, update, -1, &statement, nil) ==
            SQLITE_OK {
            let text = field.text
            sqlite3_bind_int(statement, 1, Int32(i))
            sqlite3_bind_text(statement, 2, text!, -1, nil)
        }
        if sqlite3_step(statement) != SQLITE_DONE {
            print("Error updating table")
            sqlite3_close(database)
            return
        }
        sqlite3_finalize(statement)
    }
    sqlite3_close(database)
}
}

```

Первый фрагмент нового кода относится к методу `viewDidLoad()`. Прежде всего открываем базу данных с помощью добавленного метода `dataFilePath()`. Этот метод похож на метод `dataFileURL()`, который мы использовали в предыдущих примерах, за исключением того, что он возвращает путь к файлу, а не его URL. Мы используем этот путь, чтобы открыть базу данных или создать ее, если ее не было. Если открыть базу данных не удалось, мы закрываем ее, выводим на экран сообщение об ошибке и возвращаем управление.

```

var database:OpaquePointer? = nil
var result = sqlite3_open(dataFilePath(), &database)
if result != SQLITE_OK {

```

```

    sqlite3_close(database)
    print("Failed to open database")
    return
}

```

Нам необходимо удостовериться в существовании таблицы, содержащей наши данные. Для этого можно использовать SQL-оператор CREATE TABLE. С помощью выражения IF NOT EXISTS мы защищаем базу данных от перезаписи существующих данных. Если таблица с указанным именем уже существует, эта команда не выполнит никаких действий, поэтому ее можно безопасно вызывать при каждом запуске нашего приложения, не проверяя таблицу на существование в явном виде.

```

let createSQL = "CREATE TABLE IF NOT EXISTS FIELDS " +
    "(ROW INTEGER PRIMARY KEY, FIELD_DATA TEXT);"
var errMsg:UnsafeMutablePointer<Int8>? = nil
result = sqlite3_exec(database, createSQL, nil, nil, &errMsg)
if (result != SQLITE_OK) {
    sqlite3_close(database)
    print("Failed to create table")
    return
}

```

Каждая строка в таблице базы данных содержит целое число и объект класса String. Целое число представляет собой порядковый номер строки в графическом пользовательском интерфейсе, от которого получены данные (начиная отсчет с нуля), а объект класса String — это содержимое поля редактирования в этой строке. Наконец мы должны загрузить свои данные. Сделаем это с помощью SQL-оператора SELECT. В этом простом примере мы создаем SQL-оператор SELECT, который запрашивает все строки из базы данных, и просим SQLite3 подготовить для нас оператор SELECT. Мы также требуем, чтобы строки таблицы были упорядочены по номеру, чтобы мы всегда получали их в том же порядке. В противном случае SQLite3 будет возвращать строки в том порядке, в котором они были сохранены.

```

let query = "SELECT ROW, FIELD_DATA FROM FIELDS ORDER BY ROW"
var statement:OpaquePointer? = nil
if sqlite3_prepare_v2(database, query, -1, &statement, nil) ==
    == SQLITE_OK {

```

Примените функцию sqlite3_step() для выполнения команды SELECT и пройдитесь по всем возвращаемым строкам:

```
while sqlite3_step(statement) == SQLITE_ROW {
```

Сохраните номер строки в переменной типа int, а данные из поля — в C-строке, которая впоследствии будет преобразована в объект класса String в языке Swift.

```
let row = Int(sqlite3_column_int(statement, 0))
let rowData = sqlite3_column_text(statement, 1)
let fieldValue = String.init(cString: UnsafePointer<CChar>(rowData!))
```

Присвойте соответствующему полю значение, извлеченное из базы данных.

```
lineFields[row].text = fieldValue
```

Наконец закройте соединение с базой данных.

```
}
    sqlite3_finalize(statement);
}
sqlite3_close(database);
```

Обратите внимание на то, что мы закрываем соединение с базой данных сразу после завершающих действий по созданию таблицы и загрузке данных, которые она содержит, а не оставляем ее открытой все время, пока работает наше приложение. Это — самый простой способ управления соединением, и в этом небольшом приложении мы можем открывать соединение столько раз (т.е. немного), сколько потребуется. В приложениях с более интенсивным использованием базы данных, возможно, имеет смысл оставлять соединение открытым в течение всего времени работы приложения.

Остальные внесенные нами изменения связаны с методом applicationWillResignActive(), в котором нам нужно сохранить данные нашего приложения.

Метод applicationWillResignActive() начинается с открытия базы данных. Для того чтобы сохранить данные, в цикле пройдите по всем четырем полям и на каждом проходе выполните отдельную команду обновления каждой строки таблицы базы данных.

```
for i in 0..

```

Создайте SQL-оператор INSERT OR REPLACE с двумя связанными переменными. Первая представляет номер сохраняемой строки, вторая предназначена для сохранения реального строкового значения. Используя оператор INSERT OR REPLACE вместо стандартного INSERT, мы избавляем себя от беспокойства по поводу существования обрабатываемой строки.

```
char update = "INSERT OR REPLACE INTO FIELDS (ROW, FIELD_DATA) " +
    "VALUES (?, ?);";
```

Затем объявите указатель на оператор, подготовьте оператор со связанными переменными и свяжите значения с двумя связанными переменными.

```
var statement:OpaquePointer? = nil
if sqlite3_prepare_v2(database, update, -1, &statement, nil) == SQLITE_OK {
    let text = field.text
    sqlite3_bind_int(statement, 1, Int32(i))
    sqlite3_bind_text(statement, 2, text!, -1, nil)
}
```

Теперь вызовите метод `sqlite3_step`, чтобы выполнить обновление, проверьте результат выполнения и завершите цикл.

```
if sqlite3_step(statement) != SQLITE_DONE {  
    print("Error updating table")  
    sqlite3_close(database)  
    return  
}  
sqlite3_finalize(statement)
```

Обратите внимание на то, что здесь мы просто выводим сообщение об ошибке, если возникли проблемы. В реальном приложении обработка ошибок должна быть более содержательной, чем простой вывод предупреждения.

```
sqlite3_close(database)
```

ЗАМЕЧАНИЕ. При выполнении предыдущего SQLite-кода возможно возникновение аварийной ситуации, которую может вызвать ошибка, не являющаяся ошибкой программиста. Такая ошибочная ситуация возникнет в случае, если память устройства заполнится до такой степени, что механизм SQLite не сможет сохранять изменения в базе данных. Однако эта ситуация возникает довольно редко и может привести к более глубоким проблемам, которые выходят за видимости данных нашего приложения. Другими словами, если система побывает в таком состоянии, приложение, возможно, даже не будет успешно запускаться. Поэтому проблемы такого рода нужно обходить стороной.

Скомпилируйте и запустите приложение. Введите какие-нибудь данные, а затем нажмите кнопку Home симулятора iPhone. Перезапустите приложение Persistence, и вы должны убедиться в том, что ваши данные остались на месте. С точки зрения пользователя, между версиями этого приложения нет совершенно никакой разницы, но все они используют различные механизмы сохранения данных.

Использование каркаса Core Data

Последний способ реализации сохранения данных, который мы решили продемонстрировать в этой главе, основан на использовании каркаса Core Data, разработанного компанией Apple. Этот каркас представляет собой надежный полнофункциональный инструмент обеспечения долговременного хранения данных. Здесь мы снова обратимся к уже знакомому вам приложению Persistence, но переориентируем его на использование каркаса Core Data.

ЗАМЕЧАНИЕ. Для более полного освещения каркаса Core Data рекомендуем обратиться к книге Майкла Прайвeta (Michael Privet) и Роберта Уорнера (Robert Warner) *Pro iOS Persistence: Using Core Data* (Apress, 2014).

Создайте новый проект в среде Xcode. На этот раз выберите шаблон **Single View Application** в разделе **iOS** и щелкните на кнопке **Next**. Назовите проект **Core Data Persistence** и выберите пункт **Universal** в списке **Devices**, но пока не щелкайте

на кнопке Next. Непосредственно под раскрывающимся списком Devices находится флажок Use Core Data. Добавление каркаса Core Data в существующий проект — не такая простая задача, поэтому компания Apple любезно переложила часть нашей с вами работы на некоторые шаблоны проектов приложений. Установите флажок Use Core Data (рис. 13.8) и щелкните на кнопке Next. В ответ на приглашение введите имя каталога для хранения проекта и щелкните на кнопке Create.



Рис. 13.8. Возможность использования каркаса Core Data в шаблоне Single View Application

Прежде чем переходить к коду, рассмотрим окно проекта, которое содержит новые элементы. Раскройте папку Core Data Persistence (рис. 13.9).

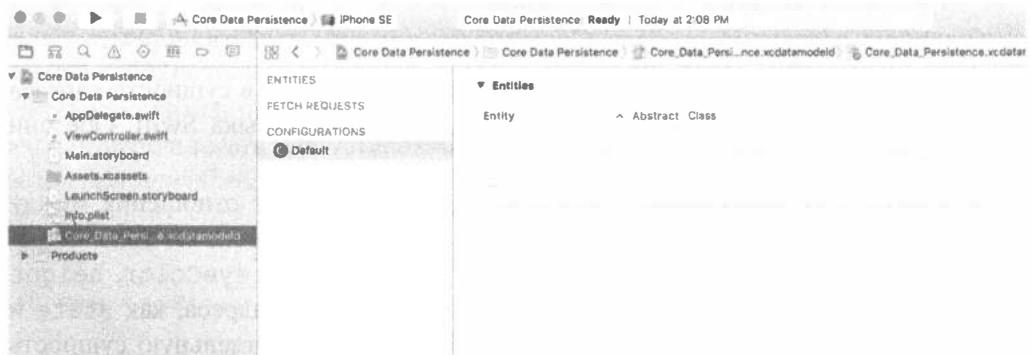


Рис. 13.9. Наш шаблон проекта с файлами, необходимыми для каркаса Core Data. Поскольку выбрана модель Core Data, на панели редактирования отображен редактор модели данных

Сущности и управляемые объекты

Большинство элементов, которые вы видите в окне навигатора проекта, должны быть вам уже знакомы: это файлы делегата приложения и каталог изображений. Кроме того, здесь есть файл Core_Data_Persistence.xcdatamodeld, который содержит нашу модель данных. В среде Xcode система Core Data допускает визуальное проектирование моделей данных без написания кода, а сами данные хранятся в файлах с расширением .xcdatamodeld.

Раскройте файл с расширением .xcdatamodeld, и вы увидите **редактор моделей данных** (data model editor), как показано на рис. 13.9. Редактор моделей данных предоставляет пользователю два режима работы, в зависимости от настройки Editor Style, расположенной в правом нижнем углу окна проекта. В режиме Table (см. рис. 13.9) элементы модели данных представляются в виде редактируемых таблиц, а в режиме Graph эти же элементы изображаются графически. Пока оба режима демонстрируют одно и то же: пустую модель.

До появления каркаса Core Data традиционным способом построения моделей данных было создание подклассов класса `NSObject` и приведение их в соответствие протоколам `NSCoding` и `NSCopying`, чтобы обеспечить возможность архивирования, как мы и поступали выше в этой главе. Каркас Core Data использует фундаментально другой подход. Вместо классов вы создаете **сущности** (entities) прямо в редакторе модели данных, а затем в коде создаете из них **управляемые объекты** (managed objects).

ЗАМЕЧАНИЕ. Термины **сущности** и **управляемые объекты** могут слегка вводить в заблуждение, поскольку оба они относятся к объектам моделей данных. Сущность имеет отношение к описанию объекта, а управляемый объект — к реальным конкретным экземплярам этой сущности, создаваемым во время выполнения приложения. Поэтому в редакторе моделей данных вы создаете сущности, а в коде — управляемые объекты. Различие между сущностями и управляемыми объектами подобно различию между классом и экземплярами этого класса.

Сущность состоит из свойств. Существуют три типа свойств.

- **Атрибуты.** Атрибут выполняет ту же самую функцию в сущности каркаса Core Data, что и переменная экземпляра в классе языка Swift. Обе они предназначены для хранения данных.
- **Связи.** Как подсказывает название, связь определяет отношения между сущностями. Например, чтобы создать сущность Person, вы должны начать с определения таких атрибутов, как `hairColor`, `eyeColor`, `height` и `weight`. Вы могли бы определить такие атрибуты адреса, как `state` и `ZIP`, либо эти атрибуты можно было бы встроить в отдельную сущность `HomeAddress`. В последнем случае вам следовало бы затем создать связь между сущностями `Person` и `HomeAddress`. Связи могут быть двух типов: “к одному” и “ко многим”. Связь от `Person` к `HomeAddress`, скорее

всего, относится к типу “к одному”, поскольку большинство людей имеют только один домашний адрес. Связь от HomeAddress к Person можно отнести к типу “ко многим”, так как может быть несколько представителей сущности Person, проживающих по адресу, определяемому сущностью HomeAddress.

- * **Извлекаемые свойства.** Извлекаемое *свойство* (fetched property) — это альтернатива связи. Извлекаемые свойства позволяют создавать запросы, которые вычисляются в момент извлечения, чтобы выявить, какие объекты принадлежат данной связи. Расширим рассмотренный выше пример и представим себе, что объект Person может иметь извлекаемое свойство Neighbors, которое находит все объекты HomeAddress в хранилище данных, которые имеют тот же самый ZIP-код, что и объект HomeAddress, принадлежащий объекту Person. По своей природе извлекаемые свойства всегда являются односторонними связями. Извлекаемые свойства представляют также единственный вид связей, позволяющий проходить по хранилищу данных.

Обычно атрибуты, связи и извлекаемые свойства определяются с использованием редактора моделей данных в среде Xcode. В нашем приложении Core Data Persistence мы построим простую сущность, и тогда вы ощутите, как это все в комплексе работает на практике.

Кодирование в стиле “ключ–значение”

В коде вместо использования get- (accessor) и set-методов (mutator) для установки свойств или извлечения их значений вы будете использовать **кодирование “ключ–значение”** (key-value coding). Термин **кодирование “ключ–значение”** может показаться несколько устрашающим, но мы с вами уже использовали его в этой книге. Каждый раз, когда мы использовали, например, класс NSDictionary, мы прибегали к кодированию в стиле “ключ–значение”, поскольку каждый объект в словаре хранится под уникальным ключевым значением. Кодирование “ключ–значение”, используемое каркасом Core Data, несколько сложнее, чем кодирование, используемое классом NSDictionary, но их основные принципы действия совпадают. При работе с управляемым объектом ключ, который вы решили использовать для установки или считывания значения свойства, является *именем атрибута*, который вы хотите установить. Ниже показано, как можно извлечь из управляемого объекта значение, хранимое в атрибуте с именем name.

```
let name = myManagedObject.valueForKey("name")
```

Аналогично, чтобы установить новое значение для свойства управляемого объекта, используйте такую строку кода:

```
myManagedObject.setValue("Gregor Overlander", forKey:"name")
```

Использование контекста

Итак, где “обитают” эти управляемые объекты? Они “прописаны” в так называемом **постоянном хранилище** (*persistent store*), также именуемом **резервным хранилищем** (*backing store*). Постоянные хранилища могут выступать в различных формах. По умолчанию каркас Core Data реализует резервное хранилище как базу данных SQLite, хранимую в каталоге `documents` приложения: Даже если вы сохраняете данные с помощью SQLite, всю работу, связанную с их загрузкой и хранением, выполняют классы каркаса Core Data. Если вы используете Core Data, вам не нужно писать никакие SQL-операторы. Вы просто работаете с объектами, а каркас Core Data сам решает, что нужно с ними делать.

В дополнение к базам данных SQLite резервные хранилища могут быть также реализованы как двоичные неструктурированные файлы или иметь формат XML. Еще один возможный вариант — *хранилище в оперативной памяти* (*in-memory store*), которое вы можете использовать в случае, если запрограммируете механизм кеширования, но этот механизм не позволяет хранить данные после завершения текущего сеанса работы. Практически во всех ситуациях в качестве постоянного хранилища следует использовать базу данных SQLite.

Несмотря на то что большинство приложений имеет только одно постоянное хранилище, не исключена возможность использования нескольких постоянных хранилищ в одном приложении. Если вас интересует, как резервное хранилище создается и конфигурируется, загляните в файл `AppDelegate.swift` из вашего проекта в среде Xcode. Шаблон проекта Xcode, который мы выбрали, обеспечил нас всем кодом, необходимым для установки единственного постоянного хранилища для нашего приложения.

Помимо создания постоянного хранилища (об этом вместо вас позаботился каркас в делегате приложения), вы обычно работаете с ним не напрямую, а используете так называемый **контекст управляемых объектов**, часто именуемый просто **контекстом**. Контекст управляет доступом к постоянному хранилищу и хранит информацию о том, какие свойства были изменены с момента последнего сохранения объекта. Контекст также регистрирует все изменения с помощью **менеджера отмены** (*undo manager*), а это значит, что вы всегда имеете возможность отменить одно изменение или полностью возвратиться к тому состоянию, когда ваши данные были изменены в последний раз.

ЗАМЕЧАНИЕ. У вас может быть несколько контекстов, указывающих на одно и то же постоянное хранилище, хотя большинство iOS-приложений ограничивается только одним.

Для того чтобы обеспечить выполнение приложения в контексте, многие вызовы каркаса Core Data в качестве параметра принимают экземпляр класса `NSManagedObjectContext`. За исключением очень сложных многопоточных iOS-приложений, можно использовать свойство `managedObjectContext` (из своего делегата приложения), которое служит стандартным контекстом,

создаваемым автоматически (также благодаря “любезности” шаблона проекта в среде Xcode).

Вы, вероятно, заметили, что в дополнение к контексту управляемых объектов и координатору постоянного хранилища этот “готовый” делегат приложения также содержит экземпляр класса `NSManagedObjectModel`. Этот класс отвечает за загрузку и представление во время работы модели данных, которую вы создадите с помощью редактора модели данных в среде Xcode. Как правило, вам не нужно взаимодействовать с этим классом напрямую. Он используется “тайно” другими классами каркаса Core Data, чтобы они могли идентифицировать, какие сущности и свойства вы определили в своей модели данных. Если вы создадите модель данных с помощью предоставленного каркасом файла, вам совсем не нужно беспокоиться об этом классе.

Создание новых управляемых объектов

Создание нового экземпляра управляемого объекта — довольно несложная задача, хотя и не такая простая, как создание экземпляра обычного объекта. Вы просто используете фабричный метод `insertNewObject(forEntityName: into:)`, определенный в классе `NSEntityDescription`. Назначение класса `NSEntityDescription` состоит в отслеживании всех сущностей, определенных в модели данных приложения. Этот метод возвращает экземпляр, представляющий одну сущность в памяти. Он возвращает либо экземпляр класса `NSManagedObject` (который создается с использованием корректных свойств для этой конкретной сущности), либо экземпляр его подкласса (если вы сконфигурировали свою сущность с помощью специального подкласса, производного от класса `NSManagedObject`). Вы же помните, что сущности подобны классам. Сущность представляет собой описание объекта и определяет, какие свойства имеет конкретный ее представитель.

Для того чтобы создать новый объект, выполните такой код:

```
let thing = NSEntityDescription.insertNewObject(
    forEntityName: "Thing",
    into: managedObjectContext)
```

Вызываемый здесь метод называется `insertNewObject(forEntityName: into:)`, поскольку в дополнение к созданию объекта он помещает в контекст только что созданный объект, а затем возвращает его. После выполнения этого вызова объект существует в контексте, но пока не является частью постоянно-го хранилища. Объект будет добавлен в него при следующем вызове метода `save()` контекста управляемого объекта.

Извлечение управляемых объектов

Для того чтобы извлечь управляемые объекты из постоянного хранилища, используется **запрос на извлечение** (`fetch request`), который представляет собой способ обработки предопределенного запроса, типичный для механизма

Core Data. Например, вы могли бы сказать “Найдите все сущности Person, значение атрибута eyeColor которых равно blue”. После первого создания запроса на извлечение вы снабжаете его экземпляром класса `NSEntityDescription`, который определяет сущность объекта или объектов, которые вы хотите извлечь. Вот пример создания запроса на извлечение:

```
let context = appDelegate.managedObjectContext
let request: NSFetchedRequest<NSFetchRequestResult> =
    NSFetchedRequest(entityName:"Thing")
```

Запрос на извлечение выполняется с помощью метода экземпляра класса `NSManagedObjectContext`.

```
do {
    let objects = try context.fetch(request)
    // Ошибки нет – можно использовать переменную objects
} catch {
    // Ошибка – переменная error содержит объект класса NSError
    print(error)
}
```

Метод `fetch()` загружает заданные объекты из постоянного хранилища и возвращает их в массив. При возникновении ошибки метод `fetch()` создает объект класса `NSError`, который описывает конкретную проблему. Эту ошибку необходимо либо перехватить и обработать, либо передать в вызывающую функцию. В данном случае мы просто выводим сообщение об ошибке на экран. Если вы не знакомы с механизмом обработки ошибок в языке Swift, прочитайте раздел “Обработка ошибок” в приложении. Если ошибок нет, вы получите корректный массив, хотя он может и не содержать ни одного объекта, поскольку не исключено, что ни один из рассматриваемых объектов не соответствует заданному критерию. С этого момента любые изменения, которые вы вносите в управляемые объекты, возвращаемые в этот массив, будут отслеживаться контекстом управляемых объектов, в котором был выполнен запрос и были сохранены данные (посредством отправки этому контексту сообщения `save:`).

Приложение Core Data

Прежде чем погрузиться в код, создадим нашу модель данных.

Разработка модели данных

Выберите файл `Core_Data_Persistence.xcdatamodel`, чтобы открыть редактор моделей данных в среде Xcode. Панель редактора моделей данных в верхнем левом углу этого окна называется `Entity`, поскольку в ней перечислены все текущие сущности, извлекаемые запросы и конфигурации, которые входят в состав вашей модели данных.

ЗАМЕЧАНИЕ. Концепция конфигураций в каркасе Core Data позволяет определять одно или несколько подмножеств сущностей, содержащихся в вашей модели данных. Они

могут оказаться полезными в определенных ситуациях. Например, если вы хотите создать набор приложений на основе общей модели данных, но некоторые модели не должны иметь доступ ко всем сущностям (например, одна модель предназначена для рядовых пользователей, а другая — для системных администраторов), данный подход позволит вам сделать это. Кроме того, в рамках одного приложения можно использовать несколько конфигураций, переключая их в зависимости от режима работы. В книге мы не рассматриваем конфигурации, но поскольку существует список конфигураций (включая конфигурацию по умолчанию, содержащуюся в каждой модели данных), который ставит перед вами проблему, связанную с сущностями и запросами на извлечение, мы решили о нем упомянуть.

Как показано на рис. 13.9, эта панель содержит пустой список, так как мы еще не создали ни одной сущности. Избавимся от “пустоты”, щелкнув на пиктограмме со знаком “плюс” в нижнем левом углу панели сущностей, чтобы создать и выбрать сущность Entity (рис. 13.10).

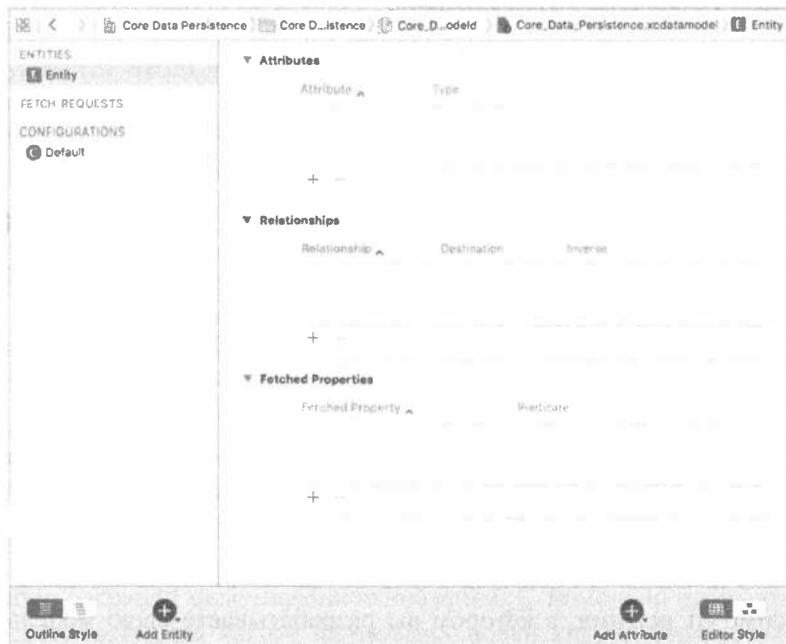


Рис. 13.10. Панель редактора моделей данных с новой сущностью

При построении модели данных, вероятно, вы будете постоянно переключаться между табличным и графическим режимами. Переключитесь пока в графический режим, в котором сущность отображается в виде небольшого блока, содержащего разделы для демонстрации атрибутов и связей (в данный момент пустых) (рис. 13.11). Графический режим очень удобен, если модель содержит много сущностей, поскольку при этом демонстрируются все связи между ними.



Рис. 13.11. С помощью элемента управления, расположенного в правом нижнем углу, мы переключили редактор моделей данных в графический режим. Обратите внимание на то, что в графическом режиме отображаются те же сущности, что и в табличном, но в графической форме. Это удобно, если в модели много сущностей и связей между ними

ЗАМЕЧАНИЕ. Если предпочитаете графический режим, то можете всю модель разрабатывать именно в нем. Мы предпочитаем в этой главе пользоваться табличным режимом, так как его проще объяснить. При разработке собственных моделей данных можете выбирать любой режим, который понравится.

Независимо от режима, в котором вы разрабатываете свою модель данных, придется открыть инспектор модели данных Core Data. Этот инспектор позволяет просматривать и редактировать релевантные детали любых элементов, выбранных в редакторе модели данных, — сущностей, атрибутов, связей или чего-то другого. Можете просматривать существующую модель, не используя инспектор модели данных, но в реальной работе без него обойтись невозможно, точно так же, как без инспектора атрибутов при работе с nib-файлами.

Нажмите комбинацию клавиш <Option+⌘+3>, чтобы открыть инспектор модели данных. В данный момент инспектор демонстрирует информацию о сущности, которую мы только что добавили. Измените поле Name с Entity на Line (рис. 13.12).

Если в данный момент вы находитесь в графическом режиме, переключитесь в табличный режим. Он отображает больше информации о сущности, с которой вы работаете, поэтому при создании новой сущности оказывается полезнее, чем графический. В табличном режиме большинство элементов в редакторе моделей данных изображается в виде таблиц, содержащих атрибуты, связи и извлекаемые свойства сущностей. Именно здесь происходит настройка наших сущностей.

Обратите внимание на то, что рядом с элементом управления **Editor Style** в правом нижнем углу находится пиктограмма **Add Attribute** со знаком +. Если вы выберете сущность, а затем будете удерживать нажатой кнопку мыши, то на экране откроется всплывающее окно, позволяющее добавить атрибут, отношение или извлекаемое свойство вашей сущности (рис. 13.13). Если же вы хотите просто добавить атрибут, то щелкните на пиктограмме +.

Примените описанный выше прием и добавьте атрибут для сущности **Line**. Новый атрибут **attribute** добавляется в раздел **Attributes** и выбирается автоматически. В таблице вы увидите, что выбрана не только строка, но и имя атрибута. Это означает, что сразу после щелчка на пиктограмме со знаком “плюс” вы можете вводить имя нового атрибута без дальнейших щелчков. Измените имя атрибута с **attribute** на **lineNumber**, щелкните на раскрывающемся списке, расположенному рядом с его именем, и измените поле **Type** с **Undefined** на **Integer 16**, чтобы этот атрибут сохранял целочисленное значение. С помощью этого атрибута мы будем распознавать, для какого из четырех полей управляемый объект хранит данные. Поскольку у нас есть только четыре возможных варианта, выберем из них наименьший целочисленный тип.

Переключите внимание на инспектор моделей данных, расположенный справа от области редактирования. Здесь уточняются дополнительные детали. Этот инспектор должен показывать свойства вновь добавленных атрибутов. Если он все еще показывает детали записи **Line**, щелкните на строке атрибутов в редакторе, чтобы переключить фокус редактора на этот атрибут. Флажок **Optional**, расположенный справа и ниже поля **Name**, устанавливается по умолчанию. Щелкните на нем, чтобы сбросить его, поскольку нам не нужно, чтобы этот атрибут

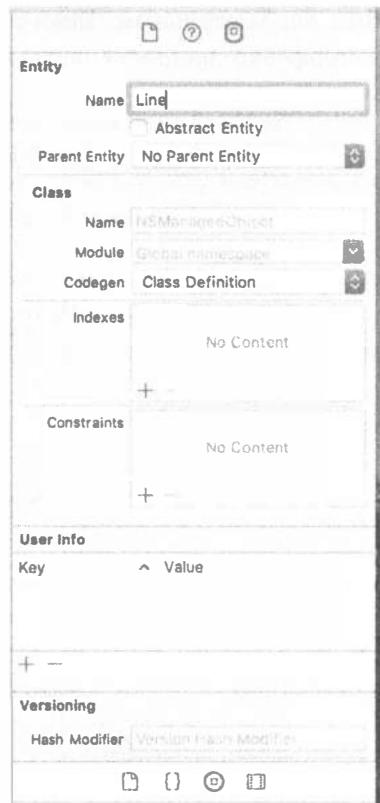


Рис. 13.12. Использование инспектора моделей данных для изменения имени сущности на **Line**

был необязательным. Ведь строка, которая не соответствует надписи в нашем интерфейсе, попросту бесполезна.



Рис. 13.13. Выбрав сущность, нажмите и удерживайте правую пиктограмму со знаком “плюс”, чтобы добавить атрибут, отношение или извлекаемое свойство сущности

Установка флагка `Transient` приводит к созданию временного атрибута. Этот атрибут используется для определения значения, которое хранится в управляемом объекте во время выполнения приложения, но не сохраняется в хранилище данных. Поскольку мы хотим, чтобы номера строк хранились в хранилище данных, оставьте флагок `Transient` сброшенным. Установка флагка `Indexed` инициирует создание индекса по соответствующему полю атрибута в используемой SQL-базе данных. Оставьте этот флагок сброшенным. Поскольку объем данных невелик и мы не планируем предоставлять пользователю возможности поиска, индексировать данные не обязательно.

Ниже расположено еще несколько настроек, позволяющих осуществлять простую оценку данных, задавая минимум и максимум для целых чисел, значение по умолчанию и другие параметры. В нашем примере эти настройки не используются.

Выберите сущность `Line` и щелкните на пиктограмме со знаком “плюс”, чтобы добавить второй атрибут с именем `lineText`, и измените его тип (поле `Type`) на `String`. Этот атрибут предназначен для хранения реальных данных из поля

редактирования. В этом случае оставьте флагок `Optional` установленным, ведь вполне допустимо, чтобы пользователь совсем не вводил значение в это поле.

ЗАМЕЧАНИЕ. Изменив поле `Type` на `String`, вы заметите появление дополнительных опций, которые позволяют установить значение, действующее по умолчанию, или ограничить длину вводимой строки. В данном приложении мы не будем использовать ни одну из этих опций, но вам все же полезно знать об их существовании.

Наша модель данных готова. Мы сделали все, что нужно. Благодаря каркасу Core Data мы смогли лишь щелчками мыши “проложить дорогу” к модели данных приложения. Теперь закончим построение приложения, чтобы узнать, как пользоваться нашей моделью данных из кода.

Модификация файла `AppDelegate.swift`

Найдите в файле `AppDelegate.swift` следующую строку:

```
// MARK: - Core Data stack
```

Под этой строкой расположены два раздела кода. В первом разделе создается объект класса `NSPersistentContainer`, а второй обеспечивает упаковку структуры Core Data. В нашем примере эти функциональные возможности использовать не будут, поэтому удалите эти два раздела.

ЗАМЕЧАНИЕ. Новая контейнерная функция — прекрасная возможность, упрощающая разработку приложений с помощью каркаса Core Data, однако на время создания этой книги эта функция оставалась ненадежной, поэтому мы от нее отказались и предпочли более устойчивый метод.

Удалите также шаблонный метод `saveContext` и замените его методом, приведенным в листинге 13.11. Здесь будет осуществляться сохранение нашего контекста. Мы будем вызывать этот метод из контроллера представления, когда будем готовы изменить активный статус нашего приложения.

Листинг 13.11. Метод `saveContext` в файле `AppDelegate.swift`

```
func saveContext () {
    if managedObjectContext.hasChanges {
        do {
            try managedObjectContext.save()
        } catch {
            // Замените эту реализацию кодом для обработки ошибок.
            // Функция abort() вынуждает приложение создать аварийную запись
            // и прекратить работу. В реальных приложениях эту функцию
            // использовать не следует, она нужна только на период разработки.
            let nserror = error as NSError
            NSLog("Unresolved error \(nserror), \(nserror.userInfo)")
            abort()
        }
    }
}
```

520 ГЛАВА 13 ■ ОСНОВЫ ДОЛГОВРЕМЕННОГО ХРАНЕНИЯ ДАННЫХ

Затем добавьте следующие методы из листинга 13.12 в файл AppDelegate.swift в строку, расположенную ниже директивы

```
// MARK: - Core Data stack
```

Листинг 13.12. Стек каркаса Core Data

```
// MARK: - Core Data stack
lazy var applicationDocumentsDirectory: URL = {
    // Каталог, который приложение использует для хранения файла Core Data.
    // Этот код использует каталог Documents в каталоге Application Support.
    let urls = FileManager.default.urls(for: .documentDirectory,
                                         in: .userDomainMask)
    return urls[urls.count-1]
}()

lazy var managedObjectModel: NSManagedObjectModel = {
    // Модель управляемого объекта для приложения.
    // Это свойство является необязательным. Если приложение не сможет
    // найти и загрузить эту модель, то возникнет фатальная ошибка.
    let modelURL = Bundle.main.url(forResource: "Core_Data_Persistence",
                                    withExtension: "momd")!
    return NSManagedObjectModel(contentsOf: modelURL) !
}()

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // Диспетчер хранилища для приложения. Эта реализация создает
    // и возвращает диспетчера при условии, что хранилище уже связано
    // с приложением. Это свойство является необязательным, поскольку
    // существуют формальные условия, при которых создание
    // хранилища невозможно.
    // Создаем диспетчера и хранилища.
    let coordinator = NSPersistentStoreCoordinator(
        managedObjectModel: self.
        managedObjectModel)
    let url = try!self.applicationDocumentsDirectory.
        appendingPathComponent("SingleViewCoreData.sqlite")
    var failureReason = "There was an error creating or loading
                        the application's saved data."
    do {
        try coordinator.addPersistentStore(
            ofType: NSSQLiteStoreType,
            configurationName:nil, at: url, options: nil)
    } catch {
        // Сообщение об ошибке.
        var dict = [String: AnyObject]()
        dict[NSTaggedPointerKey] =
            "Failed to initialize the application's saved data"
        dict[NSTaggedFailureReasonErrorKey] = failureReason
        dict[NSTaggedUnderlyingErrorKey] = error as NSError
        let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN",
                                   code: 9999, userInfo: dict)
        // Замените эту реализацию кодом для обработки ошибок.
        // Функция abort() вынуждает приложение создать аварийную запись
        // и прекратить работу. В реальных приложениях эту функцию
```

```

    // использовать не следует, она нужна только на период разработки.
    NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
    abort()
}
return coordinator
}()

lazy var managedObjectContext: NSManagedObjectContext = {
    // Возвращает контекст управляемого объекта для приложения,
    // который уже связан с диспетчером постоянного хранилища.
    // Это свойство является необязательным, поскольку
    // существуют формальные условия, при которых создание
    // контекста невозможно.
    let coordinator = self.persistentStoreCoordinator
    var managedObjectContext = NSManagedObjectContext(concurrencyType:
        .mainQueueConcurrencyType)
    managedObjectContext.persistentStoreCoordinator = coordinator
    return managedObjectContext
}()

```

ЗАМЕЧАНИЕ. В предыдущих версиях Xcode шаблон Single view генерировал большую часть этого кода. По какой-то причине в бета-версии новой версии Xcode этот код не генерируется. Однако, если выбрать шаблон проекта Master-Detail, то большая часть этого кода будет сгенерирована автоматически.

Определим путь к хранилищу Core Data.

```
lazy var applicationDocumentsDirectory: URL = {
```

Эта переменная представляет собой нашу модель управляемого объекта.

```

lazy var managedObjectModel: NSManagedObjectModel = {
    // Модель управляемого объекта для приложения.
    // Это свойство является необязательным. Если приложение не сможет
    // найти и загрузить эту модель, возникнет фатальная ошибка.
    let modelURL = Bundle.main.url(forResource: "Core_Data_Persistence",
        withExtension: "momd")!
    return NSManagedObjectModel(contentsOf: modelURL) !
}()

```

Следующий раздел кода создает ссылку на диспетчер постоянного хранилища.

```

lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // Диспетчер хранилища для приложения. Эта реализация создает
    // и возвращает диспетчер при условии, что хранилище уже связано
    // с приложением. Это свойство является необязательным, поскольку
    // существуют формальные условия, при которых создание
    // хранилища невозможно.
    // Создаем диспетчер и хранилище.
    let coordinator = NSPersistentStoreCoordinator(
        managedObjectModel:self.managedObjectModel)
    let url = try! self.applicationDocumentsDirectory.
        appendingPathComponent("SingleViewCoreData.sqlite")
    var failureReason = "There was an error creating or loading
        the application's saved data."

```

```

do {
    try coordinator.addPersistentStore(
        ofType: NSSQLiteStoreType, configurationName:nil,
        at: url, options: nil)
} catch {
    // Сообщение об ошибке.
    var dict = [String: AnyObject]()
    dict[NSLocalizedDescriptionKey] =
        "Failed to initialize the application's saved data"
    dict[NSLocalizedFailureReasonErrorKey] = failureReason
    dict[NSUnderlyingErrorKey] = error as NSError
    let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN",
                               code: 9999, userInfo: dict)
    // Замените эту реализацию кодом для обработки ошибок.
    // Функция abort() вынуждает приложение создать аварийную запись
    // и прекратить работу. В реальных приложениях эту функцию
    // использовать не следует, она нужна только на период разработки.
    NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
    abort()
}
return coordinator
}()

```

Осталось создать контекст управляемого объекта:

```

lazy var managedObjectContext: NSManagedObjectContext = {
    // Возвращает контекст управляемого объекта для приложения,
    // который уже связан с диспетчером постоянного хранилища.
    // Это свойство является необязательным, поскольку
    // существуют формальные условия, при которых создание
    // контекста невозможно.
    let coordinator = self.persistentStoreCoordinator
    var managedObjectContext = NSManagedObjectContext(
        concurrencyType:.mainQueueConcurrencyType)
    managedObjectContext.persistentStoreCoordinator = coordinator
    return managedObjectContext
}()

```

Это все, что нужно для делегата нашего приложения. Мы написали этот код, чтобы остальная часть приложения получила доступ к функциям каркаса Core Data.

Создание представления для приложения Persistence

Выберите файл `ViewController.swift` и внесите в него следующие изменения:

```

class ViewController: UIViewController {
    @IBOutlet var lineFields:[UITextField]!

```

Сохраните этот файл. Затем выберите файл `Main.storyboard`, чтобы открыть окно Interface Builder и отредактировать графический пользовательский интерфейс. Разработайте представление и соедините поля с выходами, следуя инструкциям, приведенным выше, в разделе “Разработка представления для приложения Persistence”. Завершив создание представления, выберите его,

откройте инспектор атрибутов и выберите пункт Light Grey Color во всплывающем меню Background. В качестве ориентира используйте рис. 13.13. Сохраните раскладовку.

В файл ViewController.swift вставьте код из листинга 13.13.

Листинг 13.13. Модификация файла ViewController.

swift для использования каркаса Core Data

```
import UIKit
import CoreData

class ViewController: UIViewController {
    private static let lineEntityName = "Line"
    private static let lineNumberKey = "lineNumber"
    private static let lineTextKey = "lineText"
    @IBOutlet var lineFields:[UITextField]!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Дополнительная настройка после загрузки представления,
        // обычно из nib-файла.

        let appDelegate =
            UIApplication.shared().delegate as! AppDelegate
        let context = appDelegate.managedObjectContext
        let request: NSFetchedResultsController<NSFetchRequestResult> =
            NSFetchedResultsController(entityName:ViewController.lineEntityName)
        do {
            let objects = try context.fetch(request)
            for object in objects {
                let lineNum: Int = object.value(
                    forKey: ViewController.lineNumberKey)! as! Int
                let lineText = object.value(
                    forKey: ViewController.lineTextKey) as? String ?? ""
                let textField = lineFields[lineNum]
                textField.text = lineText
            }
        }

        let app = UIApplication.shared()
        NotificationCenter.default.addObserver(self,
            selector: #selector(UIApplicationDelegate.
                applicationWillResignActive(_:)),
            name: NSNotification.Name.
        UIApplicationWillResignActive,
            object: app)
        } catch {
            // Перехват ошибки, сгенерированной методом executeFetchRequest()
            print("There was an error in executeFetchRequest(): \(error)")
        }
    }

    func applicationWillResignActive(_ notification:Notification) {
        let appDelegate =
            UIApplication.shared().delegate as! AppDelegate
```

```
let context = appDelegate.managedObjectContext
for i in 0 ..< lineFields.count {
    let textField = lineFields[i]

    let request: NSFetchedResultsController<NSFetchRequestResult> =
        NSFetchedResultsController(entityName: ViewController.lineEntityName)
    let pred = Predicate(format: "%K = %d",
        ViewController.lineNumberKey, i)
    request.predicate = pred

    do {
        let objects = try context.fetch(request)
        var theLine: NSManagedObject! = objects.first as? NSManagedObject
        if theLine == nil {
            // Нет данных для этой строки - вставляем в нее
            // новый управляемый объект
            theLine =NSEntityDescription.insertNewObject(
                forEntityName: ViewController.lineEntityName,
                into: context) as NSManagedObject
        }
        theLine.setValue(i, forKey: ViewController.lineNumberKey)
        theLine.setValue(textField.text,
            forKey: ViewController.lineTextKey)
    } catch {
        print("There was an error in executeFetchRequest(): \(error)")
    }
}
appDelegate.saveContext()
}
```

Сначала мы импортировали каркас Core Data. Затем мы изменили метод viewDidLoad(). Этот метод проверяет, есть ли данные в постоянном хранилище. Если в постоянном хранилище есть какие-либо данные, метод должен загрузить их и заполнить ими соответствующие поля. Прежде всего в этом методе мы должны получить ссылку на делегат приложения, с помощью которой затем получим контекст управляемого объекта (типа NSManagedObjectContext), который был создан вместо нас.

```
let appDelegate =
    UIApplication.shared().delegate as! AppDelegate
let context = appDelegate.managedObjectContext
```

После этого следует создать запрос на извлечение и передать ему описание сущности, чтобы он знал, какого типа объекты нужно извлечь.

```
let request: NSFetchedResultsController<NSFetchRequestResult> =
    NSFetchedResultsController(entityName: ViewController.lineEntityName)
```

Поскольку мы хотим извлечь все объекты типа Line, содержащиеся в постоянном хранилище, нам не нужно создавать предикат. Выполняя запрос без предиката, мы тем самым предписываем контексту извлечь из постоянного хранилища все (без исключения) объекты типа Line. Создав запрос на извлечение,

для его выполнения мы используем метод `fetch()` из контекста управляемого объекта. Поскольку метода `fetch()` может генерировать ошибки, мы поместили его вызов и код, использующий результат его работы, в блок `do-catch`, чтобы перехватить ошибку, как показано ниже.

```
do {
    let objects = try context.fetch(request)
```

Затем проходим в цикле по массиву найденных управляемых объектов, извлекаем из него значения `lineNum` и `lineText` и используем эту информацию для обновления соответствующего поля редактирования в интерфейсе пользователя.

```
for object in objects {
    let lineNumber: Int = object.value(forKey: ViewController.lineNumberKey) as! Int
    let lineText = object.value(forKey: ViewController.lineTextKey) as? String ?? ""
    let textField = lineFields[lineNum]
    textField.text = lineText
}
```

Разумеется, при первом выполнении этого кода в хранилище ничего нет, поэтому список объектов будет пустым.

Потом, как и при создании всех остальных приложений этой главы, мы регистрируемся на получение уведомления в момент, когда приложение готово выйти из активного состояния (в результате либо перехода в фоновый режим, либо полного завершения), что позволяет нам сохранить любые изменения, которые пользователь внес в данные.

```
let app = UIApplication.shared()
NotificationCenter.default.addObserver(self,
    selector: #selector(UIApplicationDelegate.applicationWillResignActive(_:)),
    name: NSNotification.Name.UIApplicationWillResignActive,
    object: app)
```

В заключение в разделе `catch` на экран выводится сообщение об ошибке, возникшей в методе `fetch()`.

```
} catch {
    // Ошибка, сгенерированная в методе executeFetchRequest()
    print("There was an error in executeFetchRequest(): \(error)")
```

Рассмотрим теперь метод `applicationWillResignActive()`. Получаем ссылку на делегата приложения и используем ее для получения указателя на стандартный контекст нашего приложения.

```
let appDelegate =
    UIApplication.shared().delegate as! AppDelegate
let context = appDelegate.managedObjectContext
```

526 ГЛАВА 13 ■ ОСНОВЫ ДОЛГОВРЕМЕННОГО ХРАНЕНИЯ ДАННЫХ

После этого входим в цикл, который выполняется для каждого поля редактирования и получаем корректную ссылку на поле.

```
for i in 0 ..< lineFields.count {  
    let textField = lineFields[i]
```

Далее создаем запрос на извлечение записи Line. Необходимо выяснить, существует ли в постоянном хранилище управляемый объект, соответствующий этой записи, поэтому создаем предикат, идентифицирующий правильный объект для поля по ключу записи.

```
let request: NSFetchedRequest<NSFetchedRequestResult> =  
let pred = Predicate(format: "%K = %d", ViewController.lineNumberKey, i)  
request.predicate = pred
```

Теперь мы выполняем запрос на извлечение по контексту. Как и прежде, мы погрузили этот код в раздел do-catch, чтобы сообщить об ошибках, возникших при работе с каркасом Core Data.

```
do {  
    let objects = try context.fetch(request)
```

Теперь объявляем указатель на экземпляр типа NSManagedObject, который будет ссылаться на данные из заданной строки. Стока может ничего не содержать, так что пока не знаем, будем ли загружать из постоянного хранилища управляемый объект или же создавать новый. По этой причине объект theLine следует объявить необязательным. Однако для удобства мы решили оставить его неупакованным, поскольку мы планируем использовать метод insertNewObject(forEntityName:inManagedObjectContext:) для создания управляемого объекта, предназначенного для данной строки в постоянном хранилище, если его там еще нет. В данном случае для инициализации объекта theLine используется управляемый объект.

```
var theLine: NSManagedObject! = objects.first as? NSManagedObject  
if theLine == nil {  
    // Нет данных для этой строки - вставляем в нее новый управляемый объект  
    theLine =  
        NSEntityDescription.insertNewObject(  
            forEntityName: ViewController.lineEntityName,  
            into: context)  
        as NSManagedObject
```

Используем кодирование “ключ–значение”, чтобы установить номер строки и текст для этого управляемого объекта.

```
theLine.setValue(i, forKey: ViewController.lineNumberKey)  
theLine.setValue(textField.text, forKey: ViewController.lineTextKey)
```

Наконец, завершив цикл обработки, предписываем контексту сохранить изменения.

```
appDelegate.saveContext()
```

Вот и все! Соберите и выполните приложение, чтобы убедиться в его работоспособности. Его версия для работы с каркасом Core Data должна вести себя точно так же, как и предыдущие версии.

Резюме

Из этой главы узнали о четырех способах сохранения данных приложения между сеансами работы — даже о пяти способах, если учесть механизм, с которым вы познакомились в предыдущей главе. Мы создали приложение, которое сохраняло данные с помощью списков свойств, а затем переориентировали его на использование архивов объектов. После этого на практике убедились в возможности сохранения данных на базе встроенного в систему iOS механизма управления базами данных SQLite3. Наконец мы перестроили то же самое приложение, чтобы опробовать каркас Core Data. Перечисленные выше механизмы — это базовые строительные блоки, используемые для сохранения и загрузки данных практически во всех iOS-приложениях.

ГЛАВА 14



Документы и служба iCloud

Служба iCloud, впервые появившаяся в системе iOS 5 (рис. 14.1), позволяет пользователям iOS-устройств и компьютеров с операционной системой macOS хранить данные в облачных хранилищах. Большинство пользователей системы iOS, вероятно, немедленно воспользуются возможностью резервного копирования в службе iCloud при инсталляции нового устройства или обновлении операционной системы до новейшей версии iOS на старом устройстве и быстро оценят преимущества автоматического резервного копирования, для которого даже не требуется компьютер.



Рис. 14.1. Служба iCloud, впервые появившаяся в системе iOS 5, позволяет использовать серверное хранилище для приложений iOS и macOS

Резервное копирование без использования компьютера — превосходная функция, но это лишь малая часть возможностей службы iCloud. Еще более важным

является то, что служба iCloud предоставляет разработчикам приложений механизм для прозрачного сохранения данных на облачных серверах компании Apple, требующих минимальных усилий. Вы можете сделать так, чтобы ваши приложения сохраняли данные в службе iCloud и чтобы эти данные автоматически передавались на любые другие устройства, зарегистрированные на того же пользователя службы iCloud. Пользователь может создать документ на своем устройстве iPad, а потом увидеть этот же документ на своем устройстве iPhone, не прилагая для этого никаких усилий; документ просто там появится.

Обеспечением корректности регистрации в службе iCloud и управлением передачей файлов занимается системный процесс, поэтому вам не стоит беспокоиться о сетях или аутентификации. Для использования службы iCloud в своем приложении, кроме небольшой настройки конфигурации приложения, вам нужно лишь внести незначительные изменения в свои методы хранения файлов и поиска доступных файлов.

Ключевым компонентом файловой системы iCloud является класс `UIDocument`, который выполняет часть работы, связанной с созданием документо-ориентированных приложений, реализуя некоторые из основных аспектов чтения и записи файлов. Таким образом, вы можете сосредоточиться на уникальных особенностях своих приложений вместо вставки стандартных фрагментов кода в каждое создаваемое приложение.

Независимо от того, используете ли вы службу iCloud, класс `UIDocument` представляет собой довольно мощный инструмент для управления файлами в системе iOS. Для демонстрации его возможностей в начале главы рассмотрим процесс создания простого документо-ориентированного приложения `TinyPix`, которое сохраняет файлы в локальном хранилище. Этот подход хорошо работает для всех видов приложений в системе iOS, а недавно компания Apple обеспечила возможность тестировать синхронизацию со службой iCloud на симуляторе, не выполняя приложение на реальном устройстве.

В этой главе мы покажем, как создать приложение `TinyPix`, имеющее доступ к службе iCloud.

Управление хранилищем документов с помощью класса `UIDocument`

Любое программное обеспечение, позволяющее работать с несколькими коллекциями данных и сохранять каждую коллекцию в отдельном файле, может рассматриваться как документо-ориентированное приложение. Часто между окном и документом, который оно содержит, существует взаимно однозначное соответствие, но иногда (например, в среде Xcode) одно окно может отображать несколько взаимосвязанных файлов.

На устройствах, работающих под управлением системы iOS, нет многочисленных окон, но многие приложения могут по-прежнему извлекать выгоду из документо-ориентированного подхода. Благодаря классу `UIDocument`, взявшему

на себя реализацию большинства стандартных аспектов, связанных с хранением документных файлов, теперь нет необходимости непосредственно работать с файлами (только с их указателями URL), а все, что требуется для их чтения и записи, происходит в фоновом потоке, поэтому ваше приложение продолжает реагировать даже тогда, когда открывается доступ к файлу. Кроме того, оно периодически сохраняет отредактированные документы всякий раз, когда работа приложения приостанавливается (например, при отключении устройства, нажатии главной кнопки и т.п.), поэтому кнопка для сохранения файлов не нужна. Все это помогает обеспечивать именно такое поведение приложения, какого ожидает пользователь.

Создание приложения TinyPix

Мы создадим приложение TinyPix, которое позволит редактировать простые изображения размером 8×8 пикселей в однобитовом цвете (рис. 14.2). Для удобства пользователей каждый рисунок при редактировании будет увеличиваться до размера экрана. Для представления данных для каждого изображения будет использоваться класс `UIDocument`.

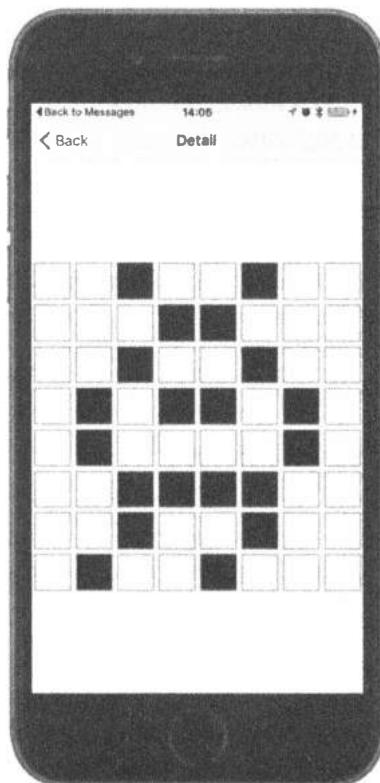


Рис. 14.2. Редактирование в приложении TinyPix пиктограммы с чрезвычайно низким разрешением

Начнем с создания нового проекта в среде Xcode. Выберите в разделе iOS Application шаблон Master-Detail Application и щелкните на кнопке Next. Назовите новое приложение TinyPix, выберите в списке Devices пункт iPhone и установите флажок Use Storyboard. Затем снова щелкните на кнопке Next и выберите место для хранения своего проекта.

В навигаторе проекта среды Xcode вы увидите, что ваш проект содержит файлы AppDelegate, MasterViewController, DetailViewController, а также файл Main.storyboard. Мы внесем в эти файлы определенные изменения, а также создадим несколько новых классов.

Создание класса TinyPixDocument

Сначала создадим документный класс, который будет содержать данные о каждом изображении, загружаемом приложением TinyPix из файлового хранилища. Выберите папку TinyPix в среде Xcode и нажмите комбинацию клавиш **<⌘+N>**, чтобы создать новый файл. Выберите в разделе iOS пункт Cocoa Touch Class и щелкните на кнопке Next. Введите в поле Class имя TinyPixDocument, в поле Subclass of — имя UIDocument и щелкните на кнопке Next. Щелкните на кнопке Create, чтобы создать файл.

Перед тем как перейти к реализации деталей этого класса, следует подумать о его открытом интерфейсе прикладного программирования. Этот класс будет представлять сетку размером 8×8 пикселей, при этом каждый пиксель будет либо включен, либо выключен. Итак, класс должен содержать метод, принимающий пару индексов, определяющих место пикселя в сетке, и возвращающий значение типа Bool. Кроме того, нам нужен метод для установки определенного состояния конкретной точки в сетке и для удобства еще один метод, который будет просто включать или выключать конкретный пиксель.

Откройте файл TinyPixDocument.swift, в котором будет реализовано хранение сетки 8×8 , методы, определенные в открытом интерфейсе программного обеспечения, и методы класса UIDocument, обеспечивающие загрузку и сохранение документов.

Начнем с определения контейнера для сетки 8×8 . Мы будем хранить эти данные в массиве типа UInt8. Добавьте в класс TinyPixDocument следующее свойство:

```
class TinyPixDocument: UIDocument {
    private var bitmap: [UInt8]
```

Класс UIDocument имеет специальный инициализатор, который должны использовать все его подклассы. Именно здесь будет создаваться первоначальное графическое изображение. Придерживаясь истинного стиля битовых масок, будем минимизировать объем требуемой памяти, используя для хранения каждой строки только один байт. Каждый бит в этом байте будет представлять значение “включен/выключен” для индекса в заданной строке. В совокупности наш документ будет состоять всего лишь из 8 байт.

ЗАМЕЧАНИЕ. В этом разделе содержатся небольшое количество побитовых операций, а также указатель из языка C и операции для манипуляции массивом. Для разработчиков программ на языке C в этом нет ничего необычного, но если у вас мало опыта в программировании на языке C, код может показаться вам запутанным и даже непостижимым. В таком случае скопируйте его и используйте по назначению (он отлично работает). Если вы действительно хотите разобраться в нем, приступайте к углубленному изучению языка C.

Добавьте в реализацию документа следующий метод:

```
override init(fileURL: NSURL) {
    bitmap = [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80]
    super.init(fileURL: fileURL as URL)
}
```

Каждое графическое изображение сначала будет представлять собой простой диагональный шаблон, растянутый из одного угла в другой. Сначала рассмотрим метод для чтения состояния отдельного бита. Вставьте в класс код, приведенный в листинге 14.1.

Листинг 14.1. Считывание состояния отдельного бита

```
func stateAt(row: Int, column: Int) -> Bool {
    let rowByte = bitmap[row]
    let result = UInt8(1 << column) & rowByte
    return result != 0
}
```

Этот метод просто будет считывать релевантный байт из нашего массива, затем выполнять разрядный сдвиг и операцию AND для определения, установлен ли указанный бит, и возвращать значение true или false. Затем сделаем нечто противоположное: метод, устанавливающий значение конкретной точки в сетке по заданным индексам строки и столбца. Снова возьмем указатель на байт, соответствующий заданным строке и столбцу, и выполним разрядный сдвиг. Однако на этот раз мы не будем определять содержимое строки, а, наоборот, будем устанавливать или сбрасывать указанный бит в строке. Добавьте в конец определения класса метод, приведенный в листинге 14.2.

Листинг 14.2. Установка и сброс бита в строке

```
func setState(state: Bool, atRow row: Int, column: Int) {
    var rowByte = bitmap[row]
    if state {
        rowByte |= UInt8(1 << column)
    } else {
        rowByte &= ~UInt8(1 << column)
    }
    bitmap[row] = rowByte
}
```

Добавим удобный метод, который позволит внешнему коду переключать отдельную ячейку.

```
func toggleStateAt(row: Int, column: Int) {
    let state = stateAt(row: row, column: column)
    setState(state: state!, atRow: row, column: column)
}
```

В нашем документном классе не достает двух заключительных фрагментов: методов для чтения и записи. Как указывалось ранее, вам не обязательно непосредственно взаимодействовать с файлом. Вам даже не обязательно заботиться об указателе URL, который ранее передавался в метод init(fileURL:). Вам нужно лишь реализовать метод, трансформирующий структуру данных документа в объект класса NSData, готовый для сохранения, а также метод, получающий вновь загруженный объект класса NSData и извлекающий из него его структуру данных. Добавьте эти два метода, приведенные в листинге 14.3, чтобы реализовать требуемый контракт класса UIDocument.

Листинг 14.3. Прямые и обратные преобразования объектов класса NSData

```
override func contents(forType typeName: String) throws -> AnyObject {
    print("Saving document to URL \(fileURL)")
    let bitmapData = NSData(bytes: bitmap, length: bitmap.count)
    return bitmapData
}

override func load(fromContents contents: AnyObject, ofType typeName: String?) throws {
    print("Loading document from URL \(fileURL)")
    if let bitmapData = contents as? NSData {
        bitmapData.getBytes(UnsafeMutablePointer<UInt8>(bitmap),
                            length: bitmap.count)
    }
}
```

Первый из этих методов, contents(forType: typeName:), вызывается каждый раз, когда документ должен быть отправлен в хранилище. Он просто возвращает неизменяемую копию графического изображения, которую система впоследствии должна сохранить. Второй метод, load(fromContents contents:), вызывается каждый раз, когда система должна загрузить данные из хранилища и передать их объекту документного класса. Здесь просто извлекаем изменяемую копию полученных данных. Мы включили несколько инструкций, связанных с регистрацией, чтобы вы могли впоследствии увидеть записи в журнале среды Xcode.

Каждый из этих методов позволяет выполнять действия, которые мы проигнорировали в своем приложении. Они имеют параметр typeName, с помощью которого можно различать разные типы хранящихся данных, которые ваш документ загружает или сохраняет. Кроме того, они имеют параметр outError,

с помощью которого можно идентифицировать ошибку во время копирования данных в структуру данных документа, находящегося в памяти, или при их загрузке оттуда. Однако в данном случае все настолько просто, что никаких ошибок быть не может.

Вот и все, что требуется для нашего документного класса. В соответствии с принципами MVC документ представляет собой модель, ничего не знающую о том, как она будет отображаться на экране. Благодаря суперклассу `UIDocument` документ даже защищен от необходимости знать о большинстве деталей своего хранения.

Основной код

Разработав документный класс, можем переходить к первому представлению, которое пользователь увидит при запуске приложения: списку существующих документов для приложения `TinyPix`, реализованному с помощью класса `MasterViewController`. Нам необходимо сообщить классу, как получить список доступных документов, создать и назвать новый документ и позволить пользователю выбрать существующий документ. Когда документ создан или выбран, он передается детализированному представлению для отображения на экране.

Выберите файл `MasterViewController.swift`. Этот файл, автоматически сгенерированный как часть шаблона `Master-Detail`, содержит код для демонстрации массива элементов. Мы не будем использовать этот код и заменим его собственным кодом, поэтому удалите все, что записано в этом файле, кроме директивы импорта каркаса `UIKit` и объявления класса. После этого код будет выглядеть примерно так:

```
import UIKit

class MasterViewController: UITableViewController { }
```

Кроме того, включаем в графический пользовательский интерфейс сегментированный элемент управления, который позволит пользователю выбирать цвет для отображения пикселей в приложении `TinyPix`. Несмотря на то что сама по себе эта функция не особенно интересна, это позволит нам продемонстрировать механизм службы `iCloud`, поскольку цветом подсветки будет выделяться путь от устройства, с которым вы работаете, до другого присоединенного устройства, на котором запущено то же самое приложение. В первом варианте приложения будем использовать цвет, заданный стандартными настройками устройства. Позднее в этой главе мы добавим код, который позволит выделить цветом путь, проходящий через службу `iCloud` к другим устройствам пользователя.

Для реализации сегментированного элемента управления для переключения цвета добавим в код дополнительные выход и действие. Кроме того, добавим свойства для хранения списка названий документов и указатель на документ, выбранный пользователем. Внесем эти изменения в файл `MasterViewController.swift`.

536 ГЛАВА 14 ■ ДОКУМЕНТЫ И СЛУЖБА ICLOUD

```
class MasterViewController: UITableViewController {
    @IBOutlet var colorControl: UISegmentedControl!
    private var documentFileURLs: [URL] = []
    private var chosenDocument: TinyPixDocument?
```

Прежде чем реализовать методы табличного представления и другие стандартные методы, напишем несколько закрытых вспомогательных методов. Первый из них принимает имя файла, объединяет его с путем к каталогу Documents, связанному с приложением, и возвращает указатель URL, указывающий на данный конкретный файл. Как указано в главе 13, каталог Documents представляет собой специальное место, которое система iOS организовывает отдельно, по одному для каждого приложения, инсталлированного на устройстве, работающем под ее управлением. Можете использовать его для хранения документов, созданных в вашем приложении, и быть полностью уверенными в том, что эти документы будут автоматически включены при резервном копировании вашего устройства, независимо от того, какая служба для этого используется — iTunes или iCloud.

Добавьте этот метод в файл MasterViewController.swift:

```
private func urlForFileName(fileName: String) -> URL {
    let urls = FileManager.default.urls(for:
        .documentDirectory, inDomains: .userDomainMask)
    var url: URL = URL(fileURLWithPath: "") // create a blank path
    do {
        try url = urls.first!.appendingPathComponent(fileName)
    } catch {
        print("Error is \(error)")
    }
    return url
}
```

Второй закрытый метод немного длиннее. Он также использует каталог Documents, на этот раз для поиска файлов, представляющих существующие документы. Этот метод сортирует найденные файлы по дате их создания, так что пользователь увидит список документов, упорядоченный в стиле блога, т.е. новейшие элементы расположены первыми. Имена документных файлов накапливаются в свойстве documentFilenames, а затем табличное представление (которые мы намеренно пока не упоминали) загружается заново. Добавьте в определение класса код, приведенный в листинге 14.4.

Листинг 14.4. Метод повторной загрузки файлов

```
private func reloadFiles() {
    let fm = FileManager.default
    let documentsURL = fm.urls(for:
        .documentDirectory, inDomains: .userDomainMask).first!
    do {
        let fileURLs = try fm.contentsOfDirectory(at: documentsURL,
            includingPropertiesForKeys: nil,
```

```

        options: [])
let sortedFileURLs = fileURLs.sorted(isOrderedBefore: {
    (file1URL, file2URL) -> Bool in
    let attr1 = try! fm.attributesOfItem(atPath: file1URL.path!)
    let attr2 = try! fm.attributesOfItem(atPath: file2URL.path!)
    let file1Date = attr1[FileAttributeKey.creationDate] as! NSDate
    let file2Date = attr2[FileAttributeKey.creationDate] as! NSDate
    let result = file1Date.compare(file2Date as Date)
    return result == ComparisonResult.orderedAscending
})
documentFileURLs = sortedFileURLs
tableView.reloadData()

} catch {
    print("Error listing files in directory \(documentsURL.path!)": \
(error)")
}
}
}

```

Перейдем к методам источника данных для табличного представления. Они должны быть вам хорошо знакомы. Добавьте в файл MasterViewController.swift методы, приведенные в листинге 14.5.

Листинг 14.5. Методы источников данных для табличного представления

```

override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}

override func tableView(_ tableView: UITableView,
                      numberOfRowsInSection section: Int) -> Int {
    return documentFileURLs.count
}

override func tableView(_ tableView: UITableView,
                      cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "FileCell")!
    let fileURL = documentFileURLs[indexPath.row]
    do {
        try cell.textLabel!.text = fileURL.deletingPathExtension().lastPathComponent
    } catch {
        print("Error is \(error)")
    }
    return cell
}

```

Эти методы основаны на содержании массива, хранящегося в свойстве documentFileURLs и созданного методом reloadFiles(). Этот массив содержит объекты класса NSURL для каждого файла, хранящегося в каталоге Documents. При этом файлы упорядочены по возрастанию в соответствии со временем их создания. Метод tableView(_:cellForRowAtindexPath:), возвращающий

объект класса `UITableViewCell`, предполагает существование ячейки, присоединенной к табличному представлению с идентификатором "FileCell", поэтому мы должны обязательно настроить ее в раскладовке немного позднее.

Не учитывая тот факт, что мы еще не прикасались к раскладовке, код, который мы создали к этому моменту, почти готов к запуску, но без заранее созданных документов для приложения `TinyPix` мы ничего не сможем вывести на экран в табличном представлении. В то же время мы еще не создали инструмент для создания новых документов. Кроме того, еще не создан элемент управления для выбора цвета, который мы собирались добавить. Итак, прежде чем запустить приложение, придется немного потрудиться.

Выбор цвета подсветки будет немедленно использоваться для окрашивания сегментированного элемента управления. В классе `UView` есть свойство `tintColor`. Оно распространяется как на представление, так и на все его дочерние представления. Когда пользователь делает выбор, мы сохраним его в объекте класса `NSUserDefaults`, чтобы впоследствии пользователь мог его оттуда извлечь. Добавьте в конец определения класса следующие два метода.

```
@IBAction func chooseColor(sender: UISegmentedControl) {
    let selectedColorIndex = sender.selectedSegmentIndex
    setTintColorForIndex(colorIndex: selectedColorIndex)

    let prefs = UserDefaults.standard
    prefs.set(selectedColorIndex, forKey: "selectedColorIndex")
    prefs.synchronize()
}

private func setTintColorForIndex(colorIndex: Int) {
    colorControl.tintColor = TinyPixUtils.getTintColorForIndex(index:
colorIndex)
}
```

Первый метод выполняется, когда пользователь изменяет выбор в сегментированном элементе управления. Он сохраняет выбранный индекс в настройках пользователя и передает его второму методу, который определяет цвет, соответствующий этому индексу, и применяет его к сегментированному элементу управления. Кроме того, нам необходимо запрограммировать преобразование индекса в цвет в детализированном контроллере представления, поэтому оно реализовано в отдельном классе. Для создания этого класса нажмите комбинацию клавиш `<⌘+N>` и откройте диалоговое окно открытия нового файла. В разделе `iOS` выберите пункт `Swift File` и щелкните на кнопке `Next`. Задайте имя файла `TinyPixUtils.swift` и щелкните на кнопке `Create`.

Теперь откройте файл `TinyPixUtils.swift`, чтобы реализовать необходимый нам метод.

```
import UIKit

class TinyPixUtils {
    class func getTintColorForIndex(index: Int) -> UIColor {
```

```

let color: UIColor
switch index {
case 0:
    color = UIColor.red

case 1:
    color = UIColor(red: 0, green: 0.6, blue: 0, alpha: 1)

case 2:
    color = UIColor.blue

default:
    color = UIColor.red
}
return color
}
}

```

Мы еще ничего не сделали с раскладовкой. Сначала нам нужно кое-что сделать в файле MasterViewController.swift. Начнем с метода viewDidLoad. После вызова его реализации в суперклассе добавим кнопку справа от панели навигации. Пользователь будет нажимать эту кнопку, чтобы создать новый документ класса TinyPix. Кроме того, мы загрузим сохраненный цвет из настроек пользователя и окрасим в этот цвет сегментированный элемент управления. Закончим работу, вызвав метод reloadFiles(), реализованный ранее. Добавьте в реализацию метода viewDidLoad() код, приведенный в листинге 14.6.

Листинг 14.6. Метод viewDidLoad в файле MasterViewController.swift

```

override func viewDidLoad() {
    super.viewDidLoad()

    let addButton = UIBarButtonItem(
        barButtonSystemItem: UIBarButtonSystemItem.add,
        target: self, action: #selector(MasterViewController.
insertNewObject))
    navigationItem.rightBarButtonItem = addButton

    let prefs = UserDefaults.standard
    let selectedColorIndex = prefs.integer(forKey: "selectedColorIndex")
    setTintColorForIndex(colorIndex: selectedColorIndex)
    colorControl.selectedSegmentIndex = selectedColorIndex

    reloadFiles()
}

```

При первом запуске приложения сегментированный элемент управления окрашивается в красный цвет. Это объясняется тем, что в настройках пользователя пока ничего не записано, и поэтому метод integerForKey() возвращает значение 0, которое метод setTintColorForIndex() интерпретирует как красный цвет.

Возможно, вы заметили, что, создавая объект класса `UIBarButtonItem`, мы указали ему при нажатии вызывать метод `insertNewObject()`. Пока мы не написали этот метод, поэтому добавьте в код следующие строки:

```
func insertNewObject() {
    let alert = UIAlertController(title: "Choose File Name",
                                  message: "Enter a name for your new TinyPix document",
                                  preferredStyle: .alert)

    alert.addTextField(configurationHandler: nil)

    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel,
                                     handler: nil)
    let createAction = UIAlertAction(title: "Create", style: .default) {
        action in
        let textField = alert.textFields![0] as UITextField
        self.createFileName(textField.text!)
    }

    alert.addAction(cancelAction)
    alert.addAction(createAction)

    present(alert, animated: true, completion: nil)
}
```

Этот метод использует класс `UIAlertController` для вывода предупреждения, содержащего поле редактирования, а также кнопки `Create` и `Cancel`. При нажатии кнопки `Create` ответственность за создание нового элемента возлагается на метод, который вызывается из блока обработки нажатия. Добавим этот метод:

```
private func createFileName(fileName: String) {
    let trimmedFileName = fileName.trimmingCharacters(
        in: NSCharacterSet.whitespaces)
    if !trimmedFileName.isEmpty {
        let targetName = trimmedFileName + ".tinypix"
        let saveUrl = urlForFileName(fileName: targetName)
        chosenDocument = TinyPixDocument(fileURL: saveUrl)
        chosenDocument?.save(to: saveUrl,
                             for: UIDocumentSaveOperation.forCreating,
                             completionHandler: { success in
            if success {
                print("Save OK")
                self.reloadFiles()
                self.performSegue(
                    withIdentifier: "masterToDetail",
                    sender: self)
            } else {
                print("Failed to save!")
            }
        })
    }
}
```

Начало этого метода довольно простое. Он отбрасывает ведущие и замыкающие пробелы из передаваемого имени. Если результат не пустой, он создает соответствующий файл, указатель URL на основе его имени (с помощью метода `urlForFileURLs()`, написанного нами ранее) и новый экземпляр класса `TinyPixDocument` на основе указателя URL.

Следующая часть метода немного сложнее. Здесь важно понимать, что простое создание нового документа по заданному указателю URL еще не означает создания файла. Фактически в момент вызова метода `init(fileURL:)` документ еще не знает, ссылается ли заданный указатель URL на существующий файл, или этот новый файл еще только предстоит создать. Мы должны сообщить ему, что делать. В данном случае с помощью приведенного ниже кода сообщаем ему, что он должен сохранить новый файл по заданному указателю URL.

```
chosenDocument?.save(to: saveUrl,
    for: UIDocumentSaveOperation.forCreating,
    completionHandler: { success in
    .
    .
    .
})
```

Особый интерес представляют цель и использование блока, который передается в качестве последнего аргумента. Этот метод, который мы назвали `save(to: saveUrl)`, не должен возвращать значение, чтобы сообщить нам, как он отработал. Фактически он возвращает управление сразу после своего вызова, задолго до того, как файл будет действительно сохранен. Он начинает процесс сохранения файла и позднее, когда он будет закончен, вызывает блок, который мы передали ему, используя параметр `success`, чтобы узнать, успешно ли выполнено задание. Для того чтобы все это работало как можно более гладко, процесс сохранения файла выполняется в фоновом потоке. Однако блок, который мы передаем методу `save(to: saveUrl)`, вызывается в главном потоке, так что можно безопасно использовать любые ресурсы, которые необходимы для главного потока, например библиотеку UIKit. Имея все это в виду, еще раз заглянем в блок.

```
if success {
    print("Save OK")
    self.reloadFiles()
    self.performSegue(withIdentifier: "masterToDetail", sender: self)
} else {
    print("Failed to save!")
}
```

Эти строки представляют собой содержимое блока, который мы передаем методу сохранения файла. Он вызывается после завершения операций с файлом. Затем проверяем, успешно ли выполнено задание; если успешно, то немедленно перезагружаем файл, а затем инициируем переход к контроллеру следующего

представления. Этот аспект переходов (*segues*) мы не освещали в главе 9, но он довольно прост. Идея заключается в том, что переход в раскладовке может иметь идентификатор, подобно обычной ячейке табличного представления, и этот идентификатор можно использовать для программного включения перехода. В данном случае нам просто следует помнить, что, достигнув перехода, мы должны выполнить его конфигурирование в раскладовке. Однако перед этим добавим в наш класс последний метод, который будет обеспечивать переход. Вставьте в файл *MasterViewController.swift* метод, приведенный в листинге 14.7.

Листинг 14.7. Метод *prepareForSegue* в файле *MasterViewController.swift*

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    let destination =
        segue.destinationViewController as! UINavigationController
    let detailVC =
        destination.topViewController as! DetailViewController
    if sender === self {
        // if sender === self, новый документ только что был создан,
        // и параметр chosenDocument уже задан.
        detailVC.detailItem = chosenDocument
    } else {
        // Найти выбранный документ из табличного представления
        if let indexPath = tableView.indexPathForSelectedRow {
            let docURL = documentFileURLs[(indexPath as NSIndexPath).row]
            chosenDocument = TinyPixDocument(fileURL: docURL)
            chosenDocument?.open() { success in
                if success {
                    print("Load OK")
                    detailVC.detailItem = self.chosenDocument
                } else {
                    print("Failed to load!")
                }
            }
        }
    }
}
```

Этот метод имеет две очевидные ветви, которые управляются условием, заданным в самом начале. В главе 9 мы указывали, что этот метод вызывается контроллером представления каждый раз, когда новый контроллер затачивается в стек навигации. Параметр *sender* указывает на объект, инициирующий переход. Мы используем его для того, чтобы выяснить, что делать дальше. Если переход инициирован программным способом, реализованным методом делегата представления предупреждений, то параметр *sender* будет равен *self*, потому что это значение параметра *sender* в методе *performSegue(withIdentifier:)* приводит к вызову метода *createFileNamed()*. В данном случае мы знаем, что свойство *chosenDocument* уже установлено, и просто передаем это значение соответствующему контроллеру представления.

В противном случае мы должны ответить на касание пользователя строки в табличном представлении, и ситуация немного усложняется. В этот момент необходимо создать указатель URL (точно так же, как при создании документа), новый экземпляр документного класса и попытаться открыть файл. Метод `open()`, который мы вызываем для открытия файла, работает аналогично методу, выполняющему сохранение файла. Мы передаем ему блок, который он сохраняет для дальнейшего выполнения. Как и метод сохранения файла, загрузка выполняется в фоновом потоке, а блок выполняется в главном потоке после завершения метода. Затем, если загрузка прошла успешно, передаем документ контроллеру детализированного представления.

Обратите внимание на то, что оба эти метода используют метод кодирования “ключ–значение”, который мы уже несколько раз использовали, позволяя задавать свойство `detailItem` контроллера, являющегося точкой назначения перехода, даже если мы не вставили в код его заголовок. Это очень удобно, поскольку класс детализированного представления `DetailViewController`, созданный как часть проекта Xcode, содержит стандартное свойство `detailItem`, но мы должны преобразовать его из типа `NSDate` в тип `AnyObject?`, как показано ниже.

```
var detailItem: AnyObject? {
```

К данному моменту мы создали довольно большую часть кода. Настало время сконфигурировать раскладовку, чтобы мы могли выполнять приложение и на экране что-то происходило. Сохраните код и следуйте за нами.

Начальная раскладовка

Выберите файл `Main.storyboard` в навигаторе проекта Xcode. Вы обнаружите в нем сцены для контроллера навигации, главного контроллера представления и контроллера детализированного представления (рис. 14.3). Вся наша работа будет связана с контроллерами главного и детализированного представлений.

Начнем со сцены контроллера главного представления. Именно здесь конфигуруется табличное представление, демонстрирующее все документы в приложении TinyPix. По умолчанию табличное представление этой сцены настраивается так, чтобы оно использовало статические ячейки, а не динамические. Мы хотим, чтобы наше табличное представление получало свое содержимое от методов источника данных, которые мы реализовали, поэтому настройки по умолчанию нас вполне устраивают. Однако нам необходимо конфигурировать прототип ячейки, поэтому выберите ее и откройте инспектор атрибутов. Измените атрибут ячейки `Identifier` с `Cell` на `FileCell`. Это откроет коду источника данных, написанному нами ранее, доступ к ячейке табличного представления.

Мы также должны создать переход, который будет переключаться в нашем коде. Для этого нажмите клавишу `<Control>`, перетащите указатель с пиктограммы контроллера главного представления (оранжевый кружок внизу сцены

или пиктограмма Master в окне Document Outline) на контроллер навигации в детализированном представлении (он расположен в нижнем левом углу, как показано на рис. 14.3; его также можно найти в окне Document Outline, раскрыв сцену Navigation Controller Scene) и выберите пункт Show Detail во всплывающем меню переходов.

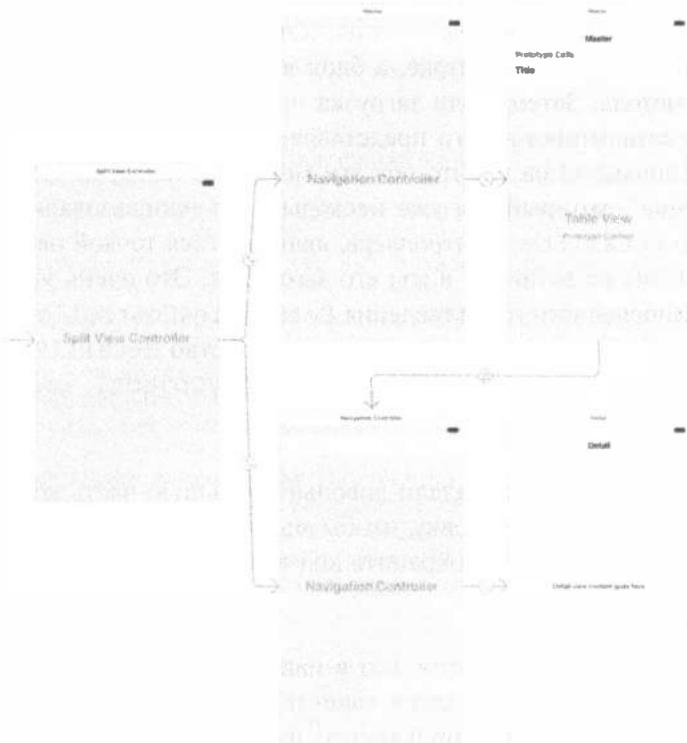


Рис. 14.3. Раскладка TinyPix с контроллером навигации, контроллерами главного и детализированного представлений

Вы увидите два перехода, соединяющих две сцены. Выбрав один из них, можете сообщить, откуда они исходят. Выбор одного перехода включает подсветку всей главной сцены; выбор другого включает подсветку только ячейки табличного представления. Выберите переход, который подсвечивает всю сцену, и с помощью инспектора атрибутов задайте значение его параметра Identifier равным masterToDetail.

Последнее вмешательство в контроллер главного представления позволит пользователю выбирать цвет, используемый для окрашивания включенной точки в детализированном представлении. Вместо реализации полноценной палитры цветов просто добавим сегментированный элемент управления, который позволит пользователю выбрать цвет из заданного заранее набора цветов.

Найдите элемент **Segmented Control** в библиотеке объектов и перетащите его на панель навигации в верхней части главного представления (рис. 14.4).

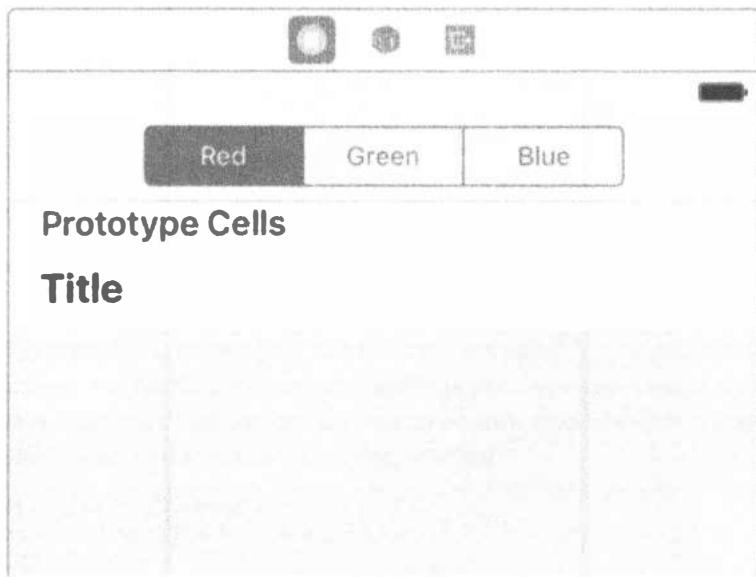


Рис. 14.4. Раскладовка TinyPix с контроллером главного представления и сегментированным элементом управления, расположенным на панели навигации контроллера

Выберите сегментированный элемент управления и откройте инспектор атрибутов. В разделе **Segmented Control**, расположенном в верхней части инспектора, измените количество сегментов с 2 на 3, используя счетчик **Segments**. Затем по очереди дважды щелкните на заголовке каждого сегмента, изменив их на **Black**, **Red** и **Green** соответственно.

Затем нажмите клавишу **<Control>**, перетащите указатель с сегментированного элемента управления на пиктограмму, представляющую главный контроллер (оранжевый кружок внизу сцены или пиктограмма **Master** в окне **Document Outline**), и выберите метод `chooseColor()`. Затем нажмите клавишу **<Control>**, перетащите указатель с главного контроллера обратно на сегментированный элемент управления и выберите выход `colorControl`. Мы связали сегментированный элемент управления с его выходом в контроллере главного представления, и при его выборе будет вызван соответствующий метод действия.

Мы достигли стадии, на которой можно скомпилировать и запустить приложение. Вы увидите его заставку и пустое табличное представление с сегментированным элементом управления в верхней части экрана, а также кнопку со знаком “плюс” в правом верхнем углу (рис. 14.5).



Рис. 14.5. Приложение TinyPix в момент первого запуска. Щелкните на пиктограмме со знаком “плюс”, чтобы добавить новый документ. Вы получите предложение назвать новый документ TinyPix. Пока детализированное представление просто отображает имя документа на метке

Нажмите кнопку Back и вернитесь к главному списку, в котором увидите добавленный вами элемент. Создайте еще один или несколько элементов, чтобы убедиться, что они правильно включаются в список. Затем вернитесь в среду Xcode, потому что у нас впереди еще много работы.

Создание класса TinyPixView

На следующем этапе необходимо создать класс представления, чтобы отображать на экране сетку и позволить пользователю ее редактировать. Выберите в навигаторе проекта папку TinyPix и нажмите комбинацию клавиш $\langle \text{⌘}+\text{T} \rangle$, чтобы создать новый файл. В разделе iOS Source выберите пиктограмму Cocoa Touch class и щелкните на кнопке Next. Назовите новый класс TinyPixView и выберите пункт UIView во всплывающем списке Subclass of. Щелкните на кнопке Next, убедитесь, что место для хранения файлов выбрано правильно, и щелкните на кнопке Create.

ЗАМЕЧАНИЕ. Реализация класса представления предусматривает операции рисования и обработки нажатий, которые мы еще не рассматривали. Вместо того чтобы заполнять эту главу множеством деталей, касающихся этих тем, мы просто продемонстрируем соответствующий код. Подробная информация о рисовании приведена в главе 16, а детали, касающиеся нажатий и перетаскивания, — в главе 18.

Выберите файл `TinyPixView.swift` и внесите в его начало перед определением класса следующие изменения:

```
struct GridIndex {
    var row: Int
    var column: Int
}
```

Здесь мы определили структуру, с помощью которой будем работать с парами “строка–столбец” сетки. Мы также определили расширение класса с несколькиими свойствами и закрытыми методами, которые нам понадобятся в дальнейшем, и синтезировали все наши свойства в реализации.

```
var document: TinyPixDocument!
var lastSize: CGSize = CGSize.zero
var gridRect: CGRect!
var blockSize: CGSize!
var gap: CGFloat = 0
var selectedBlockIndex: GridIndex = GridIndex(row: NSNotFound, column: NSNotFound)
```

Подкласс класса `UIView`, пустой по умолчанию, содержит метод `init(frame:)`, который является стандартным инициализатором класса `UIView`. Однако, поскольку этот класс будет загружаться из раскладовки, он будет инициализироваться с помощью метода `init(coder:)`. Мы реализуем оба так, чтобы каждый вызывал третий метод для инициализации свойств. Запишите в файл `TinyPixView.swift` код, представленный в листинге 14.8.

Листинг 14.8. Инициализатор файла `TinyPixView.swift`

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

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

private func commonInit() {
    calculateGridForSize(bounds.size)
}
```

Метод `calculateGridForSize()` определяет, насколько большими должны быть ячейки цветовой сетки, руководствуясь размером объекта класса `TinyPixView`. Вычисление размера сетки позволяет использовать одно и то же приложение на устройствах с экранами разных размеров и правильно изменять размер представления при вращении устройства. Добавьте в файл `TinyPixView.swift` следующую реализацию метода `calculateGridForSize()`:

```
private func calculateGridForSize(_ size: CGSize) {
    let space = min(size.width, size.height)
    gap = space/57
    let cellSide = gap * 6
    blockSize = CGSize(width: cellSide, height: cellSide)
    gridRect = CGRect(x: (size.width - space)/2, y: (size.height - space)/2,
                      width: space, height: space)
}
```

Идея, лежащая в основе этого метода, заключается в том, чтобы полностью накрыть представление сеткой по горизонтали или по вертикали в зависимости от того, какое из этих измерений больше, а затем отцентровать его вдоль более длинной оси. Для этого мы вычисляем размер каждой ячейки и просветов между ними и делим наименьшее из этих измерений на 57. Почему на 57? Потому что хотим выделить место для восьми ячеек, причем размер каждой ячейки должен быть в шесть раз больше, чем размер просветов между ними. Поскольку просвет должен находиться между двумя соседними ячейками, а также в начале и в конце каждой строки или столбца, нам нужно место для $(6 \times 8) + 9 = 57$ просветов. Определив размер просвета, мы можем вычислить размер каждой ячейки (умножив его на шесть). Мы используем эту информацию для установки значений свойства `blockSize`, которое задает размер каждой ячейки, и свойства `gridRect`, соответствующего пространству внутри представления, на котором действительно будет прорисовываться сетка ячеек.

Перейдем к функциям рисования. Мы замещаем стандартный метод `UIView drawRect()` и используем простой перебор всех блоков сетки, вызывая для каждого блока другой метод. Добавим следующий код и удалим комментарии вокруг метода `drawRect`:

```
override func draw(_ rect: CGRect) {
    if document != nil {
        let size = bounds.size
        if !size.equalTo(lastSize) {
            lastSize = size
            calculateGridForSize(size)
        }

        for row in 0 ..< 8 {
            for column in 0 ..< 8 {
                drawBlockAt(row: row, column: column)
            }
        }
    }
}
```

Перед тем как рисовать ячейки, сравниваем текущий размер представления, чтобы вычислить со свойством `lastSize`, и если они окажутся разными, то вызываем метод `calculateGridForSize()`. Это происходит, если представление прорисовывается впервые и при каждом изменении размеров, которое чаще всего происходит при вращении устройства.

Теперь добавим код, рисующий блок для каждой ячейки в сетке.

```
private func drawBlockAt(row: Int, column: Int) {
    let startX = gridRect.origin.x + gap
        + (blockSize.width + gap) * (7 - CGFloat(column)) + 1
    let startY = gridRect.origin.y + gap
        + (blockSize.height + gap) * CGFloat(row) + 1
    let blockFrame = CGRect(x: startX, y: startY,
                           width: blockSize.width, height: blockSize.height)
    let color = document.stateAt(row: row, column: column)
        ? UIColor.black() : UIColor.white()
    color.setFill()
    tintColor.setStroke()
    let path = UIBezierPath(rect: blockFrame)
    path.fill()
    path.stroke()
}
```

Этот код использует начало координат, а также размеры ячеек и просветов, заданные методом `calculateGridForSize()`, для вычисления координат каждой ячейки, а затем рисует ее с учетом текущего цвета для контура и белого или черного цвета для заполнения. Методы рисования будут описаны в главе 16.

В заключение добавим несколько методов, реагирующих на касание пользователя. Методы `touchesBegan(_:withEvent:)` и `touchesMoved(_:withEvent:)` являются стандартными методами, которые каждый подкласс класса `UIView` может реализовать самостоятельно, чтобы обработать касание пользователя в кадре представления. Мы обсудим эти методы в главе 19. Эти два метода используют другие методы, определенные в расширении класса, для вычисления координаты точки сетки в месте прикосновения и включают конкретное значение в документе. Добавьте эти четыре метода в конец файла перед закрывающей фигурной скобкой, как показано в листинге 14.9.

Листинг 14.9. Методы для обработки касаний пользователя, добавляемые в файл `TinyPixView.swift`

```
private func touchedGridIndexFromTouches(_ touches: Set<UITouch>)
    -> GridIndex {
    var result = GridIndex(row: -1, column: -1)
    let touch = touches.first!
    var location = touch.location(in: self)
    if gridRect.contains(location) {
        location.x -= gridRect.origin.x
        location.y -= gridRect.origin.y
        result.column = Int(8 - (location.x * 8.0 / gridRect.size.width))
        result.row = Int(location.y * 8.0 / gridRect.size.height)
    }
}
```

```

        }
        return result
    }

private func toggleSelectedBlock() {
    if selectedBlockIndex.row != -1
        && selectedBlockIndex.column != -1 {
        document.toggleStateAt(row: selectedBlockIndex.row,
                               column: selectedBlockIndex.column)
    document.undoManager?.prepare(withInvocationTarget: document)
        .toggleStateAt(row: selectedBlockIndex.row,
                       column: selectedBlockIndex.column)
    setNeedsDisplay()
}
}

override func touchesBegan(_ touches: Set<UITouch>,
                           with event: UIEvent?) {
    selectedBlockIndex = touchedGridIndexFromTouches(touches)
    toggleSelectedBlock()
}

override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
{
    let touched = touchedGridIndexFromTouches(touches)
    if touched.row != selectedBlockIndex.row
        && touched.column != selectedBlockIndex.column {
        selectedBlockIndex = touched
        toggleSelectedBlock()
    }
}
}

```

Внимательные читатели могли заметить, что метод `toggleSelectedBlock()` делает нечто необычное. После вызова метода `toggleStateAt(row:column:)` из класса документа он делает еще кое-что. Посмотрим еще раз.

```

private func toggleSelectedBlock() {
    if selectedBlockIndex.row != -1
        && selectedBlockIndex.column != -1 {
        document.toggleStateAt(row: selectedBlockIndex.row,
                               column: selectedBlockIndex.column)
    document.undoManager?.prepare(withInvocationTarget: document)
        .toggleStateAt(row: selectedBlockIndex.row,
                       column: selectedBlockIndex.column)
    setNeedsDisplay()
}
}

```

Вызов метода `document.undoManager()` возвращает экземпляр класса `NSUndoManager`. Мы не будем его использовать больше нигде в данной книге. Отметим лишь, что класс `NSUndoManager` является структурной основой функции “откат/повтор” в системах iOS и macOS. Идея заключается в том, чтобы при каждом выполнении действия в графическом пользовательском интерфейсе

вы использовали объект класса `NSUndoManager`, чтобы оставить навигационную цепочку при вызове метода записи, который будет выполнять откат. Объект класса `NSUndoManager` будет сохранять вызов метода в специальном стеке отката, который можно использовать для возвращения к состоянию документа при выполнении отката.

Для этого метод `prepare(withInvocationTarget:)` возвращает специальный объект-заместитель, которому можно послать любое сообщение, а это сообщение будет упаковано вместе с адресом и помещено в стек отката. Хотя это похоже на двойной вызов метода `toggleStateAt(row:column:)` в строке, на самом деле во второй раз метод не вызывается, а просто помещается в очередь для дальнейшего потенциального использования.

Зачем же мы это делаем? Ведь раньше мы обходились без использования функции “повтор/откат”, так что же изменилось сейчас? Причина заключается в том, что регистрация действия, допускающего откат, с помощью метода `NSUndoManager` из документного класса помечает документ как “черновик” и гарантирует, что в определенный момент он будет сохранен автоматически в течение нескольких секунд. Тот факт, что действия пользователя также допускают откат, — это просто “бантик”, по крайней мере в нашем приложении. В приложении с более сложной структурой документов возможность выполнять откат над документами может стать огромным преимуществом.

Сохраните внесенные изменения. Теперь, когда наш класс представления готов к использованию, вернемся к раскладовке, чтобы настроить графический пользовательский интерфейс для детализированного представления.

Раскладовка детализированного представления

Выберите файл `Main.storyboard`, найдите детализированную сцену и посмотрите на ее содержимое (в левом нижнем углу). Весь графический пользовательский интерфейс состоит из метки (“*Detail view content goes here*”), которая содержалась в описании документа при предыдущем запуске приложения. Эта метка не особенно полезна, поэтому выберите ее в контроллере детализированного представления и нажмите клавишу `<Delete>`, чтобы удалить. Найдите в библиотеке объектов элемент `UIImageView` и перетащите его на детализированное представление. Установите ширину и высоту представления так, чтобы оно заполнило всю строку заголовка (рис. 14.6).

Откройте инспектор идентичности, чтобы заменить экземпляр класса `UIImageView` экземпляром нашего собственного класса. Откройте в разделе `Custom Class` раскрывающийся список `Class` и выберите пункт `TinyPixView`. Затем откройте инспектор атрибутов и замените значение настройки `Mode` значением `Redraw`. В результате объект `TinyPixView` будет заново прорисовываться при изменении размеров. Это необходимо потому, что позиция в сетке внутри представления зависит от размера самого представления при вращении. Иерархия представления на детализированной сцене показана на рис. 14.7.



Рис. 14.6. Мы заменили метку в детализированном представлении другим представлением, отцентрированным по отношению к содержащему его представлению. Затем мы изменили его размеры, чтобы он заполнил всю область, и разместили под панелью инструментов



Рис. 14.7. Иерархия представлений в детализированной сцене

Прежде чем переходить к следующему этапу, мы должны настроить ограничения для нового представления. Мы хотим заполнить доступную область детализированным представлением. Перейдите в окно Document Outline, нажмите клавишу <Control>, перетащите указатель от объекта класса TinyPixView к его родительскому представлению и отпустите кнопку мыши. Нажмите клавишу <Shift> и во всплывающем меню выберите команды Leading Space to Container Margin, Trailing Space to Container Margin, Vertical Spacing to Top Layout Guide и Vertical Spacing to Bottom Layout Guide, а затем щелкните в любом месте за пределами меню, чтобы применить эти ограничения.

Теперь необходимо связать пользовательское представление с контроллером детализированного представления. Мы еще не подготовили выход для пользовательского представления, но система Xcode сделает это автоматически. Активизируйте помощник редактора. Рядом с окном редактора графического пользовательского интерфейса должно открыться окно текстового редактора, в котором отображается содержимое файла DetailViewController.swift. Если в нем отображается что-нибудь другое, воспользуйтесь панелью быстрых переходов, расположенной вверху окна текстового редактора, чтобы открыть файл DetailViewController.swift. Для того чтобы создать связь, нажмите клавишу <Control> и перетащите указатель с пиктограммы Tiny Pix View на код, отпустив кнопку мыши под существующим объектом IBOutlet в верхней части файла. Во всплывающем окне, которое появится на экране, убедитесь, что параметр Connection имеет значение Outlet, назовите новый выход pixView и щелкните на кнопке Connect. Затем удалите выход detailDescriptionLabel, поскольку мы не будем его использовать.

Вы должны увидеть, что добавление связи привело к появлению в файле DetailViewController.swift строки

```
@IBOutlet weak var pixView: TinyPixView!
```

Приступим к модификации метода configureView. Он не является стандартным методом класса UIViewController. Это закрытый метод, который шаблон проекта включает в класс как удобное средство для размещения кода, который должен обновлять представление после внесения в него каких-либо изменений. Поскольку мы не используем описательную метку, удалим строку, которая ее задает. Затем добавим немного кода для передачи выбранного документа пользовательскому представлению и заставим его перерисовать себя, вызвав метод setNeedsDisplay().

```
func configureView() {
    // Обновляем пользовательский интерфейс
    // для элемента детализированного представления.
    if detailItem != nil && isViewLoaded() {
        pixView.document = detailItem! as! TinyPixDocument
        pixView.setNeedsDisplay()
    }
}
```

Обратите внимание на вызов метода `isViewLoaded()` перед обновлением документа в объекте класса `TinyPixView`. Это необходимо, поскольку метод `configureView()` может быть вызван до того, как контроллер детализированного представления загрузит соответствующее приложение. В данном случае свойство `pixView` будет равным `nil` и приложение потерпит крах, если мы попробуем его использовать. Мы можем отложить обновление документа, потому что метод `configureView()` будет вызван снова из метода `viewDidLoad()` при реальной загрузке представления.

Затем нам необходимо задать цвет заполнения для объекта класса `TinyPixView`. Это необходимо делать как при первой загрузке представления, так и при каждом изменении цвета заполнения. Первоначальный цвет заполнения можно получить из настроек пользователя, поэтому мы добавим метод, извлекающий ранее сохраненное значение, преобразующий его в объект класса `UIColor` и применяющий его к объекту класса `TinyPixView`. Добавьте этот метод в любом месте класса.

```
private func updateTintColor() {
    let prefs = UserDefaults.standard
    let selectedColorIndex = prefs.integer(forKey: "selectedColorIndex")
    let tintColor = TinyPixUtils.getTintColorForIndex(selectedColorIndex)
    pixView.tintColor = tintColor
    pixView.setNeedsDisplay()
}
```

Этот метод необходимо вызывать для настройки первоначального цвета заполнения при первой загрузке представления и любом изменении цвета заполнения. Как же нам узнать, что это произошло? При изменении цвета заполнения новое значение сохраняется в настройках пользователя. Для того чтобы узнать о том, что настройки пользователя изменились, надо зарегистрировать наблюдателя для уведомления типа `NSUserDefaultsDidChangeNotification` в центре уведомлений, заданных по умолчанию. Добавьте в метод `viewDidLoad` следующий код:

```
updateTintColor()
NotificationCenter.default.addObserver(self,
    selector: #selector(DetailViewController.onSettingsChanged(_:)),
    name: UserDefaults.didChangeNotification, object: nil)
```

Теперь при любом изменении настроек пользователя будет вызываться метод `onSettingsChanged()`. В этом случае при изменении цвета необходимо задать новый цвет. Добавьте в класс следующую реализацию метода:

```
func onSettingsChanged(notification: NSNotification) {
    updateTintColor()
}
```

Добавив наблюдатель уведомлений, мы должны удалить его до того, как объект класса будет удален из памяти. Для этого нам нужен деструктор класса.

```
deinit {
    NSNotificationCenter.default.removeObserver(self,
        name: NSUserDefaults.didChangeNotification, object: nil)
}
```

Мы практически закончили работу над классом, но необходимо внести еще одно изменение. Помните, мы говорили об автосохранении документа после поступления уведомления о его редактировании, которое генерируется при регистрации действия, допускающего откат? Сохранение обычно выполняется в течение 10 секунд после редактирования. Как и остальные процедуры сохранения и загрузки, описанные ранее в этой главе, автосохранение выполняется в фоновом потоке, так что пользователи его даже не замечают. Однако этот механизм работает только при условии, что документ еще существует в памяти.

При существующих настройках есть риск, что, когда пользователь нажмет кнопку возврата, чтобы вернуться к главному списку, экземпляр документа будет удален из памяти без сохранения, и последние изменения пользователя будут удалены. Для того чтобы этого не случилось, мы должны добавить соответствующий код в метод `viewWillDisappear()`, чтобы закрыть документ, как только пользователь выйдет из детализированного представления. Закрытие документа вызывает автоматическое сохранение, которое также выполняется в фоновом потоке. В данном конкретном случае нам не надо ничего делать при сохранении, поэтому передаем вместо блока значение `nil`.

```
override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)
    if let doc = detailItem as? UIDocument {
        doc.close(completionHandler: nil)
    }
}
```

Итак, наша первая версия истинно документо-ориентированного приложения готова к испытаниям. Вы можете создавать новые документы, редактировать их, возвращаться к списку, а затем выбирать другой документ (или тот же самый), и все это прекрасно работает. Если вы откроете консоль Xcode, то при сохранении и загрузке документа каждый раз будете видеть определенный вывод на экране. Используя систему автосохранения, вы не имеете прямого контроля над процессом сохранения (только в момент закрытия документа), но бывает интересно просмотреть журналы, чтобы просто понять, что же происходит.

Добавление поддержки службы iCloud

В нашем распоряжении находится полноценное документо-ориентированное приложение, но мы не собираемся останавливаться на достигнутом. Модификация приложения TinyPix для работы со службой iCloud является чрезвычайно простой. Учитывая все, что происходит “за кулисами”, потребуется чрезвычайно мало изменений. Мы должны пересмотреть метод, загружающий список доступных файлов, и метод, задающий указатель URL для загрузки нового файла, и все.

Кроме изменения кода, нам также придется выполнить несколько административных условий. Компания Apple позволяет приложениям сохранять данные в службе iCloud, только если эти данные содержит встроенный профиль ресурсов, сконфигурированных для использования службы iCloud. Это означает, что, для того чтобы добавить в свое приложение поддержку службы iCloud, вы должны оплатить членство в организации разработчиков программ для системы iOS и инсталлировать свой сертификат разработчика. Кроме того, служба iOS работает только с реальными устройствами, а не с симулятором, поэтому, для того чтобы запустить приложение TinyPix с поддержкой службы iOS, необходимо иметь хотя бы одно устройство, работающее под управлением системы iOS и зарегистрированное в службе iCloud. С двумя устройствами вы получите еще больше удовольствия, поскольку сможете увидеть, как изменения передаются с одного устройства на другое. Сначала в окне Finder создайте копию проекта TinyPix, чтобы вносить изменения, не боясь бесповоротно испортить приложение. Откройте копию проекта, созданную в среде Xcode, и приступайте к работе.

Создание профиля ресурсов

Сначала необходимо создать профиль ресурсов iCloud для приложения TinyPix. Раньше для этого нужно было выполнить множество операций на веб-сайте разработчиков компании Apple, но сейчас среда Xcode делает легко. Перейдите в окно навигатора проекта, выберите пункт TinyPix в его верхней части и щелкните на вкладке Capabilities в области редактирования. Вы увидите что-то подобное рис. 14.8.

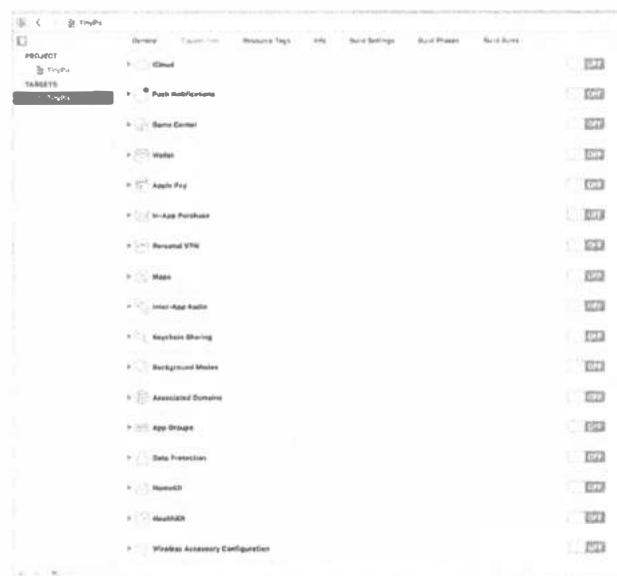


Рис. 14.8. Демонстрация легко настраиваемых технологий и служб в среде Xcode

Если вы не являетесь платным членом организации разработчиков программ, то можете использовать лишь небольшую часть функциональных возможностей, среди которых нет службы iCloud. В таком случае оформите подписку разработчика на веб-сайте компании Apple и попробуйте снова. Если ваша новая учетная запись отличается от бесплатной учетной записи, которую вы использовали до сих пор, то вам придется добавить ее в среду Xcode. Для этого выберите команду Xcode⇒Preferences на верхней панели меню и откройте диалоговое окно Preferences. Затем откройте вкладку Accounts (рис. 14.9).



Рис. 14.9. Вкладка Accounts демонстрирует доступные учетные записи разработчика

Щелкните на пиктограмме + и выберите команду Add Apple ID во всплывающем меню. Затем введите имя и пароль оплаченной учетной записи, которая будет добавлена в список доступных учетных записей на вкладке Accounts. Закройте вкладку Preferences и выберите вкладку General в окне редактора Xcode. В разделе Identity вы увидите список Team (рис. 14.10).

Выберите свою учетную запись разработчика и вернитесь на вкладку Capabilities. Если все сделано правильно, то вы увидите полный список функциональных возможностей, показанный на рис. 14.8.

Функциональные возможности, показанные на рис. 14.7, можно настроить непосредственно в среде Xcode. Для этого нужно зайти на веб-сайт, создать и загрузить профиль ресурсов и т.д. Однако сначала необходимо получить уникальный идентификатор приложения. Например, для того чтобы изменить идентификатор приложения на со.тумСо, необходимо задать идентификатор пакета, как показано на рис. 14.11.

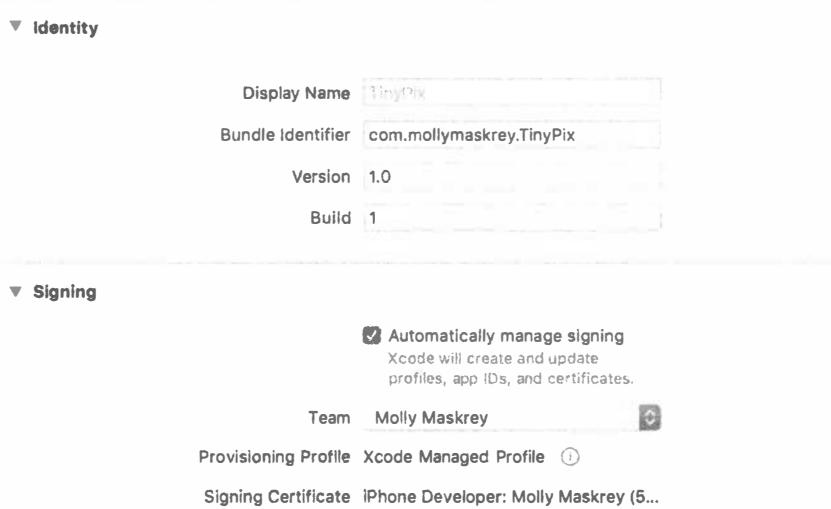


Рис. 14.10. Использование списка Team для подключения учетной записи разработчика в среде Xcode

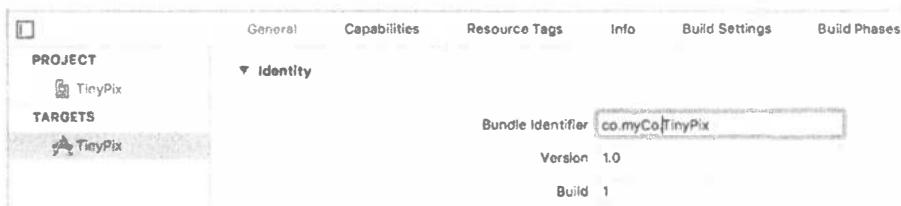


Рис. 14.11. Изменение идентификатора пакета

Конечно, вы должны использовать уникальное значение, а не со.myCo. Вернитесь на вкладку Capabilities. Мы хотим, чтобы приложение TinyPix имело доступ к службе iCloud (это первая возможность, указанная в списке), поэтому щелкните на треугольнике раскрытия, расположенному рядом с пиктограммой облака. Вы увидите информацию об этой возможности. Включите переключатель, расположенный справа. Среда Xcode свяжется с серверами компании Apple, чтобы настроить профиль ресурсов для данного приложения. Если выбранный вами идентификатор пакета уже кем-то зарегистрирован, то в нижней части раздела iCloud вы увидите красную пиктограмму и сообщение об ошибке. Вернитесь на вкладку General, выберите другой идентификатор пакета и попробуйте снова. После этого установите флагки Key-value storage и iCloud Documents (рис. 14.12).

Теперь ваше приложение имеет необходимые разрешения для доступа к службе iCloud из вашего кода. Остальное — дело обычного программирования.



Рис. 14.12. Приложение, настроенное на работу со службой iCloud

Как послать запрос

Выберите файл `MasterViewController.swift`, чтобы мы могли внести в него изменения, необходимые для использования службы iCloud. Самое крупное изменение касается способа поиска доступных документов. В первой версии приложения `TinyPix` мы использовали объект класса `FileManager`, чтобы увидеть доступные файлы в локальной файловой системе. На этот раз мы решим эту задачу немного иначе — пошлем особый запрос на поиск документов.

Для начала добавим свойство в класс, содержащий указатель на запрос.

```
@IBOutlet var colorControl: UISegmentedControl!
private var documentFileURLs: [URL] = []
private var chosenDocument: TinyPixDocument?
private var query: NSMetadataQuery!
```

Теперь напишем новый метод для просмотра списка файлов. Удалите весь метод `reloadFiles()` и замените его кодом, приведенным в листинге 14.10.

Листинг 14.10. Метод `reloadFiles` для работы со службой iCloud

```
private func reloadFiles() {
    let fileManager = FileManager.default

    // Передача значения nil здесь не вызывает проблем
    let cloudURL = fileManager.urlForUbiquityContainerIdentifier(nil)
    print("Got cloudURL \(cloudURL)")
    if (cloudURL != nil) {
        query = NSMetadataQuery()
        query.predicate = Predicate(format: "%K like '*.%tinypix%'", NSMetadataItemFSNameKey)
        query.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope]
```

```

NotificationCenter.default.addObserver(self,
    selector:#selector(MasterViewController.
    updateUbiquitousDocuments(_:)),
    name: NSNotification.Name.
        NSMetadataQueryDidFinishGathering,
    object: nil)
NotificationCenter.default.addObserver(self,
    selector: #selector(MasterViewController.
    updateUbiquitousDocuments(_:)),
    name: NSNotification.Name.
        NSMetadataQueryDidUpdate,
    object: nil)
query.start()
}
}

```

Здесь есть несколько новых моментов, достойных отдельного обсуждения. Первым является следующая строка:

```
let cloudURL = fileManager.urlForUbiquityContainerIdentifier(nil)
```

Что значит слово “ubiquity”? О чём идет речь? При работе со службой iCloud часто приходится встречать термины компании Apple, касающиеся идентификации ресурсов в хранилище iCloud и включающие в себя слова “ubiquity” и “ubiquitous”, означающие “вездесущий”, “повсеместный”, т.е. ресурс, доступный с любого устройства посредством одних и тех же сертификатов регистрации в службе iCloud.

В данном случае мы просим файловый менеджер дать нам базовый указатель URL, который откроет доступ к каталогу службы iCloud, связанному с идентификатором конкретного контейнера. Идентификатор контейнера обычно представляет собой строку, содержащую уникальный идентификатор пакета вашей компании и идентификатор приложения. Она используется для получения разрешений службы iCloud, содержащихся в вашем приложении. Передача значения nil здесь представляет упрощение: оно означает просьбу “вернуть первый элемент, содержащийся в списке”. Поскольку наше приложение содержит в списке только один элемент (это видно по списку *Containers*, показанному на рис. 14.12), такое упрощение целиком оправдано.

Создадим и настроим экземпляр класса NSMetadataQuery.

```
query = NSMetadataQuery()
query.predicate = NSPredicate(format: "%K like '*.%tinypix*'",
    NSMetadataItemFSNameKey)
query.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope]
```

Класс NSMetadataQuery изначально был написан для использования в механизме поиска Spotlight в операционной системе OS X (macOS), но в настоящее время он имеет дополнительные функции, обеспечивающие поиск в каталогах службы iCloud по запросам приложений для системы iOS. Мы передаем запросу

предикат, ограничивающий результаты поиска заданным именем файла, а также указываем диапазон поиска, чтобы служба iCloud искала файл только в папке Documents хранилища iCloud, выделенного для приложения. Затем настраиваем несколько уведомлений, позволяющих узнать, выполнен ли запрос, и посылаем сам запрос.

```
NotificationCenter.default.addObserver(self,
    selector: #selector(MasterViewController.
        updateUbiquitousDocuments(_:)),
    name: NSNotification.Name.NSMetadataQueryDidFinishGathering,
    object: nil)
NotificationCenter.default.addObserver(self,
    selector: #selector(MasterViewController.
        updateUbiquitousDocuments(_:)),
    name: NSNotification.Name.NSMetadataQueryDidUpdate,
    object: nil)

query.start()
```

Теперь необходимо реализовать метод, который вызовут эти уведомления, когда запрос будет выполнен. Добавьте сразу после метода `reloadFiles()` код, приведенный в листинге 14.11.

Листинг 14.11. Модифицированный метод обработки уведомлений

```
func updateUbiquitousDocuments(_ notification: Notification) {
    documentFileURLs = []

    print("updateUbiquitousDocuments, results = \(query.results)")
    let results = query.results.sorted() { obj1, obj2 in
        let item1 = obj1 as! NSMetadataItem
        let item2 = obj2 as! NSMetadataItem
        let item1Date =
            item1.value(forAttribute: NSMetadataItemFSCreationDateKey)
            as! Date
        let item2Date =
            item2.value(forAttribute: NSMetadataItemFSCreationDateKey)
            as! Date
        let result = item1Date.compare(item2Date)
        return result == ComparisonResult.orderedAscending
    }
    for item in results as! [NSMetadataItem] {
        let url = item.value(forAttribute: NSMetadataItemURLKey) as! URL
        documentFileURLs.append(url)
    }
    tableView.reloadData()
}
```

Результаты выполнения запроса содержат список объектов класса `NSMetadataItem`, из которого мы можем извлечь такие сведения, как указатели `URL` и даты создания файлов. Мы используем эти данные для сортировки файлов по дате, а затем запоминаем все указатели `URL` для дальнейшего использования

Местоположение файлов

Следующее изменение касается метода `urlForFileName:`, который также будет совершенно другим. Здесь мы используем повсеместный указатель URL, чтобы создать полный путь URL для заданного имени файла. Мы также вставляем слово "Documents" в сгенерированный путь, чтобы ограничиться папкой Documents. Удалите старый метод и замените его следующим:

```
private func urlForFileName(_ fileName: String) -> URL {
    // Не забудьте вставить в путь слово "Documents"
    let fm = FileManager.default
    let baseURL = fm.urlForUbiquityContainerIdentifier(nil)
    let pathURL = try! baseURL?.appendingPathComponent("Documents")
    let destinationURL = try! pathURL?.appendingPathComponent(fileName)
    return destinationURL!
}
```

Соберем и запустим наше приложение на реальном устройстве, работающем под управлением системы iOS (не на симуляторе). Если вы уже запускали свое приложение на этом устройстве, то обнаружите, что все данные, созданные приложением TinyPix ранее, увидеть невозможно. Новая версия приложения игнорирует локальный каталог Documents, выделенный для приложения, и полагается исключительно на службу iCloud. Однако мы должны иметь возможность создавать новые документы и восстанавливать их состояние после выхода и повторного запуска приложения. Более того, мы можем даже вообще удалить приложение TinyPix со своего устройства, а затем запустить его из среды Xcode и обнаружить, что все наши документы, сохраненные в службе iCloud, по-прежнему доступны. Если у вас есть еще одно устройство, работающее под управлением системы iOS и сконфигурированное для того же пользователя службы iCloud, запустите свое приложение на этом устройстве, и увидите, что те же самые документы появятся на другом устройстве! Эти документы можно найти в разделе iCloud приложения Settings на вашем устройстве iOS (выполните команду Storage⇒Manage Storage⇒TinyPix), а также в разделе iCloud приложения System Preferences на компьютере Mac под управлением системы OS X 10.8 или более современной.

Сохранение настроек в службе iCloud

Приложив еще немного усилий, можно обеспечить еще одну функциональную возможность службы iCloud. Поддержка службы iCloud в системе iOS обеспечивается классом `NSUserDefaults`, который напоминает класс `User Defaults`; однако в отличие от класса `User Defaults` его ключи и значения хранятся в облаке. Это очень важная возможность для настроек приложения, токенов регистрации и любой информации, которая не содержится в документе, но может оказаться полезной при совместной работе с другими устройствами.

В приложении TinyPix мы используем эту функциональную возможность для того, чтобы пользователь мог сохранить предпочтаемый цвет выделения. Таким образом, вместо настройки цвета для каждого устройства пользователь может задать цвет только один раз, и он будет использоваться всюду.

Опишем план действий.

- ❖ Когда пользователь изменяет цвет окраски, мы сохраняем новое значение в объекте класса `UserDefaults` и можем сохранить его в объекте класса `NSUserDefaultsKeyStore`, чтобы сделать его доступным для приложений на других устройствах.
- ❖ Мы регистрируемся на получение уведомления об изменениях в объекте класса `NSUserDefaultsKeyStore`. Вместе с уведомлением об изменении мы получим новое значение цвета. В этот момент мы должны обновить сегментированный элемент управления и изменить цвет его окраски с помощью контроллера главного представления, а также изменить цвет контура с помощью контроллера детализированного представления. Мы не будем делать это непосредственно, а просто сохраним новое значение цвета окраски в объекте класса `UserDefault`s. Изменение объекта класса `UserDefault`s приводит к генерированию уведомления. Контроллер детализированного представления уже ждет этого уведомления, поэтому он будет обновляться автоматически. Мы внесем небольшие изменения в контроллер главного представления, чтобы он делал то же самое.

Важно помнить, что модификации объекта класса `NSUserDefaultsKeyStore` не распространяются автоматически по всем другим устройствам. Если устройство не подключено к службе iCloud по какой-либо причине, она не увидит изменений до следующего сеанса связи. Таким образом, не следует ожидать немедленных изменений.

Для начала зарегистрируемся на получение уведомлений об изменениях от хранилища ключей и значений службы iCloud. Откройте файл `AppDelegate.swift` и добавьте следующий код в метод `application(_ application:, didFinishLaunchingWithOptions:)`, как показано в листинге 14.12.

Листинг 14.12. Модифицированный метод `application(_ application:, didFinishLaunchingWithOptions:)`

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions
    launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Точка замещения для настройки после запуска приложения.
    let splitViewController = self.window!.rootViewController
        as! UISplitViewController
    let navigationController = splitViewController.viewControllers[
        splitViewController.viewControllers.count-1]
        as! UINavigationController
    navigationController.topViewController!.navigationItem.leftBarButtonItem =
    splitViewController.displayModeButtonItem()
    splitViewController.delegate = self
}
```

```

// Регистрация на получение уведомлений
// об изменениях ключей и значений iCloud
NotificationCenter.default.addObserver(self,
    selector: #selector(AppDelegate.iCloudKeysChanged(_:)),
    name: NSUbiquitousKeyValueStore.didChangeExternallyNotification,
    object: nil)

// Начало обновлений ключей и значений iCloud
NSUbiquitousKeyValueStore.default().synchronize()
updateUserDefaultsFromICloud()
return true
}

```

Первая новая строка настраивает метод делегата приложения `iCloudKeysChanged()`, чтобы он вызывался при создании объекта класса `NSUbiquitousKeyValueStoreDidChangeEventNotification`, т.е. когда служба iCloud изменяет какую-нибудь пару “ключ–значение”, связанную с приложением. Метод `Synchronize` гарантирует, что локальные изменения в объекте класса `NSUbiquitousKeyValueStore` будут записываться в службе iCloud в фоновом режиме и при этом будут выпускаться уведомления об удаленных изменениях. Метод `updateUserDefaultsFromICloud()`, который мы вскоре увидим, получает текущее состояние выбранного цвета окраски из хранилища ключей и значений службы iCloud, если они заданы, и сохраняет их в локальных настройках пользователя, поэтому они будут использоваться немедленно.

Далее добавим реализации методов `iCloudKeysChanged()` и `updateUserDefaultsFromCloud()`, как показано в листинге 14.13.

Листинг 14.13. Обновление ключей и значений iCloud

```

func iCloudKeysChanged(_ notification: Notification) {
    updateUserDefaultsFromICloud()
}

private func updateUserDefaultsFromICloud() {
    let values = NSUbiquitousKeyValueStore.default().
    dictionaryRepresentation
    if values["selectedColorIndex"] != nil {
        let selectedColorIndex =
            Int(NSUbiquitousKeyValueStore.default().longLong(
                forKey: "selectedColorIndex"))
        let prefs = UserDefaults.standard
        prefs.set(selectedColorIndex, forKey: "selectedColorIndex")
        prefs.synchronize()
    }
}

```

Когда появится уведомление, мы получим индекс цвета окраски из хранилища ключей с помощью метода `longLong(forKey:)`. Этот интерфейс прикладного программирования очень напоминает класс `UserDefault`s, но в нем нет метода для хранения значений типа `integer`, поэтому мы будем хранить его

как значение типа long long. Получив это значение, мы просто скопируем в объект класса UserDefaults и синхронизируем изменения, сгенерировав уведомление. Как нам известно, получив это уведомление, контроллер детализированного представления автоматически выполняет обновление. Далее нам необходимо изменить контроллер главного представления, чтобы он делал то же самое. Вернитесь к файлу MasterViewController.swift и зарегистрируйте этот контроллер для получения уведомлений об изменениях объектов класса UserDefaults с помощью метода viewDidLoad().

```
reloadFiles()

NotificationCenter.default.addObserver(self,
    selector: #selector(MasterViewController.onSettingsChanged(_:)),
    name: UserDefaults.didChangeNotification,
    object: nil)
```

Затем добавим метод onSettingsChanged().

```
func onSettingsChanged(_ notification: Notification) {
    let prefs = UserDefaults.standard
    let selectedColorIndex = prefs.integer(forKey: "selectedColorIndex")
    setTintColorForIndex(selectedColorIndex)
    colorControl.selectedSegmentIndex = selectedColorIndex
}
```

Этот метод обновляет цвет сегментированного элемента управления с помощью того же метода, который вызывается при нажатии одного из его сегментов, но он получает индекс цвета от объекта класса UserDefaults, а не от элемента управления.

В заключение, когда пользователь изменяет цвет, необходимо сохранить новый индекс в хранилище ключей и значений iCloud. Для этого внесите в метод chooseColor() следующие строки.

```
@IBAction func chooseColor(_ sender: UISegmentedControl) {
    let selectedColorIndex = sender.selectedSegmentIndex
    setTintColorForIndex(selectedColorIndex)

    let prefs = UserDefaults.standard
    prefs.set(selectedColorIndex, forKey: "selectedColorIndex")
    prefs.synchronize()

    NSUbiquitousKeyValueStore.default()
        .set(Int64(selectedColorIndex),
              forKey: "selectedColorIndex")
    NSUbiquitousKeyValueStore.default().synchronize()
}
```

Все! Теперь можете запустить приложение на нескольких устройствах, сконфигурированных для одного и того же пользователя службы iCloud, и вы увидите, что настройка цвета на одном устройстве изменит цвет на другом устройстве, когда вы в следующий раз откроете на нем документ.

Резюме

Мы изложили основы работы со службой iCloud, а также процесс создания и запуска документо-ориентированного приложения, но обошли вниманием много вопросов, которые вы, возможно, хотели бы рассмотреть. Мы не собираемся касаться этих тем в данной книге, но если вы серьезно решили разрабатывать приложения с использованием службы iCloud, вам стоит подумать о следующем.

- Документы, хранящиеся в службе iCloud, уязвимы для конфликтов. Что произойдет, если вы редактируете один и тот же файл приложения TinyPix на нескольких устройствах одновременно? К счастью, компания Apple уже подумала об этом и предоставила несколько способов разрешения таких конфликтов, которые могут возникнуть в вашем приложении. Вы сами должны решить, хотите ли вы игнорировать конфликты, попытаться разрешить их автоматически или попросить пользователя помочь вам решить проблему. Подробности, касающиеся разрешения конфликтов между документами, можно найти в документации среды Xcode.
- Компания Apple рекомендует разрабатывать полностью автономные приложения, если у вас по каким-то причинам нет возможности использовать службу iCloud. Она также рекомендует обеспечить способ, с помощью которого пользователь мог бы осуществлять обмен файлами между хранищем iCloud и локальным хранилищем. К сожалению, компания Apple не предоставляет никакого стандартного графического пользовательского интерфейса, который мог бы помочь пользователям решить эту проблему, а современные приложения, реализующие эту функциональную возможность, такие как iWork компании Apple, решают проблему специфическим образом. Более подробную информацию по этой теме можно найти в разделе “Managing the Life Cycle of a Document” документации среды Xcode.
- Компания Apple поддерживает использование службы iCloud для реализации хранилища Core Data и даже предоставила класс `UIManagedDocument`, из которого по необходимости можно выводить подклассы. Более подробную информацию о создании приложений, использующих службу iCloud для хранения данных Core Data, следует искать в описании класса `UIManagedDocument` в справочнике. Эта архитектура намного сложнее и проблематичнее, чем обычное хранилище документов в службе iCloud. В последних версиях системы iOS компания Apple предприняла меры, чтобы исправить положение, но она пока работает не слишком хорошо.

ГЛАВА 15

Многопотоковое программирование с помощью технологии **Grand Central Dispatch**

Несмотря на то что многопотоковое программированием в любой среде, на первый взгляд, может показаться слишком сложным (рис. 15.1), компания Apple разработала новую технологию, **Grand Central Dispatch (GCD)**, которая намного упрощает такую работу. Она сочетает в себе возможности языка, библиотек времени выполнения и системных функций, обеспечивая всеобъемлющую и исчерпывающую поддержку параллельного выполнения кода на многопроцессорном аппаратном обеспечении, работающем под управлением операционных систем iOS и macOS.

К числу самых трудных задач, которые разработчикам приходится решать в настоящее время, относится написание программ, способных выполнять сложные операции в ответ на вводимые пользователем данные и в то же время оставаться чутко реагирующими на действия пользователя, чтобы ему не нужно было все время ждать завершения

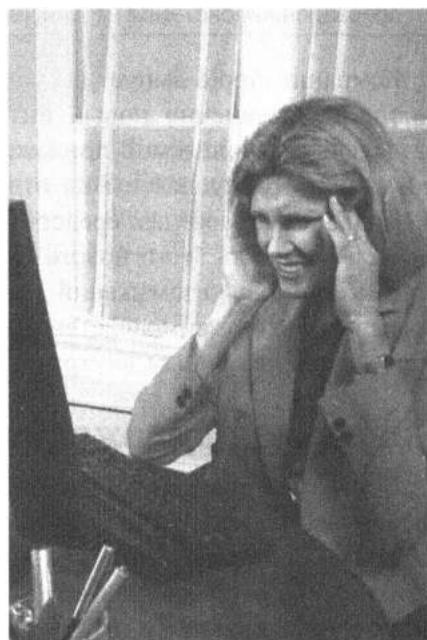


Рис. 15.1. Программирование многопотоковых приложений — работа не для слабонервных

задания, “подспудно” выполняемого процессором. Эта проблема возникла давно, но все еще остается актуальной, несмотря на все успехи вычислительной техники, способствующие неуклонному повышению быстродействия центральных процессоров — достаточно взглянуть на экран стоящего рядом компьютерного монитора. Когда вы в последний раз работали на компьютере, ваша работа, скорее всего, не раз прерывалась и на экране вращался шарик или другой курсор, свидетельствовавший о выполнении задания и вынуждавший вас ждать.

Одна из причин такого положения дел заключается в том, что программы, как правило, пишутся в виде последовательного ряда событий, обрабатываемых по очереди. Такие программы допускают вертикальное масштабирование по мере роста быстродействия центральных процессоров, но только до определенного предела. Как только выполнение программы приостанавливается в ожидании внешнего ресурса, например файла или сетевого соединения, вся последовательность событий фактически прерывается. Во всех современных операционных системах теперь можно пользоваться несколькими потоками выполнения в программе, чтобы даже в случае приостановки выполнения одного потока в ожидании конкретного события остальные потоки продолжали действовать. Тем не менее многие разработчики относятся к многопоточному программированию как к разновидности магии и стараются его избегать.

ЗАМЕЧАНИЕ. Поток — это последовательность инструкций, которые обрабатываются операционной системой независимо одна от другой.

Компания Apple выпустила каркас Grand Central Dispatch (GCD), предоставляющий совершенно новый интерфейс прикладного программирования для разделения выполняемой приложением работы на более мелкие части, которые могут быть распределены по отдельным потокам выполнения, а при наличии подходящих аппаратных средств — по нескольким центральным процессорам.

Большая часть этого нового интерфейса прикладного программирования доступна с помощью замыканий в языке Swift, предоставляющих удобный способ организовывать взаимодействие разных объектов и в то же время сохранять код более тесно связанным внутри методов.

Создание приложения SlowWorker

В качестве примера для демонстрации принципа действия каркаса GCD создадим несложное приложение SlowWorker, состоящее из простого интерфейса, управляемого единственной кнопкой и текстовым представлением. Достаточно щелкнуть на этой кнопке, чтобы сразу же началось выполнение синхронного задания, блокирующего работу интерфейса данного приложения приблизительно на десять секунд. По завершении этого задания в текстовом представлении появляется некоторый текст, как показано на рис. 15.2.



Рис. 15.2. Интерфейс приложения SlowWorker скрывается за единственной кнопкой. Если щелкнуть на этой кнопке, интерфейс блокируется приблизительно на десять секунд, а приложение продолжает свою работу

Начнем с создания проекта в среде Xcode, используя шаблон Single View Application. Присвойте новому проекту имя SlowWorker, выберите в списке Devices пункт Universal, щелкните на кнопке Next, чтобы сохранить проект, и т.д. Затем внесите в исходный код из файла ViewController.swift изменения, приведенные в листинге 15.1.

Листинг 15.1. Добавьте эти методы в файл ViewController.swift

```
@IBOutlet var startButton: UIButton!
@IBOutlet var resultsTextView: UITextView!

func fetchSomethingFromServer() -> String {
    Thread.sleep(forTimeInterval: 1)
    return "Hi there"
}
```

570 ГЛАВА 15 ■ МНОГОПОТОКОВОЕ ПРОГРАММИРОВАНИЕ...

```
func processData(_ data: String) -> String {
    Thread.sleep(forTimeInterval: 2)
    return data.uppercased()
}

func calculateFirstResult(_ data: String) -> String {
    Thread.sleep(forTimeInterval: 3)
    return "Number of chars: \(data.characters.count)"
}

func calculateSecondResult(_ data: String) -> String {
    Thread.sleep(forTimeInterval: 4)
    return data.replacingOccurrences(of: "E", with: "e")
}

@IBAction func doWork(_ sender: AnyObject) {
    let startTime = NSDate()
    self.resultsTextView.text = ""
    let fetchedData = self.fetchSomethingFromServer()
    let processedData = self.processData(fetchedData)
    let firstResult = self.calculateFirstResult(processedData)
    let secondResult = self.calculateSecondResult(processedData)
    let resultsSummary =
        "First: \(firstResult)\nSecond: \(secondResult)"
    self.resultsTextView.text = resultsSummary
    let endTime = NSDate()
    print("Completed in \(endTime.timeIntervalSince(startTime as Date)) seconds")
}
```

Как видите, вся работа приведенного выше класса, какой бы она ни была, разделяется на ряд мелких этапов. Код этого класса предназначен лишь для имитации некоторых медленно выполняемых операций, и ни один из его методов не делает ничего такого, что вообще отнимало бы сколько-нибудь времени. Однако самое интересное, что каждый его метод содержит вызов метода `sleep(forTimeInterval:)` из класса `Thread`, благодаря чему данное приложение, а точнее, поток выполнения, из которого этот метод вызывается, по существу, приостанавливается, ничего не делая в течение промежутка времени, заданного в секундах. Кроме того, в начале и в конце метода `doWork()` содержится код для расчета времени, необходимого для выполнения всей работы.

Откройте файл `Main.storyboard` и перетащите объекты `Button` и `Text View` в пустое окно `View`, расположив их так, как показано на рис. 15.3. Вы увидите текст, заданный по умолчанию. Удалите его из объекта `Text View` и измените название кнопки на `Start Working`. Для того чтобы задать ограничения `Auto Layout`, щелкните сначала на кнопке `Start Working`, а затем на кнопке `Align`, расположенной в правом нижнем углу области редактирования. Во всплывающем меню выберите команду `Horizontally in Container` и щелкните на кнопке `Add 1 Constraint`. Далее нажмите клавишу `<Control>` и перетащите указатель от данной кнопки к верхнему краю окна `View`, отпустите кнопку мыши и выберите во всплывающем меню команду `Vertical Spacing to Top Layout Guide`. Для того чтобы

завершить наложение ограничений на расположение данной кнопки, нажмите клавишу <Control> и перетащите указатель от данной кнопки вниз к текстовому представлению, отпустите кнопку мыши и выберите во всплывающем меню команду Vertical Spacing. Для того чтобы зафиксировать положение и размеры текстового представления, раскройте сцену View Controller Scene в окне Document Outline, нажмите клавишу <Control> и перетащите указатель от текстового представления в разметке на пиктограмму View в окне Document Outline. Отпустите кнопку мыши. Когда появится всплывающее меню, нажмите клавишу <Shift> и выберите команды Leading Space to Container Margin, Trailing Space to Container Margin и Vertical Spacing to Bottom Layout Guide, а затем нажмите клавишу <Return>, чтобы применить заданные ограничения. На этом автоматическое наложение ограничений на макет данного приложения завершается.

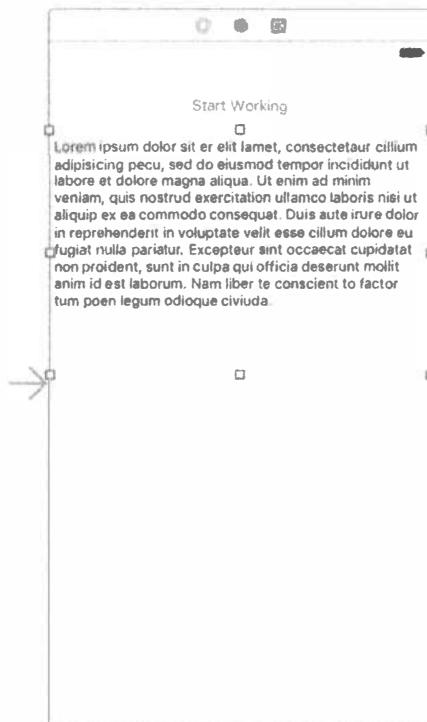


Рис. 15.3. Интерфейс приложения SlowWorker
состоит из кнопки и текстового представления

Нажмите клавишу <Control> и перетащите указатель от пиктограммы View Controller в окно Document Outline, чтобы связать два выхода контроллера представления (т.е. переменные экземпляра startButton и resultsTextView) с кнопкой и текстовым представлением.

Затем нажмите клавишу <Control> и перетащите указатель от кнопки на пиктограмму View Controller, отпустите кнопку мыши и выберите метод doWork() во

всплывающем меню. Этот метод будет вызываться при нажатии кнопки. В заключение выберите текстовое представление и перейдите к инспектору атрибутов, чтобы сбросить флагок *Editable* в правом верхнем углу окна данного инспектора, а затем удалите исходный текст из текстового представления.

Сохраните результаты своей работы и выполните команду *Run*. Приложение должно запуститься на выполнение, и если вы щелкнете на кнопке, то в течение примерно десяти секунд, т.е. суммарного времени ожидания, будет выполняться задание, прежде чем на экране появятся результаты. Ожидая этих результатов, обратите внимание на то, что кнопка *Start Working* все это время заметно обесцвечивается и не возвращается к своему обычному цвету до тех пор, пока задание не будет выполнено. Кроме того, текстовое представление данного приложения никак не реагирует до тех пор, пока не будет выполнено задание. Касание любого участка экрана не дает никакого результата. В действительности единственный способ взаимодействия с приложением в течение всего этого времени состоит в том, чтобы нажать главную кнопку и выйти из данного приложения. Однако именно этой ситуации нам и хотелось бы избежать.

Основы многопотоковой работы

Прежде чем приступить к реализации конкретных решений, сделаем краткий обзор основ многозадачности. Он, конечно, далек от полноценного описания многопотоковой работы как в системе iOS, так и вообще, но мы попробуем в достаточной мере разъяснить то, что собираемся делать далее в этой главе. В большинстве современных операционных систем, в том числе в iOS, поддерживается принцип потоков выполнения. Каждый процесс может состоять из многих параллельно выполняемых потоков. При наличии одноядерного процессора операционная система будет осуществлять переключение среди всех выполняющихся потоков подобно переключению среди всех выполняющихся процессов. При наличии многоядерного процессора потоки выполнения, как, впрочем, и процессы, будут распределяться по отдельным ядрам процессора.

Все потоки выполнения в процессе разделяют один и тот же общий код выполняемой программы и одни и те же глобальные данные. Кроме того, у каждого потока выполнения могут быть только его собственные данные. В потоках выполнения может использоваться специальная структура, называемая *мьютексом* (т.е. взаимным исключением), или *блокировкой* для гарантии того, что данный конкретный фрагмент кода не будет одновременно выполняться в нескольких потоках. С помощью такого механизма взаимного исключения обеспечиваются корректные последствия одновременного доступа к одним и тем же данным из нескольких потоков выполнения, поскольку в этом случае доступ со стороны других потоков блокируется, когда один из потоков обновляет значение в том, что иначе называется **критическим разделом кода**.

Когда приходится иметь дело с потоками выполнения, главной задачей становится соблюдение принципа **потокобезопасности кода**. Одни библиотеки

программ специально написаны с учетом параллельного выполнения потоков, а их критические разделы должным образом защищены мьютексами, тогда как другие библиотеки не являются потокобезопасными. Например, в библиотеке Cocoa Touch каркас Foundation в целом считается потокобезопасным. В то же время каркас UIKit, содержащий специальные классы для построения приложений с графическим пользовательским интерфейсом, в том числе классы UIApplication, UIView и все их подклассы, по большей части не считается потокобезопасным. (Впрочем, некоторые функции UIKit, например рисование, считаются потокобезопасными.) Это означает, что в приложении для системы iOS все вызовы методов, оперирующих любыми объектами UIKit, должны выполняться из одного и того же потока, обычно называемого **основным**. Совсем другое дело, если доступ к объектам UIKit осуществляется из другого потока выполнения. В таком случае вполне возможны необъяснимые, на первый взгляд, программные ошибки, а еще хуже, когда недостатки приложения не устраняются на стадии разработки и проявляются лишь после его передачи в эксплуатацию.

ПОДСКАЗКА. На тему потокобезопасности написано немало специальной литературы, поэтому стоит потратить время на ее как можно более углубленное изучение. Начать лучше всего с документации, предоставляемой компанией Apple. Уделите немногого времени чтению веб-страницы по указанному ниже адресу. Ее содержимое, определенно, принесет вам пользу. <http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Multithreading/ThreadSafetySummary/ThreadSafetySummary.html>

Единицы работы

Недостаток описанной выше модели заключается в том, что рядовому программисту практически невозможно написать безошибочный многопотоковый код. Это не критика в адрес отрасли в целом или способностей среднестатистического программиста, а просто наблюдение. Задача организации сложных взаимодействий, которые приходится учитывать в коде, синхронизируя данные и действия во многих потоках выполнения, оказывается просто не по плечу большинству людей. Допустим, что 5% всех людей способны вообще программировать. Из этих 5% лишь небольшая часть действительно способна решать задачи, связанные с разработкой многопоточных приложений, предназначенных для эксплуатации в интенсивном режиме. Однако даже те, кому удавалось успешно справиться с этими задачами, часто не советуют следовать их примеру.

К счастью, существуют альтернативы. Реализовать многопотоковое выполнение задач, не особенно вдаваясь в тонкости, вполне возможно. Аналогично возможности выводить данные на экран, не загружая их непосредственно в видеопамять, и читать данные с диска, не обращаясь непосредственно к контроллерам дисков, существуют программные абстракции, позволяющие выполнять код во многих потоках выполнения и вообще не требующие от нас много взаимодействовать с потоками непосредственно.

Решения, которыми компания Apple рекомендует пользоваться, основываются на принципах разделения задач с длительным временем выполнения на единицы работы и постановки этих единиц в очередь на выполнение. Система управляет очередями сама, автоматически выполняя единицы работы во многих потоках. Это избавляет нас от необходимости запускать фоновые потоки выполнения, управлять ими непосредственно и вести учет используемых системных ресурсов, т.е. решать задачи, которые обычно связаны с реализацией параллельных приложений, поскольку система берет на себя решение всех этих задач.

Организация очередей на низком уровне средствами GCD

Упомянутый выше принцип постановки единиц работы в очередь на выполнение в фоновом режиме, когда система управляет потоками выполнения автоматически, оказывается довольно эффективным и значительно упрощает многие ситуации, возникающие в процессе разработки, когда требуется многопотоковое выполнение. Всю необходимую для этого инфраструктуру предоставляет каркас Grand Central Dispatch, впервые появившийся в OS X (ныне macOS), а теперь также ставший частью системы iOS.

Одной из основных концепций каркаса GCD является *очередь*. Система предоставляет ряд предварительно заданных очередей, в том числе очередь, для которой выполнение операций всегда гарантируется в основном потоке выполнения. Это идеально подходит для каркаса UIKit, который не является потокобезопасным. Кроме того, можно организовать собственные очереди в каком угодно количестве. Очереди GCD действуют строго по принципу “первым пришел — первым вышел” (FIFO). Единицы работы, добавляемые в очередь, всегда запускаются на выполнение в том порядке, в каком они поставлены в очередь. Таким образом, их выполнение не всегда может завершаться в том же самом порядке, поскольку работа в очереди GCD будет автоматически распределяться по как можно большему числу потоков выполнения.

Каркас GCD имеет доступ к пулу потоков выполнения, многократно используемых в ходе выполнения приложения. Каркас GCD будет всегда пытаться поддерживать пул потоков выполнения, наиболее подходящий для архитектуры конкретной машины, автоматически используя преимущества многоядерного процессора более производительной машины для выполнения имеющейся работы. До недавнего времени мобильные устройства, работавшие под управлением системы iOS, имели одноядерные процессоры, поэтому данный вопрос был не актуален. Однако за последнее время компания Apple начала выпускать устройства с многоядерными процессорами, что делает теперь каркас GCD особенно полезным.

Для инкапсуляции кода, вводимого в очередь, в каркасе GCD применяются замыкания. Замыкания относятся к главным языковым средствам Swift, их можно присваивать переменным, передавать методам или возвращать как результат вызова методов. Замыкания равнозначны блокам в Objective-C и аналогичны языковым средствам, иногда еще называемым не совсем понятным термином *лямбда-функции* в других языках программирования.

Подобно методу или функции, блок может принимать один или несколько параметров и указывать возвращаемое значение, хотя те замыкания, которые применяются в GCD, не принимают аргументы и не возвращают значения. Для того чтобы объявить переменную замыкания, достаточно присвоить ей некоторый код, заключенный в фигурные скобки (дополнительно, хотя и не обязательно, с аргументами), как показано ниже.

```
// Объявление переменной замыкания "loggerClosure" без параметров
// и возвращаемого значения
var loggerClosure = {
    println("I'm just glad they didn't call it a lambda")
}
```

Замыкание выполняется таким же образом, как и вызов функции:

```
// Выполнение замыкания для вывода данных на консоль
loggerClosure()
```

Усовершенствование приложения SlowWorker

Для того чтобы показать, как замыкания применяются в GCD, вернемся к методу `doWork()` из приложения `SlowWorker`. В настоящий момент он выглядит следующим образом:

```
@IBAction func doWork(_ sender: AnyObject) {
    let startTime = NSDate()
    self.resultsTextView.text = ""
    let fetchedData = self.fetchSomethingFromServer()
    let processedData = self.processData(fetchedData)
    let firstResult = self.calculateFirstResult(processedData)
    let secondResult = self.calculateSecondResult(processedData)
    let resultsSummary =
        "First: [\"firstResult\"]\nSecond: [\"secondResult\"]"
    self.resultsTextView.text = resultsSummary
    let endTime = NSDate()
    print("Completed in \(endTime.timeIntervalSinceDate(startTime)) seconds")
}
```

Этот метод можно полностью выполнить в фоновом режиме, заключив весь его код в замыкание и передав функции GCD под названием `DispatchQueue`. Эта функция принимает два параметра: очередь GCD и замыкание для постановки в очередь. Итак, внесите в метод `doWork()` изменения, приведенные в листинге 15.2.

Листинг 15.2. Модификации метода `doWork`, необходимые для использования каркаса GCD

```
@IBAction func doWork(sender: AnyObject) {
    let startTime = NSDate()
    resultsTextView.text = ""
    let queue = DispatchQueue.global(qos: .default)
```

```

queue.async {
    let fetchedData = self.fetchSomethingFromServer()
    let processedData = self.processData(fetchedData)
    let firstResult = self.calculateFirstResult(processedData)
    let secondResult = self.calculateSecondResult(processedData)
    let resultsSummary =
        "First: \(firstResult)\nSecond: \(secondResult)"
    self.resultsTextView.text = resultsSummary
    let endTime = NSDate()
    print("Completed in \(endTime.timeIntervalSinceStartTime as Date) seconds")
}
}

```

В первой измененной строке с помощью функции `DispatchQueue.global()` выбирается уже существующая глобальная очередь, которая всегда доступна. Эта функция получает один аргумент, который задает приоритет. Если же указать в этом аргументе другой приоритет, то фактически будет получена совсем другая глобальная очередь, приоритетность для которой система устанавливает иначе. Остановимся пока что на используемой по умолчанию приоритетной очереди. Пока мы будем использовать стандартную глобальную очередь.

Затем очередь передается функции `queue.async()` вместе с замыканием. Каркас GCD ставит замыкание в очередь, где его поочередное выполнение планируется в фоновом потоке, как если бы это делалось в основном потоке выполнения.

Обратите внимание на то, что мы определили переменную `startTime` непосредственно перед созданием замыкания, а затем использовали ее значение в конце замыкания. Интуитивно ясно, что это нецелесообразно, поскольку к моменту, когда завершается выполнение замыкания, возврат из метода `doWork()` уже произойдет, и поэтому экземпляр класса `NSDate`, на который ссылается переменная `startTime`, уже будет удален из памяти! Это очень важный момент для понимания замыканий: если во время выполнения замыкание обращается к какой-либо переменной “извне”, то при создании замыкания происходит специальная настройка, обеспечивающая замыканию доступ к этим переменным. Все это делается автоматически компилятором Swift и выполняющей системой и не требует от вас никаких специальных действий.

Не забывайте об основном потоке

Единственный недостаток рассматриваемого здесь кода заключается в том, что каркас UIKit не является потокобезопасным. Напомним, что передача любого объекта графического пользовательского интерфейса, в том числе `resultsTextView` — текстового представления результатов, из фонового потока запрещена. Если выполнить код рассматриваемого здесь приложения теперь, то через десять секунд при попытке обновить текстовое представление в замыкании возникнет исключение. К счастью, каркас GCD предоставляет способ преодолеть и это препятствие. В замыкании можно вызвать другую функцию

диспетчеризации, передав выполняемую работу обратно основному потоку. Внесите в метод `doWork()` еще одно изменение, приведенное в листинге 15.3.

Листинг 15.3. Модифицированный метод `doWork()`

```
@IBAction func doWork(sender: AnyObject) {
    let startTime = NSDate()
    resultsTextView.text = ""
    let queue = DispatchQueue.global(attributes:
        DispatchQueue.GlobalAttributes.qosDefault)
    queue.async {
        let fetchedData = self.fetchSomethingFromServer()
        let processedData = self.processData(fetchedData)
        let firstResult = self.calculateFirstResult(processedData)
        let secondResult = self.calculateSecondResult(processedData)
        let resultsSummary =
            "First: \"\nSecond: \""
        DispatchQueue.main.async {
            self.resultsTextView.text = resultsSummary
        }
        let endTime = NSDate()
        print("Completed in \(endTime.timeIntervalSinceDate(startTime as Date)) seconds")
    }
}
```

Организация обратной связи

Если скомпилировать и выполнить рассматриваемое здесь приложение в данный момент, то можно заметить, что оно теперь работает чуть более гладко, по крайней мере в некотором отношении. В частности, кнопка не “залипает” в выбранном положении после ее касания, что, вероятно, вызывает желание нажать ее еще и еще раз. Если же просмотреть журнал консольных сообщений в среде Xcode, то можно обнаружить результаты каждого нажатия кнопки, хотя в текстовом представлении будут показаны результаты только последнего нажатия. Мы должны усовершенствовать графический пользовательский интерфейс данного приложения таким образом, чтобы после нажатия кнопки пользователем выводимая на экран информация сразу же обновлялась, указывая на то, что действие совершается и кнопка недоступна до тех пор, пока работа выполняется. С этой целью введем представление класса `UIActivityIndicatorView` в интерфейс данного приложения. В этом классе предоставляется счетчик, применявшийся во многих приложениях и на веб-сайтах. Зададим для него выход в самом начале файла `ViewController.swift`.

```
@IBOutlet var spinner : UIActivityIndicatorView!
```

Далее откройте файл `Main.Storyboard`, найдите в библиотеке элемент `ActivityIndicator View` и перетащите его в основное представление, расположив рядом с кнопкой. Для того чтобы зафиксировать положение индикатора активности относительно кнопки, придется наложить ограничения Auto Layout на его макет.

С этой целью нажмите клавишу <Control>, перетащите указатель от кнопки к индикатору активности и выберите пункт *Horizontal Spacing* из всплывающего меню, чтобы зафиксировать пространство, разделяющее эти элементы по горизонтали, а затем нажмите клавишу <Control> и снова перетащите указатель в указанном направлении, но на этот раз выберите пункт *Center Vertically* из всплывающего меню, чтобы выровнять центры обоих элементов по вертикали.

Выбрав индикатор активности в виде счетчика, перейдите к инспектору атрибутов и установите флагок *Hides When Stopped*, чтобы счетчик появлялся только в том случае, когда мы даем ему команду на вращение, поскольку застывший счетчик в графическом пользовательском интерфейсе приложения никому не нужен. Далее нажмите клавишу <Control> и снова перетащите указатель от пиктограммы *View Controller* к счетчику, установив связь с его выходом. Сохраните внесенные изменения.

Откройте файл *ViewController.swift*. Теперь займемся методом *doWork()* и внесем в него несколько строк кода для управления внешним видом кнопки и счетчика, когда пользователь нажимает кнопку и когда работа завершена. Сначала мы должны установить значение *false* свойства кнопки *enabled*, чтобы воспрепятствовать регистрации любых нажатий и показать, что кнопка недоступна, выделив текст ее надписи серым цветом и сделав его немного прозрачным. Затем мы должны привести “вертушку” в движение, вызвав ее метод *setAnimating()*. В конце замыкания кнопка должна быть активизирована снова, а счетчик — остановлен, в результате чего он снова исчезает. Изменения, вносимые в метод *doWork()*, приведены в листинге 15.4.

Листинг 15.4. Добавление функций счетчика в метод *doWork*

```
@IBAction func doWork(sender: AnyObject) {
    let startTime = NSDate()
    resultsTextView.text = ""
    startButton.isEnabled = false
    spinner.startAnimating()
    let queue = DispatchQueue.global(qos: .default)
    queue.async {
        let fetchedData = self.fetchSomethingFromServer()
        let processedData = self.processData(fetchedData)
        let firstResult = self.calculateFirstResult(processedData)
        let secondResult = self.calculateSecondResult(processedData)
        let resultsSummary =
            "First: [\"(firstResult)\"]\nSecond: [\"(secondResult)\"]"
        DispatchQueue.main.async {
            self.resultsTextView.text = resultsSummary
            self.startButton.isEnabled = true
            self.spinner.stopAnimating()
        }
        let endTime = NSDate()
        print("Completed in \(endTime.timeIntervalSinceDate(startTime as Date)) seconds")
    }
}
```

Скомпилируйте и запустите приложение на выполнение, а затем нажмите кнопку. Как видите, в данном приложении кое-что изменилось к лучшему. Несмотря на то что работа выполняется несколько секунд, пользователь не остается в неведении относительно того, что происходит в приложении: кнопка явно выглядит недоступной, а оживляемый счетчик дает пользователю знать, что приложение не “зависло”, и можно ожидать, что оно через некоторое время вернется к正常ной работе в диалоговом режиме.

Параллельные замыкания

Возможно, внимательные читатели заметили, что, несмотря на показанные выше модификации, мы не изменили основной последовательный характер реализуемого алгоритма, если, конечно, этот простой ряд шагов можно назвать алгоритмом. Мы просто переносим часть метода `doWork()` в фоновый поток выполнения, а завершаем его в основном потоке выполнения, что и подтверждает вывод на консоль Xcode: работа, как и прежде, выполняется 10 секунд. Однако все дело в том, что методы `calculateFirstResult()` и `calculateSecondResult()` совсем не обязательно должны выполняться последовательно, а если распараллелить их выполнение, то можно добиться существенного ускорения всего процесса.

К счастью, в GCD этого можно добиться с помощью так называемой *диспетчерской группы*. Все замыкания, которые асинхронно диспетчерируются в контексте группы, высвобождаются функцией `dispatch_group_async()` для как можно более быстрого выполнения, включая их распределение по многим потокам для параллельного выполнения. В то же время с помощью функции `dispatch_group_notify()` можно указать дополнительное замыкание, которое будет выполняться по завершении всех замыканий в группе.

Учитывая все сказанное выше, внесите в данную версию метода `doWork()` изменения, приведенные в листинге 15.5.

Листинг 15.5. Окончательная версия метода `doWork()`

```
@IBAction func doWork(_ sender: AnyObject) {
    let startTime = Date()
    self.resultsTextView.text = ""
    startButton.isEnabled = false
    spinner.startAnimating()
    let queue = DispatchQueue.global(qos: .default)
    queue.async {
        let fetchedData = self.fetchSomethingFromServer()
        let processedData = self.processData(fetchedData)
        var firstResult: String!
        var secondResult: String!
        let group = DispatchGroup()

        queue.async(group: group) {
            firstResult = self.calculateFirstResult(processedData)
        }
    }
}
```

```

queue.async(group: group) {
    secondResult = self.calculateSecondResult(processedData)
}

group.notify(queue: queue) {
    let resultsSummary = "First: \"\((firstResult!))\nSecond:
                           \"\((secondResult!))\""
    DispatchQueue.main.async {
        self.resultsTextView.text = resultsSummary
        self.startButton.isEnabled = true
        self.spinner.stopAnimating()
    }
    let endTime = Date()
    print("Completed in \(endTime.timeIntervalSince(startTime))
          iseconds")
}
}
}

```

В данном коде возникает следующее затруднение: каждый метод типа, имя которого содержит слово `calculate`, возвращает значение, которое требуется захватить. Поэтому нужно сделать так, чтобы значения присваивались переменным `firstResult` и `secondResult` в замыканиях. Для этого они объявляются с помощью оператора `var`, а не `let`. Однако в языке Swift требуется, чтобы переменная, на которую делается ссылка из замыкания, была инициализирована, и поэтому следующие объявления не действуют:

```

var firstResult: String
var secondResult: String

```

Это препятствие можно, конечно, обойти, инициализировав обе переменные произвольным значением, но проще объявить их как неявно разворачиваемые необязательные типы, добавив в конце их объявления восклицательный знак, как выделено ниже полужирным шрифтом.

```

var firstResult: String!
var secondResult: String!

```

Теперь компилятор языка Swift не потребует инициализации этих переменных, но у них все равно должны быть конкретные значения, когда их придется в конечном итоге считывать. В данном случае значениячитываются из переменных в завершающем замыкании асинхронной группы, и к этому моменту им должны быть присвоены конкретные значения. Итак, сделав упомянутые выше изменения, еще раз скомпилируйте и запустите рассматриваемое здесь приложение на выполнение, чтобы убедиться в том, что ваши усилия оказались ненапрасными. То, на что раньше требовалось 10 секунд, теперь отнимает всего семь секунд, благодаря тому, что оба расчета выполняются параллельно.

Очевидно, что в данном искусственном примере нам удается добиться максимального эффекта из-за того, что в обоих этих расчетах на самом деле ничего не делается, кроме перевода в состояние ожидания того потока выполнения,

в котором они производятся. В реальном приложении ускорение будет зависеть от характера выполняемой работы и доступных ресурсов. Данный способ распараллеливания может принести пользу при выполнении расчетов, требующих интенсивного использования вычислительных мощностей центрального процессора, только в том случае, если имеется многоядерный центральный процессор, а по мере увеличения количества ядер на устройствах iOS производительность будет возрастать почти даром. Впрочем, заметного ускорения работы можно добиться и на одноядерном центральном процессоре, например, при одновременной выборке данных из нескольких сетевых соединений.

Как видите, каркас GCD не является панацеей. Его применение совсем не гарантирует автоматическое повышение быстродействия каждого приложения. Однако, если аккуратно применять рассмотренные выше способы параллельной работы в тех местах, где быстродействие имеет существенное значение, или же там, где приложение реагирует на действия пользователя с некоторым запаздыванием, то в конечном итоге можно добиться улучшения взаимодействия с пользователем в тех случаях, когда не удается повысить реальную производительность.

Фоновая работа

Еще одним важным технологическим средством многопотокового выполнения операций является фоновая работа, которая позволяет выполнять приложения в фоновом режиме, а в некоторых случаях — даже после того, как пользователь нажмет главную кнопку.

Эти функциональные возможности не следует путать с подлинной много задачностью, характерной для всех современных настольных операционных систем, в которых все программы, запускаемые на выполнение, остаются в оперативной памяти системы вплоть до явного их завершения. У мобильных устройств, работающих под управлением системы iOS, все еще недостаточно оперативной памяти, чтобы извлечь реальную пользу из многопотоковой работы. Напротив, фоновый режим предназначен для того, чтобы дать возможность тем приложениям, которым требуются особые функциональные возможности системы, продолжать работу в ограниченном виде. Так, если имеется приложение, воспроизводящее потоковый звук из радиостанции в Интернете, система iOS может предоставить такому приложению возможность продолжить работу даже в том случае, если пользователь перейдет к другому приложению. Более того, в то время как приложение воспроизводит звук, система iOS может даже предоставить элементы управления паузой, воспроизведением и громкостью звука на прозрачной панели управления, которая обычно появляется в нижней части экрана, если провести пальцем снизу вверх по экрану.

Допустим, что вы создаете приложение, выполняющее одно из следующих действий: воспроизведение звука, когда пользователь выполняет другое приложение, непрерывное запрашивание обновления местоположения, реагирование

на специальные извещающие запросы сервера на загрузку новых данных или реализация системы связи VoIP, чтобы предоставить пользователям возможность отправлять и принимать телефонные звонки через Интернет. Все эти требования к системе можно объявить в файле Info.plist приложения, и тогда система будет обращаться с таким приложением особым образом. Хотя такая задача может оказаться интересной сама по себе, вряд ли ее придется решать большинству читателей этой книги, поэтому не будем углубляться в ее рассмотрение.

Помимо выполнения приложений в фоновом режиме, в системе iOS имеется возможность переводить приложение в состояние приостановки после того, как пользователь нажмет главную кнопку. Это состояние концептуально подобно переводу компьютера Macintosh в режим ожидания. Вся рабочая область памяти приложения сохраняется в оперативной памяти, но оно не выполняется в состоянии приостановки. В итоге возврат к такому приложению происходит практически мгновенно. Такой режим работы распространяется не только на специальные приложения, но, по существу, является стандартным для любого приложения, скомпилированного в среде Xcode, хотя он и может быть отменен другой установкой в файле Info.plist. Для того чтобы проверить данный режим в действии, откройте на мобильном устройстве стандартное приложение Mail и перейдите к сообщению, нажмите главную кнопку, откройте еще одно стандартное приложение Notes и выберите заметку. Затем дважды нажмите главную кнопку, чтобы сразу же вернуться к приложению Mail. При этом вы не обнаружите сколько-нибудь заметной задержки в переходе к первому приложению. Возврат к приложению Mail произойдет настолько естественно, как будто оно и не переставало работать все это время.

Подобного рода автоматическая приостановка и возобновление работы — это практически все, что нужно для большинства приложений. Однако в некоторых случаях приложению, возможно, потребуется знать, когда именно его выполнение будет приостановлено и возобновлено. Система предоставляет средства, позволяющие уведомлять приложение о предстоящем переходе в состояние приостановки. Данной цели служат методы делегата и уведомления из класса UIApplication, применение которых будет рассмотрено далее в этой главе.

Когда система собирается перевести приложение в состояние приостановки, независимо от того, предназначено ли оно специально для работы в фоновом режиме, оно может запросить еще немного времени для выполнения в фоновом режиме. Это делается для того, чтобы у приложения было достаточно времени на закрытие любых открытых файлов, сетевых ресурсов и т.д. Ниже будет приведен характерный тому пример.

Жизненный цикл приложения

Прежде чем рассматривать порядок действий, изменяющих состояние выполнения приложения, остановимся вкратце на особенностях различных приложений.

Состояние невыполнения. В этом состоянии находятся все приложения на вновь перезагруженном устройстве. Приложение, запущенное в любой момент после включения мобильного устройства, вернется в это состояние только в особых случаях, в частности:

если его файл Info.plist содержит ключ UIApplicationExitsOnSuspend со значением YES;

если перед этим оно находилось в состоянии приостановки и системе требуется частично очистить оперативную память;

если во время выполнения произойдет аварийный отказ.

- ❖ **Активное состояние.** Обычное состояние приложения, когда оно отображается на экране. В этом состоянии оно может принимать вводимую пользователем информацию и обновлять отображаемую информацию.
- ❖ **Фоновое состояние.** В этом состоянии приложению предоставляется некоторое время для выполнения определенного кода, но без непосредственного доступа к экрану или получения информации от пользователя. Все приложения переходят в это состояние, как только пользователь нажимает главную кнопку, а большинство из них быстро переходит в состояние приостановки. Те же приложения, которым требуется выполнение в фоновом состоянии, остаются в нем до тех пор, пока они вновь не будут переведены в активное состояние.
- ❖ **Состояние приостановки.** В этом состоянии выполнение приложения останавливается. Как правило, это происходит с приложениями после их краткого пребывания в фоновом состоянии. Вся оперативная память, использовавшаяся приложением в активном состоянии, сохраняется в прежнем виде. Если же пользователь переведет приложение обратно в активное состояние, оно продолжит свою работу с того места, где она была остановлена. В то же время, если системе потребуется дополнительная оперативная память для любого приложения, находящегося в настоящий момент в активном состоянии, любые приостановленные приложения будут завершены (и переведены в состояние невыполнения), а используемая ими оперативная память будет освобождена для применения в других целях.
- ❖ **Неактивное состояние.** В это состояние приложение переводится только для временного пребывания между двумя другими состояниями. Единственная возможность надолго перевести приложение в неактивное состояние появляется в том случае, если система приглашает пользователя к определенным действиям, например ответить на входящие звонки или сообщения SMS, либо если пользователь заблокировал экран. Это своего рода переходное состояние.

Уведомления о смене состояния

Для организации смены упомянутых выше состояний в классе UIApplication определен ряд методов, которые может реализовать его делегат. Помимо методов делегата, в классе UIApplication определен согласованный ряд имен уведомлений (табл. 15.1). Это дает возможность другим объектам, кроме делегата приложения, зарегистрировать уведомления при смене состояния приложения.

Таблица 15.1. Методы делегата, отслеживающие состояние выполнения приложения, и соответствующие имена уведомлений

Метод делегата	Имя уведомления
application(_:didFinishLaunchingWithOptions:)	UIApplicationDidFinishLaunching
applicationWillResignActive()	UIApplicationWillResignActive
applicationDidBecomeActive()	UIApplicationDidBecomeActive
applicationDidEnterBackground()	UIApplicationDidEnterBackground
applicationWillEnterForeground()	UIApplicationWillEnterForeground
applicationWillTerminate()	UIApplicationWillTerminate

Следует, однако, иметь в виду, что перечисленные выше методы делегата и имена уведомлений непосредственно связаны с состояниями выполнения: активным, неактивным и фоновым. Каждый метод делегата вызывается (и каждое уведомление посыпается) только в одном из этих состояний. К числу самых важных переходов между состояниями относится переход из активного в другие состояния и обратно, а остальные переходы, например из фонового в состояние приостановки, происходят безо всяких уведомлений. Рассмотрим вкратце все эти методы и их основное назначение.

Первый метод, application(_: didFinishLaunchingWithOptions:), уже не раз упоминался на страницах этой книги. Его основное назначение — выполнять код на уровне приложения после запуска последнего. Имеется аналогичный метод application(_:willFinishLaunchingWithOptions:), который вызывается первым и предназначен для приложений, пользующихся средством сохранения состояния, построенным на основе контроллера представления. Этот метод не перечислен выше, поскольку он не связан с изменением состояния.

Два следующих метода, applicationWillResignActive() и applicationDidBecomeActive(), применяются в целом ряде случаев. Так, если пользователь нажмет главную кнопку, вызывается метод applicationWillResignActive(). Если пользователь в дальнейшем возвратит приложение обратно в фоновое состояние, то будет вызван метод applicationDidBecomeActive(). Аналогичная последовательность событий происходит и в том случае, если пользователь получает телефонный звонок. Кроме того, метод applicationDidBecomeActive() вызывается и при запуске приложения на выполнение! Как правило, эта пара

методов заключает в себе все действия по переводу приложения из активного состояния в неактивное и служит удобным местом для активизации или отмены любой анимации, звука или других элементов в приложении, имеющих отношение к представлению приложения пользователю. В связи с тем, что метод `applicationDidBecomeActive()` применяется в целом ряде случаев, может возникнуть потребность ввести код инициализации именно здесь, а не в методе `application(_:didFinishLaunchingWithOptions:)`. Но, делая это в методе `applicationWillResignActive()`, не следует думать, что приложение находится на грани перехода в фоновое состояние, поскольку подобная смена состояний может оказаться временной, а в конечном итоге — привести в активное состояние.

Далее следуют еще два метода, `applicationDidEnterBackground()` и `applicationWillEnterForeground()`, имеющих несколько иную область применения: они предназначены для работы с приложениями, которые определенно переводятся в фоновое состояние. При вызове метода `applicationDidEnterBackground()` приложение должно освободить все ресурсы, которые могут быть восстановлены в дальнейшем, сохранить все данные пользователя, разорвать сетевые соединения и т.д. Именно здесь можно также сделать запрос дополнительного времени на выполнение в фоновом режиме, если в этом возникнет потребность, как будет показано ниже. Однако если потратить слишком много времени на выполнение операций в методе `applicationDidEnterBackground()` (более пяти секунд), система решит, что приложение ведет себя неверно, и завершит его принудительно. Для восстановления того, что было нарушено в методе `applicationDidEnterBackground()`, например перезагрузка данных пользователя, повторное установление соединения и так далее, нужно реализовать метод `applicationWillEnterForeground()`. Следует также иметь в виду, что при вызове метода `applicationDidEnterBackground()` можно безо всякого риска предположить, что недавно вызывался и метод `applicationWillResignActive()`, подобно тому, как при вызове метода `applicationWillEnterForeground()` вполне допустимо предположить, что вскоре будет вызван метод `applicationDidBecomeActive()`.

Последний из рассматриваемых здесь методов, `applicationWillTerminate()`, применяется крайне редко, если вообще применяется. На самом деле он вызывается лишь в том случае, если приложение уже находится в фоновом состоянии и система решает отказаться по той или иной причине от приостановки его выполнения и просто завершает приложение.

Вот, собственно, и все теоретические основы, которые следует знать о состояниях и переходах приложения между ними. Теперь попробуем применить эти теоретические знания на практике, написав приложение, которое просто записывает сообщение в журнал консольных сообщений среды Xcode всякий раз, когда вызывается один из рассмотренных выше методов. Затем попробуем

манипулировать выполняющимся приложением самыми разными способами, подобно тому, как это мог бы сделать пользователь, чтобы проверить, к переходам в какие именно состояния это приведет. Для того чтобы извлечь наибольшую пользу из рассматриваемого здесь примера приложения, вам потребуется устройство iOS. Если же у вас нет такого устройства, можете воспользоваться симулятором, пропустив те части данного примера, в которых это устройство необходимо.

Создание приложения State Lab

Итак, создайте в среде Xcode новый проект на основе шаблона Single View Application и присвойте ему имя State Lab. В данном приложении не предполагается (по крайней мере, первоначально) выводить на экран ничего, кроме исходного серого фона при его запуске на выполнение. В дальнейшем мы заставим это приложение делать нечто более интересное, а до тех пор весь вывод будет направляться в журнал консольных сообщений среды Xcode. Исходный файл AppDelegate.swift уже содержит все интересующие нас методы, и поэтому нам остается лишь добавить в него немного кода для регистрации сообщений, приведенного в листинге 15.6. Обратите также внимание на то, что из этих методов для краткости удалены комментарии.

Листинг 15.6. Методы регистрации сообщений в файле AppDelegate.swift

```
func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
        [NSObject: AnyObject]?) -> Bool {
    print(#function)
    return true
}

func applicationWillResignActive(_ application: UIApplication) {
    print(#function)
}

func applicationDidEnterBackground(_ application: UIApplication) {
    print(#function)
}

func applicationWillEnterForeground(_ application: UIApplication) {
    print(#function)
}

func applicationDidBecomeActive(_ application: UIApplication) {
    print(#function)
}

func applicationWillTerminate(_ application: UIApplication) {
    print(#function)
}
```

В приведенном выше коде прежде всего обращает на себя внимание значение, передаваемое функции `print()` в каждом из вызываемых методов. Им является литеральное выражение `#function`, вычисляющее имя метода, в котором оно появляется. В данном случае оно служит для получения имени текущего метода без его повторного набора или копирования и вставки в каждый метод жизненного цикла.

Исследование состояний выполнения

Скомпилируйте и запустите приложение на выполнение и посмотрите на консоль (`View⇒Debug Area⇒Activate Console`). На экране должны появиться строки сообщений, аналогичные приведенным ниже.

```
application(_:didFinishLaunchingWithOptions:)
applicationDidBecomeActive
```

Кроме того, на экране появится много других сообщений. Для того чтобы отфильтровать их, вы можете использовать поле поиска, расположеннное ниже окна, для вывода результатов на экран.

Как видите, данное приложение было успешно запущено на выполнение и затем переведено в активное состояние. Теперь вернитесь в симулятор и нажмите главную кнопку (для этого в симуляторе следует выбрать команду меню `Hardware⇒Home` или нажать комбинацию клавиш `<Shift+⌘+H>`). На консоли должны появиться приведенные ниже строки сообщений.

```
applicationWillResignActive
applicationDidEnterBackground
```

Как следует из приведенных выше строк сообщений, данное приложение фактически переходит в одно из следующих двух состояний: сначала оно становится неактивным, а затем переходит в фоновое состояние. Однако из этих строк совсем не следует, что приложение переходит также в третье состояние — приостановки. Напомним, что уведомления о подобном переходе нельзя получить, поскольку они нам неподвластны. Следует, однако, иметь в виду, что данное приложение оказывается в каком-то смысле по-прежнему действующим и связанным со средой Xcode, несмотря на то что ему не предоставляется время центрального процессора. Для того чтобы убедиться в этом, попробуйте нажать пиктограмму данного приложения, чтобы перезапустить его. В итоге на консоль будут выведены следующие строки сообщений.

```
applicationWillEnterForeground
applicationDidBecomeActive
```

Итак, приложение возвратилось в свое рабочее состояние. Находясь прежде в состоянии приостановки, оно было переведено в неактивное, а затем опять в активное состояние. Что же происходит, когда приложение действительно завершается? Нажмите главную кнопку еще раз, чтобы на консоли появились приведенные ниже строки сообщений.

```
applicationWillResignActive
applicationDidEnterBackground
```

Далее дважды нажмите главную кнопку (а по существу, комбинацию клавиш <Shift+⌘+H+H>, т.е. дважды — клавишу <H>). В итоге должен появиться экран приложений с боковой прокруткой. Нажмите моментальный снимок экрана приложения State Lab и проведите пальцем вверх по экрану, чтобы вывести этот моментальный снимок за пределы экрана, удалив тем самым приложение. Вы должны увидеть на экране примерно такие строки.

```
2016-07-21 10:15:40.201746 temp[2825:864732] [Common] <FBSUIApplicationWork
spaceClient
[0x6080000f8700]: Received exit event
applicationDidEnterBackground
applicationWillTerminate
```

ПОДСКАЗКА. Для того чтобы сохранить состояние приложения, не следует полагаться на метод `applicationWillTerminate()`. Вместо этого лучше вызвать метод `applicationDidEnterBackground()`.

Теперь исследуем еще одно интересное взаимодействие, возникающее в том случае, когда система выводит предупреждающее диалоговое окно, временно перенимая поток ввода у приложения и переводя его в неактивное состояние. Приложение нетрудно перевести в это состояние с помощью приложения Messages, но только если оно выполняется на настоящем устройстве, например iPhone, а не в симуляторе. Приложение Messages, как и многие другие приложения, может получать сообщения извне и отображать их несколькими способами.

Для того чтобы увидеть, как выполняется настройка этого приложения, запустите на своем устройстве приложение Settings. Затем выберите вариант **Notifications** из списка и далее — приложение **Messages** из списка приложений. Относительно новый эффективный механизм демонстрации сообщений в системе iOS 5 называется **Banners**. Он выводит на экран небольшой баннер, который перекрывает верхнюю часть экрана, не требуя прерывания текущего приложения. Мы же хотим продемонстрировать старый механизм **Alert**, выведивший модальную панель перед текущим приложением, требуя от пользователя выполнить какое-нибудь действие. Для этого достаточно выбрать вариант **Alert** под заголовком **ALERT STYLE WHEN UNLOCKED** (Стиль предупреждений в отсутствие блокировки), чтобы приложение **Messages** сразу же перешло в старый режим, знакомый пользователям системы iOS 4.

Вернитесь к своему компьютеру. Находясь в среде Xcode, откройте список, расположенный слева вверху, и перейдите в режим симулятора. Затем нажмите кнопку **Run**, чтобы построить и выполнить приложение на своем устройстве. Теперь вы должны принять на свое мобильное устройство сообщение из внешнего мира. Если вы работаете на мобильном устройстве iPhone, то можете

отправить SMS-сообщение с другого устройства. Если же вы работаете на мобильных устройствах iPod и iPad, ваш выбор ограничивается службой iMessage компании Apple, которая работает на всех устройствах с операционной системой iOS 5 (включая iPhone). Сделайте подходящие настройки и отправьте сами или попросите кого-нибудь прислать сообщение (SMS или iMessage) на ваше устройство iPhone. Когда на экране iPhone появится системное предупреждение с отображаемым SMS-сообщением, на консоль Xcode будет выведена следующая строка сообщения:

```
applicationWillResignActive
```

Обратите внимание на то, что данное приложение не было переведено в фоновое состояние, но находится в неактивном состоянии и по-прежнему видно позади системного предупреждения. Если бы это было игровое приложение или если бы оно воспроизводило видеоизображение, звук или анимацию, то именно в этот момент у нас могла бы возникнуть потребность приостановить воспроизведение.

Нажмите кнопку Close в предупреждающем диалоговом окне, и на консоль будет выведена следующая строка сообщения:

```
applicationDidBecomeActive
```

Теперь посмотрим, что произойдет, если поступить иначе: ответить на полученное сообщение. Отправьте сами или попросите кого-нибудь прислать сообщение на ваше мобильное устройство, чтобы на консоль была выведена следующая строка сообщения:

```
applicationWillResignActive
```

На этот раз нажмите кнопку Reply, чтобы запустить приложение Messages и увидеть на консоли приведенную ниже вспышку активности.

```
applicationDidBecomeActive
applicationWillResignActive
applicationDidEnterBackground
```

Любопытно, что рассматриваемое здесь сообщение быстро становится активным, затем опять неактивным и наконец переходит в фоновое состояние и далее — незаметно в состояние приостановки.

Практическое применение смены состояний выполнения

Какую же пользу можно извлечь из всего изложенного выше? Основываясь на рассмотренном выше примере, можно выработать ясную стратегию, чтобы придерживаться ее в дальнейшем, когда придется иметь дело с изменениями состояния.

Переход из активного состояния в неактивное

Пользуйтесь методом `applicationWillResignActive()` и уведомлением типа `UIApplicationWillResignActive` для приостановки отображения своего приложения. Если ваше приложение является игровым, значит, в нем уже должна быть предусмотрена возможность приостанавливать каким-то образом ход игры. Что касается других разновидностей приложений, то они не должны предъявлять жестких по времени требований к получению вводимых пользователем данных, поскольку ваше приложение не сможет получать эти данные в течение некоторого времени.

Переход из неактивного состояния в фоновое

Пользуйтесь методом `applicationDidEnterBackground()` и уведомлением `UIApplicationDidEnterBackground` для освобождения любых ресурсов, которые необязательно сохранять, когда приложение находится в фоновом состоянии (например, кэшируемые изображения или другие легко перезагружаемые данные), либо тех ресурсов, которые все равно не уцелеют в фоновом режиме работы, в том числе сетевые соединения. Отказ от использования лишней оперативной памяти, скорее всего, приведет к тому, что получаемый в итоге моментальный снимок состояния приостановки вашего приложения уменьшится, благодаря чему снизится риск полного удаления приложения из оперативной памяти. Непременно воспользуйтесь такой возможностью и для сохранения любых данных приложения, чтобы после его перезапуска пользователь смог продолжить работу с того места, где она была остановлена. Если ваше приложение вернется в активное состояние, то, как правило, это не будет иметь особого значения. Однако если приложение удалено из оперативной памяти и должно быть перезапущено, то его пользователи по достоинству оценят возможность начать работу с того же самого места.

Переход из фонового состояния в неактивное

Пользуйтесь методом `applicationWillEnterForeground()` и уведомлением типа `UIApplicationWillEnterForeground` для отмены любой работы, проделанной при переходе из неактивного состояния в фоновое. В данный момент можно, например, восстановить постоянные сетевые соединения.

Переход из неактивного состояния в активное

Пользуйтесь методом `applicationDidBecomeActive()` и уведомлением типа `UIApplicationDidBecomeActive` для отмены любой работы, проделанной при переходе из активного состояния в неактивное. Однако если ваше приложение является игровым, то это, вероятно, не должно означать приостановку игры. Вы должны предоставить пользователям возможность делать это по своему усмотрению. Следует также иметь в виду, что данный метод и соответствующее уведомление используются при запуске приложения заново, и поэтому все, что ни делается здесь, должно быть работоспособным и в данном контексте.

У перехода из неактивного состояния в фоновое имеется еще одна особенность. Он требует не только самого подробного описания, но и больше всего кода и времени из-за объема операций учета используемых системных ресурсов, которые, возможно, придется вести в вашем приложении. Когда совершается подобный переход, система не предоставляет неограниченное время на сохранение внесенных изменений, но дает на это не более пяти секунд. Если вашему приложению потребуется больше времени для возврата из метода делегата (и обработки любых зарегистрированных вами уведомлений), то ваше приложение будет в конечном итоге удалено из оперативной памяти и переведено в состояние невыполнения. Если же это вас не устраивает, можете получить отсрочку. На время обращения к данному методу делегата или обработки соответствующего уведомления можете дать системе некоторую дополнительную работу, чтобы она выполнила ее для вас в очереди фоновых заданий, и тем самым выиграть немного времени. Этот прием будет продемонстрирован в следующем разделе.

Обработка неактивного состояния

Самая простая смена состояний, которая может произойти в приложении, состоит в переходе из активного в неактивное состояние, а затем обратно в активное. Напомним, что именно это и происходит, когда в ходе выполнения приложения мобильное устройство iPhone получает SMS-сообщение, которое выводится на экран для уведомления пользователя. В этом разделе мы сначала доработаем рассматривавшееся ранее приложение State Lab, чтобы оно наглядно показывало происходящее в том случае, если проигнорировать данную смену состояний, а затем покажем, как выйти из создавшегося положения.

В данном случае нам нужно добавить объект типа UILabel в отображаемое представление и заставить его двигаться средствами Core Animation, которые позволяют выполнять изящную анимацию объектов в iOS.

С этой целью прежде всего введите в исходный код из файла ViewController.swift объект типа UILabel в качестве переменной экземпляра и соответствующее свойство, выделенное ниже полужирным шрифтом.

```
class ViewController: UIViewController {
    private var label:UILabel!
```

Затем установите метку, появляющуюся при загрузке представления. С этой целью введите в метод viewDidLoad() строки кода из листинга 15.7.

Листинг 15.7. Модифицированный метод viewDidLoad

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.
    let bounds = view.bounds
```

592 ГЛАВА 15 ■ МНОГОПОТОКОВОЕ ПРОГРАММИРОВАНИЕ...

```
let labelFrame = CGRect(origin: CGPoint(x: bounds.origin.x,
                                         y: bounds.midY - 50),
                        size: CGSize(width: bounds.size.width, height: 100))
label = UILabel(frame: labelFrame)
label.font = UIFont(name: "Helvetica", size: 70)
label.text = "Bazinga!"
label.textAlignment = NSTextAlignment.center
label.backgroundColor = UIColor.clear()
view.addSubview(label)
}
```

Далее организуем анимацию метки. С этой целью мы должны определить два метода: один — для переворачивания метки, другой — для ее возврата в исходное положение.

```
func rotateLabelDown() {
    UIView.animate(withDuration: 0.5, animations: {
        self.label.transform = CGAffineTransform(rotationAngle: CGFloat(M_PI))
    },
    completion: { (Bool) -> Void in
        self.rotateLabelUp()
    }
)
}

func rotateLabelUp() {
    UIView.animate(withDuration: 0.5, animations: {
        self.label.transform = CGAffineTransform(rotationAngle: 0)
    },
    completion: { (Bool) -> Void in
        self.rotateLabelDown()
    }
)
}
```

Приведенный выше код требует некоторых пояснений. В классе `UIView` определяется метод `animate(withDuration:completion)`, который организует всю анимацию метки. Любые атрибуты, используемые в замыкании анимации, не оказывают непосредственного влияния на ее получателя. Напротив, механизм `Core Animation` организует плавный переход оживляемого атрибута от его текущего значения к новому указываемому значению. Это так называемая **неявная анимация**, являющаяся главной особенностью `Core Animation`. В завершающем замыкании можно указать, что же произойдет по завершении анимации. Обратите особое внимание на следующий синтаксис замыканий в приведенных выше методах:

```
completion: { (Bool) -> Void in
    if self.animate {
        self.rotateLabelDown()
    }
}
```

Код, выделенный полужирным шрифтом, определяет сигнатуру замыкания и указывает на то, что замыкание вызывается с единственным логическим

аргументом и ничего не возвращает. Этот аргумент принимает логическое значение `true`, если анимация завершается нормально, а если она отменена, логическое значение `false`. В рассматриваемом здесь примере этот аргумент не находит никакого применения.

Таким образом, в каждом из рассматриваемых здесь методов свойство `transform` метки устанавливается на конкретный вид ее вращения в радианах. Кроме того, в каждом из них устанавливается блок завершения только для вызова другого метода, чтобы анимация текста надписи метки продолжалась в бесконечном цикле.

В заключение необходимо задать способ запуска анимации. Ограничимся пока что вводом с этой целью следующей строки кода в конце метода `viewDidLoad()`:

```
rotateLabelDown();
```

Постройте и запустите данное приложение на выполнение. На экране должна появиться метка **Bazinga!**, поворачивающаяся то вверх, то вниз (рис. 15.4).



Рис. 15.4. Анимация поворачивания метки в приложении State Lab

Для проверки перехода данного приложения из активного состояния в неактивное его, опять же, нужно запустить на выполнение непосредственно на мобильном устройстве iPhone и затем отправить на него SMS-сообщение откуда-нибудь еще. Постройте и запустите данное приложение на выполнение непосредственно на мобильном устройстве iPhone, чтобы посмотреть анимацию поворачивающейся метки. Затем отправьте SMS-сообщение на мобильное устройство iPhone, и, когда на экране появится системное предупреждение для отображения этого сообщения, вы обнаружите, что анимация продолжается как ни в чем не бывало. Возможно, такая ситуация и покажется вам немного комичной, но пользователя данного приложения она, скорее всего, выведет из себя. Для того чтобы выйти из этой затруднительной ситуации, воспользуемся далее уведомлениями о смене состояний, чтобы остановить анимацию, когда происходит нечто подобное.

Классу контроллера потребуется некоторое внутреннее состояние для отслеживания потребности выполнять анимацию в любой момент времени. С этой целью введите в класс ViewController свойство, выделенное ниже полужирным шрифтом.

```
class ViewController: UIViewController {
    private var label:UILabel!
    private var animate = false
```

Как пояснялось ранее, уведомления об изменениях в состоянии приложения делегату не направляются, но поскольку данный класс не выполняет функцию делегата приложения, мы не можем просто реализовать методы делегата, надеясь, что они будут действовать так, как мы и предполагаем. Вместо этого мы должны подписаться на получение уведомлений от приложения при смене состояния его выполнения. С этой целью введите следующий фрагмент кода в конце метода viewDidLoad() из файла ViewController.swift:

```
let center = NotificationCenter.default
center.addObserver(self,
    selector:#selector(ViewController.applicationWillResignActive),
    name: Notification.Name.UIApplicationWillResignActive, object: nil)
center.addObserver(self,
    selector: #selector(ViewController.applicationDidBecomeActive),
    name: NSNotification.Name.UIApplicationDidBecomeActive, object: nil)
```

В этом фрагменте кода задаются два уведомления, отсылаемые в соответствующее время на каждый вызов метода в данном классе. Введите следующие методы в классе ViewController:

```
func applicationWillResignActive() {
    print("VC: \(#function)")
    animate = false
}
```

```
func applicationDidBecomeActive() {
    print("VC: \(#function)")
    animate = true
    rotateLabelDown()
}
```

Как видите, в данный класс мы включили такую же регистрацию методов, как и прежде, только для того, чтобы вам было понятнее, где именно это происходит на консоли Xcode. Мы добавили префикс VC:, чтобы отличать этот вызов от вызовов делегата (аббревиатура “VC” означает “view controller” — “контроллер представления”). Первый из этих методов просто сбрасывает признак анимации в переменной `animate`, а второй устанавливает этот признак, чтобы возобновить анимацию. Для того чтобы первый из этих методов начал действовать, придется ввести еще немного кода, в переменной `animate` которого проверяется состояние признака анимации, продолжающейся только в том случае, если данный признак установлен.

```
func rotateLabelUp() {
    UIView.animate(withDuration: 0.5, animations: {
        self.label.transform = CGAffineTransform(rotationAngle: 0)
    },
    completion: {(Bool) -> Void in
        if self.animate {
            self.rotateLabelDown()
        }
    }
)
}
```

Он был введен только в замыкание, завершающее метод `rotateLabelUp()`, чтобы анимация останавливалась лишь после того, как текст надписи метки вернется в исходное положение.

В заключение отметим, что анимация не начинается в том момент, когда активизируется приложение, а это происходит сразу же после его запуска, и поэтому вызывать метод `rotateLabelDown()` из метода `viewDidLoad()` больше не нужно. Следовательно, его можно удалить, как показано ниже.

```
override func viewDidLoad() {
    rotateLabelDown();
    let center = NotificationCenter.default
```

Еще раз постройте и запустите данное приложение на выполнение, чтобы посмотреть, что же произойдет. Снова отправьте SMS-сообщение на свое мобильное устройство iPhone. На этот раз при появлении системного сообщения вы должны увидеть, что анимация на заднем плане останавливается, как только текст метки возвращается в нормальное положение. Нажмите кнопку *Close*, и анимация метки возобновится.

Теперь вы знаете, что следует делать в простом случае перехода из активного состояния в неактивное и обратно. Однако более сложная и, вероятно, важная задача возникает при переходе в фоновый режим работы приложения и обратно в приоритетный.

Обработка фонового состояния

Как упоминалось ранее, переход в фоновое состояние очень важен для обеспечения наилучшего взаимодействия с пользователем. Ведь в этом случае требуется отказаться от любых ресурсов, которые можно легко запросить повторно (или потерять, если приложение перестанет работать), а также сохранить информацию о текущем состоянии приложения, причем сделать это, не занимая текущий поток выполнения более пяти секунд.

Для того чтобы продемонстрировать некоторые виды подобного поведения, расширим возможности рассматриваемого здесь приложения State Lab несколькими способами. Сначала добавим изображение в отображаемое представление, чтобы в дальнейшем показать, каким образом можно избавиться от изображения, загруженного в оперативную память. Затем покажем, как сохранить информацию о состоянии приложения, чтобы ее можно было легко восстановить впоследствии. В заключение покажем, что все эти действия не должны отнимать много времени в основном потоке выполнения, если поставить всю эту работу в очередь фоновых заданий.

Удаление ресурсов при переходе в фоновое состояние

Поместите в папку State Lab файл smiley.png из папки 15 – Image архива проектов из этой книги. Установите флагок в режим, предписывающий среде Xcode копировать файл в каталог текущего проекта. Только не вводите этот файл в каталог ресурсов Assets.xcassets, поскольку это приведет к автоматическому кешированию, способному помешать управлению конкретными ресурсами, которое мы собираемся реализовать.

Введите свойства, выделенные ниже полужирным шрифтом, как для самого изображения, так и для его представления, в исходный код из файла ViewController.swift.

```
class ViewController: UIViewController {
    private var label:UILabel!
    private var smiley:UIImage!
    private var smileyView:UIImageView!
    private var animate = false
```

Затем установите представление изображения и разместите его на экране, внеся в метод viewDidLoad() корректины, приведенные в листинге 15.8.

Листинг 15.8. Модифицированный метод viewDidLoad

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

```

// Дополнительная настройка после загрузки представления,
// обычно из nib-файла.

let bounds = view.bounds
let labelFrame = CGRect(origin: CGPoint(x: bounds.origin.x,
                                         y: bounds.midY - 50),
                        size: CGSize(width: bounds.size.width,
                                     height: 100))
label = UILabel(frame:labelFrame)
label.font = UIFont(name:"Helvetica", size:70)
label.text = "Bazinga!"
label.textAlignment = NSTextAlignment.center
label.backgroundColor = UIColor.clear()

// Размер изображения smiley.png равен 84 x 84
let smileyFrame = CGRect(x: bounds.midX - 42,
                         y: bounds.midY/2 - 42, width: 84, height: 84)

smileyView = UIImageView(frame:smileyFrame)
smileyView.contentMode = UIViewContentMode.center
let smileyPath =
    Bundle.main.pathForResource("smiley", ofType: "png")!
smiley = UIImage(contentsOfFile: smileyPath)
smileyView.image = smiley
view.addSubview(smileyView)

view.addSubview(label)

let center = NotificationCenter.default
center.addObserver(self,
                   selector: #selector(ViewController.
applicationWillResignActive),
                   name: NSNotification.Name.UIApplicationWillResignActive,
object: nil)
center.addObserver(self,
                   selector: #selector(ViewController.applicationDidBecomeActive),
                   name: NSNotification.Name.UIApplicationDidBecomeActive, object: nil)
}

```

Постройте и запустите данное приложение на выполнение, чтобы увидеть на экране счастливо улыбающуюся рожицу над вращающимся текстом (рис. 15.5).

Попробуйте сначала нажать главную кнопку, чтобы перевести данное приложение в фоновое состояние, а затем коснуться пиктограммы, чтобы запустить его снова. Как видите, приложение запускается из того же состояния, в котором оно было остановлено. Это, конечно, удобно для пользователя, но мы еще не оптимизировали использование ресурсов в той степени, в какой могли бы это сделать. Напомним, что чем меньше ресурсов используется, когда приложение находится в фоновом состоянии, тем меньше риска, что система iOS прекратит его выполнение вообще. Поэтому удалим из памяти по возможности все восстанавливаемые ресурсы, чтобы повысить шансы, что приложение останется активным и восстановит свою работу максимально быстро.



Рис. 15.5. Вид приложения State Lab с вращающимся текстом метки и пиктограммой улыбки

Посмотрим, что можно сделать с изображением. Было бы желательно освободиться от изображения в оперативной памяти при переходе в фоновое состояние и восстановить его при возврате из этого состояния. С этой целью нужно ввести еще две регистрации уведомлений в теле метода viewDidLoad(), как показано ниже.

```
center.addObserver(self,
    selector: #selector(ViewController.applicationDidEnterBackground),
    name: NSNotification.Name.UIApplicationDidEnterBackground, object: nil)
center.addObserver(self,
    selector: #selector(ViewController.applicationWillEnterForeground),
    name: NSNotification.Name.UIApplicationWillEnterForeground, object: nil)
```

Далее нужно реализовать два новых метода следующим образом.

```
func applicationDidEnterBackground() {
    print("VC: \(#function)")
    self.smiley = nil;
    self.smileyView.image = nil;
}
```

```

func applicationWillEnterForeground() {
    print("VC: \(_FUNCTION_)")
    let smileyPath =
        Bundle.main.path(forResource: "smiley", ofType: "png")!
    smiley = UIImage(contentsOfFile: smileyPath)
    smileyView.image = smiley
}

```

Постройте и запустите данное приложение на выполнение и повторите те же самые действия для его перевода сначала в фоновое, а затем в активное состояние. С точки зрения пользователя, поведение данного приложения почти не меняется. Если же вы хотите убедиться, что некоторые изменения в нем все же происходят, закомментируйте метод applicationWillEnterForeground() и еще раз постройте и запустите данное приложение на выполнение. Теперь вы должны ясно увидеть, как исчезает изображение.

Сохранение состояния при переходе в фоновое состояние

После того как был продемонстрирован пример освобождения некоторых ресурсов при переходе в фоновое состояние, настало время подумать о сохранении состояния. Напомним, что основной смысл данной операции состоит в том, чтобы сохранить всю информацию, касающуюся действий пользователя, если приложение будет удалено из оперативной памяти, а пользователь смог в следующий раз продолжить работу с того места, где она была остановлена.

Конкретное состояние, о котором здесь идет речь, сильно зависит от самого приложения, а не от представления. Сохранение состояния не следует путать с сохранением местоположения представлений или экрана, просматривавшегося пользователем в последний раз, когда приложение было активно. Для этого в iOS предоставляется отдельный механизм сохранения и восстановления состояния, подробно описанный в руководстве *App Programming Guide for iOS*, доступном по адресу <https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/StrategiesforImplementingYourApp/StrategiesforImplementingYourApp.html>). В данном случае речь идет о чем-то вроде пользовательских установок в приложениях, для которых не хотелось бы реализовывать отдельный комплект параметров настройки. Используя тот же самый интерфейс прикладного программирования класса UserDefaults, представленного в главе 12, можно быстро и просто сохранить пользовательские установки в приложении, чтобы в дальнейшем восстановить их обратно. Разумеется, если приложение не является визуально сложным или же если применять механизм сохранения и восстановления состояния не требуется, то информацию, позволяющую восстановить визуальное состояние приложения, можно сохранить и в пользовательских установках.

Пример приложения State Lab слишком простой, чтобы по-настоящему реализовать в нем пользовательские установки. Поэтому выберем кратчайший путь к достижению цели, введя состояние, характерное для данного приложения, в

один и только один его контроллер представления. С этой целью добавьте в файл `ViewController.swift` свойство `index` и сегментированный элемент управления.

```
class ViewController: UIViewController {
    private var label:UILabel!
    private var smiley:UIImage!
    private var smileyView:UIImageView!
    private var segmentedControl:UISegmentedControl!
    private var index = 0
    private var animate = false
```

Сначала мы собираемся предоставить пользователю возможность установить значение свойства `index` с помощью сегментированного элемента управления и сохранить его в пользовательских установках по умолчанию. Затем мы остановим и перезапустим приложение, чтобы продемонстрировать возможность восстановления сохраненного значения свойства.

Далее перейдите к середине тела метода `viewDidLoad()`, в котором создается сегментированный элемент управления, чтобы ввести его в представление следующим образом.

```
smileyView.image = smiley
segmentedControl =
    UISegmentedControl(items: ["One", "Two", "Three", "Four"])
segmentedControl.frame = CGRect(x: bounds.origin.x + 20, y: 50,
    width: bounds.size.width - 40, height: 30)
segmentedControl.addTarget(self,
    action: #selector(ViewController.selectionChanged(_:)),
    for: UIControlEvents.valueChanged)

view.addSubview(segmentedControl)
view.addSubview(smileyView)
```

Для связывания сегментированного элемента управления с методом `selectionChanged()`, который требуется вызывать, когда изменяется выбранный сегмент, мы воспользовались методом `addTarget (_:action:forControlEvents)`. Реализовать этот метод можно где-нибудь в классе `ViewController`, как показано ниже. Если теперь пользователь изменит выбранный сегмент, значение свойства `index` будет обновлено.

```
func selectionChanged(_ sender:UISegmentedControl) {
    index = segmentedControl.selectedSegmentIndex;
}
```

Постройте и выполните данное приложение. Вы должны увидеть сегментированный элемент управления и иметь возможность щелкать на его сегментах по очереди. При этом значение свойства `index` будет изменяться, хотя это и незаметно. Еще раз переведите приложение в фоновый режим, щелкнув на главной кнопке, откройте панель задач (дважды щелкнув на главной кнопке), удалите приложение, а затем запустите его снова. Когда приложение запустится

на выполнение снова, свойство `index` будет иметь нулевое значение и ни один сегмент не будет выбран. Устраним этот недостаток.

Сохранить значение свойства `index` совсем не трудно. Для этого достаточно ввести одну строку кода в конце метода `applicationDidEnterBackground()` из файла `ViewController.swift`.

```
func applicationDidEnterBackground() {
    print("VC: \(#function)")
    self.smiley = nil;
    self.smileyView.image = nil;
    UserDefaults.standard.set(self.index,
        forKey: "index")
}
```

Однако где именно мы должны восстановить индекс выделенного сегмента, чтобы воспользоваться им для настройки сегментированного элемента управления? Противоположный данному метод `applicationWillEnterForeground()` нам, скорее всего, не поможет. Ведь когда этот метод вызывается, приложение уже выполняется, а данная его настройка остается по-прежнему неизменной. Вместо этого нам требуется доступ к данной настройке после нового запуска приложения, что возвращает нас назад к методу `viewDidLoad()`. Введите в конце этого метода несколько строк кода, выделенных ниже полужирным шрифтом.

```
view.addSubview(label)

index = UserDefaults.standard.integer(forKey: "index")
segmentedControl.selectedSegmentIndex = index;
```

Когда данное приложение запускается на выполнение в первый раз, в пользовательских настройках по умолчанию отсутствует какое-либо значение. В данном случае метод `integerForKey()` возвращает нулевое значение, которое оказывается правильным начальным значением свойства `index`. Если же требуется использовать другое начальное значение этого свойства, то для этого достаточно зарегистрировать его как значение индексного ключа по умолчанию, как поясняется в разделе “Регистрация значений по умолчанию” главы 12.

Снова постройте и запустите данное приложение на выполнение. Вы сразу же заметите, что в сегментированном элементе управления предварительно выбран первый сегмент, поскольку в методе `viewDidLoad()` был установлен индекс выбранного сегмента этого элемента управления. Теперь коснитесь этого сегмента и затем выполните последовательность действий перехода в фоновое состояние, удаления и перезапуска приложения. Значение индекса восстановлено, а следовательно, в сегментированном элементе управления выбран правильный сегмент.

Очевидно, что приведенный здесь пример достаточно прост, но сам продемонстрированный в нем принцип может быть распространен на все виды состояний приложения. Вам решать, до какой степени следует его распространять, чтобы создать у пользователя иллюзию постоянного присутствия приложения, просто ожидающего возврата к нему.

Запрос дополнительного времени на фоновую работу

Как упоминалось выше, приложение может быть удалено из оперативной памяти, если его пребывание в фоновом состоянии отнимает слишком много времени. Например, в ходе выполнения приложения может происходить пересылка файлов, которую опасно оставлять незавершенной, но заставлять метод `applicationDidEnterBackground()` в любом случае завершать свою работу нецелесообразно. Вместо этого метод `applicationDidEnterBackground()` следует использовать для того, чтобы дать системе команду выполнить дополнительную работу, а затем выполнить блок кода, решающий эту задачу. При условии, что система имеет достаточный объем оперативной памяти, когда пользователь делает что-то еще, система выполнит данную ей команду и оставит приложение активным.

Продемонстрируем это на примере пересылки файлов, а на примере простого вызова метода, переводящего приложение в спящий режим. Для этого воспользуемся уже известными нам свойствами каркаса GCD, чтобы выполнить метод `applicationDidEnterBackground()` в отдельной очереди.

Внесите в исходный код метода `applicationDidEnterBackground()` из файла `ViewController.swift` изменения, приведенные в листинге 15.9.

Листинг 15.9. Модифицированный метод `applicationDidEnterBackground`

```
func applicationDidEnterBackground() {
    print("VC: \(#function)")
    UserDefaults.standard.set(self.index,
        forKey:"index")

    let app = UIApplication.shared()
    var taskId = UIBackgroundTaskInvalid
    let id = app.beginBackgroundTask() {
        print("Background task ran out of time and was terminated.")
        app.endBackgroundTask(taskId)
    }
    taskId = id

    if taskId == UIBackgroundTaskInvalid {
        print("Failed to start background task!")
        return
    }

    DispatchQueue.global(qos: .default).async {
        print("Starting background task with " +
            "\((app.backgroundTimeRemaining) seconds remaining")
        self.smiley = nil;
        self.smileyView.image = nil;

        // Имитация продолжительной (25 с) процедуры
        Thread.sleep(forTimeInterval: 25)

        print("Finishing background task with " +
            "\((app.backgroundTimeRemaining) seconds remaining")
```

```

        app.endBackgroundTask(taskId)
    });
}

```

Проанализируем приведенный выше код по частям. Сначала в нем выбирается разделяемый экземпляр объекта типа `UIApplication`, поскольку мы будем неоднократно пользоваться им в данном методе. Затем следует приведенный ниже фрагмент кода.

```

var taskId = UIBackgroundTaskInvalid
let id = app.beginBackgroundTask() {
    print("Background task ran out of time and was terminated.")
    app.endBackgroundTask(taskId)
}
taskId = id

```

Вызывая метод `app.beginBackgroundTask()`, мы, по существу, сообщаем системе, что нам нужно больше времени на выполнение определенной работы, но обещаем уведомить ее, когда эта работа будет завершена. Замыкание, которое мы передаем данному методу в качестве параметра, может быть вызвано в том случае, если система посчитает, что мы выполняем работу слишком долго и что пора завершить фоновую задачу. В результате вызова метода `app.beginBackgroundTask()` возвращается идентификатор, сохраняемый нами в локальной переменной `taskId` (если это в большей степени отвечает конструкции созданного вами класса контроллера представления, можете сохранить данное значение в свойстве данного класса).

Обратите внимание на то, что передаваемое нами замыкание завершается вызовом метода `endBackgroundTask()`, которому локальная переменная `taskId` передается в качестве параметра. Этим мы сообщаем системе о завершении работы, на выполнение которой запросили ранее дополнительное время. Каждый вызов метода `app.beginBackgroundTask()` очень важно согласовать с вызовом метода `endBackgroundTask()`, чтобы своевременно уведомить систему о завершении работы.

ЗАМЕЧАНИЕ. В зависимости от уровня вашей подготовки в области компьютерных наук употребляемый здесь термин **задание** может вызвать у вас ассоциации с тем, что обычно называют **процессом**, состоящим из выполняемой в нескольких потоках программы. В данном контексте термин **задание** означает "нечто, требующее выполнения". Любое формируемое задание по-прежнему выполняется в пределах выполняемого приложения.

Затем проводится следующая проверка:

```

if taskId == UIBackgroundTaskInvalid {
    print("Failed to start background task!")
    return
}

```

В предыдущем вызове метода `app.beginBackgroundTask()` возвращалось специальное значение `UIBackgroundTaskInvalid`, которое означает, что система отказывается вообще предоставлять дополнительное время. В таком случае можно попытаться выполнить самую быструю часть необходимой работы в надежде уложиться в отведенный пятисекундный срок. К такому приему, вероятнее всего, придется прибегнуть с целью выполнения приложения на старых моделях мобильных устройств, например iPhone 3G, которые не поддерживают многозадачность. Однако в данном случае мы обходим такую возможность, просто сообщая о том, что запустить фоновое задание на выполнение не удалось. Далее следует самая интересная часть кода, в которой фактически выполняется сама работа.

```
DispatchQueue.global(qos: .default).async {
    print("Starting background task with " +
        "\u2022\app.backgroundTimeRemaining) seconds remaining")
    self.smiley = nil;
    self.smileyView.image = nil;

    // Имитация продолжительной (25 с) процедуры
    Thread.sleep(forTimeInterval: 25)

    print("Finishing background task with " +
        "\u2022\app.backgroundTimeRemaining) seconds remaining")
    app.endBackgroundTask(taskId)
};
```

В этой части кода работа, выполнявшаяся рассматриваемым здесь методом, в первую очередь, помещается в очередь фоновых заданий. Следует, однако, иметь в виду, что код, в котором класс `UserDefaults` применяется для сохранения состояния, не был перемещен в отдельный блок кода. Это сделано потому, что данное состояние очень важно сохранить независимо от того, предоставит ли система iOS дополнительное время для выполнения приложения, когда оно переходит в фоновое состояние. В конце рассматриваемого здесь фрагмента кода вызывается метод `endBackgroundTask()`, чтобы уведомить систему о завершении задания.

Итак, внеся упомянутые выше изменения в исходный код рассматриваемого здесь приложения, постройте и запустите его на выполнение, а затем переведите в фоновое состояние, нажав кнопку Home. Следите за консолью Xcode, и спустя 25 с увидите результаты окончательного вывода в журнал консольных сообщений. Если приложение выполняется полностью до данного момента, на консоль должны быть выведены следующие строки сообщений.

```
application(_ : didFinishLaunchingWithOptions:)
applicationDidBecomeActive
VC: applicationDidBecomeActive()
applicationWillResignActive
VC: applicationWillResignActive()
applicationDidEnterBackground
```

```
VC: applicationWillEnterBackground()
Starting background task with 179.808078499991 seconds remaining
Finishing background task with 154.796897583336 seconds remaining
```

Как видите, система намного более щедро выделила время на выполнение операций в фоновом режиме, чем в основном потоке выполнения приложения. Благодаря этому, придерживаясь рассмотренной выше процедуры, вы сможете поспособствовать успешному выполнению любых текущих заданий в своем приложении.

Следует иметь в виду, что в данном примере мы запросили единственный идентификатор фонового задания, но на практике их можно запросить сколько угодно. Так, если во время работы в фоновом режиме происходят сетевые пересылки данных, которые нужно непременно завершить, на каждую из них можно запросить отдельный идентификатор, чтобы продолжить их выполнение в очереди фоновых заданий. Это дает возможность распараллелить выполнение многих операций в течение имеющегося времени.

Резюме

Материал этой главы оказался довольно насыщенным новыми понятиями и непростым для усвоения. Из нее вы узнали о совершенно новых принципах распараллеливания заданий, позволяющих не особенно задумываться об организации потоков выполнения и выборе средств, обеспечивающих правильное поведение приложений в многозадачной среде iOS. Теперь, когда вы освоили столь сложный материал, можно переходить к следующей главе, посвященной рисованию.

ГЛАВА 16



Графика и рисование

Все созданные ранее приложения были построены из представлений и элементов управления, входящих в состав каркаса UIKit. Из готовых компонентов UIKit можно сделать немало полезного. В частности, интерфейсы подавляющего большинства приложений можно построить, используя только эти объекты. Однако некоторые визуальные компоненты (рис. 16.1) нельзя понять до конца, не рассмотрев их отдельно от каркаса UIKit.

От приложения, например, иногда требуется способность выполнять специальные графические функции. Правда, в состав iOS входит каркас Core Graphics, позволяющий решать самые разные задачи рисования. В этой главе рассматриваются потенциальные возможности данной графической среды и поясняются основные принципы ее действия.

Библиотека QUARTZ 2D

Одним из главных компонентов Core Graphics является набор прикладных программных интерфейсов под общим названием “Quartz 2D”. Это библиотека функций, типов данных и объектов, предоставляющих возможность рисовать непосредственно в представлении или изображении, находящемся в оперативной памяти. В библиотеке Quartz 2D представление или изображение трактуется как рисуемое на виртуальном холсте, следуя так называемой модели художника. Это оригинальное название означает, что команды рисования применяются подобно краске, накладываемой на холст.

Если художник сначала раскрасит весь холст красной краской, а затем нижнюю его

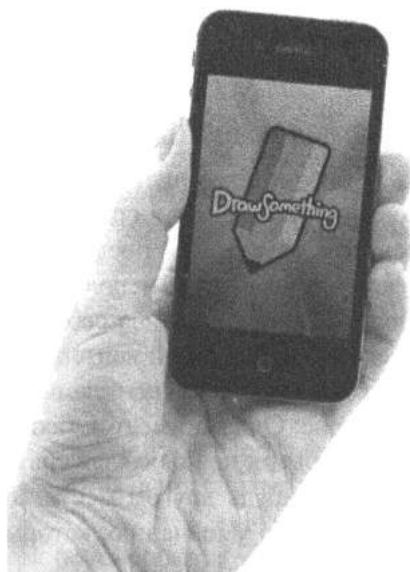


Рис. 16.1. Приложение, интенсивно использующее графику, которая выходит за пределы возможностей библиотеки UIKit

половину — синей краской, то холст получится наполовину красным и наполовину синим (если синяя краска непрозрачная) или фиолетовым (если синяя краска полупрозрачная). По такому же принципу действует и виртуальный холст в библиотеке Quartz 2D. Если заполнить все представление сначала красным цветом, а затем нижнюю его половину — синим цветом, то представление получится наполовину красным и наполовину синим, в зависимости от того, было ли второе рисующее действие непрозрачным или полупрозрачным по своему характеру. Каждое рисующее действие накладывается на виртуальный холст поверх любых предыдущих рисующих действий.

В библиотеке Quartz 2D предоставляются разнообразные функции рисования линий, форм и изображений. Теперь, когда у вас сложилось общее представление о принципах рисования средствами Quartz 2D, попробуем воспользоваться ими на практике. Начнем с самых основ применения Quartz 2D, а затем построим простое графическое приложение средствами этой библиотеки.

Подход к рисованию в библиотеке Quartz 2D

Пользуясь библиотекой Quartz 2D, нужно, как правило, вводить в представление код, выполняющий рисование. Например, можно создать подкласс, производный от класса `UIView`, и ввести вызовы функций из библиотеки Quartz 2D в метод `draw(_ rect:)` этого класса. Метод `draw(_ :rect)` является частью определения класса `UIView` и вызывается всякий раз, когда представлению требуется перерисовка. Если ввести вызов функции из библиотеки Quartz 2D в метод `draw(_ rect:)`, то в результате вызова этой функции представление будет перерисовано.

Графические контексты Quartz 2D

В библиотеке Quartz 2D, как и в остальной части каркаса Core Graphics, рисование происходит в **графическом контексте**, обычно называемом просто **контекстом**. С каждым представлением связан определенный контекст. Этот контекст извлекается и используется для вызова различных графических функций Quartz 2D, принимая на себя все хлопоты по визуализации рисуемой графики в представлении. Такой контекст можно рассматривать как своего рода холст. Система предоставляет контекст по умолчанию, в котором содержимое появляется на экране. Однако имеется также возможность создать свой контекст для рисования, чтобы он не появлялся сразу, а был сохранен для последующего применения в каком-нибудь другом месте. Здесь и далее основное внимание будет уделено контексту по умолчанию, который можно получить с помощью кода

```
let context = UIGraphicsGetCurrentContext()
```

Графический контекст относится к типу `CGContext`. Этот тип получается в результате преобразования из языка C в Swift типа указателя `CGContextRef`, являющегося платформенно-ориентированным представлением графического

контекста в каркасе Core Graphics. Конкретным выводимым типом переменной графического контекста в приведенной выше строке кода оказывается CGContext. Однако это совсем необязательно, поскольку в результате вызовов функций на языке С теоретически может быть возвращено значение NULL, которое само разворачивается, избавляя от необходимости разворачивать каждую ссылку на графический контекст. Несмотря на это функция UIGraphicsGetCurrentContext() не возвращает значение NULL при условии, что она применяется там, где гарантируется текущий контекст.

ЗАМЕЧАНИЕ. Каркас Core Graphics представляет собой интерфейс прикладного программирования, написанный на языке С. Имена всех функций данного каркаса, начинающиеся с префикса CG и демонстрируемые в примерах кода из этой главы, на самом деле написаны на языке С, а не на Swift.

Определив графический контекст, можно приступить к рисованию в нем, передавая его различным графическим функциям Core Graphics. Например, в приведенном ниже листинге 16.1 сначала создается контур, описывающий простую линию, а затем рисуется этот контур.

Листинг 16.1. Рисование в графическом контексте

```
context?.setLineWidth(4.0)
context?.setStrokeColor(UIColor.red.cgColor)
context?.moveTo(x: 10.0, y: 10.0)
context?.addLineTo(x: 20.0, y: 20.0)
context?.strokePath()
```

При вызове первой функции указывается, что любые последующие команды рисования, создающие текущий контур, должны быть выполнены с толщиной кисти 4 пикселя в заданном графическом контексте. Это все равно что выбрать размер кисти для рисования. И до тех пор, пока эта функция не будет вызвана снова с другим числовым значением второго аргумента, все линии будут рисоваться толщиной 4 пикселя. Затем указывается красный цвет обводки. С действиями рисования в Core Graphics связаны следующие две разновидности цвета.

- ➊ **Цвет обводки**, используемый для рисования линий по очертаниям форм.
- ➋ **Цвет заливки**, используемый для заполнения форм.

С графическим контекстом связано своего рода невидимое перо, которым рисуется линия. По ходу выполнения команд рисования движениями этого пера образуется контур. При вызове функции `moveTo(x:, y:)` виртуальное перо поднимается и конечная точка текущего контура перемещается в заданное место, но при этом ничего не рисуется. Любая последующая операция будет выполнена относительно той точки, в которую перемещено перо. В приведенном выше примере кода перо было перемещено в точку с координатами (10,10). При вызове следующей функции линия рисуется от текущего местоположения

пера в точке с координатами (10,10) до указанного местоположения в точке с координатами (20,20), которая становится новым местом расположения пера.

Рисуя с помощью каркаса Core Graphics, на самом деле вы не видите нарисованное, — по крайней мере видите не сразу; вы только создаете контур, который может стать формой, линией или каким-нибудь другим графическим объектом, но он не обладает цветом или иными свойствами, делающими его видимым. Это все равно что писать симпатическими чернилами. Созданный контур остается невидимым до тех пор, пока вы не сделаете что-нибудь конкретное для того, чтобы он стал видимым. Поэтому далее необходимо указать библиотеке Quartz 2D нарисовать линию, используя функцию `CGContextStrokePath()`. В этой функции заданные ранее толщина линии и цвет обводки фактически используются для “раскраски” контура с целью сделать его видимым.

Система координат

В листинге 16.1 в качестве параметров функциям `context!.move(x:, y:)` и `context!.addLineTo(x:, y:)` были переданы числовые значения с плавающей точкой. Эти числовые значения обозначают местоположение в системе координат Core Graphics. Местоположение в этой системе координат описывается координатами x и y , которые обычно обозначаются как (x, y) . Верхний левый угол графического контекста находится в точке с координатами (0,0). При перемещении вниз увеличивается значение координаты y , а при перемещении вправо — значение координаты x . В приведенном выше фрагменте кода мы нарисовали диагональную линию, проведенную из точки с координатами (10,10) в точку с координатами (20,20), как показано на рис. 16.2.

Система координат относится к уловкам, определяющим особенности рисования средствами Quartz 2D, поскольку система координат Quartz 2D существенно отличается от тех, которые обычно применяются во многих графических библиотеках, а также от традиционной декартовой системы координат. В других графических библиотеках, включая OpenGL и даже версию Quartz для OS X, точка с координатами (0,0) располагается в левом нижнем углу, а значение координаты y увеличивается при перемещении по контексту или представлению вверх, как показано на рис. 16.3.

Для задания точки в системе координат одним функциям из библиотеки Quartz 2D в качестве параметров требуются два числовых значения с плавающей точкой, а другим функциям — встраивание точки в структуру `CGPoint`, состоящую из двух значений с плавающей точкой: x и y . Для описания размеров представления или другого объекта в Quartz 2D используется структура `CGSize`, также состоящая из двух значений с плавающей точкой: `width` и `height`. Кроме того, в библиотеке Quartz 2D определен тип данных `CGRect`, который служит для определения прямоугольника в системе координат. Он состоит из двух элементов: `CGPoint` — начало отсчета, определяющего верхний левый угол прямоугольника, а также `CGSize` — размер, определяющий ширину и высоту прямоугольника, как показано в листинге 16.2.

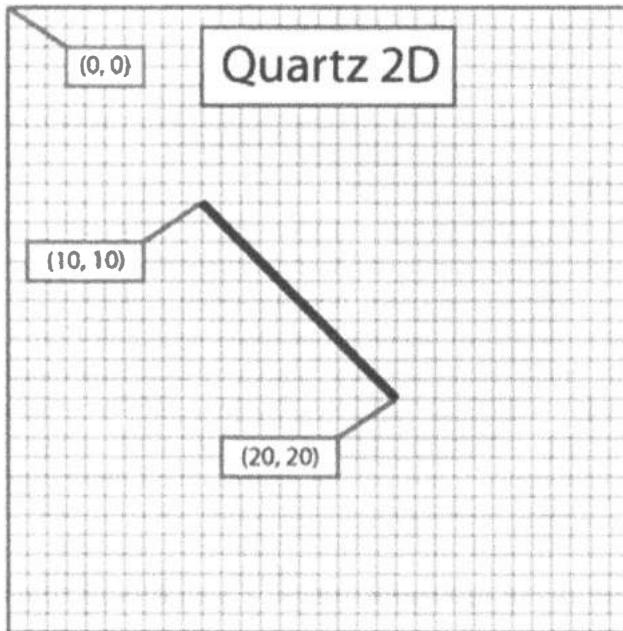


Рис. 16.2. Линия, нарисованная в системе координат Quartz 2D

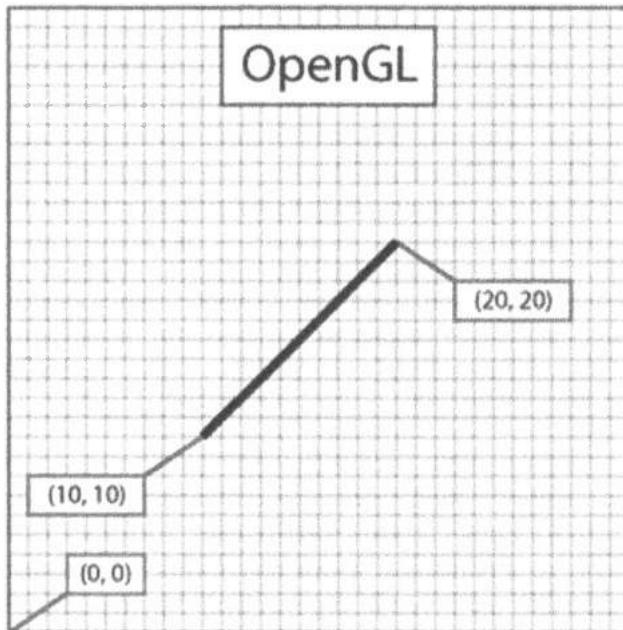


Рис. 16.3. Во многих графических библиотеках, в том числе в OpenGL, линия, проведенная из точки с координатами (10,10) в точку с координатами (20,20), будет выглядеть так, как показано на этом рисунке, а не так, как показано на рис. 16.2

Листинг 16.2. Рисование прямоугольника

```
var startingPoint = CGPoint(x: 1.0, y: 1.0)
var sizeOfrect = CGSize(width: 10.0, height: 10.0)
var rectangle = CGRect(origin: startingPoint, size: sizeOfrect)
```

Задание цветов

Важной составляющей рисования является цвет, поэтому очень важно иметь ясное представление о том, как цвета воспроизводятся в системе iOS. Для представления цвета в библиотеке UIKit предназначен класс `UIColor`. Этот класс нельзя использовать непосредственно в вызовах функций Core Graphics, но поскольку он служит лишь оболочкой для класса `cgColor`, который как раз и требуется для вызова функций Core Graphics, ссылку на объект типа `cgColor` можно извлечь из экземпляра объекта типа `UIColor`, используя его свойство `cgColor`, как это было сделано ранее и показано в приведенной ниже строке кода.

```
context!.setStrokeColor(UIColor.red.cgColor)
```

Немного теории цвета применительно к машинной графике

В современной машинной графике отображаемый на экране цвет состоит из данных, хранящихся тем или иным способом в зависимости от так называемой *цветовой модели*. Цветовая модель, называемая иначе *цветовым пространством*, просто обозначает способ представления настоящих цветов в виде дискретных значений, пригодных для обработки на компьютере. К числу наиболее распространенных способов представления цветов относится их разложение на четыре составляющие: красный, зеленый, синий (основные цвета) и альфа-канал. В библиотеке Quartz 2D каждая из этих составляющих цвета представлена значением типа `CGFloat` и должна находиться в пределах от 0,0 до 1,0.

ПРЕДУПРЕЖДЕНИЕ. В 32-разрядных системах тип `CGFloat` представлен 32-разрядным числом с плавающей точкой, и поэтому он преобразуется непосредственно в тип `Float` языка Swift. Однако в 64-разрядных системах этот тип представлен 64-разрядным числом, соответствующим типу `Double` в языке Swift. Поэтому будьте внимательны, манипулируя значениями типа `CGFloat` в коде Swift.

Красную, зеленую и синюю составляющие цвета очень легко понять, поскольку они представляют собой *основные аддитивные цвета*, или *цветовую модель RGB* (рис. 16.4). Если сложить вместе свет этих трех основных цветов в равных пропорциях, то в конечном итоге получится воспринимаемый зрением человека белый свет или же оттенок серого, если пропорции смешиваемых основных цветов окажутся не совсем равными. Сочетая эти три основных цвета в разных пропорциях, можно получить самые разные цвета, образующие *цветовую гамму*.

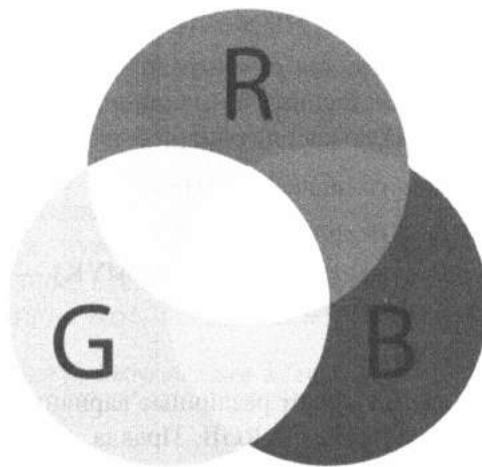


Рис. 16.4. Простое представление основных аддитивных цветов, составляющих цветовую модель RGB

На уроках рисования в школе вас, вероятно, учили, что основными цветами являются красный, желтый и синий. Эти основные цвета принято называть *исторически сложившимися основными субтрактивными цветами или цветовой моделью RYB*, но о них очень мало упоминается в современной теории цвета, а в машинной графике они почти никогда не применялись. Цветовая гамма цветовой модели RYB весьма ограничена, а кроме того, эта модель не очень легко поддается математическому определению. Как ни прискорбно об этом говорить, но вас неверно учили теории цвета в школе, по крайней мере в контексте машинной графики. Однако на будущее имейте в виду, что в машинной графике вообще и в программировании графических приложений в частности в качестве основных цветов принято использовать красный, зеленый и синий, но не красный, желтый и синий.

Помимо красного, зеленого и синего, в Quartz 2D применяется еще одна составляющая цвета, называемая *альфа-каналом* и определяющая степень прозрачности цвета. Когда один цвет наносится на другой в процессе рисования, альфа-канал используется для определения окончательно нарисованного цвета. Так, если значение альфа-канала равно 1, 0, нарисованный цвет получается совершенно непрозрачным, полностью закрывая любые находящиеся под ним другие цвета. Когда используется составляющая альфа-канала, соответствующую цветовую модель иногда еще называют *цветовой моделью RGBA*, хотя формально альфа-канал не относится к цвету, а только определяет взаимодействие одного цвета с другими цветами в процессе рисования.

Другие цветовые модели

Цветовая модель RGB считается самой распространенной в машинной графике, но она далеко не единственная. В машинной графике применяется ряд других цветовых моделей, включая следующие.

- ※ “Оттенок–насыщенность–значение” (HSV)
- ※ “Оттенок–насыщенность–яркость” (HSL)
- ※ “Голубой–пурпурный–желтый–черный” (CMYK) — применяется в четырехкрасочной офсетной печати
- ※ “Полутон” (Grayscale)

Некоторые из этих моделей имеют различные варианты, в том числе несколько вариантов цветового пространства RGB. Правда, для выполнения большинства графических операций особенно беспокоиться о применяемой цветовой модели не нужно. В большинстве случаев достаточно передать ссылку на объект типа `CGColor` объекту типа `UIColor`, а все необходимые преобразования будут выполнены в каркасе Core Graphics автоматически.

Рисование изображений в графическом контексте

Изображения можно рисовать средствами Quartz 2D непосредственно в графическом контексте. Примером тому служит еще один класс `UIImage`, который можно использовать в качестве альтернативы применению структуры данных `CGImage` из оболочки Core Graphics. Класс `UIImage` содержит методы для рисования изображений в текущем графическом контексте. Место появления рисуемого изображения в графическом контексте обозначается одним из двух следующих способов:

- ※ указанием структуры `CGPoint` для обозначения левого верхнего угла изображения;
- ※ указанием типа данных `CGRect` для заключения изображения в рамку и подгонки, если потребуется, его размеров под рамку.

Изображение типа `UIImage` можно нарисовать в текущем графическом контексте, используя код, приведенный в листинге 16.3.

Листинг 16.3. Рисование объекта класса `UIImage` в текущем контексте

```
var image:UIImage // Если допустить, что этот графический контекст
                  // существует, то указывает на экземпляр класса UIImage
let drawPoint = CGPointMake(100.0, 100.0)
image.drawAtPoint(drawPoint)
```

Рисование форм: прямоугольников, прямых и кривых линий

В библиотеке Quartz 2D предусмотрен целый ряд функций, упрощающих создание сложных форм. Для того чтобы нарисовать прямоугольник или многоугольник, не нужно рассчитывать вручную углы или выполнять иные геометрические расчеты. Достаточно вызвать соответствующую функцию из этой библиотеки, которая сделает все необходимое автоматически. Так, если требуется нарисовать эллипс, нужно лишь определить прямоугольник, в который необходимо вписать эллипс, предоставив Core Graphics возможность сделать все остальное, как показано в коде, приведенном в листинге 16.4.

Листинг 16.4. Рисование прямоугольника в текущем контексте

```
let startingPoint = CGPointMake(x: 1.0, y: 1.0)
let sizeOfrect = CGSizeMake(width: 10.0, height: 10.0)
let theRect = CGRectMake(origin: startingPoint, size:sizeOfrect)
context!.addEllipse(inRect: theRect)
context!.addRect(theRect)
```

Аналогичные методы применяются и для рисования прямоугольников. Имеются также методы, позволяющие создавать и более сложные формы, например дуги и криволинейные контуры Безье.

ЗАМЕЧАНИЕ. В примерах из этой главы сложные формы не используются. Подробнее о рисовании дуг и криволинейных контуров Безье средствами Quartz 2D читайте в руководстве *Quartz 2D Programming Guide*, доступном на сайте iOS Dev Center по адресу <http://developer.apple.com/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/> или среди имеющейся в Интернете документации по Xcode.

Образцы инструментальных средств Quartz 2D: узоры, градиенты и пунктиры

В библиотеке Quartz 2D предоставляется довольно внушительный набор инструментальных средств. В частности, средствами Quartz 2D можно заполнять многоугольники градиентами, а не только сплошными цветами. И помимо сплошных линий, в этой библиотеке имеется богатый выбор средств для рисования пунктирных линий. На рис. 16.5 приведены копии экрана iPhone с образцами графики, демонстрирующими возможности библиотеки Quartz 2D. Эти образцы взяты из примера программного проекта QuartzDemo, предоставленного компанией Apple. Теперь, когда вы получили основное представление о принципе действия библиотеки Quartz 2D и ее возможностях, попробуем воспользоваться ею на практике.

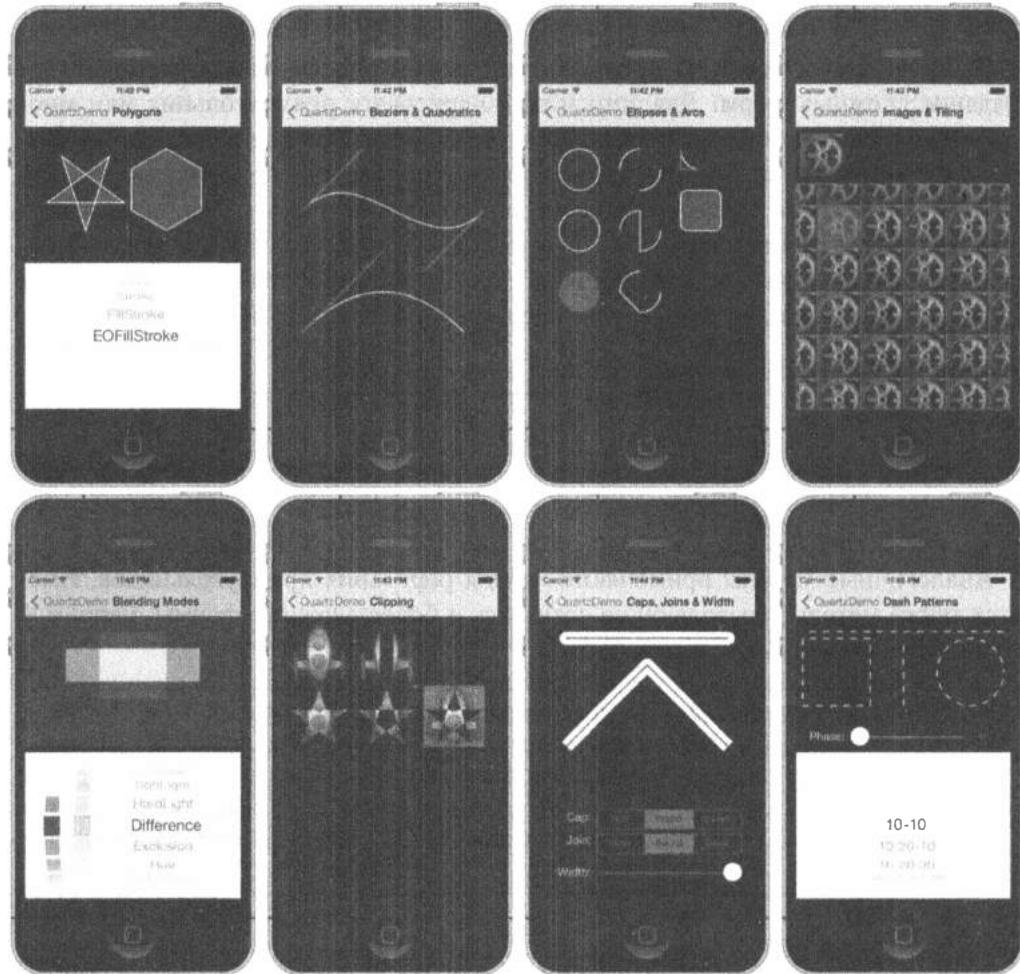


Рис. 16.5. Некоторые примеры возможностей библиотеки Quartz 2D, взятые из примера проекта QuartzDemo, предоставляемого компанией Apple

Приложение QuartzFun

Нашим следующим приложением будет простая прикладная программа рисования (рис. 16.6). Это приложение будет построено средствами Quartz 2D, чтобы наглядно показать, как согласуются между собой описанные выше принципы рисования графических объектов.

Создание приложения QuartzFun

Создайте в среде Xcode новый проект, используя шаблон Single View Application, и присвойте ему имя QuartzFun. В этом шаблоне уже имеются делегат приложения и контроллер представления. Однако поскольку мы

собираемся выполнять особые операции рисования в представлении, нам нужно создать подкласс, производный от класса `UIView`, где мы и будем рисовать, переопределяя метод `draw(_ rect:)`. Выбрав папку `QuartzFun` (которая содержит файлы делегата приложения и контроллера представления), нажмите комбинацию клавиш `<⌘+N>`, чтобы вызвать помощник для создания новых файлов, а затем выберите шаблон `Cocoa Touch Class` в разделе `iOS`. Назовите новый класс `QuartzFunView` и сделайте его подклассом, производным от класса `UIView`.

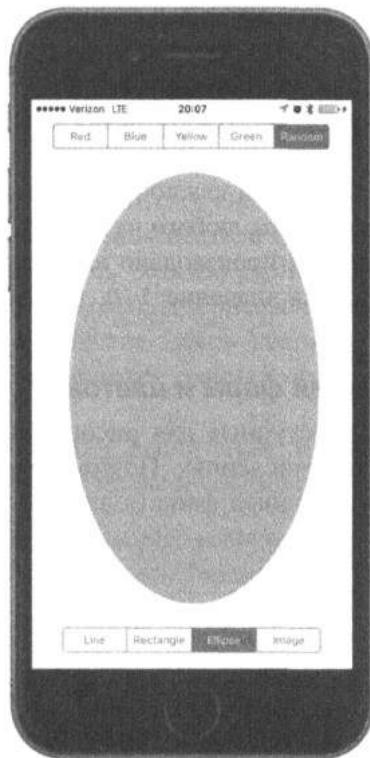


Рис. 16.6. Приложение QuartzFun в действии

Далее нам нужно ввести два перечисления: одно — для типов рисуемых форм, а другое — для имеющихся цветов. Поскольку один из этих цветов выбирается произвольно (`Random`), потребуется также метод, возвращающий произвольный цвет всякий раз, когда он вызывается. Итак, начнем с создания этого метода и двух перечислений.

Формирование произвольного цвета

Безусловно, мы могли бы определить глобальную функцию, возвращающую произвольный цвет, но ее лучше ввести в виде расширения в класс `UIColor`. С этой целью откройте исходный файл `QuartzFunView.swift` и введите ближе к его началу следующий код.

618 ГЛАВА 16 ■ ГРАФИКА И РИСОВАНИЕ

```
// Расширение класса UIColor для формирования произвольного цвета
extension UIColor {
    class func randomColor() -> UIColor {
        let red = CGFloat(Double(arc4random_uniform(255))/255)
        let green = CGFloat(Double(arc4random_uniform(255))/255)
        let blue = CGFloat(Double(arc4random_uniform(255))/255)
        return UIColor(red: red, green: green, blue: blue, alpha:1.0)
    }
}
```

Этот код довольно прост. Для каждой составляющей цвета вызывается функция `arc4random_uniform()`, формирующая числовое значение с плавающей точкой в пределах от 0,0 до 1.0. Поэтому остаток от деления произвольного значения на 256, что дает целое число в пределах от 0 до 255, делится далее на 255. Почему именно на 255? Потому что в версии Quartz 2D для iOS поддерживаются 256 разных уровней яркости каждой составляющей цвета, а число 255 гарантирует произвольность выбора любого из этих уровней яркости. И наконец новый цвет формируется из трех произвольно выбранных составляющих цвета. В альфа-канале устанавливается значение 1,0, чтобы все формируемые цвета были непрозрачными.

Определение перечислений форм и цветов

Как упоминалось выше, доступные для рисования формы и цвета должны быть представлены в двух перечислениях. Поэтому введите следующие определения этих перечислений в исходный файл QuartzFunView.swift.

```
enum Shape : Int {
    case line = 0, rect, ellipse, image
}

// Индекс закладок для выбора цветов
enum DrawingColor : Int {
    case red = 0, blue, yellow, green, random
}
```

Оба приведенные выше перечисления являются производными от класса `UInt`. Ведь для сопоставления формы или цвета с сегментом, выбираемым в сегментированном элементе управления, придется пользоваться исходными перечисляемыми значениями.

Реализация заготовки представления QuartzFunView

Рисование мы собираемся выполнять в подклассе, производном от класса `UIView`, и поэтому снабдим этот класс всем необходимым, кроме конкретного кода для рисования, который будет добавлен в дальнейшем. Итак, введите в класс `QuartzFunView` следующие шесть свойств.

```
// Свойства, которые можно задать в приложении
var shape = Shape.line
var currentColor = UIColor.red
var useRandomColor = false
```

```
// Внутренние свойства
private let image = UIImage(named:"iphone")
private var firstTouchLocation = CGPoint.zero
private var lastTouchLocation = CGPoint.zero
```

Свойство `shape` служит для отслеживания формы, которую пользователю требуется нарисовать; свойство `currentColor` определяет цвет, выбранный пользователем для рисования; а свойство `useRandomColor` принимает логическое значение `true`, если пользователь выбирает произвольный цвет для рисования. Все эти свойства предназначены для применения за пределами своего класса и на самом деле используются в контроллере представления.

Следующие три свойства требуются только в реализации своего класса и поэтому помечены как закрытые. Первые два из них служат для слежения за пальцем пользователя по мере его перемещения по экрану. Место, где пользователь первый раз коснулся экрана, хранится в свойстве `firstTouchLocation`, а место, где останавливается палец пользователя при перетаскивании по экрану, — в свойстве `lastTouchLocation`. Эти два свойства будут использоваться в коде рисования, чтобы определить место для рисования запрашиваемой формы. В заключение отметим, что в свойстве `image` хранится изображение, рисуемое на экране, когда пользователь выбирает крайний справа элемент на нижней панели инструментов (рис. 16.7).

Перейдем непосредственно к реализации. Введите следующие методы для программирования реакции на прикосновения пользователя (листинг 16.5).

Листинг 16.5. Методы для программирования реакции на касание в файле `QuardFunView.swift`

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let touch = touches.first {
        if useRandomColor {
            currentColor = UIColor.randomColor()
        }
        firstTouchLocation = touch.location(in: self)
        lastTouchLocation = firstTouchLocation
        setNeedsDisplay()
    }
}

override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let touch = touches.first {
        lastTouchLocation = touch.location(in: self)
        setNeedsDisplay()
    }
}

override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let touch = touches.first {
        lastTouchLocation = touch.location(in: self)
        setNeedsDisplay()
    }
}
```



Рис. 16.7. Рисование изображения типа `UIImage` в приложении QuartzFun

Эти методы наследуются из класса `UIView`, который, в свою очередь, наследует их от своего родительского класса `UIResponder`. Их можно переопределить, чтобы выяснить, где именно пользователь касается экрана. Ниже поясняется, как действуют эти методы.

- ❖ Метод `touchesBegan(_:withEvent:)` вызывается, когда пользователь первый раз касается пальцем экрана. В этом методе изменяется цвет, если пользователь выбрал произвольный цвет, для чего используется новый метод `randomColor()`, введенный ранее в класс `UIColor`. После этого сохраняется текущее местоположение, чтобы знать, где именно пользователь коснулся экрана первый раз. Для указания на необходимость перерисовки представления метод `setNeedsDisplay()` вызывается по ссылке `self`.
- ❖ Метод `touchesMoved(_:withEvent:)` вызывается в течение всего времени, пока пользователь проводит пальцем по экрану. В этом методе лишь сохраняется новое местоположение в свойстве `lastTouchLocation` и указывается на необходимость перерисовки экрана.

- Метод `touchesEnded(_:withEvent:)` вызывается, когда пользователь отнимает палец от экрана. Как и в методе `touchesMoved(_:withEvent:)`, в данном методе лишь сохраняется конечное местоположение в свойстве `lastTouchLocation` и указывается на необходимость перерисовки представления.

Если остальная часть рассматриваемого здесь кода вам не совсем понятна, не переживайте. Особенности обработки событий, связанных с прикосновением к экрану, и применения методов `touchesBegan(_:withEvent:)`, `touchesMoved(_:withEvent:)` и `touchesEnded(_:withEvent:)` будут рассмотрены более подробно в главе 18.

Мы еще вернемся к данному классу, когда заготовка рассматриваемого здесь приложения будет готова к применению. Все основные действия в данном приложении выполняются в методе `draw(_ rect:)`, который в настоящий момент закомментирован, поскольку код для него еще не написан. Прежде чем вводить код рисования, следует завершить написание приложения.

Создание и связывание выходов и действий

Прежде чем приступить к рисованию, мы должны ввести сегментированные элементы управления в графическом пользовательском интерфейсе рассматриваемого здесь приложения, а затем связать действия с выходами. С этой целью щелкните на файле `Main.storyboard`. В первую очередь, нужно изменить класс представления. Для этого перейдите к схеме документа, разверните элементы сцены и содержащего ее контроллера, а затем щелкните на элементе `View`. Нажмите комбинацию клавиш `<Shift+⌘+3>`, чтобы открыть инспектор идентичности, и смените класс с `UIView` на `QuartzFunView`.

Далее найдите в библиотеке объектов сегментированный элемент управления и перетащите его в верхнюю часть представления, опустив прямо под строкой состояния. Расположите его приблизительно по центру, как показано на рис. 16.8. Особая точность в расположении этого элемента управления не требуется, поскольку для выравнивания по центру далее будут наложены ограничения на его компоновку.

Если сегментированный элемент управления все еще выбран, откройте инспектор атрибутов и измените количество сегментов с 2 на 5. Дважды щелкните на каждом сегменте по очереди, изменив их метки слева направо в следующем порядке: `Red`, `Blue`, `Yellow`, `Green` и `Random`. Затем наложите ограничения на компоновку. С этой целью, находясь в схеме документа, нажмите клавишу `<Control>` и перетащите указатель от сегментированного элемента управления к элементу `Quartz Fun View`, отпустите кнопку мыши и выберите пункт `Vertical Spacing to Top Layout Guide` из всплывающего меню. Повторите эту операцию снова, но на этот раз выберите пункт `Center Horizontally in Container` из всплывающего меню, а затем нажмите клавишу `<Return>`. Выберите в окне `Document Outline` пиктограмму `View Controller`, вернитесь к редактору раскладовки, щелкните на кнопке `Resolve Auto Layout`, расположенной слева от кнопки `Pin`,

и выберите пункт меню **Update Frames**. Если этот пункт не выбран, выберите сегментированный элемент управления в схеме документа. В итоге этот элемент управления должен быть расположен надлежащим образом, имея установленные размеры (рис. 16.8).

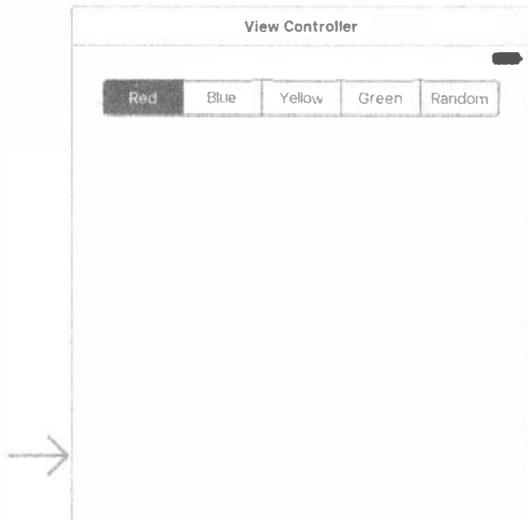


Рис. 16.8. Добавление сегментированного элемента управления для выбора цвета с правильной меткой и позицией

Откройте помощник редактора, если он еще не открыт, и выберите файл `ViewController.swift` на панели быстрых переходов. Нажмите клавишу `<Control>` и перетащите указатель от сегментированного элемента управления в схеме документа к файлу `ViewController.swift`, расположенному справа от линии, проходящей под объявлением класса. Отпустите кнопку мыши, чтобы создать новый выход. Назовите его `colorControl` и оставьте все остальные параметры заданными по умолчанию. Класс `ViewController` должен выглядеть следующим образом:

```
class ViewController: UIViewController {
    @IBOutlet var colorControl: UISegmentedControl!
```

Теперь добавим действие. С этой целью откройте файл `ViewController.swift` в помощнике редактора, снова выберите файл `Main.storyboard`, нажмите клавишу `<Control>` и перетащите указатель сегментированного элемента управления вниз, к определению данного класса в файле контроллера представления, отпустив кнопку мыши прямо над закрывающей фигурной скобкой. На этот раз измените тип связи на `Action` и присвойте новому действию имя `changeColor`. По умолчанию во всплывающем меню должно появиться событие `Value Changed`, что нам и требуется. Следует также установить тип `UISegmentedControl` данного элемента управления.

Введем второй сегментированный элемент управления. Он послужит для выбора рисуемой формы. С этой целью перетащите сегментированный элемент управления из библиотеки объектов, опустив его у нижнего края представления. Выберите новый элемент управления в схеме документа, откройте инспектор атрибутов и смените количество сегментов с 2 на 4. Дважды щелкните на каждом сегменте и измените поочереди их названия на Line, Rect, Ellipse и Image. Далее нужно наложить ограничения на компоновку данного элемента управления, чтобы зафиксировать его размеры и положение таким же образом, как это было сделано для элемента управления выбором цвета. С этой целью выполните следующие действия.

1. Находясь в окне Document Outline, нажмите клавишу <Control> и перетащите указатель от нового сегментированного элемента управления к элементу Quartz Fun View, отпустите кнопку мыши, выберите команды Vertical Spacing to Bottom Layout Guide и Center Horizontally in Container из всплывающего меню и нажмите клавишу Return.
2. Выберите пиктограмму View Controller в окне Document Outline, а затем вернитесь в редактор раскладовки, щелкните на кнопке Resolve Auto Layout Issues и выберите пункт Update Frames из всплывающего меню.

Сделав все это, снова откройте файл ViewController.swift в помощнике редактора, нажмите клавишу <Control> и перетащите указатель от нового сегментированного элемента управления в самый конец файла ViewController.swift, чтобы создать еще одно действие. Измените тип связи на Action, присвойте новому действию имя changeShape, а тип данного элемента управления измените на UISegmentedControl. В итоге раскладовка должна выглядеть так, как показано на рис. 16.9. Следующей нашей задачей является реализация методов действия.



Рис. 16.9. Вид раскладовки представления с двумя сегментированными элементами управления, расположенными на своих местах

Реализация методов действия

Сохраните раскладовку и закройте окно помощника редактора. Откройте исходный файл ViewController.swift, найдите в нем заготовочную реализацию метода changeColor(), автоматически созданную в Xcode, и введите в нее строки кода, приведенные в листинге 16.6.

Листинг 16.6. Метод changeColor в файле ViewController.swift

```
@IBAction func changeColor(_ sender: UISegmentedControl) {
    let drawingColorSelection =
        DrawingColor(rawValue: UInt(sender.selectedSegmentIndex))
    if let drawingColor = drawingColorSelection {
        let funView = view as! QuartzFunView
        switch drawingColor {

            case .red:
                funView.currentColor = UIColor.red
                funView.useRandomColor = false

            case .blue:
                funView.currentColor = UIColor.blue
                funView.useRandomColor = false

            case .yellow:
                funView.currentColor = UIColor.yellow
                funView.useRandomColor = false

            case .green:
                funView.currentColor = UIColor.green
                funView.useRandomColor = false

            case .random:
                funView.useRandomColor = true
        }
    }
}
```

Эти изменения в коде довольно просты. В методе changeColor() мы просто анализируем, какой именно сегмент был выбран, и на основании этого выбора формируем новый цвет для рисования. Для сопоставления выбранного индекса сегментированного элемента управления с перечисляемым значением соответствующего цвета воспользуемся следующим конструктором перечисления, принимающим исходное значение:

```
let drawingColorSelection =
    DrawingColor(rawValue: UInt(sender.selectedSegmentIndex))
```

После этого мы устанавливаем свойство currentColor таким образом, чтобы в классе было известно, каким именно цветом следует рисовать, если только не был выбран произвольный цвет. В этом случае в свойстве useRandomColor устанавливается значение true, а следовательно, новый цвет будет выбираться

всякий раз, когда пользователь начинает новое действие рисования, как следует из исходного кода введенного ранее метода touchesBegan(_:withEvent:). Нам не нужно ничего больше добавлять в этот метод, поскольку весь код рисования будет находиться в самом представлении.

Теперь найдите существующую реализацию метода changeShape() и введите в нее следующие строки кода.

```
@IBAction func changeShape(_ sender: UISegmentedControl) {
    let shapeSelection = Shape(rawValue: UInt(sender.
selectedSegmentIndex))
    if let shape = shapeSelection {
        let funView = view as! QuartzFunView
        funView.shape = shape
        colorControl.isHidden = shape == Shape.image
    }
}
```

В методе changeShape() просто устанавливается тип формы в зависимости от выбранного сегмента элемента управления. Четыре элемента перечисления Shape соответствуют четырем сегментам панели инструментов, расположенной в нижней части представления рассматриваемого здесь приложения. Поэтому мы задаем такую же форму, как и в выбранном в настоящий момент сегменте, а затем скрываем и показываем элемент управления выбором цвета в зависимости от того, был ли выбран сегмент Image.

Проверьте, правильно ли вы все сделали, скомпилировав и выполнив приложение. Пока вы еще не сможете рисовать на экране, но сегментированные элементы управления должны работать, и, когда вы нажмете кнопку Image, элементы управления цветом должны исчезнуть.

Если все это работает правильно, можно перейти непосредственно к рисованию.

Ввод кода рисования из библиотеки Quartz 2D

Итак, все готово ко вводу кода, выполняющего рисование. В рассматриваемое здесь приложение нам предстоит ввести функции рисования линии, некоторых форм, а также изображения.

Рисование линии

Реализуем сначала самое простое: рисование одной линии. С этой целью откройте исходный файл QuartzFunView.swift и замените в нем закомментированный метод draw(_ rect: CGRect) кодом, приведенным в листинге 16.7.

Листинг 16.7. Метод draw(rect:)

```
override func draw(_ rect: CGRect) {
    let context = UIGraphicsGetCurrentContext()
    context!.setLineWidth(2.0)
    context!.setStrokeColor(currentColor.cgColor)
```

```

switch shape {
case .line:
    context?.moveTo(x: firstTouchLocation.x,
                    y: firstTouchLocation.y)
    context?.addLineTo(x: lastTouchLocation.x,
                    y: lastTouchLocation.y)
    context?.strokePath()

case .rect:
    break

case .ellipse:
    break

case .image:
    break
}
}

```

Приведенный выше код начинается с извлечения ссылки на текущий графический контекст, чтобы знать, где следует рисовать.

```
let context = UIGraphicsGetCurrentContext()
```

Далее задается толщина линии 2,0, а это означает, что любая рисуемая линия будет иметь толщину обводки 2 пикселя.

```
context!.setLineWidth(context, 2.0);
```

После этого задается цвет обводки линий. В классе UIColor имеется свойство cgColor, которое, собственно, требуется функции установки цвета обводки линий, поэтому оно используется как свойство данного свойства currentColor для правильной передачи цвета этой функции.

```
context!.SetStrokeColor(currentColor.cgColor);
```

Затем используется оператор switch для непосредственного перехода к коду рисования формы каждого типа. Сначала мы ввели код для рисования линии в приведенной ниже ветви оператора switch, а далее в этой главе будет поочереди введен код для рисования остальных форм.

```
switch shape {
case .Line:
```

Для того чтобы нарисовать линию, дается команда создать в графическом контексте контур, начиная с того места, где пользователь коснулся пальцем экрана в первый раз, как показано ниже. Напомним, что координаты этого места запоминаются в методе touchesBegan(_:withEvent:), и поэтому оно всегда будет отражать исходную точку большинства последних касаний или скольжений пальцем по экрану.

```
context?.moveTo(x: firstTouchLocation.x,
                y: firstTouchLocation.y)
```

Затем рисуется линия от места первого касания и до последнего касания экрана пользователем, как показано ниже. Если палец пользователя по-прежнему касается экрана, в свойстве `lastTouchLocation` содержится текущее местоположение его пальца на экране. Если же пользователь больше не касается экрана пальцем, то в свойстве `lastTouchLocation` содержится местоположение его пальца в тот момент, когда он отнял его от экрана.

```
context?.addLineTo(x: lastTouchLocation.x,
                    y: lastTouchLocation.y)
```

Эта функция на самом деле не рисует линию, а только добавляет ее к контуру в текущем контексте. Для того чтобы линия появилась на экране, нужно обвести ее по контуру. Это делает следующая функция, используя заданные ранее цвет и толщину обводимой линии:

```
context?.strokePath(context);
```

На данном этапе вы должны быть в состоянии успешно скомпилировать и запустить на выполнение рассматриваемое здесь приложение. Варианты выбора `Rect`, `Ellipse` и `Image` в нем пока еще не действуют, но рисование линий любым выбранным цветом должно быть вам уже доступно, как показано на рис. 16.10.

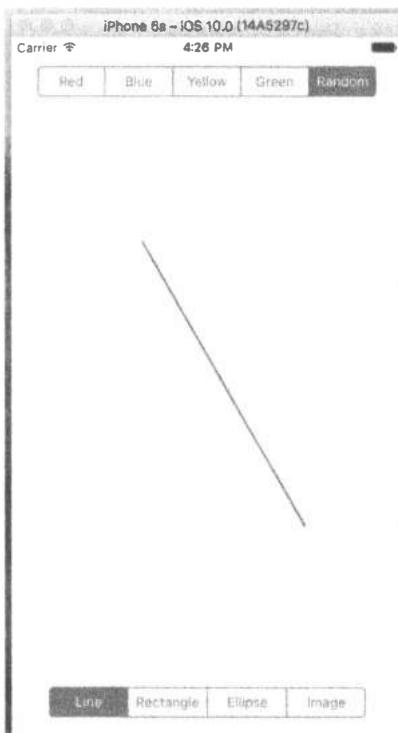


Рис. 16.10. Часть приложения, рисующая линии, уже готова к применению. На этом рисунке показана линия, нарисованная красным цветом

Рисование прямоугольника и эллипса

Теперь напишем код одновременно для рисования прямоугольника и эллипса, поскольку в библиотеке Quartz 2D эти графические объекты реализуются, по существу, одинаково. Итак, внесите в метод `draw(_ rect:)` изменения, приведенные в листинге 16.8.

Листинг 16.8. Изменения в методе `draw(rect:)` для рисования прямоугольника и эллипса

```
override func draw(_ rect: CGRect) {
    let context = UIGraphicsGetCurrentContext()
    context?.setLineWidth(2.0)
    context?.setStrokeColor(currentColor.cgColor)
    context?.setFillColor(currentColor.cgColor)
    let currentRect = CGRect(x: firstTouchLocation.x,
                             y: firstTouchLocation.y,
                             width: lastTouchLocation.x - firstTouchLocation.x,
                             height: lastTouchLocation.y -
firstTouchLocation.y)
    switch shape {
        case .line:
            context?.moveTo(x: firstTouchLocation.x,
                            y: firstTouchLocation.y)
            context?.addLineTo(x: lastTouchLocation.x,
                               y: lastTouchLocation.y)
            context?.strokePath()

        case .rect:
            context?.addRect(currentRect)
            context?.drawPath(using: .fillStroke)

        case .ellipse:
            context?.addEllipse(inRect: currentRect)
            context?.drawPath(using: .fillStroke)

        case .image:
            break
    }
}
```

Прямоугольник и эллипс нужно закрасить сплошным цветом, поэтому в начале данного метода вводится вызов следующей функции, которой передается цвет заливки с помощью свойства `currentColor`:

```
context?.SetFillColorWithColor(currentColor.cgColor);
```

Затем объявляется переменная экземпляра `CGRect`. Это делается здесь потому, что прямоугольник и эллипс рисуются на одном и том же прямоугольном основании. Переменная `currentRect` служит для хранения формы прямоугольника, описываемой скольжением пальцем по экрану. Напомним, что объект типа `CGRect(x: y: width: height:)` создается объект типа `CGRect`. Код рисования

прямоугольника довольно прост, как показано ниже. В качестве начала отсчета используется точка, хранящаяся в свойстве `firstTouchLocation`. Затем определяются размеры прямоугольника по разности двух значений `x` и двух значений `y`. Следует, однако, иметь в виду, что в зависимости от направления движения пальца по экрану один или оба размера прямоугольника могут получить отрицательные значения, но так и должно быть. Объект типа `CGRect` с отрицательным размером может быть просто нарисован в противоположном направлении от начала его отсчета, т.е. влево при отрицательной ширине или вправо при отрицательной высоте.

```
let currentRect = CGRect(x: firstTouchLocation.x,
    y: firstTouchLocation.y,
    width: lastTouchLocation.x - firstTouchLocation.x,
    height: lastTouchLocation.y - firstTouchLocation.y)
```

Как только такой прямоугольник будет определен, останется лишь вызвать две функции: одну — для рисования прямоугольника или эллипса в определенном ранее объекте типа `CGRect`, а другую — для его заполнения и обводки выбранным цветом, как показано ниже.

```
case .rect:
    context?.addRect(currentRect)
    context?.drawPath(using: .fillStroke)

case .ellipse:
    context?.addEllipse(inRect: currentRect)
    context?.drawPath(using: .fillStroke)
```

Скомпилируйте и запустите на выполнение рассматриваемое здесь приложение, а затем попробуйте нарисовать прямоугольник и эллипс инструментами `Rect` и `Ellipse`, чтобы проверить, насколько они удобны в использовании. Не забудьте поменять цвета, а также опробовать рисование обеих форм произвольным цветом.

Рисование изображения

В заключение организуем рисование изображения. В папке 16 – `Image` находятся три файла изображений, `iphone.png`, `iphone@2x.png` и `iphone@3x.png`, которые можно добавить в каталог ресурсов. Выберите каталог `Assets.xcassets` в окне навигатора проекта, а затем выберите все три изображения в окне `Finder`, перетащите их в область редактирования и создайте новую группу изображений с именем `iphone` в каталоге ресурсов. Затем введите в тело метода `draw(_ rect:)` строки кода, приведенные в листинге 16.9.

Листинг 16.9. Модифицированный метод `draw(_ rect:)` для создания изображений

```
override func draw(_ rect: CGRect) {
    let context = UIGraphicsGetCurrentContext()
    context?.setLineWidth(2.0)
    context?.setStrokeColor(currentColor.cgColor)
    context?.setFillColor(currentColor.cgColor)
```

630 ГЛАВА 16 ■ ГРАФИКА И РИСОВАНИЕ

```
let currentRect = CGRect(x: firstTouchLocation.x,
                         y: firstTouchLocation.y,
                         width: lastTouchLocation.x - firstTouchLocation.x,
                         height: lastTouchLocation.y -
firstTouchLocation.y)
switch shape {
case .line:
    context?.moveTo(x: firstTouchLocation.x,
                     y: firstTouchLocation.y)
    context?.addLineTo(x: lastTouchLocation.x,
                      y: lastTouchLocation.y)
    context?.strokePath()

case .rect:
    context?.addRect(currentRect)
    context?.drawPath(using: .fillStroke)

case .ellipse:
    context?.addEllipse(inRect: currentRect)
    context?.drawPath(using: .fillStroke)

case .image:
    let horizontalOffset = image!.size.width / 2
    let verticalOffset = image!.size.height / 2
    let drawPoint =
        CGPoint(x: lastTouchLocation.x - horizontalOffset,
                y: lastTouchLocation.y - verticalOffset)
    image!.draw(at: drawPoint)
}
}
```

В приведенном выше коде сначала рассчитывается центр изображения, поскольку его нужно отцентровать относительно того места, где пользователь коснулся экрана в последний раз. Без такого выравнивания изображение оказалось бы нарисованным начиная с левого верхнего угла в месте касания экрана в последний раз, что было бы неверно. После этого создается новый объект типа `CGPoint` вычитанием смещения относительно значений координат `x` и `y`, хранящихся в свойстве `lastTouchLocation`, как показано ниже.

```
let horizontalOffset = image!.size.width / 2
let verticalOffset = image!.size.height / 2
let drawPoint =
    CGPoint(x: lastTouchLocation.x - horizontalOffset,
            y: lastTouchLocation.y - verticalOffset)
```

В заключение дается команда нарисовать изображение. В следующей строке показано, как это делается:

```
image!.draw(at: drawPoint)
```

Постройте и запустите данное приложение на выполнение. Выберите вариант `Image` из сегментированного элемента управления и проверьте, сможете ли вы разместить изображения на холсте для рисования.

Оптимизация приложения QuartzFun

Итак, созданное нами приложение делает то, что нам нужно, но ему не помешает незначительная оптимизация. Если в небольшом приложении какие-либо задержки несущественны, то в более сложном приложении, выполняющемся на менее быстродействующем процессоре, такие задержки уже становятся заметными. Главные трудности оптимального выполнения кода скрываются в исходном файле QuartzFunView.swift, а точнее — в методах touchesMoved(_:withEvent:) и touchesEnded(_: withEvent:). Оба метода содержат следующую строку кода:

```
setNeedsDisplay()
```

Очевидно, что именно так мы можем сообщить текущему представлению о каких-то изменениях и необходимости его перерисовки. Такой код вполне работоспособен, но приводит к стиранию и перерисовке всего представления в целом, даже если произошедшие изменения незначительны. Представление приходится стирать полностью в том случае, если требуется подготовить его к рисованию новой формы, но совсем не обязательно очищать экран несколько раз в секунду.

Вместо того чтобы многократно перерисовывать все представление в процессе рисования формы, можно воспользоваться методом setNeedsDisplayInRect(). Этот метод относится к классу NSView и помечает только одну прямоугольную часть области представления как требующую перерисовки. Используя этот метод для пометки той части представления, которая оказывает влияние на текущие операции рисования и требует перерисовки, можно существенно повысить эффективность рассматриваемого здесь приложения.

Нам нужно перерисовать не только прямоугольный фрагмент экрана между точками касания, хранящимися в свойствах firstTouchLocation и lastTouchLocation, но и любой другой фрагмент экрана, охваченный в настоящий момент скольжением пальцем по нему. Так, если пользователь сначала коснется экрана, а затем небрежно проведет пальцем по всему экрану, а мы перерисуем только фрагмент между точками первого и последнего касания, то среди нарисованного на экране останется немало из того, что нам больше не нужно.

Решение задачи оптимизации состоит в том, чтобы отслеживать в переменной экземпляра CGRect весь фрагмент, охваченный движением пальца по экрану. С этой целью установим в методе touchesBegan(_:withEvent:) переменную экземпляра CGRect в состояние, соответствующее одной только точке касания пользователем экрана. Затем воспользуемся в методах touchesMoved(_:withEvent:) и touchesEnded(_:withEvent:) функцией Core Graphics, позволяющей объединить текущий и сохраненный прямоугольники и сохранить полученный в итоге прямоугольник. Им мы воспользуемся, чтобы указать ту часть представления, которая требует перерисовки. Такой подход

632 ГЛАВА 16 * ГРАФИКА И РИСОВАНИЕ

дает нам промежуточный итог для фрагмента, охваченного в настоящий момент с помощью перемещения пальца по экрану.

Итак, рассчитаем текущий прямоугольник в методе `draw(_ rect:)` для рисования форм эллипса и прямоугольника. Затем перенесем этот расчет в новый метод, чтобы воспользоваться им во всех трех местах, не повторяя код.

Ведите в класс `QuartzFunView` новое свойство `redrawRect`.

```
// Внутренние свойства
private let image = UIImage(named:"iphone")
private var firstTouchLocation = CGPoint.zero
private var lastTouchLocation = CGPoint.zero
private var redrawRect = CGRect.zero
```

Это свойство послужит для сложения за перерисовываемой областью экрана. Кроме того, расчет текущей прямоугольной области перерисовки следует вынести в отдельный метод в конце класса `QuartzFunView`, как показано ниже.

```
func currentRect() -> CGRect {
    return CGRect(x: firstTouchLocation.x,
                  y: firstTouchLocation.y,
                  width: lastTouchLocation.x - firstTouchLocation.x,
                  height: lastTouchLocation.y - firstTouchLocation.y)
}
```

Теперь замените в методе `draw(_ rect:)` все ссылки `currentRect` ссылками `currentRect()`, чтобы использовать в коде только что созданный метод. Затем удалите зачеркнутые строки кода, как показано в листинге 16.10.

Листинг 16.10. Последние изменения в методе `draw(_ rect:)`

```
override func draw(_ rect: CGRect) {
    let context = UIGraphicsGetCurrentContext()
    context?.setLineWidth(2.0)
    context?.setStrokeColor(currentColor.cgColor)
    context?.setFillColor(currentColor.cgColor)

    switch shape {
    case .line:
        context?.moveTo(x: firstTouchLocation.x,
                        y: firstTouchLocation.y)
        context?.addLineTo(x: lastTouchLocation.x,
                           y: lastTouchLocation.y)
        context?.strokePath()

    case .rect:
        context?.addRect(currentRect())
        context?.drawPath(using: .fillStroke)

    case .ellipse:
        context?.addEllipse(inRect: currentRect())
        context?.drawPath(using: .fillStroke)

    case .image:
        let horizontalOffset = image!.size.width / 2
        let verticalOffset = image!.size.height / 2
```

```

let drawPoint =
    CGPoint(x: lastTouchLocation.x - horizontalOffset,
            y: lastTouchLocation.y - verticalOffset)
image!.draw(at: drawPoint)
}
}

```

Нам нужно также внести ряд изменений в методы touchesEnded(_:withEvent:) и touchesMoved(_:withEvent:). В частности, рассчитаем пространство, охватываемое текущей операцией рисования, чтобы указать лишь ту часть текущего представления, которую требуется перерисовать. Итак, замените существующие методы touchesEnded(_:withEvent:) и touchesMoved(_:withEvent:) следующими их новыми версиями (листинг 16.11).

Листинг 16.11. Модифицированные подпрограммы обработки касаний

```

override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let touch = touches.first {
        if useRandomColor {
            currentColor = UIColor.randomColor()
        }
        firstTouchLocation = touch.location(in: self)
        lastTouchLocation = firstTouchLocation
        redrawRect = CGRect.zero
        setNeedsDisplay()
    }
}

override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let touch = touches.first {
        lastTouchLocation = touch.location(in: self)

        if shape == .image {
            let horizontalOffset = image!.size.width / 2
            let verticalOffset = image!.size.height / 2
            redrawRect = redrawRect.union(
                CGRect(x: lastTouchLocation.x -
                    horizontalOffset,
                       y: lastTouchLocation.y -
                           verticalOffset,
                       width: image!.size.width,
                       height: image!.size.height))
        } else {
            redrawRect = redrawRect.union(currentRect())
        }
        setNeedsDisplay(redrawRect)
    }
}

override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let touch = touches.first {
        lastTouchLocation = touch.location(in: self)
    }
}

```

```

if shape == .image {
    let horizontalOffset = image!.size.width / 2
    let verticalOffset = image!.size.height / 2
    redrawRect = redrawRect.union(
        CGRect(x: lastTouchLocation.x -
            horizontalOffset,
            y: lastTouchLocation.y -
            verticalOffset,
            width: image!.size.width,
            height: image!.size.height))
} else {
    redrawRect = redrawRect.union(currentRect())
}
setNeedsDisplay(redrawRect)
}
}

```

Соберите и снова запустите на выполнение данное приложение, чтобы проверить его в окончательном виде. Возможно, вы и не заметите особых отличий, но благодаря лишь нескольким дополнительным строкам кода нам удалось существенно сократить количество операций, требующихся для перерисовки представления, избавившись от необходимости стирать и перерисовывать любую часть представления, не затрагиваемую в настоящий момент движением пальца по экрану. Оптимизируя свои приложения под iOS, особенно по мере их усложнения, вы постепенно научитесь экономно расходовать вычислительные ресурсы своего мобильного устройства.

ЗАМЕЧАНИЕ. Если вас интересует более основательное изучение возможностей библиотеки Quartz 2D, обратите внимание на книгу Джека Наттинга (Jack Nutting), Дэйва Вудбриджа (Dave Wooldridge) и Дэвида Марка (David Mark) *Beginning iPad Development for iPhone Developers: Mastering the iPad SDK*, вышедшую в издательстве Apress в 2010 году. В этой книге подробно рассматриваются вопросы, связанные с рисованием средствами Quartz 2D, а весь приведенный в ней код и пояснения к нему применимы не только к устройству iPad, но и к iPhone.

Резюме

В этой главе мы рассмотрели лишь в самых общих чертах возможности рисования под управлением системы iOS. Теперь вы должны почувствовать себя увереннее в обращении с библиотекой Quartz 2D. Если вы будете периодически обращаться за справкой к предоставляемой компанией Apple документации к этой библиотеке, то, вероятно, сможете удовлетворить все свои насущные потребности в рисовании средствами Quartz 2D.

Теперь самое время еще больше усовершенствовать свои графические навыки. В главе 17 будет представлен каркас Sprite Kit, внедренный в версии iOS 7 и позволяющий очень быстро воспроизводить растровую графику в создаваемых играх или в другом динамичном интерактивном содержимом.

ГЛАВА 17



Введение в каркас Sprite Kit

В версии iOS 7 компания Apple внедрила каркас Sprite Kit для высокоеффективного воспроизведения двухмерной графики. В отличие от каркаса Core Graphics, предназначенного в основном для рисования графики по модели художника, или каркаса Core Animation, поддерживающего анимационные свойства элементов графического пользовательского интерфейса, каркас Sprite Kit служит совсем другой цели — разработке видеоигр. Это первая попытка компании Apple овладеть графической стороной программирования игр в эпоху iOS. С этой целью каркас Sprite Kit был выпущен одновременно для систем OS 7 и OS X 10.9 (Mavericks) с одинаковыми интерфейсами прикладного программирования для обеих платформ, упрощающими перенос разработанных приложений с одной платформы на другую. Несмотря на то что компания Apple никогда прежде не предоставляла каркас, подобный Sprite Kit, у него имеется немало общего с различными открытыми библиотеками наподобие Cocos2D. И если вам приходилось раньше пользоваться такими библиотеками, то работать с каркасом Sprite Kit вам будет очень комфортно.

В каркасе Sprite Kit не реализуется такая же удобная универсальная система рисования, как и в Core Graphics. В нем отсутствуют методы для рисования контуров, градиентов или заполнения пространств цветом. Вместо этого предоставляется **граф сцены**, аналогичный иерархии представлений в библиотеке UIKit; возможность преобразовывать положение, масштаб и вращение каждого узла этого графа, а также рисовать сам узел. Рисование происходит главным образом в экземпляре класса SKSprite (или одного из его подклассов), который представляет единое графическое изображение для вывода на экран.

В этой главе мы воспользуемся каркасом Sprite Kit для построения простой игры-“стрелялки” под названием TextShooter. Вместо того чтобы применять готовую графику, мы построим для этой игры графические объекты с фрагментами текста, используя специально предназначенный для этой цели подкласс, производный от класса SKSprite. При таком подходе не придется извлекать графику из библиотеки проекта или другого аналогичного источника.

Мы собираемся создать приложение, имеющее простой внешний вид, но легко поддающееся видоизменению.

СОЗДАНИЕ ПРИЛОЖЕНИЯ TEXTSHOOTER

Находясь в среде Xcode, нажмите комбинацию клавиш **<⌘+N>** или выберите сначала команду меню **File⇒New⇒Project...** и шаблон Game в разделе iOS. Щелкните на кнопке **Next**, присвойте новому проекту имя **TextShooter**, выберите пункт **Universal** в списке **Devices**, пункт **SpriteKit** в списке **Game Technology**, а затем создайте проект. Здесь уместно хотя бы вкратце рассмотреть и другие варианты выбора технологий. Так, варианты **OpenGL ES** и **Metal** (последний появился лишь в версии iOS 8) обозначают низкоуровневые графические интерфейсы прикладного программирования, предоставляющие практически полный контроль над имеющимися графическими аппаратными средствами, хотя пользоваться ими намного сложнее, чем **Sprite Kit**. Если **Sprite Kit** — это интерфейс прикладного программирования для написания приложений двухмерной графики, то **SceneKit** — это набор инструментальных средств (также появившийся только в версии iOS 8) для приложений трехмерной графики. Прочитав эту главу, обратитесь за справкой к документации по **SceneKit** по адресу https://developer.apple.com/library/prerelease/ios/documentation/SceneKit/Reference/SceneKit_Framework/index.html, если вас заинтересует программирование трехмерных игр.

Если теперь запустить проект **TextShooter** на выполнение, то на экране появится стандартное для **Sprite Kit** приложение, как показано на рис. 17.1. Первоначально оно выводит на экран традиционный текст "Hello, World". Если щелкнуть мышью или коснуться экрана пальцем, то на экране появятся еще несколько вращающихся реактивных самолетов. На протяжении всей этой главы мы постепенно заменим весь шаблон этого стандартного приложения, чтобы построить свое, хотя и простое приложение.

Проанализируем проект, только что созданный в среде Xcode. Он состоит из стандартного вида класса **AppDelegate** и небольшого класса контроллера представления **GameViewController**, выполняющего первоначальную настройку объекта типа **SKView**. Этот объект загружается из раскадровки приложения **Sprite Kit** и обозначает представление, предназначенное для отображения всего содержимого этого приложения. Ниже приведен исходный код метода **viewDidLoad()**, определенного в классе **GameViewController** и инициализирующего объект типа **SKView**.

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

    if let view = self.view as! SKView? {
        // Загрузка SKScene из файла 'GameScene.sks'
        if let scene = SKScene(fileNamed: "GameScene") {
            // Масштабирование по экрану
```

```
scene.scaleMode = .aspectFill
// Представляем сцену
view.presentScene(scene)
}

view.ignoresSiblingOrder = true

view.showsFPS = true
view.showsNodeCount = true
}
}
```

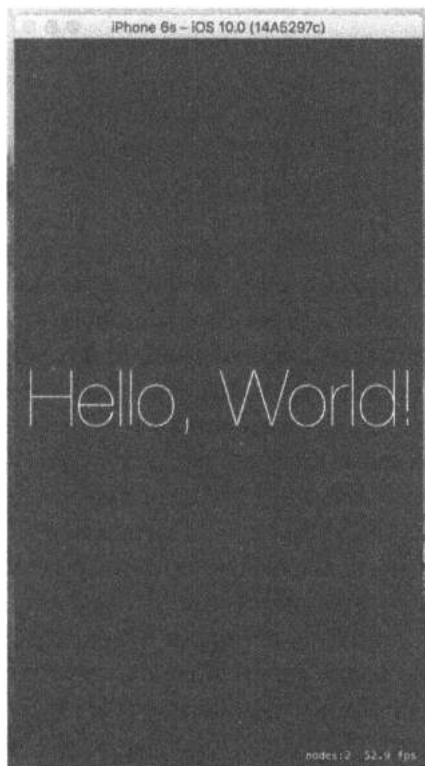


Рис. 17.1. Стандартное приложение Sprite Kit в действии с текстом в центре экрана

В первых строках кода данного метода из раскадровки получается экземпляр класса SKView, который настраивается на демонстрацию некоторых характеристик производительности во время выполнения игры. Приложения Sprite Kit строятся в виде ряда сцен, представленных классом SKScene. В процессе разработки приложения Sprite Kit для каждой визуально различимой его части, скорее всего, придется создать новый подкласс, производный от класса SKScene. Сцена может служить местом для анимации десятков объектов в ходе динамично развивающейся игры или простого отображения начального меню.

В этой главе будут продемонстрированы многочисленные примеры применения класса SKScene. По шаблону первоначально формируется пустая сцена в виде класса GameScene.

Во взаимосвязи классов SKView и SKScene прослеживается определенная параллель с классами типа UIViewController, применяемыми в примерах, представленных на протяжении всей этой книги. В частности, класс SKView действует аналогично классу UINavigationController в том отношении, что он служит своего рода чистой доской, просто управляющей доступом к отображению для других контроллеров. Однако на этом сходство и заканчивается. В отличие от класса UINavigationController, объекты верхнего уровня, управляемые классом SKView, являются производными не от класса UIViewController, а от класса SKScene, которому известно, как управлять графиком отображаемых объектов, воздействовать на них физическим механизмом и т.д.

Далее в методе viewDidLoad() создается первоначальная сцена:

```
if let scene = SKScene(fileNamed: "GameScene") {
```

Создать сцену можно двумя способами: выделить и инициализировать ее экземпляр вручную или загрузить его из **файла сцены** Sprite Kit. В шаблоне Xcode выбран второй способ. Для этого в нем формируется файл сцены Sprite Kit, называемый GameScene.sks и содержащий архивную копию объекта типа SKScene. Как и большинство других классов Sprite Kit, класс SKScene согласуется с протоколом NSCoder, обсуждавшимся в главе 13. Файл GameScene.sks служит стандартным архивом, который можно читать и записывать, используя классы NSKeyedUnarchiver и NSKeyedArchiver. Однако, как правило, применяется метод SKScene(fileNamed:), загружающий объект типа SKScene из архива и инициализирующий его как экземпляр конкретного подкласса, для которого он вызывается. В данном случае это заархивированные данные типа SKScene, используемые для инициализации объекта типа GameScene.

В связи с изложенным выше может возникнуть следующий вопрос: зачем в коде шаблона загружается объект пустой сцены из файла сцены, если его можно было бы просто создать? Все дело в конструкторе *Sprite Kit Level Designer*, встроенным в среду Xcode, который позволяет конструировать сцену подобно пользовательскому интерфейсу в программе Interface Builder. Сконструировав сцену, вы можете сохранить ее в файле сцены и снова запустить приложение на выполнение. На этот раз сцена, безусловно, не окажется пустой, но будет иметь то оформление, которое было создано вами в конструкторе Level Designer. Загрузив первоначальную сцену, вы вольны программно ввести в нее дополнительные элементы. И этим нам с вами придется еще не раз заняться в данной главе. С другой стороны, вы можете построить все свои сцены полностью в коде, если не найдете никакой для себя пользы в конструкторе Level Designer.

Если выбрать файл GameScene.sks в окне навигатора проекта, он откроется в конструкторе Level Designer, как показано на рис. 17.2.



Рис. 17.2. Конструктор Sprite Kit Level Designer в среде Xcode с первоначальной игровой сценой GameScene, содержащей метку

Сцена отображается в области редактора. В настоящий момент она пуста и обозначена желтой прямоугольной рамкой на сером фоне. Справа от сцены находится окно инспектора узлов SKNode, в котором устанавливаются свойства узла, выбранного в редакторе сцены. Все элементы сцены Sprite Kit являются узлами — экземплярами класса SKNode, а сам класс SKScene — производным от класса SKNode. В данном случае выбран узел типа SKScene, и поэтому в окне инспектора узлов SKNode отображаются его свойства. Ниже этого инспектора (в правом нижнем углу экрана) находится обычная для среды Xcode библиотека объектов, которая автоматически фильтруется для отображения только тех типов объектов, которые можно вводить в сцену Sprite Kit. Конструирование сцены осуществляется перетаскиванием объектов из этой библиотеки в область редактора.

Вернемся, однако, к методу `viewDidLoad()`, чтобы завершить его обсуждение:

```
/* Установка режима масштабирования с учетом размеров окна */
scene.scaleMode = .aspectFill
```

В результате присвоения свойству `ScaleMode` значения `.aspectFill` происходит масштабирование каждой размерности в зависимости от того, какая из них больше — высота или ширина. Существуют четыре режима масштабирования.

- SKSceneScaleMode.aspectFill. Размеры сцены изменяются таким образом, чтобы она заполнила весь экран, сохранив свои пропорции (отношение ширины и высоты). В этом режиме масштабирования каждый пиксель экрана охватывается представлением типа SKView, но при этом теряется часть сцены (в данном случае сцена обрезается слева и справа). Содержимое сцены также масштабируется, и поэтому текст меток становится мельче, чем в исходной сцене, хотя их местоположение относительно сцены сохраняется.
- SKSceneScaleMode.aspectFit. Пропорции сцены также сохраняются, но в то же время гарантируется, что она будет видна полностью. В итоге сцена приобретает вид установленного в черную рамку изображения, на котором выше и ниже содержимого сцены видны части представления типа SKView.
- SKSceneScaleMode.Fill. Масштаб сцены изменяется по обеим осям координат, чтобы она точно вписалась в представление. Этим гарантируется, что все содержимое сцены будет видно, но поскольку пропорции исходной сцены не сохраняются, то возможны искажения ее содержимого. В данном случае текст меток оказывается сжатым по горизонтали.
- SKSceneScaleMode.ResizeFill. Левый нижний угол сцены размещается в левом нижнем углу представления, не изменяя исходные размеры.

Выполняя следующую инструкцию, мы игнорируем отношения между родительской и дочерней сценами, которые выводятся на экран:

```
view.ignoresSiblingOrder = true
```

В зависимости от того, как именно вы играете, вы можете создавать стеки элементов. Следующая инструкция осуществляет анимацию изменений сцены:

```
view.presentScene(scene)
```

Вызов этого метода во время демонстрации сцены приводит к ее немедленной замене новой сценой. Позднее в этой главе мы продемонстрируем соответствующие примеры. В данном случае, поскольку мы визуализируем начальную сцену, никакого перехода не происходит, поэтому можно применить метод presentScene().

```
view.showsFPS = true
view.showsNodeCount = true
```

Последние две строки просто выводят в нижней части экрана определенную информацию об анимации (рис. 17.3).

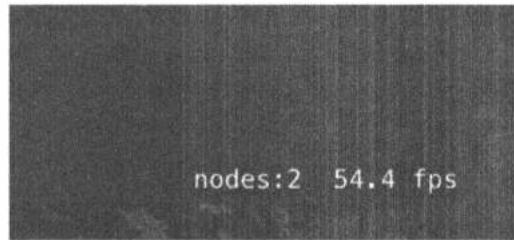


Рис. 17.3. Дополнительная информация об анимации, демонстрируемой в представлении

Первоначальная настройка сцены

Выберите класс GameScene. Большая часть исходного кода этого класса, сгенерированная в шаблоне Xcode автоматически, не потребуется, поэтому ее нужно удалить. Сначала удалите весь метод `didMoveToView()`. Этот метод вызывается всякий раз, когда сцена воспроизводится в представлении типа `SKView`. Он, как правило, вызывается для внесения в сцену изменений непосредственно перед тем, как она становится видимой. Затем удалите большую часть кода из тела метода `touchesBegan(_:_withEvent:)`, оставив лишь цикл и первую строку кода.

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    /* Этот метод вызывается в момент касания */
    for touch in touches {
        let location = touch.locationIn(in: self)
    }
}
```

Мы не собираемся загружать сцену из файла `GameScene.sks`, поэтому нам потребуется метод, автоматически создающий сцену с некоторым первоначальным содержимым. Кроме того, нам придется ввести свойства для хранения номера текущего уровня игры, количества жизней, имеющихся в распоряжении игрока, а также признака, обозначающего завершение игры на данном уровне. С этой целью введите в исходный файл `GameScene.swift` строки кода, приведенные в листинге 17.1.

Листинг 17.1. Первые модификации файла `GameScene.swift`

```
class GameScene: SKScene {

    private var levelNumber: Int
    private var playerLives: Int
    private var finished = false

    class func scene(size:CGSize, levelNumber:Int) -> GameScene {
        return GameScene(size: size, levelNumber: levelNumber)
    }
}
```

642 ГЛАВА 17 ■ ВВЕДЕНИЕ В KAPKAC SPRITE KIT

```
override convenience init(size:CGSize) {
    self.init(size: size, levelNumber: 1)
}

init(size:CGSize, levelNumber:Int) {
    self.levelNumber = levelNumber
    self.playerLives = 5
    super.init(size: size)

    backgroundColor = SKColor.lightGray()

    let lives = SKLabelNode(fontNamed: "Courier")
    lives.fontSize = 16
    lives.fontColor = SKColor.black()
    lives.name = "LivesLabel"
    lives.text = "Lives: \(playerLives)"
    lives.verticalAlignmentMode = .top
    lives.horizontalAlignmentMode = .right
    lives.position = CGPoint(x: frame.size.width,
                             y: frame.size.height)
    addChild(lives)

    let level = SKLabelNode(fontNamed: "Courier")
    level.fontSize = 16
    level.fontColor = SKColor.black()
    level.name = "LevelLabel"
    level.text = "Level \(levelNumber)"
    level.verticalAlignmentMode = .top
    level.horizontalAlignmentMode = .left
    level.position = CGPoint(x: 0, y: frame.height)
    addChild(level)
}

required init?(coder aDecoder: NSCoder) {
    levelNumber = aDecoder.decodeInteger(forKey: "level")
    playerLives = aDecoder.decodeInteger(forKey: "playerLives")
    super.init(coder: aDecoder)
}

override func encode(with aCoder: NSCoder) {
    aCoder.encode(Int(levelNumber), forKey: "level")
    aCoder.encode(playerLives, forKey: "playerLives")
}
}
```

Первый метод, `scene(size:levelNumber:)`, является фабричным. Он поможет нам кратчайшим путем для создания уровня игры и установки его номера. Во втором методе, `init()`, мы переопределяем используемый по умолчанию инициализатор класса, передавая управление третьему методу вместе с устанавливаемым по умолчанию значением номера уровня. В свою очередь, третий метод вызывает назначенный инициализатор из реализации суперкласса после установки исходных значений в свойствах `levelNumber` и `playerLives`. На первый взгляд, такой путь кажется окольным, но на самом деле он является

общепринятым, когда требуется ввести новые инициализаторы в класс и в то же время пользоваться назначенным для него инициализатором. После вызова инициализатора из суперкласса мы задаем цвет фона сцены. Обратите внимание на использование с этой целью класса `SKColor` вместо класса `UIColor`. На самом деле `SKColor` — это вообще не класс, а лишь псевдоним типа, преобразуемый в класс `UIColor` для приложений системы iOS и в класс `NSColor` для приложений системы OS X. Благодаря этому немного упрощается перенос игр между платформами iOS и OS X.

После этого мы создаем два экземпляра класса `SKLabelNode`. Этот удобный класс действует аналогично классу `UILabel`, позволяя ввести немного текста в сцену, выбрать для него шрифт, задать конкретное текстовое значение и выравнивание. В частности, мы создаем одну метку для отображения количества жизней в правом верхнем углу экрана, а другую — для номера уровня в левом верхнем углу экрана. Обратите особое внимание на код, предназначенный для расположения этих меток. Так, в следующей строке кода устанавливается положение метки для обозначения жизней:

```
lives.position = CGPointMake(x: frame.size.width, y: frame.size.height);
```

Если вы считаете, что для указания положения данной метки следует передать координаты отдельных точек, то вас, возможно, удивит, что вместо них передается высота сцены. Если в UIKit расположение объекта по высоте представления типа `UIView` приводит к тому, что он оказывается в нижней части этого представления, то в SceneKit ось у перевернута, т.е. начало координат находится в левом нижнем углу экрана, а ось у направлена вверх. В итоге максимальное значение высоты сцены соответствует положению на ее верхнем краю. А что же координата x расположения метки? Она задается по ширине сцены. Если сделать это в представлении типа `UIView`, то метка оказалась бы за правым краем экрана. Однако в данном случае этого не происходит благодаря следующей строке кода:

```
lives.horizontalAlignmentMode = .right;
```

В этой строке кода устанавливается режим выравнивания `SKLabelNode.horizontalAlignmentMode.right` в свойстве `horizontalAlignmentMode` из класса `SKLabelNode`. Благодаря этому точка узла метки, используемая для ее расположения (а на самом деле свойство `position`), перемещается вправо от текста. Поскольку нам требуется выровнять текст по правому краю экрана, мы должны установить в свойстве `position` ширину сцены в качестве координаты x расположения метки. С другой стороны, текст метки, обозначающей уровень игры, выравнивается по левому краю экрана и располагается на левом краю сцены, и для этого его координата x задается равной нулю, как показано ниже.

```
level.horizontalAlignmentMode = .left;
level.position = CGPointMake(x: 0, y: frame.size.height);
```

Нам нужно также присвоить имя каждой метке. Оно действует подобно декріптору или идентификатору в других частях каркаса UIKit. Это даст нам в дальнейшем возможность извлекать метки по их имени.

В рассматриваемый здесь класс были введены также методы `init(coder:)` и `encode(with aCoder:)`, поскольку все узлы Sprite Kit, включая класс `SKScene`, соответствуют протоколу `NSCoding`. Для этого требуется переопределить метод `init(coder:)`, и поэтому для согласованности мы реализуем также метод `encode(with aCoder:)`, несмотря на то что в данном приложении архивирование объекта сцены не предполагается. По такому же образцу будут построены и все остальные подклассы, производные от класса `SKNode`, хотя мы не будем реализовывать метод `encode(with aCoder:)`, если у самого подкласса отсутствует дополнительное состояние, поскольку все действия, необходимые в данном случае, выполняются в базовой версии класса.

Теперь выберите исходный файл `GameViewController.swift` и внесите в метод `viewDidLoad()` следующие изменения:

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

    let scene = GameScene(size: view.frame.size, levelNumber: 1)

    // Конфигурирование представления.
    let skView = self.view as! SKView
    skView.showsFPS = true
    skView.showsNodeCount = true

    /* Sprite Kit выполняет дополнительную оптимизацию,
       чтобы повысить производительность рендеринга */
    skView.ignoresSiblingOrder = true

    /* Выбор режима масштабирования по экрану */
    scene.scaleMode = .aspectFill

    skView.presentScene(scene)
}
```

Вместо загрузки сцены из файла вызывается метод `scene(size: levelNumber:)`, только что введенный в класс `GameScene`, чтобы создать сцену, инициализировать ее и задать ее размеры такими же, как и размеры представления типа `SKView`. Благодаря тому что сцена и представление имеют одинаковые размеры, устанавливать свойство `scaleMode` для сцены больше не нужно, и поэтому из метода `viewDidLoad()` можно удалить строку кода, в которой это делается. Ближе к концу исходного файла `GameViewController.swift` находится следующий метод:

```
override func prefersStatusBarHidden() -> Bool {
    return true
}
```

В результате вызова этого метода строка состояния iOS исчезает во время игры. Исходный код этого метода входит в шаблон Xcode потому, что в играх-

боевиках, подобных данной, строка состояния обычно скрывается. Теперь запустите игру на выполнение, чтобы увидеть расположение самой элементарной ее структуры (рис. 17.4).

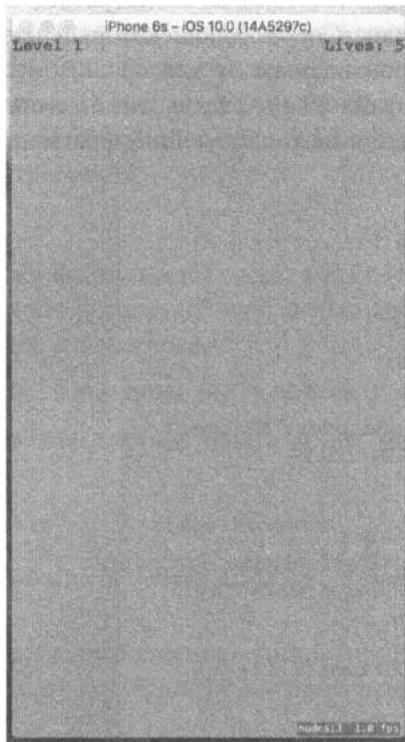


Рис. 17.4. Наша игра пока что мало привлекательна, но по крайней мере у нее высокая частота кадров!

ПОДСКАЗКА. Подсчет узлов и частота кадров в правой нижней части сцены удобны для целей отладки, но в окончательном варианте игры они не нужны. Их можно исключить, установив логическое значение `false` свойств `showsFPS` и `showsNodeCount` из класса `SKView` в методе `viewDidLoad` из класса `GameViewController`. В классе `SKView` имеются и другие свойства для получения дополнительной отладочной информации. Подробнее о них можно узнать из документации на соответствующий интерфейс прикладного программирования.

Движение игрока

Настало время придать игре немного интерактивности. С этой целью мы создадим класс, представляющий игрока. Этот класс предназначен для рисования метки игрока в сцене с помощью внутренних компонентов и ее перемещения в новое положение изящным анимационным способом. Экземпляр этого

нового класса мы вставим в сцену и затем напишем код, позволяющий игроку перемещать объект по сцене, касаясь пальцем экрана. Всякий объект, который предполагается сделать частью сцены, должен относиться к подклассу, производному от класса `SKNode`. Поэтому воспользуйтесь меню `File` в среде Xcode, чтобы создать новый класс `Cocoa Touch`, который называется `PlayerNode` и является подклассом, производным от класса `SKNode`. В созданный при этом почти пустой исходный файл `PlayerNode.swift` импортируйте каркасы `Sprite Kit` и `Foundation` и введите в него следующий фрагмент кода:

```
import SpriteKit

class PlayerNode: SKNode {
    override init() {
        super.init()
        name = "Player \(self)"
        initNodeGraph()
    }

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

    private func initNodeGraph() {
        let label = SKLabelNode(fontNamed: "Courier")
        label.fontColor = SKColor.darkGray()
        label.fontSize = 40
        label.text = "v"
        label.zRotation = CGFloat(M_PI)
        label.name = "label"
        self.addChild(label)
    }
}
```

Узел игрока `PlayerNode` сам ничего не отображает, поскольку у простого узла типа `SKNode` отсутствуют необходимые для этого средства. Вместо этого в методе `init()` устанавливается подузел, в котором фактически будет выполняться рисование. Этот подузел является экземпляром класса `SKLabelNode` аналогично узлу, созданному для отображения номера уровня игры и оставшегося количества жизней. Класс `SKLabelNode` является подклассом, производным от класса `SKNode` и наделенным нужными средствами для рисования. Другим аналогичным подклассом является класс `SKSpriteNode`. В данном случае положение метки не устанавливается, а следовательно, ее положение оказывается в точке с координатами $(0, 0)$. Подобно представлениям, каждый узел типа `SKNode` существует в системе координат, наследуемой от его родительского объекта. Установка этого узла в нулевое положение означает, что на экране он появится на месте экземпляра класса `PlayerNode`. Любые ненулевые значения будут, по существу, означать смещение относительно данной точки.

Кроме того, мы устанавливаем величину вращения для метки, чтобы перевернуть строчную букву `v`, которую она содержит. Имя `zRotation` свойства

вращения может показаться не совсем уместным, но на самом деле оно обозначает ось z в пространстве координат Sprite Kit. Отображение на экране осуществляется по осям x и y, а ось z служит как для упорядочения элементов сцены в целях их отображения, так и для их вращения. Значения, присваиваемые свойству zRotation, должны быть указаны в радианах, а не в градусах, и поэтому данному свойству присваивается значение константы M_PI, равнозначное математическому значению числа π (именно для доступа к этой константе и был импортирован каркас Foundation). Поскольку π радиан равно 180°, то это именно то, что нам требуется.

Ввод игрока в сцену

Вернитесь к исходному файлу GameScene.swift. Нам предстоит ввести экземпляр класса PlayerNode в сцену. С этой целью введите сначала следующее свойство, представляющее узел игрока.

```
private let playerNode: PlayerNode = PlayerNode()
```

Далее введите следующие строки кода в конце метода init(size:levelNumber:).

```
level.position = CGPoint(x: 0, y: frame.height)
addChild(level)
playerNode.position = CGPoint(x: frame.midX, y: frame.height * 0.1)
addChild(playerNode)
```

Построив и запустив на выполнение данное приложение, вы обнаружите, что метка игрока появляется посередине у нижнего края экрана, как показано на рис. 17.5.

Обработка касаний для перемещения игрока

Далее нам предстоит добавить немного логики в метод touchesBegan(_ :withEvent:), оставленный ранее пустым. С этой целью введите приведенные ниже строки кода в исходный файл GameScene.swift. (После ввода этих строк кода компилятор выдаст ошибку, но мы вскоре устраним ее.)

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    /* Вызывается в момент касания */

    for touch in touches {
        let location = touch.location(in: self)
        if location.y < frame.height * 0.2 {
            let target = CGPoint(x: location.x, y: playerNode.position.y)
            playerNode.moveToward(target)
        }
    }
}
```

В приведенном выше фрагменте кода местоположение любого касания в нижней пятой части экрана служит основанием для выбора нового местоположения, к которому требуется переместить узел игрока в сцене. Оно также дает

узлу игрока команду переместиться к нему. Компилятор выдает ошибку потому, что мы еще не определили метод moveToward() для перемещения узла игрока. Поэтому перейдите к исходному файлу `PlayerNode.swift` и введите в нем следующий код, реализующий данный метод:

```
func moveToward(_ location: CGPoint) {
    removeAction(forKey: "movement")

    let distance = pointDistance(position, location)
    let screenWidth = UIScreen.main.bounds.size.width
    let duration = TimeInterval(2 * distance/screenWidth)

    run(SKAction.move(to: location, duration: duration),
         forKey:"movement")
}
```

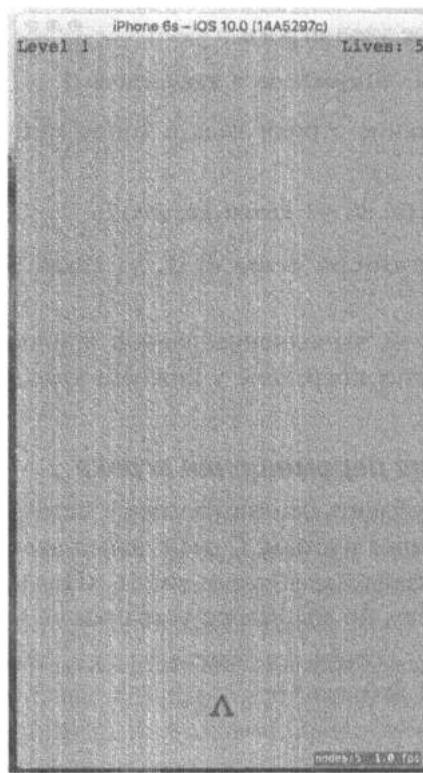


Рис. 17.5. Метка игрока в виде перевернутой буквы V в нижней части экрана

Не будем пока что обращать внимание на первую строку в приведенном выше фрагменте кода, чтобы вскоре вернуться к ней. В данном методе новое местоположение сравнивается с текущим и определяются расстояние и количество пикселей для перемещения. Далее с помощью числовой константы, задающей скорость движения вообще, определяется время, необходимое для пе-

ремещения. В заключение создается действие типа `SKAction`, выполняющее перемещение узла игрока. Класс `SKAction` входит в состав каркаса Sprite Kit и предназначен для внесения изменений в узлы сцены с течением времени, что дает возможность легко осуществить анимацию изменения положения, размера, прозрачности, вращения и прочих свойств узла. В данном случае узлу игрока сначала дается команда выполнить простое перемещающее действие в течение определенного промежутка времени, а затем этому действию присваивается ключ "movement". Как видите, это такой же ключ, как и в первой строке кода в теле данного метода, где действие удаляется. В начале этого метода по тому же самому ключу удаляется любое существующее действие, чтобы пользователь мог быстро нажать пальцем в нескольких подряд местах экрана, не порождая много лишних действий при попытке перемещаться по экрану как-то иначе!

Геометрические вычисления

Нетрудно заметить, что после ввода упомянутого выше кода у нас возникло затруднение в связи с тем, что в среде Xcode не удается найти ни одной функции, называемой `pointDistance()`. Это одна из ряда простых геометрических функций, применяемых в данном приложении для выполнения расчетов по точкам и векторам в формате чисел с плавающей точкой. Для того чтобы поставить все на свои места, создайте в среде Xcode новый исходный файл Swift, присвойте ему имя `Geometry.swift`, а затем введите в него следующие строки кода.

```
import UIKit
// Получает объекты классов CGVector и CGFloat.
// Возвращает новый объект класса CGFloat, в котором каждый компонент
// объекта умножен на m.
func vectorMultiply(_ v: CGVector, _ m: CGFloat) -> CGVector {
    return CGVector(dx: v.dx * m, dy: v.dy * m)
}

// Получает два объекта класса CGPoints.
// Возвращает объект класса CGVector, представляющий направление от p1 к p2.
func vectorBetweenPoints(_ p1: CGPoint, _ p2: CGPoint) -> CGVector {
    return CGVector(dx: p2.x - p1.x, dy: p2.y - p1.y)
}

// Получает объект класса CGVector.
// Возвращает объект класса CGFloat, содержащий длину вектора,
// вычисленное по теореме Пифагора.
func vectorLength(_ v: CGVector) -> CGFloat {
    return CGFloat(sqrtf(powf(Float(v.dx), 2) + powf(Float(v.dy), 2)))
}

// Получает два объекта класса CGPoints.
// Возвращает объект класса CGFloat, содержащий расстояние между ними,
// вычисленную по теореме Пифагора.
func pointDistance(_ p1: CGPoint, _ p2: CGPoint) -> CGFloat {
    return CGFloat(sqrtf(powf(Float(p2.x - p1.x), 2) + powf(Float(p2.y - p1.y), 2)))
}
```

Приведенные выше функции служат простыми реализациями некоторых типичных операций, которые оказываются полезными во многих играх: умножение векторов, создание векторов, указывающих в направлении от одной точки к другой, а также вычисление расстояний. Снова постройте и выполните данное приложение. Как только появится метка, обозначающая самолет пользователя, нажмите пальцем в любом месте нижней части экрана, чтобы посмотреть, как самолет скользит слева направо, достигая точки, на которую вы нажали пальцем. Можете нажать пальцем еще раз, прежде чем самолет достигнет точки своего назначения, и тотчас начнется новая анимация перемещения самолета в новое место. Это, конечно, замечательно, но было бы лучше, если бы самолет пользователя двигался живее.

Покачивание

Придадим эффект незначительного покачивания движению самолета, добавив еще одну анимацию. С этой целью введите выделенные ниже полужирным строки кода в метод moveToward: из класса PlayerNode.

```
func moveToward(_ location: CGPoint) {
    removeAction(forKey: "movement")
    removeAction(forKey: "wobbling")

    let distance = pointDistance(position, location)
    let screenWidth = UIScreen.main.bounds.size.width
    let duration = TimeInterval(2 * distance/screenWidth)

    run(SKAction.move(to: location, duration: duration),
        forKey:"movement")

    let wobbleTime = 0.3
    let halfWobbleTime = wobbleTime/2
    let wobbling = SKAction.sequence([
        SKAction.scaleX(to: 0.2, duration: halfWobbleTime),
        SKAction.scaleX(to: 1.0, duration: halfWobbleTime)
    ])
    let wobbleCount = Int(duration/wobbleTime)
    run(SKAction.repeat(wobbling, count: wobbleCount),
        forKey:"wobbling")
}
```

Итак, мы создали действие, аналогичное определенному ранее перемещающему действию, но у него имеются некоторые важные отличия. Для анимации элементарного перемещения мы сначала рассчитали его продолжительность, а затем создали и сразу же выполнили действие. На этот раз дело обстоит несколько сложнее. Сначала мы определяем время для одного покачивания (самолет должен покачиваться во время перемещения неоднократно, но с постоянной частотой). Анимация самого покачивания осуществляется следующим образом: сначала выполняется масштабирование самолета по оси x (т.е. по его ширине) на 2/10 его обычного размера, а затем обратное масштабирование до полного размера. Каждая из этих операций составляет отдельное действие, образующее

вместе с другим действием своего рода *последовательность*, в которой все действия выполняются одно за другим. Далее мы выясняем, сколько раз должно происходить такое покачивание во время перемещения самолета, и заключаем последовательность покачивающих действий в повторяющееся действие, сообщая ему количество выполняемых полностью циклов покачивания. Как и прежде, рассматриваемый здесь метод начинается с удаления предыдущего покачивающего движения, чтобы исключить лишние покачивания.

Снова постройте и запустите данное приложение на выполнение, чтобы посмотреть, как самолет приятно покачивается во время перемещения. Это движение похоже на шаткую походку.

Создание противников

Все это, конечно, хорошо, но в данной игре должно быть хотя бы немного противников, в которых могли бы стрелять игроки. С этой целью создайте в среде Xcode новый класс Cocoa Touch, назвав его `EnemyNode` и воспользовавшись в качестве его родителя классом `SKNode`. Не будем пока что сообщать этому классу никакого настоящего поведения, а только придадим ему подходящий внешний вид противника. Прибегнем к той же самой методике, что и для игрока, используя текст для построения тела противника. Без сомнения, лучшего текстового символа для обозначения противника, чем строчная буква `x`, трудно подыскать. Итак, добавьте в исходный файл `EnemyNode.swift` следующий код.

```
import SpriteKit

class EnemyNode: SKNode {
    override init() {
        super.init()
        name = "Enemy \(self)"
        initNodeGraph()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
    private func initNodeGraph() {
        let topRow = SKLabelNode(fontNamed: "Courier-Bold")
        topRow.fontColor = SKColor.brown()
        topRow.fontSize = 20
        topRow.text = "x x"
        topRow.position = CGPoint(x: 0, y: 15)
        addChild(topRow)

        let middleRow = SKLabelNode(fontNamed: "Courier-Bold")
        middleRow.fontColor = SKColor.brown()
        middleRow.fontSize = 20
        middleRow.text = "x"
        addChild(middleRow)

        let bottomRow = SKLabelNode(fontNamed: "Courier-Bold")
        bottomRow.fontColor = SKColor.brown()
```

```

        bottomRow.fontSize = 20
        bottomRow.text = "x x"
        bottomRow.position = CGPoint(x: 0, y: -15)
        addChild(bottomRow)
    }
}

```

В этих строках кода нет ничего особенно нового. Мы просто вводим текст, смещающий значение по оси у для его расположения рядами.

Размещение противников на сцене

Теперь организуем появление немногих противников на сцене, внеся некоторые изменения в исходный файл GameScene.swift. Сначала введите следующее новое свойство для хранения количества противников, добавляемых на данном уровне.

```
private let enemies = SKNode()
```

На первый взгляд, для этой цели можно было бы использовать класс `Array<SKNode>`, но в данном случае идеально подходит обычный узел типа `SKNode`, поскольку в нем можно хранить любое количество подчиненных узлов. Нам все равно придется ввести в сцену всех противников, поэтому мы можем хранить их в одном узле типа `SKNode`, чтобы упростить доступ к ним. Создайте метод `spawnEnemies()`, как показано ниже.

```

private func spawnEnemies() {
    let count = Int(log(Float(levelNumber))) + levelNumber
    for _ in 0..

```

Ведите выделенные ниже полужирным строки кода в конце метода `init(size:levelNumber:)`, чтобы сначала добавить узел противников в сцену, а затем вызвать метод `spawnEnemies()`.

```

addChild(playerNode)

addChild(enemies)
spawnEnemies()

```

Итак, мы ввели узел объекта `enemies` в сцену, и поэтому любые вводимые далее порожденные от него узлы объектов `enemies` будут также появляться на сцене. Обратите внимание на то, что для получения случайных координат для узлов объекта `enemies` `x` и `y` мы используем функцию `arc4random_uniform()`.

Для того чтобы повысить степень случайности этих чисел, необходимо правиль- но настроить генератор случайных чисел. Откройте файл AppDelegate.swift и добавьте в него следующую строку, выделенную полужирным шрифтом в ме- тоде application(_: didFinishLaunchingWithOptions:).

```
func application(application: UIApplication,
didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Точка замещения для настройки после запуска приложения.
arc4random_stir()
    return true
}
```

Теперь постройте и запустите данное приложение на выполнение, чтобы посмотреть, как грозные противники произвольно размещаются в верхней части экрана (рис. 17.6). Не возникает ли у вас желание немножко пострелять?

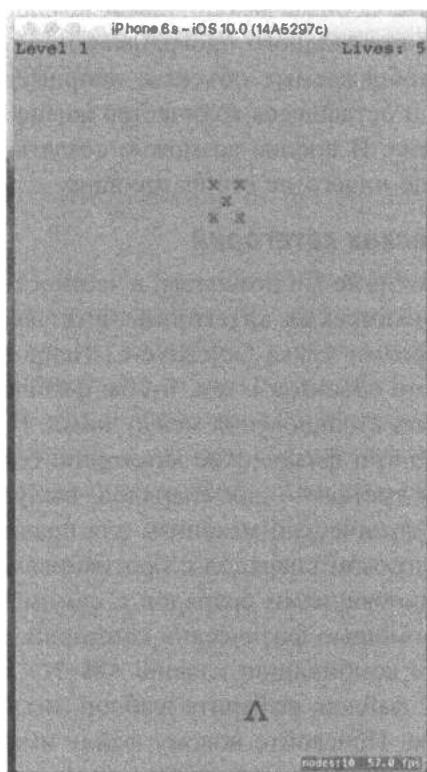


Рис. 17.6. Согласитесь, что в противников, обоз- наченных буквами X, следовало бы пострелять

Начало стрельбы

Настало время реализовать следующую, вполне логичную стадию в процессе разработки рассматриваемой здесь игры: предоставление игроку возможности

атаковать противников. В частности, игроку должна быть предоставлена возможность нажать пальцем на 80% площади верхней половины экрана, чтобы выпустить снаряд в противника. Для этого мы воспользуемся *физическими механизмами*, входящим в состав Sprite Kit, как для перемещения снаряда, выпущенного игроком, так и для определения момента ее попадания в противника.

Однако сначала нужно хотя бы вкратце пояснить, что собой представляет физический механизм. По существу, это программный компонент, отслеживающий во внешнем мире несколько физических объектов, обычно называемых *телами*, наряду с силами, которые на них действуют. Он также обеспечивает реалистичность движения объектов, принимая во внимание силу тяжести, столкновения объектов, чтобы они не занимали одновременно одно и то же место, и даже имитируя такие физические свойства, как трение и упругость. Следует иметь в виду, что физический механизм, как правило, отделен от графического механизма. Для работы с обоими механизмами компания Apple предоставляет удобные интерфейсы прикладного программирования, но они, по существу, разделены. Зачастую отображаемые объекты, например метки, обозначающие номер текущего уровня и оставшееся количество жизней, совершенно отделены от физического механизма. И вполне возможно создать объекты, имеющие физическое тело, но вообще ничего не отображающие.

Определение физических категорий

Физический механизм Sprite Kit позволяет, в частности, присваивать объектам несколько отдельных **физических категорий**. Физическая категория не имеет ничего общего с категориями языка Objective-C. Напротив, это способ группирования связанных вместе объектов с тем, чтобы физический механизм мог по-разному интерпретировать столкновения между ними. В рассматриваемом здесь примере игры создаются три физические категории: одна — для противников, другая — для игрока и третья — для снарядов, выпускаемых игроком. Нам, определенно, требуется физический механизм для правильной имитации столкновений выпускаемых игроком снарядов с противниками, но мы, скорее всего, можем пренебречь столкновениями снарядов с самим игроком. И все это не-трудно организовать с помощью физических категорий. Итак, создадим нужные нам категории. Нажмите комбинацию клавиш **<⌘+N>**, чтобы открыть помощник для создания новых файлов, выберите шаблон Swift File в разделе iOS и щелкните на кнопке Next. Присвойте новому файлу имя PhysicsCategories.swift и сохраните его, а затем введите в него следующий фрагмент кода:

```
import Foundation

let PlayerCategory: UInt32 = 1 << 1
let EnemyCategory: UInt32 = 1 << 2
let PlayerMissileCategory: UInt32 = 1 << 3
```

В этом фрагменте кода объявляются три константы категорий. Следует иметь в виду, что категории действуют как поразрядные маски, а следовательно, значение

каждой из них должно быть кратно степени двух. Этого нетрудно добиться по-разрядным сдвигом влево. Эти категории организованы в виде поразрядных масок, чтобы немного упростить интерфейс прикладного программирования физического механизма. С помощью поразрядных масок можно объединять несколько значений по логической операции OR. Это дает возможность использовать единый интерфейс прикладного программирования, чтобы сообщать физическому механизму, каким образом он должен интерпретировать столкновения на разных уровнях. И ниже будет показано, как это происходит на практике.

Создание класса BulletNode

Итак, заложив прочное основание для организации стрельбы в данной игре, создадим немного снарядов, чтобы начать стрельбу. С этой целью создайте новый класс Cocoa Touch, назвав его BulletNode и снова сделав его производным от класса SKNode. Сначала импортируйте каркас SpriteKit, а затем введите в этот класс свойство для хранения вектора нанесения удара снарядом, выделенное ниже полужирным шрифтом.

```
import SpriteKit

class BulletNode: SKNode {
    var thrust: CGVector = CGVectorMake(dx: 0, dy: 0)
}
```

Далее реализуем метод init(). Как и в остальных методах init() данного приложения, в нем создается граф объекта для снаряда. Он будет состоять из единственной точки. Раз уж мы находимся в этом классе, установим для него физическое тело, создав и настроив экземпляр класса SKPhysicsBody и присоединив его к текущему объекту по ссылке self. Между тем мы сообщаем новому физическому телу категорию, к которой оно относится, а также те категории, которые следует проверять на столкновения с данным объектом.

```
override init() {
    super.init()

    let dot = SKLabelNode(fontNamed: "Courier")
    dot.fontColor = SKColor.black()
    dot.fontSize = 40
    dot.text = "."
    addChild(dot)

    let body = SKPhysicsBody(circleOfRadius: 1)
    body.isDynamic = true
    body.categoryBitMask = PlayerMissileCategory
    body.contactTestBitMask = EnemyCategory
    body.collisionBitMask = EnemyCategory
    body.fieldBitMask = GravityFieldCategory
    body.mass = 0.01

    physicsBody = body
    name = "Bullet \(self)"
}
```

Добавим также методы `init(coder aDecoder:)` и `encode(with aCoder:)`.

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    let dx = aDecoder.decodeFloat(forKey: "thrustX")
    let dy = aDecoder.decodeFloat(forKey: "thrustY")
    thrust = CGVector(dx: CGFloat(dx), dy: CGFloat(dy))
}

override func encode(with aCoder: NSCoder) {
    super.encode(with: aCoder)
    aCoder.encode(Float(thrust.dx), forKey: "thrustX")
    aCoder.encode(Float(thrust.dy), forKey: "thrustY")
}
```

Применение законов физики

Далее напишем фабричный метод, создающий новый снаряд и сообщающий ему вектор нанесения удара, как показано ниже. Этот вектор будет использоваться физическим механизмом для выпуска снаряда в сторону цели.

```
class func bullet(from start: CGPoint,
                  toward destination: CGPoint) -> BulletNode {
    let bullet = BulletNode()
    bullet.position = start
    let movement = vectorBetweenPoints(start, destination)
    let magnitude = vectorLength(movement)
    let scaledMovement = vectorMultiply(movement, 1/magnitude)
    let thrustMagnitude = CGFloat(100.0)
    bullet.thrust = vectorMultiply(scaledMovement, thrustMagnitude)
    bullet.run(SKAction.playSoundFileNamed("shoot.wav",
                                           waitForCompletion: false))
    return bullet
}
```

Основные расчеты в этом методе довольно просты. Сначала определяется вектор движения, направленный из начального местоположения игрока к цели, а затем определяется его величина (т.е. длина). В результате деления вектора движения на его длину получается нормализованный **единичный вектор**, указывающий в том же самом направлении, что и исходный вектор, но имеющий единичную длину, которая в данном случае соответствует точке на экране, например двум пикселям на экране типа Retina или одному пикセルю на более старых экранах. Польза от создания единичного вектора состоит в том, что для определения вектора нанесения удара, одинаково действующего независимо от удаления места на экране, на котором пользователь нажал пальцем, достаточно умножить единичный вектор на фиксированную величину (в данном случае — 100). В заключение в данный класс следует ввести приведенный ниже метод, применявший силу удара к физическому телу. Этот метод будет вызываться в сцене один раз на каждый кадр.

```
func applyRecurringForce() {
    physicsBody!.applyForce(thrust)
}
```

Ввод снарядов в сцену

Перейдите к исходному файлу GameScene.swift, чтобы ввести снаряды в сцену. Прежде всего введите в этом файле очередное свойство для хранения всех снарядов в одном узле типа SKNode аналогично противникам, как показано ниже.

```
private let playerBullets = SKNode()
```

Найдите в теле метода init(size:levelNumber:) ту часть, где ранее были введены противники. Именно в этом месте устанавливается узел playerBullets, как выделено ниже полужирным шрифтом.

```
addChild(enemies)
spawnEnemies()

addChild(playerBullets)
}
```

Теперь можно приступить к программированию пусков снарядов. Введите выделенный ниже условный оператор else в тело метода touchesBegan(_:withEvent:), чтобы всякий раз, когда пользователь нажимает пальцем на любом месте в верхней части экрана, воспроизводился выстрел снаряда, а не движение самолета.

```
} else {
    let bullet = BulletNode.bullet(from: playerNode.position,
                                    toward: location)
    playerBullets.addChild(bullet)
}
```

Таким образом, снаряд вводится в сцену, но ни один из вводимых в нее снарядов на самом деле не движется, если приложить к нему силу удара в каждом кадре. В рассматриваемом здесь классе игровой сцены уже имеется пустой метод update(). Этот метод вызывается в каждом кадре и служит идеальным местом для ввода игровой логики, требующейся для воспроизведения в сцене того, что должно происходить в каждом кадре. Однако вместо обновления всех снарядов непосредственно в данном методе код игровой логики лучше разместить в отдельном методе, вызываемом из метода update().

```
override func update(_ currentTime: TimeInterval) {
    /* Вызывается перед прорисовкой каждого кадра */
    updateBullets()
}

private func updateBullets() {
    var bulletsToRemove: [BulletNode] = []
    for bullet in playerBullets.children as! [BulletNode] {
        // Удаляем все снаряды, вышедшие за пределы экрана
        if !frame.contains(bullet.position) {
            // Помечаем снаряды для удаления
            bulletsToRemove.append(bullet)
            continue
        }
    }
}
```

658 ГЛАВА 17 ■ ВВЕДЕНИЕ В KAPKAC SPRITE KIT

```
// Применить силу удара к остальным снарядам  
bullet.applyRecurringForce()  
}  
  
playerBullets.removeChildren(in: bulletsToRemove)  
}
```

Прежде чем сообщать каждому снаряду повторяющуюся силу удара, следует также проверить, не оказался ли снаряд за пределами экрана. Любой снаряд, оказавшийся за пределами экрана, размещается во временном массиве, а в конце все подобные снаряды удаляются из узла playerBullets. Следует заметить, что такой двухэтапный процесс требуется потому, что в цикле `for`, организованном в данном методе, перебираются все узлы, порожденные от узла `playerBullets`. Внесение изменений в коллекцию, когда перебираются ее элементы, не только не целесообразно, но и может легко привести к аварийному сбою. Снова постройте и запустите данное приложение на выполнение. Теперь вы можете не только перемещать самолет игрока, но и выпускать снаряды, нажимая пальцем на экран, как показано на рис. 17.7.

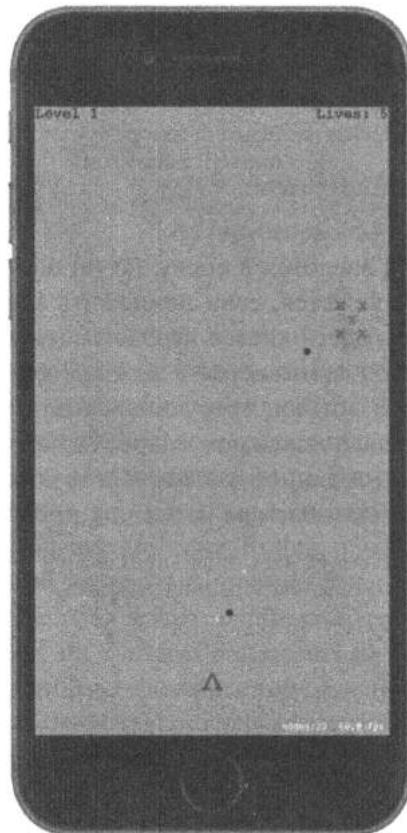


Рис. 17.7. Стрельба

Атака на противников с соблюдением законов физики

Рассматриваемой здесь игре недостает еще двух игровых элементов: противники не атакуют игрока, а он не может избавиться от противников, стреляя в них. Восполним теперь этот последний пробел в игре. Нам предстоит устроить ход игры таким образом, чтобы меткая стрельба по противнику приводила к его выбиванию из той позиции, где он находится в настоящий момент на экране. Для этого придется воспользоваться физическим механизмом, чтобы учесть силу тяжести, внеся соответствующие изменения в классы `PlayerNode`, `EnemyNode` и `GameScene`.

Прежде всего, нужно снабдить физическими телами те узлы сцены, которые их еще не имеют. Начните с узла из файла `EnemyNode.swift`, введя следующую строку кода в конце метода `init()`.

```
initPhysicsBody()
```

Затем введите приведенный ниже фрагмент кода, чтобы сделать тела действительно физическими. Это делается таким же образом, как и в классе `PlayerBullet`.

```
private func initPhysicsBody() {
    let body = SKPhysicsBody(rectangleOf: CGSize(width: 40, height: 40))
    body.affectedByGravity = false
    body.categoryBitMask = EnemyCategory
    body.contactTestBitMask = PlayerCategory | EnemyCategory
    body.mass = 0.2
    body.angularDamping = 0
    body.linearDamping = 0
    body.fieldBitMask = 0
    physicsBody = body
}
```

Далее выберите исходный файл `PlayerNode.swift`, в который предстоит внести аналогичные изменения. Прежде всего, введите следующую строку кода в конце метода `init()`.

```
initPhysicsBody()
```

В заключение введите новый метод `initPhysicsBody()` следующим образом:

```
private func initPhysicsBody() {
    let body = SKPhysicsBody(rectangleOf: CGSize(width: 20, height: 20))
    body.affectedByGravity = false
    body.categoryBitMask = PlayerCategory
    body.contactTestBitMask = EnemyCategory
    body.collisionBitMask = 0
    body.fieldBitMask = 0
    physicsBody = body
}
```

Попробуйте выполнить приложение на данной стадии его разработки. Теперь снаряды должны одним ударом сбивать противников. Однако, когда вы начинете игру и сбиваете одного противника, игра “зависает”. Похоже, что теперь самое время ввести в игру управление ее уровнями.

Завершение игры на разных уровнях

Прежде всего, введите приведенный ниже метод `updateEnemies()` в файл `GameScene.swift`. Он действует во многом так же, как и введенный ранее метод `updateBullets()`.

```
private func updateEnemies() {
    var enemiesToRemove:[EnemyNode] = []
    for node in enemies.children as! [EnemyNode] {
        if !frame.contains(node.position) {
            // Помечаем противника для удаления
            enemiesToRemove.append(node)
        }
    }
    enemies.removeChildren(in: enemiesToRemove)
}
```

В этом методе организуется удаление каждого противника из массива противников на данном уровне игры всякий раз, когда он исчезает с экрана. Теперь внесите в метод `update()` изменения, чтобы вызвать метод `updateEnemies()`, а также новый, еще не реализованный метод.

```
override func update(currentTime: CFTimeInterval) {
    if finished {
        return
    }
    updateBullets()
    updateEnemies()
    checkForNextLevel()
}
```

В начале этого метода проверяется свойство `finished`. Мы уже дошли до той стадии, когда следует ввести код, официально завершающий уровень игры, и поэтому нужно убедиться, что по завершении данного уровня дополнительная обработка не потребуется. Затем мы вызываем метод `checkForNextLevel` в каждом кадре, чтобы проверить, завершен ли текущий уровень игры. Это делается аналогично проверке вылета снарядов или противников за переделы экрана в каждом кадре. Введите исходный код метода `checkForNextLevel()` следующим образом:

```
private func checkForNextLevel() {
    if enemies.children.isEmpty {
        goToNextLevel()
    }
}
```

Переход на следующий уровень игры

В свою очередь, в методе `checkForNextLevel()` вызывается другой метод, который мы еще не реализовали. Этот метод называется `goToNextLevel()`, помечает данный уровень как завершенный, выводит на экран текст, уведомляющий об этом игрока, а затем начинает игру на следующем уровне, как показано ниже.

```
private func goToNextLevel() {
    finished = true

    let label = SKLabelNode(fontNamed: "Courier")
    label.text = "Level Complete!"
    label.fontColor = SKColor.blue()
    label.fontSize = 32
    label.position = CGPoint(x: frame.size.width * 0.5,
                            y: frame.size.height * 0.5)
    addChild(label)

    let nextLevel = GameScene(size: frame.size, levelNumber: levelNumber + 1)
    nextLevel.playerLives = playerLives
    view!.presentScene(nextLevel, transition:
        SKTransition.flipHorizontal(withDuration: 1.0))
}
```

Во второй половине метода `goToNextLevel()` создается новый экземпляр класса `GameScene`, в котором устанавливаются все значения, требующиеся для начала игры на новом уровне. Затем текущему представлению дается команда воспроизвести новую сцену, сделав плавный переход к следующему кадру. Класс `SKTransition` позволяет выбрать самые разные стили переходов. Запустите данное приложение на выполнение и завершите игру на текущем уровне, чтобы посмотреть, как это выглядит на экране (рис. 17.8).

В данном случае выбран переход, похожий на переворачивание карты по горизонтальной оси, хотя для этой цели можно выбрать и немало других переходов. Подробнее об этом можно узнать из документации к классу `SKTransition`.

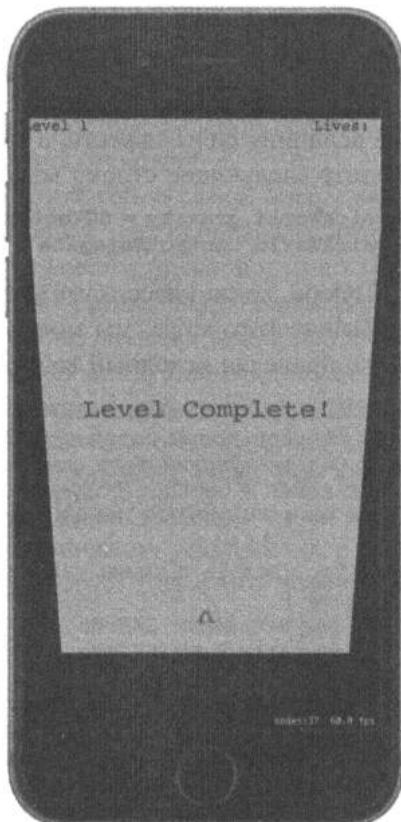


Рис. 17.8. Моментальный снимок экрана, сделанный во время плавного перехода от завершенного уровня игры к следующему

Настройка столкновений

Итак, мы разработали игру, в которую можно по-настоящему играть. Выбивая противников с экрана, игрок может переходить с одного уровня игры на другой. Все это, конечно, замечательно, но игре явно недостает состязательности. Как упоминалось ранее, одна из игровых задач состоит в том, чтобы и противник атаковал игрока, и теперь настало время ее решить. Усложним немного положение игрока, организовав падение противников от попадания снаряда или от соприкосновения с другим противником. Кроме того, “соударение” с падающим противником должно отбирать у игрока жизнь. Вы, вероятно, обратили внимание и на то, что, попав в противника, снаряд описывает виток вокруг него и продолжает двигаться вверх по своей траектории, что выглядит очень странно. Мы устраним все эти недостатки в игре, реализовав процедуру обработки столкновений в классе GameScene.swift.

Для обработки обнаруженных столкновений служит метод делегата из класса SKPhysicsWorld. По умолчанию сцена обладает физическими свойствами внешнего мира, но ее нужно немного настроить, прежде чем она что-нибудь нам сообщит. Сначала было бы неплохо уведомить компилятор, что мы собираемся реализовать протокол делегата. С этой целью введите в класс GameScene объявление, выделенное ниже полужирным шрифтом.

```
class GameScene: SKScene, SKPhysicsContactDelegate {
```

Нам по-прежнему нужно настроить внешний физический мир, немного убавив величину силы тяжести, а также сообщить ему о его делегате. С этой целью введите следующие строки кода в тело метода `init(size: levelNumber:)`:

```
physicsWorld.gravity = CGVectorMake(dx: 0, dy: -1)
physicsWorld.contactDelegate = self
```

Теперь, когда класс GameScene стал делегатом `contactDelegate` внешнего физического мира, мы можем реализовать соответствующий метод делегата. Ниже приведен исходный код, положенный в основу этого метода.

```
func didBegin(_ contact: SKPhysicsContact) {
    if contact.bodyA.categoryBitMask == contact.bodyB.categoryBitMask {
        // Оба физических тела относятся к одной категории
        let nodeA = contact.bodyA.node!
        let nodeB = contact.bodyB.node!

        // Что сделать с этими узлами?
    } else {
        var attacker: SKNode
        var attackee: SKNode

        if contact.bodyA.categoryBitMask
            > contact.bodyB.categoryBitMask {
            // Тело A атакует тело B
            attacker = contact.bodyA.node!
            attackee = contact.bodyB.node!
        } else {
```

```

    // Тело B атакует тело A
    attacker = contact.bodyB.node!
    attackee = contact.bodyA.node!
}

if attackee is PlayerNode {
    playerLives -= 1
}

// Что сделать с атакующим и атакуемым телами?
}
}

```

Ведите этот метод, хотя он пока еще не делает ничего особенного. На самом деле единственным конкретным результатом выполнения этого метода является сокращение количества жизней игрока всякий раз, когда обломки падающего противника попадают в самолет игрока, но ведь противники пока еще не падают!

Основной замысел реализации данного метода делегата состоит в том, чтобы обнаружить два соударяющихся объекта и выяснить, относятся ли они к одной и той же или к разным категориям. В первом случае они считаются “дружественными”, а в последнем случае придется определить среди них атакующего и атакуемого. Если посмотреть на порядок объявления категорий в исходном файле PhysicsCategories.swift, то можно обнаружить, что они указаны в порядке увеличения “атакуемости”. В частности, узлы игроков (Player) могут быть атакованы из узлов противников (Enemy), а те — из узлов снарядов (PlayerMissile). Это означает, что с помощью простой операции сравнения > можно определить атакующего в данной игровой обстановке.

Ради простоты и модульности мы не станем выяснять в самой сцене, каким образом каждый объект должен реагировать на атаку противника или соударение с другим объектом. Намного лучше встроить все эти подробности в классы узлов, подвергаемых атаке. Однако, как следует из приведенного выше метода, убедиться можно лишь в том, что у каждой стороны имеется экземпляр класса SKNode. Вместо составления длинной цепочки условных операторов if-else, чтобы выяснить, к какому именно подклассу, производному от класса SKNode, принадлежит каждый узел, мы можем воспользоваться обычным принципом полиморфизма, чтобы в каждом классе узлов это делалось по-своему. Следовательно, нам придется ввести в класс SKNode соответствующие методы, но так, чтобы они ничего не делали в своих реализациях по умолчанию, а затем переопределить их в тех подклассах, где это уместно. Для этого потребуется расширение класса.

Ввод расширения в класс SKNode

Для того чтобы ввести расширение в класс SKNode, щелкните правой кнопкой мыши на папке TextShooter в окне навигатора проекта среды Xcode и выполните команду New File... из всплывающего меню. Выберите шаблон Swift File в разделе iOS/Source и щелкните на кнопке Next. Присвойте новому файлу

664 ГЛАВА 17 ■ ВВЕДЕНИЕ В KARAKAS SPRITE KIT

имя SKNode+Extra.swift и нажмите кнопку Create. Откройте этот файл в редакторе и введите в нем строки кода, приведенные ниже.

```
import SpriteKit

extension SKNode {
    func receiveAttacker(_ attacker: SKNode, contact: SKPhysicsContact) {
        // В реализации по умолчанию этот метод ничего не делает
    }

    func friendlyBumpFrom(_ node: SKNode) {
        // В реализации по умолчанию этот метод ничего не делает
    }
}
```

Вернитесь к исходному файлу GameScene.swift, чтобы завершить ту его часть, где обрабатываются столкновения. Найдите в этом файле метод didBegin(_ contact:) и введите в него следующий код.

```
func didBegin(_ contact: SKPhysicsContact) {
    if contact.bodyA.categoryBitMask == contact.bodyB.categoryBitMask {
        // Оба физических тела относятся к одной категории
        let nodeA = contact.bodyA.node!
        let nodeB = contact.bodyB.node!

        // Что сделать с этими узлами?
        nodeA.friendlyBumpFrom(nodeB)
        nodeB.friendlyBumpFrom(nodeA)
    } else {
        var attacker: SKNode
        var attackee: SKNode

        if contact.bodyA.categoryBitMask
            > contact.bodyB.categoryBitMask {
            // Тело A атакует тело B
            attacker = contact.bodyA.node!
            attackee = contact.bodyB.node!
        } else {
            // Тело B атакует тело A
            attacker = contact.bodyB.node!
            attackee = contact.bodyA.node!
        }

        if attackee is PlayerNode {
            playerLives -= 1
        }

        // Что сделать с атакующим и атакуемым телами?
        attackee.receiveAttacker(attacker, contact: contact)
        playerBullets.removeChildren(in: [attacker])
        enemies.removeChildren(in: [attacker])
    }
}
```

В теле этого метода было введено несколько вызовов новых методов. Если противники сталкиваются друг с другом, каждый из них уведомляется о “дружественном” столкновении. В противном случае сначала выясняется, кто кого

атакует, а затем атакуемый уведомляется, что он атакован другим объектом. В заключение атакующий удаляется из любого узла типа playerBullets или типа Enemy, к которому он может принадлежать. Каждому из этих узловдается команда удалить атакующего, даже если он может быть одним из них, но так и должно быть. Команда узлу удалить узел, который он не порождал, не считается ошибкой — она просто не вызывает никакого действия.

Наделение противников особым поведением при столкновении

Итак, расставив все на свои места, можно реализовать в узлах ряд особых видов поведения, переопределив методы физических категорий, введенные в класс SKNode. С этой целью выберите исходный файл EnemyNode.swift и введите в него следующие переопределяемые методы:

```
override func friendlyBumpFrom(_ node: SKNode) {
    physicsBody!.affectedByGravity = true
}
override func receiveAttacker(_ attacker: SKNode, contact: SKPhysicsContact) {
    physicsBody!.affectedByGravity = true
    let force = vectorMultiply(attacker.physicsBody!.velocity,
                                contact.collisionImpulse)
    let myContact =
        scene!.convert(contact.contactPoint, to: self)
    physicsBody!.applyForce(force, at: myContact)
}
```

В первом из этих методов, friendlyBumpFrom(), просто активизируется действие силы тяжести на противника. Так, если один движущийся противник столкнется с другим противником, последний сразу же обнаружит силу тяжести и начнет падать вниз.

Второй метод, receiveAttacker(_:contact:), вызывается в том случае, если в противника попадает снаряд. Кроме того, в нем используются передаваемые ему данные соприкосновения, чтобы выяснить точку этого соприкосновения и приложить к ней силу, придав противнику дополнительный толчок в направлении выстрела снаряда.

Точное отображение количества жизней игрока

Запустите данное игровое приложение на выполнение. В этой игре вы можете не только сбивать противников, но и видеть, как одни падающие противники сталкиваются с другими, и те, в свою очередь, тоже начинают падать.

ЗАМЕЧАНИЕ. В начале игры на каждом уровне выполняется имитация физических свойств внешнего мира, чтобы физические тела не перекрывались. Это даст интересный побочный эффект на более высоких уровнях игры, поскольку увеличится вероятность, что несколько произвольно размещаемых противников займут перекрывающиеся места. Всякий раз, когда происходит нечто подобное, противники сразу же смещаются, чтобы не перекрываться, а затем вызывается код обработки столкновений, активизирующий силу тяжести, позволяя противникам падать! Такое поведение

мы вообще не предполагали, начиная разрабатывать эту игру, но по счастливой случайности оказалось, что чем выше уровень, тем труднее играть. Поэтому предоставим законам физики действовать своим чередом.

Если игрок допустит столкновение с падающими противниками, количество его жизней уменьшится. Тем не менее на экране все время отображается 5 жизней! Отображение количества жизней игрока организуется при создании уровня игры, но затем оно вообще не обновляется. Правда, этот недостаток нетрудно устранить, открыв исходный файл GameScene.swift и введя наблюдатель свойств в свойство playerLives:

```
private var playerLives: Int {
    didSet {
        let lives = childNode(withName: "LivesLabel") as! SKLabelNode
        lives.text = "Lives: \(playerLives)"
    }
}
```

В приведенном выше фрагменте кода используется имя, связанное ранее с меткой (в методе init(size:levelNumber:)). Оно дает возможность обнаружить метку снова и установить новое текстовое значение. Запустив данное приложение на выполнение, вы теперь обнаружите, что количество жизней игрока уменьшится до нуля, если допустить падение противников ливнем на игрока, но и тогда игра не закончится. Ведь в результате следующего столкновения с противниками количество жизней игрока станет отрицательным, как показано на рис. 17.9.

Этот недостаток проявляется потому, что мы еще не написали код для обнаружения конца игры, т.е. того момента, когда количество жизней игрока достигает нуля. Мы сделаем это вскоре, но прежде повысим немного интенсивность столкновений в сцене.

Придание игре остроты с помощью частиц

К числу примечательных особенностей каркаса Sprite Kit относится наличие в его составе системы частиц. Такие системы применяются в играх для создания визуальных эффектов, имитирующих дым, огонь, взрывы и пр. На данном этапе разработки игры снаряды, попадающие в противника, или сами противники, сталкивающиеся с игроком, просто перестают существовать как атакующие объекты, исчезая бесследно. Попробуем исправить это положение, создав пару систем частиц.

Итак, нажмите комбинацию клавиш **<⌘+N>**, чтобы открыть помощник для создания новых файлов. Выберите сначала раздел iOS Resource в левой части этого окна, а затем шаблон SpriteKit Particle File в правой части. Щелкните на кнопке Next и выберите на следующем экране шаблон частиц Spark. Щелкните на кнопке Next еще раз и присвойте этому шаблону имя файла MissileExplosion.sks.

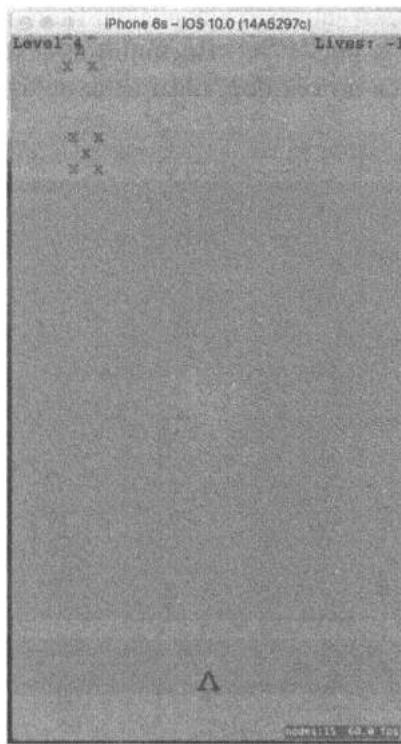


Рис. 17.9. Поскольку мы не можем распознать конец игры, количество жизней игрока стало отрицательным!

Первая система частиц

Как видите, в среде Xcode создается файл частиц и в проект вводится новый ресурс под названием spark.png. Одновременно вся область редактирования в Xcode переключается на новый файл частиц, отображая крупным планом оживляемую картину взрыва с разлетающимися в разные стороны частицами. Нам вряд ли понадобится столь причудливая и непомерная картина, когда снаряды попадают в противников, и поэтому внесем в нее некоторые корректизы. Все частицы, определяющие эту анимационную картину взрыва, находятся в инспекторе узлов типа SKNode, который можно открыть, нажав комбинацию клавиш **<Option+⌘+7>**. Картина мощного взрыва вместе с инспектором узлов типа SKNode приведена на рис. 17.10.

Теперь сделаем намного меньшим взрыв от попадания снаряда, воспользовавшись параметрами настройки системы частиц в правой части инспектора. Сначала, нужно подобрать цвета, подходящие для данной игры, щелкнув на немалом образце цвета Color Ramp справа внизу и выбрав черный цвет. Затем измените цвет фона (Background) на белый, а режим наложения (blend mode) на Alpha. В итоге весь фонтан пламени от взрыва станет черным как смоль. Все

остальные параметры имеют числовые значения. Измените их по очереди, уставив так, как показано на рис. 17.11. Внешний вид эффекта частиц от взрыва будет постепенно меняться до тех пор, пока цель не будет достигнута.

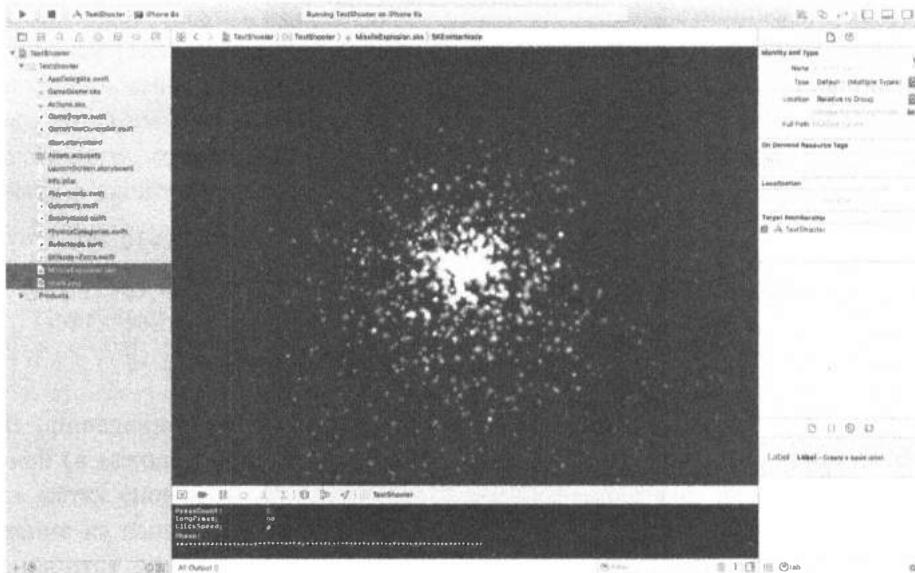


Рис. 17.10. Параметры, показанные справа, определяют внешний вид частиц по умолчанию

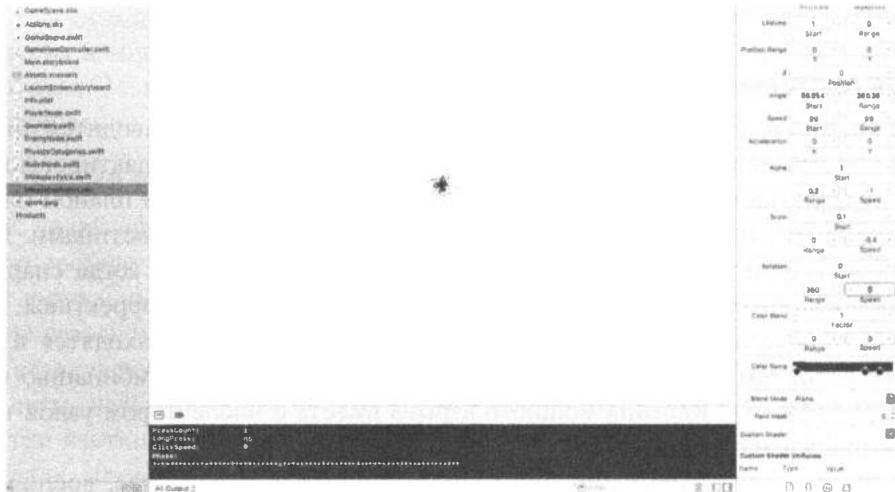


Рис. 17.11. Окончательный вид требующегося эффекта частиц от взрыва при попадании снаряда

Теперь создайте еще одну систему частиц из того же самого шаблона Spark. Присвойте новому файлу частиц имя EnemyExplosion.sks и установите параметры настройки этой системы частиц так, как показано на рис. 17.12.

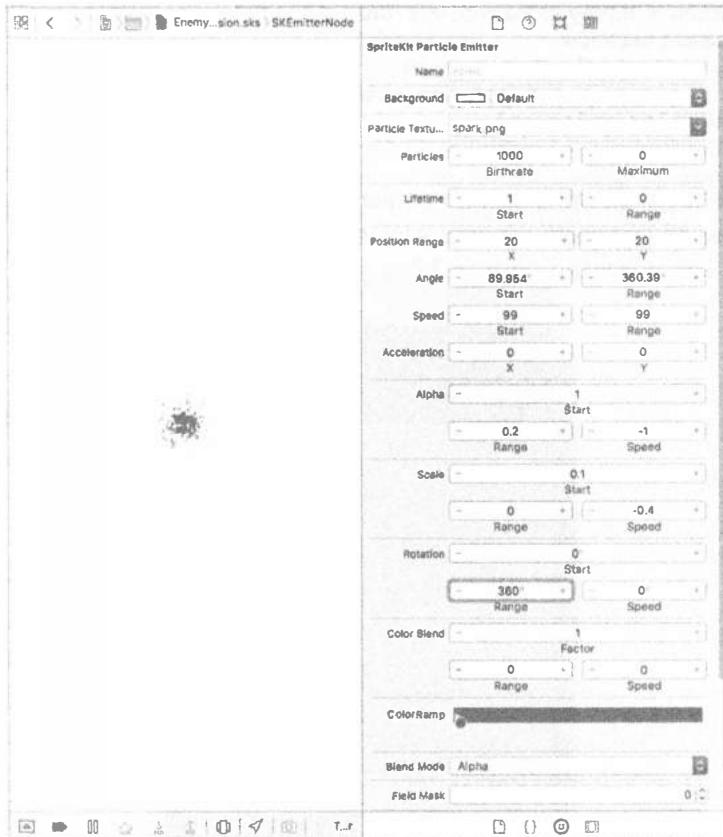


Рис. 17.12. Окончательный вид требующегося эффекта частиц от взрыва самолета противника. Для тех, кто читает данную книгу в черно-белом варианте, поясним, что в образце цвета Color Ramp на этот раз выбран темно-красный цвет

Размещение частиц на сцене

Теперь нужно разместить созданные системы частиц на сцене. Перейдите к исходному файлу `EnemyNode.swift` и введите выделенные полужирным шрифтом строки кода в конце тела метода `receiveAttacker(:contact:)`.

670 ГЛАВА 17 ■ ВВЕДЕНИЕ В KAPKAC SPRITE KIT

```
let myContact =  
    scene!.convert(contact.contactPoint, to: self)  
physicsBody!.applyForce(force, at: myContact)  
let path = Bundle.main.path(forResource: "MissileExplosion",  
    ofType: "sks")  
let explosion = NSKeyedUnarchiver.unarchiveObject(withFile: path!)  
    as! SKEmitterNode  
explosion.numParticlesToEmit = 20  
explosion.position = contact.contactPoint  
scene!.addChild(explosion)  
}
```

Запустите игровое приложение на выполнение. Начните стрелять по противникам, чтобы обнаружить мелкие взрывы в тех местах, где снаряд попадает в противника, как показано на рис. 17.13.

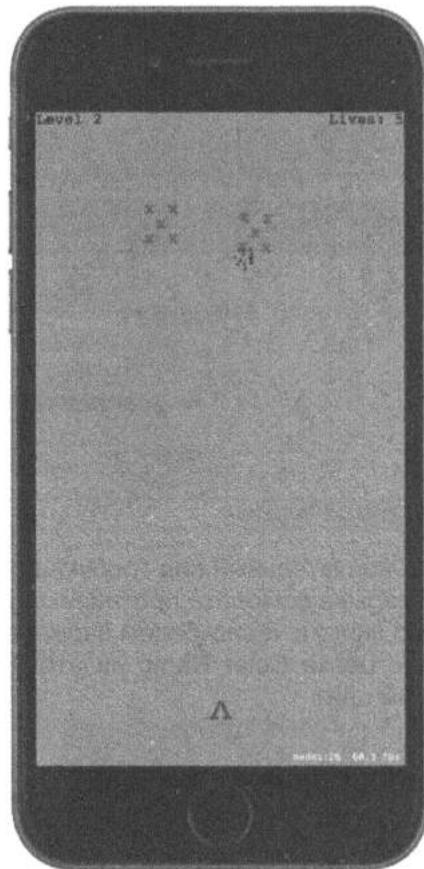


Рис. 17.13. Взрывы снарядов при попадании

Теперь сделаем нечто подобное на тот случай, если столкнутся самолеты противника и игрока. С этой целью выберите исходный файл `PlayerNode.swift` и введите в нем следующий метод:

```

override func receiveAttacker(_ attacker: SKNode, contact: SKPhysicsContact) {
    let path = Bundle.main.path(forResource: "EnemyExplosion",
                                ofType: "sks")
    let explosion = NSKeyedUnarchiver.unarchiveObject(withFile: path!)
        as! SKEmitterNode
    explosion.numParticlesToEmit = 50
    explosion.position = contact.contactPoint
    scene!.addChild(explosion)
}

```

Снова запустите данное приложение на выполнение и начните игру. Всякий раз, когда самолет противника столкнется с самолетом игрока, на экране появится эффектный взрыв красного цвета, как показано на рис. 17.14.

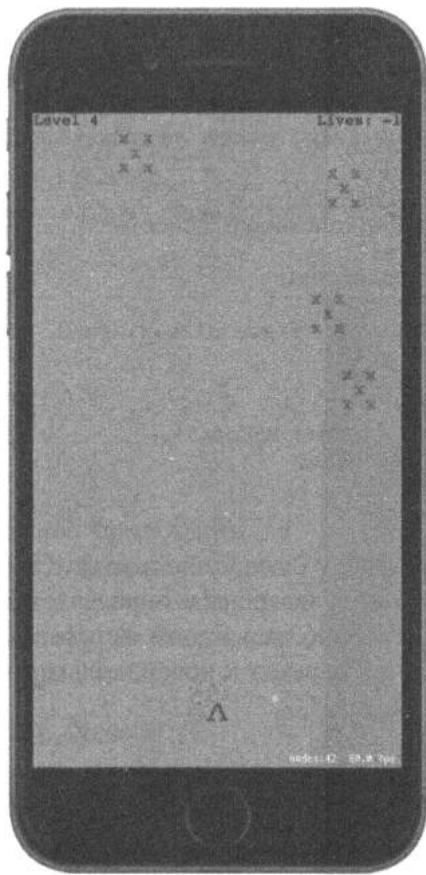


Рис. 17.14. Взрыв, который возникает, когда противник попадает в игрока

Несмотря на всю свою простоту описанные выше изменения значительно улучшают внешний вид игры. Теперь, когда объекты сталкиваются, хорошо видны последствия, ясно показывающие, что же произошло.

Завершение игры

Как упоминалось ранее, у рассматриваемой игры имеется следующий мелкий недостаток: когда количество жизней игрока достигает нуля, нужно завершать игру. Для того чтобы устранить этот недостаток, создайте новый класс сцены для перехода к ней по завершении игры. Аналогичный переход вы уже наблюдали при смене уровня игры. Таким же образом совершается и переход в конце игры, но только с помощью нового класса сцены. Итак, создайте новый класс iOS/Cocoa Touch. В качестве его суперкласса выберите класс SKScene и присвойте ему имя GameOverScene. Его реализацию мы начнем с простого вывода текстового сообщения "Game Over", не делая больше ничего. С этой целью введите в исходный файл GameOverScene.swift следующие строки кода:

```
import SpriteKit

class GameOverScene: SKScene {

    override init(size: CGSize) {
        super.init(size: size)
        backgroundColor = SKColor.purple
        let text = SKLabelNode(fontNamed: "Courier")
        text.text = "Game Over"
        text.fontColor = SKColor.white
        text.fontSize = 50
        text.position = CGPoint(x: frame.size.width/2, y: frame.size.height/2)
        addChild(text)
    }

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

Вернемся к исходному файлу GameScene.swift. Основное действие, которое следует выполнить, когда игра завершена, определяется в новом методе triggerGameOver(). Ниже показано, как в этом методе воспроизводится дополнительный взрыв и начинается переход к новой, только что созданной сцене.

```
private func triggerGameOver() {
    finished = true

    let path = Bundle.main.path(forResource:"EnemyExplosion",
                                ofType: "sks")
    let explosion = NSKeyedUnarchiver.unarchiveObject(withFile: path!)
        as! SKEmitterNode
    explosion.numParticlesToEmit = 200
    explosion.position = playerNode.position
    scene!.addChild(explosion)
    playerNode.removeFromParent()

    let transition = SKTransition.doorsOpenVertical(withDuration: 1)
    let gameOver = GameOverScene(size: frame.size)
    view!.presentScene(gameOver, transition: transition)
}
```

Далее введите приведенный ниже метод, в котором проверяется, завершена ли игра, вызывается метод `triggerGameOver()`, если подходящий момент настал, и возвращается логическое значение `true`, означающее завершение игры, или же логическое значение `false`, означающее продолжение игры.

```
private func checkForGameOver() -> Bool {
    if playerLives == 0 {
        triggerGameOver()
        return true
    }
    return false
}
```

В заключение введите в уже имеющийся метод `update()` проверку состояния завершения игры, а если игра продолжается, то и проверку возможного перехода на следующий уровень. В противном случае существует риск, что последний противник на текущем уровне может захватить последнюю жизнь игрока и сделать сразу два перехода между сценами.

```
override func update(_ currentTime: TimeInterval) {
    /* Этот метод вызывается перед прорисовкой каждого кадра */
    if finished {
        return
    }
    updateBullets()

    updateEnemies()
    if (!checkForGameOver()) {
        checkForNextLevel()
    }
}
```

Снова запустите данное приложение на выполнение и позвольте противникам пять раз упасть на игрока. В итоге должна появиться сцена с сообщением об окончании игры, как показано на рис. 17.15.

Создание начальной сцены

В связи с изложенным выше возникает следующий вопрос: что делать по завершении игры? Можно было бы, конечно, дать пользователю возможность нажать пальцем на экран, чтобы перезапустить игру. Не лучше ли создать нечто вроде начального экрана, чтобы не вводить игрока в игру сразу после ее запуска, а по окончании игры сделать переход к ее начальной сцене? Разумеется, ответ на последний вопрос может быть только

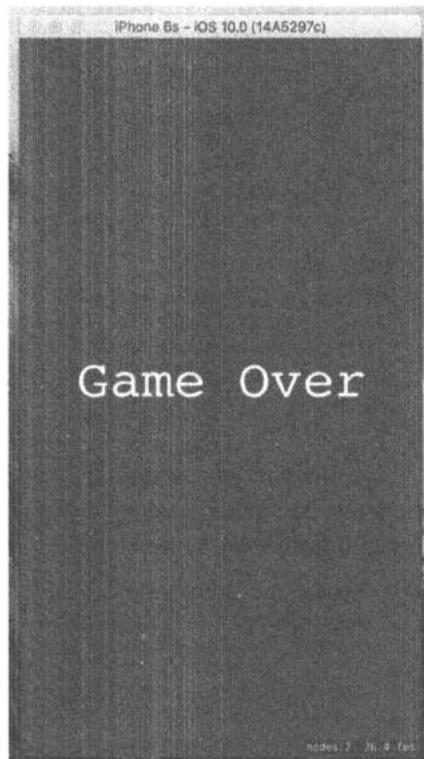


Рис. 17.15. Финальный экран

положительным! Итак, создайте еще один новый класс iOS/Cocoa Touch, снова выбрав суперкласс SKScene в качестве его родителя, но на этот раз присвоив ему имя StartScene. Мы собираемся сделать начальную сцену крайне простой, воспроизведя в ней немного текста и начав игру, как только пользователь нажмет пальцем где угодно на экране. Для того чтобы завершить создание класса StartScene, введите в исходном файле StartScene.swift следующий код:

```
import SpriteKit

class StartScene: SKScene {

    override init(size: CGSize) {
        super.init(size: size)
        backgroundColor = SKColor.green()

        let topLabel = SKLabelNode(fontNamed: "Courier")
        topLabel.text = "TextShooter"
        topLabel.fontColor = SKColor.black()
        topLabel.fontSize = 48
        topLabel.position = CGPoint(x: frame.size.width/2,
                                    y: frame.size.height * 0.7)
        addChild(topLabel)

        let bottomLabel = SKLabelNode(fontNamed: "Courier")
        bottomLabel.text = "Touch anywhere to start"
        bottomLabel.fontColor = SKColor.black()
        bottomLabel.fontSize = 20
        bottomLabel.position = CGPoint(x: frame.size.width/2,
                                      y: frame.size.height * 0.3)
        addChild(bottomLabel)
    }

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

    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
        let transition = SKTransition.doorway(withDuration: 1.0)
        let game = GameScene(size: frame.size)
        view!.presentScene(game, transition: transition)
    }
}
```

Вернитесь к исходному файлу GameOverScene.swift, чтобы организовать переход от завершающей игру сцены к ее начальной сцене. Введите в этот файл следующий фрагмент кода:

```
override func didMove(to view: SKView) {
    DispatchQueue.main.after(
        when: DispatchTime.now() + Double(3 * Int64(NSEC_PER_SEC)) /
            Double(NSEC_PER_SEC)) {
            let transition = SKTransition.flipVertical(withDuration: 1)
            let start = StartScene(size: self.frame.size)
            view.presentScene(start, transition: transition)
    }
}
```

Как было показано ранее, метод `didMoveToView()` вызывается для любой сцены после его размещения в представлении. В данном случае просто делается пауза длительностью три секунды, после чего происходит переход обратно к начальной сцене. Еще одна задача состоит в том, чтобы упорядочить все переходы между сценами. В частности, процедуру запуска данного приложения нужно изменить таким образом, чтобы вместо перехода непосредственно к игре появлялась ее начальная сцена. Для этого придется вернуться к исходному файлу `GameViewController.swift`, чтобы заменить строку кода создания класса одной сцены другой в теле метода `viewDidLoad()`, как показано ниже.

```
/* Выбрать размеры сцены */
let scene = GameScene(size: view.frame.size, levelNumber: 1)
let scene = StartScene(size: view.frame.size)
```

Теперь запустите данное приложение на выполнение. На экране появится начальная сцена с предложением начать игру. Коснитесь пальцем экрана, поиграйте, потеряв все жизни, чтобы появилась сцена, завершающая игру. Подождите несколько секунд, чтобы снова появилась начальная сцена, как показано на рис. 17.16.

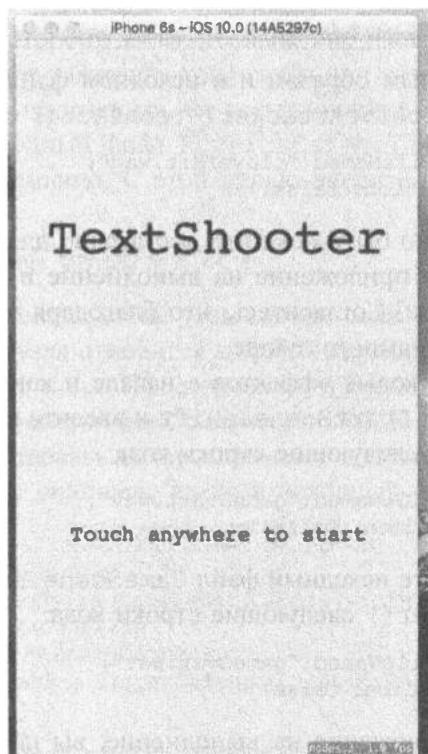


Рис. 17.16. Наконец-то, добрались до начального экрана!

Добавление звуковых эффектов

Если речь идет о разработке видеоигр, то они известны как довольно шумные, тогда как рассматриваемая здесь игра совершенно бесшумная. Правда, в состав Sprite Kit входят средства для воспроизведения звука, которыми очень просто пользоваться. В папке 17-Sound Effects с исходным кодом, прилагаемым к этой книге, вы найдете следующие звуковые файлы, заранее подготовленные для данной игры: enemyHit.wav, gameOver.wav, gameStart.wav, playerHit.wav и shoot.wav. Перетащите все эти файлы в окно навигатора проекта среды Xcode.

Теперь остается лишь организовать воспроизведение каждого из этих звуковых эффектов. Начните с звукового файла BulletNode.swift, введя в конце метода bullet(from:toward:) выделенные следующие строки непосредственно перед строкой с оператором return.

```
bullet.run(SKAction.playSoundFileNamed("shoot.wav",
    waitForCompletion: false))
```

Затем перейдите к исходному файлу EnemyNode.swift и введите в конце метода receiveAttacker(_:contact:) следующие строки кода:

```
run(SKAction.playSoundFileNamed("enemyHit.wav",
    waitForCompletion: false))
```

Поступите аналогичным образом и в исходном файле PlayerNode.swift, введя в конце метода receiveAttacker(_:contact:) следующие строки кода:

```
run(SKAction.playSoundFileNamed("playerHit.wav",
    waitForCompletion: false))
```

Этих эффектов должно быть пока что достаточно для звукового оформления игры. Запустите данное приложение на выполнение и опробуйте внедренные в него звуковые эффекты. Согласитесь, что благодаря частицам и звукам игра воспринимается теперь намного лучше.

Добавим немного звуковых эффектов в начале и конце игры. С этой целью откройте исходный файл StartScene.swift и введите в конце метода touchesBegan(_:withEvent:) следующие строки кода:

```
run(SKAction.playSoundFileNamed("gameStart.wav",
    waitForCompletion: false))
```

В заключение откройте исходный файл GameScene.swift и введите в конце метода triggerGameOver() следующие строки кода:

```
run(SKAction.playSoundFileNamed("gameOver.wav",
    waitForCompletion: false))
```

Запустив данное приложение на выполнение, вы погрузитесь в атмосферу игры, издающей рев и писк, как в вашем детстве, а возможно, и в детстве ваших родителей или даже других предков! Поверьте, что практически все игры издают подобные звуки.

Усложнение игры с помощью силовых полей

К числу других интересных новых средств, внедренных в каркас Sprite Kit, относится возможность вносить в сцену силовые поля. У силового поля имеется тип, местоположение, область действия и ряд других свойств, определяющих его поведение. Принцип действия такого поля состоит в том, что оно вносит возмущение в движение объектов, пересекающих область его действия. Имеются различные типы силовых полей, которыми можно воспользоваться, получив и настроив соответствующий экземпляр и затем введя его в сцену. Если вас снедает честолюбие, можете даже создать специальные силовые поля. С перечнем стандартных силовых полей, в том числе электрических, магнитных, тяготения и турбулентности, а также их поведением можно ознакомиться, обратившись к документации к классу `SKFieldNode` из интерфейса прикладного программирования каркаса Sprite Kit.

Для того чтобы немного усложнить рассматриваемую здесь игру, внесем в сцену несколько полей радиального тяготения. Такие поля действуют подобно крупной массе, сосредоточенной в одной точке. Когда объект пересекает область действия поля радиального тяготения, он отклоняется в сторону этого поля подобно метеориту, пролетающему очень близко от Земли (или в противоположную от силового поля сторону, если его действие настроено именно таким образом). Мы устроим поля тяготения так, чтобы они воздействовали на снаряды, и тогда игрок, целясь в противника, не всегда сможет в него попасть.

Прежде всего, в исходный файл `PhysicsCategories.swift` нужно ввести новую физическую категорию. С этой целью внесите в него следующее изменение:

```
let GravityFieldCategory: UInt32 = 1 << 4
```

Поле тяготения воздействует на узел сцены, если свойство `fieldBitMask` физического тела этого узла относится к любой категории, указанной в свойстве `categoryBitMask` этого поля. По умолчанию в свойстве `fieldBitMask` установлены физические категории всех силовых полей. В то же время поле тяготения не должно воздействовать на противников, и поэтому свойство `fieldBitMask` этого поля нужно очистить, введя в исходный файл `EnemyNode.swift` следующий код.

```
private func initPhysicsBody() {
    let body = SKPhysicsBody(rectangleOf: CGSize(width: 40, height: 40))
    body.affectedByGravity = false
    body.categoryBitMask = EnemyCategory
    body.contactTestBitMask = PlayerCategory | EnemyCategory
    body.mass = 0.2
    body.angularDamping = 0
    body.linearDamping = 0
    body.fieldBitMask = 0
    physicsBody = body
}
```

678 ГЛАВА 17 ■ ВВЕДЕНИЕ В KAPKAC SPRITE KIT

Внесите аналогичное изменение в исходный файл `PlayerNode.swift`.

```
private func initPhysicsBody() {
    let body = SKPhysicsBody(rectangleOf: CGSize(width: 20, height: 20))
    body.affectedByGravity = false
    body.categoryBitMask = PlayerCategory
    body.contactTestBitMask = EnemyCategory
    body.collisionBitMask = 0
    body.fieldBitMask = 0
    physicsBody = body
}
```

Узлы снарядов будут реагировать на поле тяготения, даже если игрок не предпринимает никаких действий, поскольку их физические тела по умолчанию относятся ко всем категориям силовых полей. Однако ради ясности это лучше сделать явно. С этой целью внесите в исходный файл `BulletNode.swift` следующие изменения:

```
override init() {
    super.init()

    let dot = SKLabelNode(fontNamed: "Courier")
    dot.fontColor = SKColor.black()
    dot.fontSize = 40
    dot.text = "."
    addChild(dot)

    let body = SKPhysicsBody(circleOfRadius: 1)
    body.isDynamic = true
    body.categoryBitMask = PlayerMissileCategory
    body.contactTestBitMask = EnemyCategory
    body.collisionBitMask = EnemyCategory
    body.fieldBitMask = GravityFieldCategory
    body.mass = 0.01

    physicsBody = body
    name = "Bullet \(self)"
}
```

Остальные изменения следует внести в исходный файл `GameScene.swift`. С этой целью мы внесем в сцену три поля тяготения, расположив их в произвольных точках чуть ниже центра сцены. Аналогично узлам снарядов и противников мы введем узлы силовых полей в родительский узел, а тот — в сцену. Итак, введите определение родительского узла в качестве нового свойства.

```
class GameScene: SKScene, SKPhysicsContactDelegate {
    private var levelNumber: Int
    private var playerLives: Int {
        didSet {
            let lives = childNode(withName: "LivesLabel") as! SKLabelNode
            lives.text = "Lives: \(playerLives)"
        }
    }
    private var finished = false
```

```
private let playerNode: PlayerNode = PlayerNode()
private let enemies = SKNode()
private let playerBullets = SKNode()
private let forceFields = SKNode()
```

Далее введите в конце метода `init(size:levelNumber:)` строки кода, приведенные ниже. В этих строках кода узел `forceFields` вводится в сцену и создаются отдельные узлы силовых полей.

```
addChild(forceFields)
createForceFields()

physicsWorld.gravity = CGVector(dx: 0, dy: -1)
physicsWorld.contactDelegate = self
}
```

В заключение введите следующую реализацию метода `createForceFields()`:

```
private func createForceFields() {
    let fieldCount = 3
    let size = frame.size
    let sectionWidth = Int(size.width) / fieldCount
    for i in 0..<fieldCount {
        let x = CGFloat(UInt32(i * sectionWidth) +
            arc4random_uniform(UInt32(sectionWidth)))
        let y = CGFloat(arc4random_uniform(UInt32(size.height * 0.25)) +
            UInt32(size.height * 0.25))
        let gravityField = SKFieldNode.radialGravityField()
        gravityField.position = CGPoint(x: x, y: y)
        gravityField.categoryBitMask = GravityFieldCategory
        gravityField.strength = 4
        gravityField.falloff = 2
        gravityField.region = SKRegion(size: CGSize(width: size.width * 0.3,
                                                      height: size.height * 0.1))
        forceFields.addChild(gravityField)

        let fieldLocationNode = SKLabelNode(fontNamed: "Courier")
        fieldLocationNode.fontSize = 16
        fieldLocationNode.fontColor = SKColor.red()
        fieldLocationNode.name = "GravityField"
        fieldLocationNode.text = "*"
        fieldLocationNode.position = CGPoint(x: x, y: y)
        forceFields.addChild(fieldLocationNode)
    }
}
```

Все силовые поля представлены экземплярами класса `SKFieldNode`. Для каждого типа поля в классе `SKFieldNode` имеется свой фабричный метод, позволяющий создать узел силового поля данного типа. В данном случае метод `radialGravityField()` используется для создания трех экземпляров поля радиального тяготения, а затем они располагаются в ряд чуть ниже центра сцены. Свойства `strength` и `falloff` определяют соответственно силу тяжести и степень ее убывания по мере отдаления от узла данного силового поля. Так, при значении 2 свойства `falloff` сила тяжести действует обратно пропорционально

квадрату расстояния между узлом данного силового поля и объектом, на который оно воздействует, как это обычно и происходит на практике. Если сила тяжести положительная, то узел данного силового поля притягивает объект. Попытайтесь экспериментировать с разными значениями свойства `strength`, в том числе с отрицательными, чтобы понаблюдать за действием полей тяготения. На тех же позициях, где и узлы полей тяготения, были созданы три узла меток типа `SKLabelNode`, чтобы игрок видел, где эти поля находятся. Вот, собственно, и все, что нужно было сделать для внесения силовых полей в данную игру. Постройте и запустите данное приложение на выполнение, а затем понаблюдайте за тем, что происходит, когда снаряды пролетают рядом с одной из красных звездочек в сцене.

Резюме

Несмотря на простой внешний вид приложения `TextShooter` методики, представленные в этой главе, послужат вам прочным основанием для разработки разнообразных игр средствами `Sprite Kit`. Из этой главы вы узнали, как распределять прикладной код по классам нескольких узлов, группировать объекты в графе сцены и о многом другом. Кроме того, вы получили некоторое представление о поэтапном процессе разработки подобного рода игр с постепенным раскрытием возможностей игры на каждом этапе. Разумеется, в рамках одной главы невозможно продемонстрировать все особенности разработки игровых приложений, в том числе ложные шаги, совершенные нами по ходу дела. Даже если учесть наши оплошности, на разработку данного игрового приложения с самого начала приблизительно в том порядке, в каком оно описано в этой главе, нам потребовалось лишь несколько часов.

Приступив к изучению возможностей каркаса `Sprite Kit`, вы обнаружите, что с его помощью можно быстро построить большую часть структуры приложения. Как было показано в этой главе, при отсутствии подходящих изображений можно пользоваться и текстовыми спрайтами. Если их потребуется в дальнейшем заменить конкретной графикой, то сделать это будет совсем нетрудно. Один из читателей рукописи этой книги предложил такой средний путь: “Вместо обычного текста в коде ASCII в строках исходного кода можно ввести символы эмодзи, используя приложение `Character Viewer` от компании Apple”. Оставляем реализацию такого пути вам, читатель, в качестве упражнения.

ГЛАВА 18



Нажатия, касания и жесты

Яркие четкие сенсорные экраны мобильных устройств iPhone, iPod touch и iPad представляют собой настоящие шедевры дизайнерской и инженерной мысли. Мультисенсорный экран является характерным элементом всех мобильных устройств, работающих под управлением системы iOS, и поэтому он относится к одним из самых главных факторов, во многом определяющих удобство использования данной платформы. Такой экран способен обнаруживать много одновременно совершаемых касаний и отслеживать их по отдельности, поэтому соответствующие приложения должны распознавать самые разные жесты, предоставляя пользователю возможности, выходящие за рамки привычного интерфейса.

Допустим, вы работаете с приложением Mail и вам требуется удалить длинный список “макулатурных” сообщений электронной почты. Для этого у вас имеется несколько возможностей. Вы можете коснуться сначала каждого сообщения электронной почты в отдельности, а затем — пиктограммы корзины, чтобы удалить это сообщение и подождать загрузки следующего сообщения, удалив его, в свою очередь. Такой способ лучше всего подходит для тех случаев, когда каждое сообщение электронной почты требуется прочитать, прежде чем удалить. Если у вас iPhone 6s или iPhone 6s Plus, можете воспользоваться функцией 3D Touch, чтобы предварительно просмотреть электронную почту, не открывая ее. В качестве альтернативы можно нажать кнопку Edit в правом верхнем углу, а затем коснуться каждой строки сообщения электронной почты в списке, чтобы пометить его, и далее нажать кнопку Delete, чтобы удалить все помеченные сообщения. Такой способ лучше всего подходит для тех случаев, когда читать каждое сообщение электронной почты перед его удалением не нужно. Кроме того, можно провести пальцем по сообщению электронной почты от одного края списка к другому, чтобы активизировать кнопки More и Trash для удаления данного сообщения. Нажмите кнопку Trash, и сообщение будет удалено.

Приведенные выше варианты удаления сообщений электронной почты служат лишь некоторыми примерами бесчисленного множества жестов, которые

способен распознавать мультисенсорный дисплей. Так, касаясь экрана и собирая пальцы в щепотку, вы сможете уменьшать масштаб просматриваемого изображения, а разводя пальцы — увеличивать его масштаб. Если же долго нажимать на пиктограмму, то произойдет переход в режим “покачивания”, в котором можно удалять приложения из мобильного устройства, работающего под управлением системы iOS; на устройствах iPhone 6s и iPhone 6s Plus можно открыть список быстрых клавиш вызова для приложений, поддерживающих технологию 3D Touch. В этой главе рассматривается базовая мультисенсорная архитектура, позволяющая обнаруживать и распознавать жесты. Из нее вы узнаете, как обнаруживается большинство жестов и как создаются и распознаются совершенно новые жесты.

Мультисенсорная терминология

Прежде чем переходить к рассмотрению мультисенсорной архитектуры, приведем краткий словарь некоторых наиболее употребительных терминов. **Жест** — это любая последовательность событий, начиная с момента касания экрана одним или несколькими пальцами и заканчивая отнятием пальцев от экрана. Продолжительность события не имеет особого значения. Событие продолжается до тех пор, пока один или несколько пальцев касаются экрана, если только оно не прерывается системным событием, например входящим телефонным звонком. Следует иметь в виду, что в среде Cocoa Touch не раскрывается ни один из классов или структур, представляющих жест. В каком-то смысле жест — это команда, появление которой отслеживается выполняющимся приложением в потоке ввода данных пользователем. Жест, совершаемый пользователем, распознается системой как последовательность **событий**. События содержат сведения об одном или нескольких произошедших касаниях. События передаются по цепочке реагирующих элементов, рассматриваемой в следующем разделе.

Касание означает прикладывание пальца к экрану, проведение пальцем по экрану и отнятие пальца от экрана. Количество касаний, входящих в жест, равно количеству пальцев, одновременно находящихся на экране. К экрану можно приложить сразу все пять пальцев, и если они не расположены слишком близко друг к другу, то система iOS сумеет распознать и отследить их по отдельности. И хотя полезных жестов пятью пальцами не так уж и много, приятно осознавать, что система iOS способна их распознать и обработать, если потребуется. Как показали эксперименты, мобильное устройство iPad может обрабатывать до 11 одновременных касаний! На первый взгляд, это может показаться чрезмерным, но в то же время полезным, если речь идет о многопользовательской игре, где несколько игроков одновременно взаимодействуют с экраном.

Нажатие имеет место в том случае, когда пользователь касается экрана одним пальцем и сразу же отнимает его от поверхности, не перемещая по экрану. Устройство, работающее под управлением системы iOS, отслеживает количество нажатий и способно распознавать двойные, тройные и даже двадцатикратные

нажатия. Оно выполняет всю необходимую синхронизацию и обработку, чтобы, например, отличать одиночные нажатия от двойных.

Распознаватель жестов — это объект, способный следить за потоком порождаемых пользователем событий и распознавать моменты, когда пользователь касается экрана и скользит по нему пальцем так, как это обычно согласуется с заранее определенным жестом. В состав iOS входят класс `UIGestureRecognizer` и различные его подклассы, помогающие автоматизировать большую часть операций по слежению за типичными жестами. Этот класс изящно инкапсулирует операции поиска жеста и без особого труда применяется в любом представлении создаваемого приложения.

В первой части этой главы будут продемонстрированы события, о которых система сообщает, когда пользователь касается экрана одним пальцем или более, а также показано, как отслеживаются движения пальцев по экрану. Используя эти события, можно обрабатывать жесты в специальном представлении или делегате приложения. Затем будут рассмотрены некоторые распознаватели жестов, входящие в состав комплекта iOS SDK. И в заключение будет показано, как построить свой распознаватель жестов.

Цепочка реагирующих элементов

Жесты проходят систему, состоящую из последовательности событий, а события передаются по **цепочке реагирующих элементов**, и поэтому необходимо ясно понимать, каким образом эта цепочка действует, чтобы правильно обрабатывать жесты. Если у вас имеется опыт работы в среде Cocoa для системы OS X, вам должно быть знакомо понятие цепочки реагирующих элементов. Аналогичный основной механизм применяется и в среде Cocoa Touch. Если же этот механизм вам не знаком, не переживайте: ниже поясняется принцип его действия.

Реакция на события

На страницах этой книги уже неоднократно упоминалось о первом реагирующем элементе. Как правило, им является объект, с которым пользователь взаимодействует в настоящий момент. Первый реагирующий элемент находится в самом начале цепочки реагирующих элементов, где имеются и другие похожие элементы. Любой класс, для которого `UIResponder` служит одним из его суперклассов, является **реагирующим элементом (responder)**. Так, класс `UIView` является подклассом класса `UIResponder`, а класс `UIControl` — подклассом класса `UIView`, и поэтому все представления и элементы управления являются реагирующими элементами. Класс `UIViewController` также является подклассом класса `UIResponder`, а это означает, что он и все его подклассы, в том числе `UINavigationController` и `UITabBarController`, относятся к реагирующим элементам. Реагирующие элементы называются так потому, что реагируют на порождаемые системой события, например касания экрана.

Если первый реагирующий элемент не обрабатывает конкретное событие, например жест, он передает его далее по цепочке реагирующих элементов. Если же следующий по цепочке объект реагирует на это конкретное событие, он, как правило, обрабатывает данное событие, прекращая его распространение по цепочке реагирующих элементов. Если реагирующий элемент обрабатывает событие лишь частично, он предпринимает действие, чтобы направить событие следующему по цепочке реагирующему элементу, хотя такое случается редко. Как правило, если объект реагирует на событие, то на этом цепочка его распространения обрывается. Если же событие проходит всю цепочку реагирующих элементов и не обрабатывается ни одним из элементов, оно отвергается.

Рассмотрим действие цепочки реагирующих элементов более подробно. Событие доставляется сначала объекту типа `UIApplication`, который, в свою очередь, передает его в окно приложения, представленное классом `UIWindow`. Для обработки события в объекте типа `UIWindow` выбирается исходный реагирующий элемент. Этот элемент выбирается следующим образом.

- Если речь идет о событии касания, то объект типа `UIWindow` сначала определяет представление, которого касается пользователь, а затем предоставляет событие любому из распознавателей жестов, зарегистрированных для данного представления или любого другого представления, расположенного выше по иерархии. Если любой распознаватель жестов обработает событие, оно не распространяется дальше. В противном случае исходный распознаватель жестов оказывается именно тем представлением, в котором произошло касание, и поэтому событие будет доставлено именно ему.
- Событие, генерируемое в том случае, когда пользователь встряхивает мобильное устройство (подробнее об этом — в главе 20), или же в устройстве дистанционного управления доставляется первому реагирующему элементу.

Если исходный реагирующий элемент не обрабатывает событие, он передает его своему родительскому представлению, при условии, что оно имеется, или же контроллеру представления, если это представление контроллера представления. Если же контроллер представления не обрабатывает событие, оно передается вверх по цепочке реагирующих элементов через иерархию представлений родителю этого контроллера представления, при условии, что таковой имеется.

Если событие доходит до вершины иерархии представлений, так и оставшись необработанным, оно передается окну приложения. Если же оно не обрабатывается и в окне приложения, то объект типа `UIApplication` передаст его далее делегату приложения, если делегат представления является подклассом, производным от класса `UIResponder` (что естественно, если вы создаете свой проект по шаблону приложения Apple). В заключение, если делегат приложения не является подклассом, производным от класса `UIResponder`, или не обрабатывает данное событие, то с ним вежливо прощаются.

Этот процесс важен по целому ряду причин. Прежде всего, он определяет порядок обработки жестов. Допустим, пользователь смотрит на таблицу, отображаемую на экране, и скользит пальцем по ее строке. Какой объект должен обрабатывать такой жест? Если такое скольжение пальцем по экрану оказывается в пределах действия представления или элемента управления, подчиненного по отношению к ячейке табличного представления, это представление или элемент управления получает возможность отреагировать на подобный жест. В противном случае эту возможность получает ячейка табличного представления. В таком приложении, как Mail, где скольжение пальцем по экрану может быть использовано для удаления сообщения, ячейке табличного представления, вероятно, требуется проверить, содержит ли данное событие жест скольжения. Большинство ячеек табличного представления не реагируют на жесты. В таком случае событие распространяется вверх к табличному представлению и далее по цепочке реагирующих элементов до тех пор, пока не достигнет ее конца.

Передача события по цепочке реагирующих элементов, поддерживаемой в активном состоянии

Вернемся к ячейке табличного представления в приложении Mail. Нам, конечно, неизвестен внутренний механизм работы этого стандартного приложения, но допустим, что ячейка табличного представления обрабатывает удаляющее скольжение пальцем по экрану, и только этот жест. Для этой цели в данной ячейке должны быть реализованы методы, связанные с получением событий касания (подробнее об этом — в следующем разделе), чтобы иметь возможность проверить, содержит ли данное событие скольжение. Если оно содержит такой жест, ячейка табличного представления предпринимает соответствующее действие, а событие не распространяется дальше.

Если событие не содержит скольжение, ячейка табличного представления несет ответственность за передачу данного события вручную следующему объекту в цепочке реагирующих элементов. Если же ячейка не сделает этого, таблица и другие объекты, расположенные вверх по цепочке, вообще не получат возможность отреагировать на данное событие, а приложение, вероятнее всего, не будет работать так, как того ожидает пользователь. Таким образом, ячейка табличного представления может воспрепятствовать другим представлениям распознать данный жест.

Когда бы ни нужно было реагировать на событие касания, необходимо иметь в виду, что код не работает в вакууме. Так, если объект перехватывает событие, которое он не в состоянии обработать, он должен передать его дальше, вызвав тот же самый метод для следующего реагирующего элемента. В листинге 18.1 приведен фрагмент вымышленного кода, реализующего это правило.

Листинг 18.1. Передача события для дальнейшей обработки

```
func respondToFictionalEvent(event: UIEvent) {
    if shouldHandleEvent(event) {
        handleEvent(event)
    } else {
        nextResponder().respondToFictionalEvent(event)
    }
}
```

Обратите внимание на порядок вызова того же самого метода для следующего реагирующего элемента. Именно таким образом должны вести себя надежные звенья цепочки реагирующих элементов. Правда, методы, реагирующие на событие, как правило, и обрабатывают его. Но очень важно иметь в виду, что, если этого не происходит, следует организовать передачу события дальше по цепочке реагирующих элементов.

Мультисенсорная архитектура

Теперь, когда вы немного ознакомились с цепочкой реагирующих элементов, рассмотрим процесс обработки жестов. Как пояснялось ранее, жесты передаются по цепочке реагирующих элементов и встроены в события. Это означает, что код, обрабатывающий любой вид взаимодействия с мультисенсорным экраном, должен содержаться в объекте, находящемся в цепочке реагирующих элементов. Как правило, это предполагает две возможности выбора: встроить данный код в подкласс класса `UIView` или же в класс `UIViewController`. В связи с этим возникает вопрос: чему принадлежит код — представлению или контроллеру представления?

Если представление должно выполнять какие-то самостоятельные действия, исходя из факта касания пользователем экрана, то соответствующий код, вероятно, должен принадлежать классу, в котором определяется данное представление. Например, классы многих элементов управления, в том числе `UISwitch` и `UISlider`, реагируют на события, связанные с касанием экрана. Так, класс `UISwitch`, возможно, придется приводить в активное или неактивное состояние по факту касания. Разработчики класса `UISwitch` встроили в него код обработки жестов, и поэтому он способен реагировать на касание. Однако нередко обрабатываемый жест оказывает воздействие не только на тот объект, на котором произошло касание, и тогда код обработки жестов принадлежит классу контроллера представления. Так, если пользователь, касаясь одной строки, делает жест, который означает удаление всех строк, такой жест должен быть обработан кодом, находящимся в контроллере представления. Порядок реагирования на касания и жесты в обоих случаях остается одним и тем же, независимо от класса, к которому принадлежит код их обработки.

Четыре метода уведомления о касаниях

Для уведомления реагирующего элемента о касаниях имеются четыре метода. Когда пользователь впервые касается экрана мобильного устройства, система находит реагирующий элемент, у которого имеется метод touchesBegan(_ :withEvent:). Для того чтобы выяснить, когда пользователь только начинает жест или нажимает экран, следует реализовать данный метод в текущем представлении или в его контроллере. В листинге 18.2 приведен пример того, как может выглядеть такой метод.

Листинг 18.2. Распознавание начала жеста или касания

```
override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {
    if let touch = touches.first{
        let numTaps = touch.tapCount
        let numTouches = event?.allTouches()?.count
    }
    // Делаем что-нибудь еще
}
```

Каждый раз, когда пользователь в первый раз касается экрана пальцем, создается новый объект класса UITouch, соответствующий данному пальцу. Этот объект добавляется в множество, которое передается вместе с каждым объектом класса UIEvent и может быть получено с помощью вызова его метода allTouches(). Все последующие события, связанные с этим же пальцем, будут содержать *тот же самый экземпляр класса UITouch* в множестве, которое передается методу allTouches(), пока пользователь не отнимет палец от экрана. Этот экземпляр также будет содержаться в множестве touches, если с данным пальцем будет связано новое действие. Следовательно, чтобы отслеживать деятельность, связанную с конкретным пальцем, необходимо следить за объектом класса UITouch.

Определить количество пальцев, коснувшихся в настоящий момент экрана, можно по количеству подсчитанных объектов, возвращаемых методом allTouches(). Если событие сообщает о касании, являющемся частью целой серии касаний любым пальцем, то подсчитанное количество касаний можно определить из свойства tapCount объекта типа UITouch, соответствующего данному пальцу. Если экрана касается только один палец или же если это вообще все равно, то по запросу можно быстро получить объект типа UITouch, используя свойство first из структуры Set. В предыдущем примере значение 2 переменной numTaps обозначает, что произошло два быстрых нажатия экрана подряд, по крайней мере одним пальцем. Аналогично значение 2 переменной numTouches означает, что пользователь коснулся экрана двумя пальцами.

Не все объекты, содержащиеся в множестве touches или возвращаемые методом allTouches(), могут соответствовать представлению или контроллеру представления, в котором реализован рассматриваемый здесь метод. Например,

ячейку табличного представления, вероятно, мало должны заботить касания других строк таблицы или навигационной панели. Подмножество касаний, приходящихся только на конкретное представление, можно получить из события с помощью инструкции

```
let myTouches = event?.touchesForView(self.view)
```

Каждое событие типа `UITouch` соответствует отдельному пальцу, а каждый палец находится в разном положении на экране. Положение отдельного пальца на экране можно определить, используя объект типа `UITouch`. Он может даже преобразовать точку касания в локальную систему координат представления, если обратиться к нему следующим образом:

```
let point = touch.locationInView(self.view) // Точка типа CGPoint
```

Уведомление можно получить и в том случае, если пользователь движет пальцами по экрану, для чего достаточно реализовать метод `touchesMoved(_:withEvent:)`. Этот метод вызывается несколько раз в течение длинного скольжения пальцем по экрану. Всякий раз, когда этот метод вызывается, в итоге получается совсем другой ряд касаний и другое событие. Помимо определения текущего положения каждого пальца на экране из объектов типа `UITouch`, можно выявлять предыдущее местоположение данного касания, т.е. положение пальца, когда в последний раз вызывался метод `touchesMoved(_:withEvent:)` или `touchesBegan(_:withEvent:)`.

Когда пользователь отнимает пальцы от экрана, порождается очередное событие и вызывается уведомляющий о нем метод `touchesEnded(_:withEvent:)`. Если этот метод вызывается, значит, пользователь завершил свой жест. Последний метод, который имеет отношение к касаниям экрана и может быть реализован реагирующими элементами, называется `touchesCancelled(_:withEvent:)`. Этот метод вызывается в том случае, если где-то посредине жеста пользователя происходит его прерывание, например, входящим телефонным звонком. Именно в этот момент можно произвести любую необходимую очистку, чтобы начать с нового жеста. Когда данный метод вызывается, метод `touchesEnded(_:withEvent:)` не будет вызываться для текущего жеста.

Создание приложения TouchExplorer

В качестве первого примера рассмотрим небольшое приложение, которое дает более ясное представление о том, как упомянутые выше четыре метода, имеющие отношение к касаниям экрана, вызываются из реагирующих элементов. С этой целью создайте в среде Xcode новый проект, используя шаблон `Single View Application`. Введите в поле `Product Name` имя `TouchExplorer` и выберите пункт `Universal` в списке `Devices`. Приложение `TouchExplorer` должно выводить на экран сообщения, содержащие количество касаний и нажатий экрана, всякий раз, когда вызывается метод, имеющий отношение к касаниям

экрана. На устройствах с функцией 3D Touch можно даже определить силу, с которой палец касался экрана при последнем событии (рис. 18.1).



Рис. 18.1. Приложение TouchExplorer

ЗАМЕЧАНИЕ. Несмотря на то что приложения, рассматриваемые в этой главе, выполняются на симуляторе, вы не сможете увидеть все доступные мультисенсорные функции или возможности технологии 3D Touch до тех пор, пока не выполните эти приложения на настоящем мобильном устройстве, работающем под управлением iOS. Функция 3D Touch доступна только на устройствах iPhone 6s и iPhone 6s Plus.

Для рассматриваемого здесь приложения нам потребуются три метки: одна — для обозначения метода, вызывавшегося последним; другая — для сообщения о текущем количестве нажатий экрана; третья — для сообщения о текущем количестве касаний экрана. Щелкните на исходном файле `ViewController.swift` и введите три выделенных ниже полужирным выхода в классе контроллера представления.

```
class ViewController: UIViewController {
    @IBOutlet var messageLabel:UILabel!
    @IBOutlet var tapsLabel:UILabel!
    @IBOutlet var touchesLabel:UILabel!
    @IBOutlet var forceLabel:UILabel!
```

Выберите файл раскладовки Main.storyboard, чтобы отредактировать графический пользовательский интерфейс. В итоге появится пустое представление, обычное для всех новых проектов подобного рода. Нажмите клавишу <Option> и, удерживая ее, перетащите еще две метки, скопировав оригинал, располагая их одну под другой, как показано на рис. 18.1. Если вы чувствуете в себе художественные наклонности, поэкспериментируйте со шрифтами и цветами меток. После этого выберите нижнюю метку и с помощью инспектора атрибутов установите значение свойства Lines равным 0, поскольку мы планируем использовать его для демонстрации нескольких строк текста.

Затем нужно задать ограничения Auto Layout на компоновку меток. С этой целью перейдите в окно Document Outline, нажмите клавишу <Control>, перетащите указатель от первой метки к главному представлению и отпустите кнопку мыши. Нажмите клавишу <Shift> и выберите команды Vertical Spacing to Top Layout Guide и Leading Space to Container Margin из всплывающего меню, а затем нажмите клавишу <Return>. Сделайте то же самое и для двух других меток. Следующий шаг состоит в том, чтобы связать метки с их выходами. С этой целью нажмите клавишу <Control> и перетащите указатель от пиктограммы View Controller к каждой из трех меток, соединив верхнюю из них с выходом messageLabel, вторую — с выходом tapsLabel, третью — с выходом touchesLabel, а нижнюю — с выходом forceLabel. Затем дважды щелкните на каждой метке и нажмите клавишу <Delete>, чтобы избавиться от текста метки.

Далее щелкните один раз на фоне главного представления или на пиктограмме View в окне Document Outline, чтобы вызвать инспектор атрибутов (рис. 18.2). Перейдите в раздел View открывшегося окна инспектора и убедитесь в том, что установлены флагки User Interaction Enabled и Multiple Touch. Если флагок Multiple Touch не установлен, методы обработки касаний экрана в классе текущего контроллера будут всегда получать одно и только одно касание, независимо от того, сколькими пальцами пользователь на самом деле касается экрана.



Рис. 18.2. Убедитесь в том, что среди атрибутов представления установлены флагки User Interaction Enabled и Multiple Touch

По завершении перейдите к исходному файлу ViewController.swift и внесите в него следующие изменения (листинг 18.3).

Листинг 18.3. Файл ViewController.swift
с поддержкой приложения TouchExplorer

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.
}

private func updateLabelsFromTouches(_ touch: UITouch?,
                                      allTouches: Set<UITouch>?) {
    let numTaps = touch?.tapCount ?? 0
    let tapsMessage = "\((numTaps) taps detected"
    tapsLabel.text = tapsMessage

    let numTouches = allTouches?.count ?? 0
    let touchMsg = "\((numTouches) touches detected"
    touchesLabel.text = touchMsg

    if traitCollection.forceTouchCapability == .available {
        forceLabel.text = "Force: \(touch?.force ?? 0)\nMax force: \(touch?.
            maximumPossibleForce ?? 0)"
    } else {
        forceLabel.text = "3D Touch not available"
    }
}

override func touchesBegan(_ touches: Set<UITouch>,
                           with event: UIEvent?) {
    messageLabel.text = "Touches Began"
    updateLabelsFromTouches(touches.first,
                           allTouches: event?.allTouches())
}

override func touchesCancelled(_ touches: Set<UITouch>,
                               with event: UIEvent?) {
    messageLabel.text = "Touches Cancelled"
    updateLabelsFromTouches(touches.first,
                           allTouches: event?.allTouches())
}

override func touchesEnded(_ touches: Set<UITouch>,
                           with event: UIEvent?) {
    messageLabel.text = "Touches Ended"
    updateLabelsFromTouches(touches.first, allTouches: event?.allTouches())
}

override func touchesMoved(_ touches: Set<UITouch>,
                           with event: UIEvent?) {
    messageLabel.text = "Drag Detected"
    updateLabelsFromTouches(touches.first, allTouches: event?.allTouches())
}
```

В этом классе контроллера мы реализуем все четыре рассматривавшихся ранее метода, имеющих отношение к касаниям экрана. В каждом из них устанавливается переменная `messageLabel`, чтобы пользователь видел, когда именно вызывается каждый метод. Далее во всех четырех методах вызывается метод `updateLabelsFromTouches()` для обновления двух других меток. Метод `updateLabelsFromTouches()` получает подсчитанное количество нажатий экрана от одного из событий касания, определяет количество пальцев, коснувшихся экрана, анализируя подсчитанное число касаний, установленное во множестве `touches`, а затем обновляет метки этой информацией. Обратите внимание на следующий фрагмент кода.

```
if traitCollection.forceTouchCapability == .available {
    forceLabel.text = "Force: \(touch?.force ?? 0)\nMax force:
    \(touch?.maximumPossibleForce ?? 0)"
} else {
    forceLabel.text = "3D Touch not available"
}
```

Функция 3D Touch доступна не на всех устройствах, поэтому в первой строке кода для проверки наличия поддержки этой функции используется свойство `forceTouchCapability` класса `UITraitCollection`. Каждый контроллер представления имеет свою коллекцию характеристик, и в данном случае для проверки мы используем коллекцию характеристик единственного контроллера представления нашего приложения. Если функция 3D Touch поддерживается, то мы используем свойство `force` класса `UITouch`, чтобы выяснить, насколько сильно пользователь нажимает на экран в данный момент, а также свойство `maximumPossibleForce`, чтобы выяснить максимально возможную величину этой силы.

Скомпилируйте и запустите данное приложение на выполнение. Если вы выполняете его в симуляторе, попробуйте несколько раз щелкнуть на экране, чтобы привести в действие подсчет нажатий экрана. Затем попробуйте щелкнуть и, не отпуская кнопку мыши, перетащить указатель по представлению, чтобы сымитировать касание и скольжение пальцем по экрану.

Нажав клавишу `<Option>`, щелкнув кнопкой мыши и перетащив указатель, можете воспроизвести щипок двумя пальцами в симуляторе. Если вы нажмете клавишу `<Option>`, чтобы воспроизвести щипок, переместите указатель мыши, чтобы обозначить два виртуальных пальца рядом друг с другом, а затем нажмите клавишу `<Shift>`, не отпуская клавишу `<Option>`, то сможете сымитировать скольжение двумя пальцами по экрану мобильного устройства. При нажатии клавиши `<Shift>` фиксируется взаимное расположение двух пальцев, после чего можно имитировать скольжение и другие жесты, совершаемые двумя пальцами. Несмотря на то что вам не удастся сымитировать жесты тремя и более пальцами, вы сможете воспроизвести на симуляторе большинство жестов двумя пальцами, используя комбинацию клавиш `<Option+Shift>`.

Если вам удастся запустить данное приложение на мобильном устройстве, выясните, сколько касаний вы сможете зарегистрировать одновременно.

Попробуйте сначала скользнуть по экрану одним пальцем, а затем двумя и тремя. Кроме того, попробуйте нажать экран двумя и тремя пальцами, чтобы проверить, увеличивается ли количество подсчитываемых нажатий, если они совершаются двумя пальцами.

Экспериментируйте с приложением TouchExplorer до тех пор, пока не прочувствуете как следует происходящие в нем события и принцип действия всех четырех методов, имеющих отношение к касаниям экрана. Когда вы достигнете подобного ощущения, перейдите к следующему разделу, в котором показано, как обнаруживается один из самых распространенных жестов: скольжение пальцем по экрану.

Создание приложения **Swipes**

Следующее приложение, которое мы собираемся написать, предназначено исключительно для обнаружения скольжения пальцем по экрану (как по вертикали, так и по горизонтали). Когда вы проводите пальцем по экрану слева направо, справа налево, сверху вниз или снизу вверх, несколько мгновений спустя приложение отображает в верхней части экрана сообщение о том, что такое скольжение было обнаружено (рис. 18.3).



Рис. 18.3. Приложение Swipes обнаруживает как вертикальное, так и горизонтальное скольжение пальца по экрану

Обнаружение скольжения с помощью событий касания

Скольжение пальца по экрану обнаруживается относительно просто. Для этого мы должны определить минимальную длину такого жеста в пикселях, т.е. минимально допустимое расстояние, на которое пользователь должен провести пальцем по экрану, чтобы этот жест считался скольжением. Кроме того, нам нужно определить отклонение, т.е. минимально допустимое смещение, на которое пользователь может отклониться от прямой линии, чтобы его жест считался горизонтальным или вертикальным скольжением. Диагональная линия обычно считается не скольжением, а лишь незначительным отклонением от желательного перемещения по горизонтали или по вертикали.

Когда пользователь касается экрана, мы должны сохранить в переменной местоположение первого касания. Затем нужно проверить, перемещается ли палец пользователя по экрану настолько далеко и прямо, чтобы этот жест можно было считать скольжением. На самом деле для этой цели служит встроенный распознаватель жестов, но мы все же воспользуемся приобретенными знаниями о событиях касания, чтобы создать свой распознаватель жестов. Итак, приступим к его построению. Создайте новый проект в среде Xcode, используя шаблон Single View Application, и выберите пункт Universal в списке Devices. Присвойте новому проекту имя Swipes. Щелкните на исходном файле ViewController.swift и введите в нем строки кода, выделенные ниже полужирным шрифтом. В этих строках кода объявляются выход для новой метки и переменная для хранения места первого касания пользователем экрана.

```
class ViewController: UIViewController {
    @IBOutlet var label:UILabel!
    private var gestureStartPoint:CGPoint!
```

Дважды щелкните на файле Main.storyboard, чтобы открыть его для редактирования. Вызовите инспектор атрибутов и убедитесь в том, что для текущего представления установлены флагки User Interaction Enabled и Multiple Touch. Затем перетащите метку из библиотеки стандартных объектов в окно View. Выровняйте метку по центру и подберите по своему усмотрению текстовые атрибуты таким образом, чтобы метка легко читалась на экране. Перейдите в окно Document Outline, нажмите клавишу <Control>, перетащите указатель от метки к текущему представлению и отпустите кнопку мыши. Нажмите клавишу <Shift> и выберите команды Vertical Spacing to Top Layout Guide и Center Horizontally in Container из всплывающего меню, а затем нажмите клавишу <Return>. Нажмите клавишу <Control> и перетащите указатель от пиктограммы View Controller к метке, связав ее с выходом label. Дважды щелкните на метке и удалите ее первоначальный текст. Затем перейдите к исходному файлу ViewController.swift и введите и модифицируйте его, как показано в листинге 18.4.

Листинг 18.4. Изменения файла ViewController.

swift для поддержки приложения Swipes

```

class ViewController: UIViewController {
    @IBOutlet var label: UILabel!
    private var gestureStartPoint: CGPoint!
    private static let minimumGestureLength = Float(25.0)
    private static let maximumVariance = Float(5)

    override func viewDidLoad() {
        super.viewDidLoad()
        // Дополнительная настройка после загрузки приложения,
        // обычно из nib-файла
    }

    override func touchesBegan(_ touches: Set<UITouch>,
                               with event: UIEvent?) {
        if let touch = touches.first {
            gestureStartPoint = touch.location(in: self.view)
        }
    }

    override func touchesMoved(_ touches: Set<UITouch>,
                               with event: UIEvent?) {
        if let touch = touches.first,
           gestureStartPoint = self.gestureStartPoint {
            let currentPosition = touch.location(in: self.view)

            let deltaX = fabsf(Float(gestureStartPoint.x - currentPosition.x))
            let deltaY = fabsf(Float(gestureStartPoint.y - currentPosition.y))

            if deltaX >= ViewController.minimumGestureLength
                && deltaY <= ViewController.maximumVariance {
                label.text = "Horizontal swipe detected"
                DispatchQueue.main.after(when: DispatchTime.now() +
                    Double(Int64(2 * NSEC_PER_SEC)) /
                    Double(NSEC_PER_SEC)) {
                    self.label.text = ""
                }
            } else if deltaY >= ViewController.minimumGestureLength
                && deltaX <= ViewController.maximumVariance {
                label.text = "Vertical swipe detected"
                DispatchQueue.main.after(when: DispatchTime.now() +
                    Double(Int64(2 * NSEC_PER_SEC)) /
                    Double(NSEC_PER_SEC)) {
                    self.label.text = ""
                }
            }
        }
    }
}

```

Рассмотрим сначала метод `touchesBegan(_:withEvent:)`. В этом методе мы только извлекаем любое касание из множества `touches` и сохраняем его точку. В настоящий момент нас, прежде всего, интересует скольжение по экрану

одним пальцем, поэтому мы извлекаем лишь одно касание, не обращая внимания на их количество, как следует из приведенного ниже кода.

```
if let touch = touches.first {
    gestureStartPoint = touch.location(in: self.view)
}
```

Во множестве `touches`, передаваемом данному методу в качестве аргумента, используются объекты типа `UITouch` вместо тех, которые содержатся в объекте типа `UIEvent`, поскольку нас больше интересует отслеживание происходящих изменений, чем общее состояние всех активных касаний. В следующем методе, `touchesMoved(_:withEvent:)`, мы уже делаем нечто более существенное. Сначала получаем текущее положение пальца пользователя следующим образом:

```
if let touch = touches.first, gestureStartPoint = self.gestureStartPoint {
    let currentPosition = touch.location(in: self.view)
```

Здесь мы используем инструкцию `if` для проверки двух условий — произошло ли текущее касание и была ли сохранена ранее начальная точка жеста. На практике оба эти условия всегда выполняются, но тот факт, что свойство `touches.first`, которое мы используем здесь и в методе `touchesBegan(_:withEvent:)`, возвращает необязательное значение, означает, что мы обязаны выполнить проверки, чтобы убедиться, что приложение не потерпит крах при попытке распарковать нулевое необязательное значение в неожиданном событии.

После этого мы определяем, насколько далеко палец пользователя переместился как по горизонтали, так и по вертикали относительно исходного положения. Функция `fabsf()` принадлежит стандартной библиотеке математических функций на языке C и возвращает абсолютную величину числового значения с плавающей точкой. Это дает нам возможность вычесть одно числовое значение из другого, не особенно беспокоясь о том, какое из них больше:

```
let deltaX = fabsf(Float(gestureStartPoint.x - currentPosition.x))
let deltaY = fabsf(Float(gestureStartPoint.y - currentPosition.y))
```

Получив оба разностных значения, проверим, переместился ли пользователь достаточно далеко в одном направлении, но не слишком далеко в другом, чтобы считать такой жест скольжением пальца по экрану. Если это так, то задаем текст метки, чтобы сообщить о том, что было обнаружено горизонтальное или вертикальное скольжение. Кроме того, для стирания текста через две секунды после его появления на экране вызываем функцию `DispatchQueue.main.after()` из библиотеки GCD. Благодаря этому пользователь может поупражняться в скольжении пальцем по экрану, не особенно задумываясь о том, что именно обозначает метка: более раннюю или самую последнюю попытку.

```
if deltaX >= ViewController.minimumGestureLength
    && deltaY <= ViewController.maximumVariance {
    label.text = "Horizontal swipe detected"
```

```
DispatchQueue.main.after(when: DispatchTime.now() +
    Double(Int64(2 * NSEC_PER_SEC)) / Double(NSEC_PER_SEC)) {
    self.label.text = ""
} else if deltaY >= ViewController.minimumGestureLength
    && deltaX <= ViewController.maximumVariance {
    label.text = "Vertical swipe detected"
DispatchQueue.main.after(when: DispatchTime.now() +
    Double(Int64(2 * NSEC_PER_SEC)) / Double(NSEC_PER_SEC)) {
    self.label.text = ""
}
}
```

Скомпилируйте и запустите данное приложение на выполнение. Если вы обнаружите, что щелчки кнопкой мыши и перетаскивания указателя не дают никаких видимых результатов, наберитесь немного терпения. Щелкайте и перетаскивайте указатель прямо вниз или же прямо по горизонтали до тех пор, пока не научитесь имитировать скольжение пальца по экрану.

Автоматическое распознавание жестов

Процедура, использованная нами для распознавания скольжения пальца по экрану, сама по себе не так уж и плоха. Вся ее сложность заключается в методе `touchesMoved(_:withEvent:)`, хотя и он не такой уж сложный. Но то же самое можно сделать и более простым способом. В состав системы iOS входит класс `UIGestureRecognizer`, исключающий необходимость следить за всеми событиями, чтобы выяснить, каким образом пальцы двигаются по экрану. Вместо того чтобы использовать класс `UIGestureRecognizer` непосредственно, создается экземпляр одного из его подклассов, каждый из которых специально предназначен для обнаружения жеста определенного типа, в том числе скольжения, щипка, двойного или тройного нажатия и т.д. Посмотрим, каким образом можно видоизменить приложение `Swipes`, чтобы использовать в нем распознаватель жестов вместо процедуры, составленной нами вручную. Как всегда, можете сделать копию папки своего проекта `Swipes`, чтобы начать с нее новый проект.

Выберите сначала исходный файл `ViewController.swift` и удалите из него методы `touchesBegan(_:withEvent:)` и `touchesMoved(_:withEvent:)`, поскольку они нам больше не понадобятся. Затем добавьте вместо них два новых метода, как показано ниже.

```
func reportHorizontalSwipe(_ recognizer:UIGestureRecognizer) {
    label.text = "Horizontal swipe detected"
    DispatchQueue.main.after(when: DispatchTime.now() +
        Double(Int64(2 * NSEC_PER_SEC)) / Double(NSEC_PER_SEC)) {
    self.label.text = ""
}
}

func reportVerticalSwipe(_ recognizer:UIGestureRecognizer) {
    label.text = "Vertical swipe detected"
    DispatchQueue.main.after(when: DispatchTime.now() +
```

```

        Double(Int64(2 * NSEC_PER_SEC)) / Double(NSEC_PER_SEC)) {
    self.label.text = ""
}
}

```

Приведенные выше методы реализуют конкретные “функциональные” (если можно так выразиться) возможности, обусловленные жестами скольжения, аналогично рассматривавшему ранее методу touchesMoved(_:withEvent:). Теперь введите в метод viewDidLoad() следующий код.

```

super.viewDidLoad()
// Дополнительная настройка после загрузки представления,
// обычно из nib-файла.

let vertical = UISwipeGestureRecognizer(target: self,
                                         action: "reportVerticalSwipe:")
vertical.direction = [.up, .down]
view.addGestureRecognizer(vertical)

let horizontal = UISwipeGestureRecognizer(target: self,
                                         action: "reportHorizontalSwipe:")
horizontal.direction = [.left, .right]
view.addGestureRecognizer(horizontal)

```

В данном методе создаются лишь два распознавателя жестов: один — для обнаружения вертикального движения по экрану, а другой — горизонтального. Когда один из них распознает жест, на который он настроен, вызывается метод reportVerticalSwipe() или reportHorizontalSwipe() и соответственно устанавливается текст метки. Для дополнительной “очистки” можно также удалить строки кода с объявлением свойства gestureStartPoint и двух констант из исходного файла ViewController.swift. Теперь постройте и запустите данное приложение на выполнение, чтобы испытать новые распознаватели жестов.

Если исходить из общего количества строк кода, то применение распознавателей жестов мало чем отличается от предыдущего подхода, если речь идет о столь простом приложении, как рассматриваемое здесь. Но код, в котором применяются распознаватели жестов, безусловно, проще для понимания и написания. Ведь вам не нужно даже задумываться над проблемой расчета движений пальцами по экрану во времени, поскольку это уже сделано в классе UISwipeGestureRecognizer. Более того, система распознавания жестов от компании Apple допускает расширение. Это означает, что если в приложении требуется распознавать по-настоящему сложные жесты, не воспринимаемые ни одним из стандартных распознавателей, можно создать свой распознаватель, вынеся весь его сложный код в отдельный класс, как было показано выше, чтобы не засорять им код контроллера представления. Характерный тому пример будет продемонстрирован далее в этой главе. Между тем, запустив рассматриваемую здесь версию данного приложения на выполнение, вы обнаружите, что оно ведет себя таким же образом, как и предыдущая его версия.

Распознавание скольжения несколькими пальцами по экрану

В приложении *Swipes* мы просто выбираем любой объект из множества `touches`, чтобы выяснить местоположение пальца пользователя, когда он проводит им по экрану. Такой подход вполне допустим в том случае, если нас интересует распознавание скольжения по экрану только одним пальцем, т.е. наиболее распространенных жестов данного типа. Но что если нам требуется обрабатывать скольжение по экрану двумя или тремя пальцами? В предыдущих изданиях этой книги мы потратили около 50 строк кода и немало текстовых строк пояснения, чтобы добиться желаемого результата, отслеживая многие экземпляры объекта типа `UITouch` среди множества событий касания. Правда, теперь, когда в нашем распоряжении имеются распознаватели жестов, данная проблема может считаться решенной. Класс `UISwipeGestureRecognizer` может быть настроен на распознавание любого количества одновременных касаний экрана. По умолчанию каждый его экземпляр ожидает касания одним пальцем, но его можно настроить на обнаружение одновременных касаний экрана любым количеством пальцев. Каждый экземпляр этого класса реагирует только на точно заданное количество касаний. Поэтому в качестве обновления рассматриваемого здесь приложения создадим в цикле целый ряд распознавателей жестов.

Сначала сделайте еще одну копию папки проекта *Swipes*. Затем отредактируйте исходный файл `ViewController.swift`, заменив код в теле метода `viewDidLoad()` следующим кодом.

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // как правило, из nib-файла

    for touchCount in 0..<5 {
        let vertical = UISwipeGestureRecognizer(target: self,
            action: #selector(ViewController.reportVerticalSwipe(_:)))
        vertical.direction = [.up, .down]
        vertical.numberOfTouchesRequired = touchCount
        view.addGestureRecognizer(vertical)

        let horizontal = UISwipeGestureRecognizer(target: self,
            action: #selector(ViewController.reportHorizontalSwipe(_:)))
        horizontal.direction = [.left, .right]
        horizontal.numberOfTouchesRequired = touchCount
        view.addGestureRecognizer(horizontal)
    }
}
```

Здесь мы добавляем в представление 10 разных распознавателей жестов: первый — для распознавания вертикального скольжения одним пальцем, второй — для распознавания вертикального скольжения двумя пальцами и т.д. Все они вызывают метод `reportVerticalSwipe()`, когда распознают соответствующие жесты. Второй набор распознавателей обрабатывает горизонтальное скольжение и при этом вызывается метод `reportHorizontalSwipe()`. Следует иметь в

виду, что в реальном приложении для реализации иных видов поведения, возможно, придется отслеживать другое количество пальцев, которыми одновременно проводят по экрану. Это нетрудно сделать, используя распознаватели жестов и предоставив каждому из них возможность вызывать другой метод действия.

Теперь нам нужно лишь изменить порядок регистрации, введя метод, представляющий нам удобное описание количества касаний, чтобы использовать эти сведения в рассматриваемых здесь методах уведомления о касаниях. Введите приведенный ниже код данного метода в конце класса `ViewController` прямо перед методами, распознающими скольжение двумя пальцами по экрану.

```
func descriptionForTouchCount(_ touchCount:Int) -> String {
    switch touchCount {
        case 1:
            return "Single"
        case 2:
            return "Double"
        case 3:
            return "Triple"
        case 4:
            return "Quadruple"
        case 5:
            return "Quintuple"
        default:
            return ""
    }
}
```

Внесите приведенные ниже изменения в оба метода уведомления о касаниях.

```
func reportHorizontalSwipe(_ recognizer:UIGestureRecognizer) {
    label.text = "Horizontal swipe detected"
    let count = descriptionForTouchCount(recognizer.numberOfTouches())
    label.text = "\((count)-finger horizontal swipe detected"
    DispatchQueue.main.after(when: DispatchTime.now() +
        Double(Int64(2 * NSEC_PER_SEC)) / Double(NSEC_PER_SEC)) {
        self.label.text = ""
    }
}

func reportVerticalSwipe(_ recognizer:UIGestureRecognizer) {
    label.text = "Vertical swipe detected"
    let count = descriptionForTouchCount(recognizer.numberOfTouches())
    label.text = "\((count)-finger vertical swipe detected"
    DispatchQueue.main.after(when: DispatchTime.now() +
        Double(Int64(2 * NSEC_PER_SEC)) / Double(NSEC_PER_SEC)) {
        self.label.text = ""
    }
}
```

Скомпилируйте и запустите на выполнение эту версию рассматриваемого здесь приложения. Вы должны суметь инициировать скольжение по экрану двумя и тремя пальцами в обоих направлениях, но в то же время сохранить возможность скольжения одним пальцем. Если же у вас маленькие пальцы, вы сможете даже инициировать скольжение по экрану четырьмя или пятью пальцами.

ПОДСКАЗКА. Если нажать клавишу <Option>, то в симуляторе появится пара точек, символизирующая пару пальцев. Сведите их вместе, а затем нажмите клавишу <Shift>. Тогда точки зафиксируют свое положение одна относительно другой, позволяя вам перемещать пару пальцев по всему экрану. Щелкните на них и перетащите вниз по экрану, симулируя двойное скольжение.

Скользя по экрану несколькими пальцами, будьте внимательны, чтобы ваши пальцы не располагались слишком близко один к другому. Так, если два пальца находятся очень близко один к другому, их движения могут быть зарегистрированы как одиночное касание. Вследствие этого не рекомендуется особенно полагаться на скольжение по экрану четырьмя или пятью пальцами для применения любых важных жестов в приложении, поскольку у многих людей слишком большие пальцы, чтобы эффективно делать подобные жесты. Кроме того, распознавание на мобильном устройстве iPad некоторых жестов четырьмя или пятью пальцами активизируется на уровне системы по умолчанию для перехода между приложениями и к начальному экрану. И хотя такие жесты могут быть отменены в приложении *Settings*, ими все же лучше не пользоваться в своих приложениях.

Распознавание многократных нажатий экрана

В приложении *TouchExplorer* мы выводили подсчитанное количество нажатий экрана. Следовательно, вы уже знаете, насколько просто обнаруживаются многократные нажатия экрана. Однако в реальном приложении далеко не все бывает так просто, как кажется на первый взгляд, поскольку нередко требуется предпринимать разные действия в зависимости от количества нажатий. Если пользователь трижды нажимает экран, система уведомляется об этом три раза по отдельности: когда происходит однократное, двойное и наконец тройное нажатие. Поэтому если в приложении требуется сделать что-нибудь одно при двойном нажатии и нечто совсем другое при тройном нажатии, то наличие трех отдельных уведомлений о нажатиях может вызвать серьезные осложнения, поскольку сначала получается уведомление о двойном, а затем о тройном нажатии. Если не написать свой искусный код, принимающий это во внимание, то в конечном итоге придется выполнить оба действия. Правда, разработчики из компании Apple предвидели подобную ситуацию и предоставили специальный механизм, позволяющий некоторым распознавателям жестов нормально взаимодействовать друг с другом, даже если им приходится иметь дело с неоднозначными входными данными, способными инициировать любой из них. В основу такого механизма положен следующий принцип: на распознаватель жестов накладывается ограничение, запрещающее ему запускать на выполнение связанный с ним метод, если только это не удастся сделать какому-то другому распознавателю жестов.

Данный принцип кажется немного абстрактным, поэтому переведем его в практическую плоскость. Одним из наиболее часто применяемых распознавателей

жестов является класс `UITapGestureRecognizer`. Этот распознаватель нажатий может быть настроен на выполнение своих функций, когда происходит конкретное количество нажатий экрана. Допустим, имеется представление, для которого нужно определить отдельные действия, совершаемые в тех случаях, когда пользователь нажимает экран один или два раза. Реализацию подобного алгоритма работы приложения можно было бы начать, например, со следующего фрагмента кода:

```
let singleTap = UITapGestureRecognizer(target: self,
action: #selector(ViewController.singleTap))
singleTap.numberOfTapsRequired = 1
singleTap.numberOfTouchesRequired = 1
view.addGestureRecognizer(singleTap)

let doubleTap = UITapGestureRecognizer(target: self,
action: #selector(ViewController.doubleTap))
doubleTap.numberOfTapsRequired = 2
doubleTap.numberOfTouchesRequired = 1
view.addGestureRecognizer(doubleTap)
```

Недостаток этого фрагмента кода заключается в том, что оба распознавателя жестов ничего не знают ни о существовании друг друга, ни о том, что действия пользователя могут больше подойти другому распознавателю. Если реализовать приведенный выше фрагмент кода в приложении, где пользователь дважды нажмет экран с текущим представлением, то в конечном итоге будет вызван метод `doDoubleTap()`, но также будет вызван метод `doSingleMethod()`, причем **дважды**: по одному разу на каждое нажатие.

Данный недостаток можно преодолеть, создав запрос на сбой программы. В частности, мы можем дать распознавателю `singleTap` команду инициировать свое действие только в том случае, если распознаватель `doubleTap` не распознает и не отреагирует на введенные пользователем данные. Это можно сделать в одной строке кода, как показано ниже.

```
singleTap.require(toFail: doubleTap)
```

Это означает, что, если пользователь нажмет экран один раз, распознаватель `singleTap` не сразу приступит к своим обязанностям. Вместо этого он будет ждать до тех пор, пока не узнает, что распознаватель `doubleTap` решил не обращать больше внимания на текущий жест, поскольку пользователь не нажал экран дважды. Именно на таком алгоритме работы мы и попробуем построить приложение в следующем проекте.

Итак, создайте в среде Xcode новый проект с шаблоном *Single View Application*, присвойте новому проекту имя *Taps* и выберите пункт *Universal* в списке *Devices*. У рассматриваемого здесь приложения должно быть четыре метки, уведомляющие о том, что ему удалось обнаружить однократное, двойное, тройное и четырехкратное нажатие экрана (рис. 18.4).

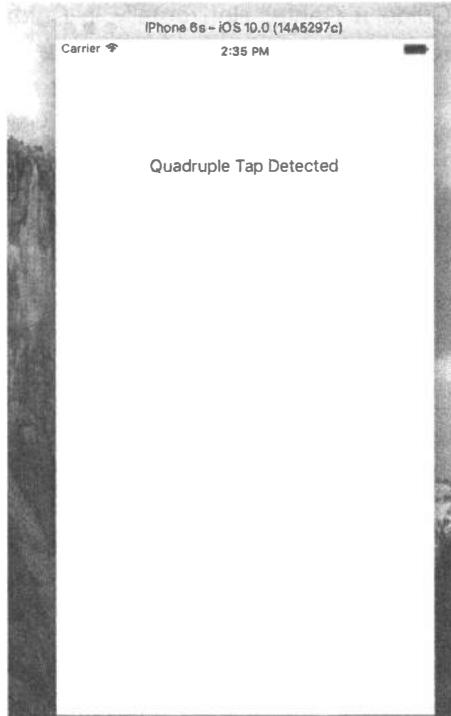


Рис. 18.4. Приложение TapTaps обнаруживает до четырех последовательных нажатий экрана

Для четырех меток нам нужны выходы, а также отдельные методы для каждого варианта нажатия экрана, чтобы сымитировать то, что может произойти в реальном приложении. Кроме того, нам нужно ввести в данное приложение отдельный метод для стирания текстовых полей. С этой целью откройте исходный файл ViewController.swift и внесите в него изменения, выделенные ниже полужирным шрифтом.

```
class ViewController: UIViewController {
    @IBOutlet var singleLabel:UILabel!
    @IBOutlet var doubleLabel:UILabel!
    @IBOutlet var tripleLabel:UILabel!
    @IBOutlet var quadrupleLabel:UILabel!
```

Сохраните исходный файл ViewController.swift и откройте файл раскладовки Main.storyboard, чтобы отредактировать графический пользовательский интерфейс. Добавьте четыре метки в текущее представление из библиотеки стандартных объектов и расположите их одну под другой. Перейдите сначала к инспектору атрибутов и установите выравнивание каждой метки по центру (Center). Затем перейдите в окно Document Outline, нажмите клавишу <Control>, перетащите указатель от верхней метки к ее родительскому представлению и отпустите кнопку мыши. Нажмите клавишу <Shift> и выберите команды Vertical

Spacing to Top Layout Guide и Center Horizontally in Container из всплывающего меню, а затем нажмите клавишу <Return>. Сделайте то же самое и для трех других меток, чтобы задать ограничения Auto Layout на их компоновку. По завершении нажмите клавишу <Control> и перетащите указатель от пиктограммы View Controller к каждой метке в отдельности, связав каждую из них с выходами singleLabel, doubleLabel, tripleLabel и quadrupleLabel соответственно. Щелкните дважды на каждой метке и нажмите клавишу <Delete>, чтобы избавиться от первоначального текста меток. Выберите исходный файл ViewController.swift и введите в нем строки кода из листинга 18.5.

Листинг 18.5. Изменения файла ViewController.

swift для поддержки приложения Taps

```

override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // как правило, из nib-файла

    let singleTap = UITapGestureRecognizer(target: self,
                                          action: #selector(ViewController.singleTap))
    singleTap.numberOfTapsRequired = 1
    singleTap.numberOfTouchesRequired = 1
    view.addGestureRecognizer(singleTap)

    let doubleTap = UITapGestureRecognizer(target: self,
                                         action: #selector(ViewController.doubleTap))
    doubleTap.numberOfTapsRequired = 2
    doubleTap.numberOfTouchesRequired = 1
    view.addGestureRecognizer(doubleTap)
    singleTap.require(toFail: doubleTap)

    let tripleTap = UITapGestureRecognizer(target: self,
                                         action: #selector(ViewController.tripleTap))
    tripleTap.numberOfTapsRequired = 3
    tripleTap.numberOfTouchesRequired = 1
    view.addGestureRecognizer(tripleTap)
    doubleTap.require(toFail: tripleTap)

    let quadrupleTap = UITapGestureRecognizer(target: self,
                                             action: #selector(ViewController.quadrupleTap))
    quadrupleTap.numberOfTapsRequired = 4
    quadrupleTap.numberOfTouchesRequired = 1
    view.addGestureRecognizer(quadrupleTap)
    tripleTap.require(toFail: quadrupleTap)
}

func singleTap() {
    showText("Single Tap Detected", inLabel: singleLabel)
}

func doubleTap() {
    showText("Double Tap Detected", inLabel: doubleLabel)
}

```

```

func tripleTap() {
    showText("Triple Tap Detected", inLabel: tripleLabel)
}

func quadrupleTap() {
    showText("Quadruple Tap Detected", inLabel: quadrupleLabel)
}

private func showText(_ text: String, inLabel label: UILabel) {
    label.text = text
    DispatchQueue.main.after(when: DispatchTime.now() +
        Double(Int64(2 * NSEC_PER_SEC)) / Double(NSEC_PER_SEC)) {
        label.text = ""
    }
}

```

Четыре приведенных выше метода обработки нажатия не делают в данном приложении ничего, кроме установки одной из четырех меток и стирания той же самой метки через 2 секунды с помощью метода `DispatchQueue.main.after()`. Но самое интересное происходит в самом методе `viewDidLoad()`. Его код начинается довольно просто: с установки распознавателя нажатий и его присоединения к текущему представлению, как показано ниже.

```

let singleTap = UITapGestureRecognizer(target: self,
    action: #selector(ViewController.singleTap))
singleTap.numberOfTapsRequired = 1
singleTap.numberOfTouchesRequired = 1
view.addGestureRecognizer(singleTap)

```

Обратите внимание на то, что мы задаем количество *нажатий* (поочередных касаний в одном и том же положении), необходимых для инициирования действия, а также количество *касаний* (числа пальцев, одновременно касающихся экрана) равным 1. После этого мы устанавливаем еще один распознаватель жестов для обработки двойного нажатия экрана, как показано ниже.

```

let doubleTap = UITapGestureRecognizer(target: self,
    action: #selector(ViewController.doubleTap))
doubleTap.numberOfTapsRequired = 2
doubleTap.numberOfTouchesRequired = 1
view.addGestureRecognizer(doubleTap)
singleTap.require(toFail: doubleTap)

```

Этот код очень похож на предыдущий распознаватель, кроме последней строки, в которой мы передаем распознавателю `singleTap` некоторый дополнительный контекст. По существу, мы даем распознавателю `singleTap` команду инициировать свое действие только при условии, что какой-нибудь другой распознаватель (в данном случае — `doubleTap`) решит, что данные, введенные в настоящий момент пользователем, не относятся к тому, что он ожидает получить.

Давайте подумаем, что все это означает. Когда оба распознавателя жестов находятся на своих местах, однократное нажатие экрана в текущем представлении вызовет мгновенную реакцию у распознавателя `singleTap`, который посчитает,

что этот жест имеет отношение к нему. В то же время распознаватель double Tap посчитает, что данный жест *может* иметь отношение к нему. Поскольку распознаватель singleTap настроен на ожидание “отказа” распознавателя doubleTap, он не сразу передает свой метод действия, а ожидает, что же произойдет с распознавателем doubleTap.

Если после первого нажатия сразу же произойдет второе, то распознаватель doubleTap посчитает, что этот жест точно имеет к нему отношение, и тотчас инициирует свое действие. В этот момент распознавателю singleTap станет ясно, что именно произошло, и он откажется от обработки данного жеста. Если через определенный промежуток времени, который определен в системе как максимальный промежуток времени между последовательными нажатиями при двойном нажатии, распознаватель doubleTap откажется от обработки данного жеста, распознаватель singleTap обнаружит этот отказ и, наконец, инициирует свое действие по обработке данного события. В остальной части рассматриваемого здесь метода определяются распознаватели жестов для трех и четырех нажатий. В каждом случае один распознаватель жестов настраивается на зависимость от отказа другого, как показано ниже.

```
let tripleTap = UITapGestureRecognizer(target: self,
    action: #selector(ViewController.tripleTap))
tripleTap.numberOfTapsRequired = 3
tripleTap.numberOfTouchesRequired = 1
view.addGestureRecognizer(tripleTap)
doubleTap.require(toFail: tripleTap)

let quadrupleTap = UITapGestureRecognizer(target: self,
    action: #selector(ViewController.quadrupleTap))
quadrupleTap.numberOfTapsRequired = 4
quadrupleTap.numberOfTouchesRequired = 1
view.addGestureRecognizer(quadrupleTap)
tripleTap.require(toFail: quadrupleTap)
```

Обратите внимание на то, что нам не нужно явно настраивать распознаватель жестов на зависимость от отказа каждого из распознавателей жестов с большим количеством нажатий экрана. Такая множественная зависимость возникает естественным путем в результате цепочки отказов, установленных в данном коде. Так, если распознавателю singleTap требуется отказ распознавателя double Tap, то распознавателю doubleTap — отказ распознавателя tripleTap, а тому — отказ распознавателя quadrupleTap. Таким образом, распознавателю singleTap требуется отказ всех остальных распознавателей.

Скомпилируйте и запустите на выполнение данное приложение. В зависимости от того, какое нажатие экрана вы совершили (однократное, двойное, тройное или четырехкратное), на экране появится лишь одна соответствующая метка. Приблизительно через полторы секунды метка исчезнет, и можно будет сделать следующую попытку.

Распознавание щипков и вращения

Еще одним распространенным жестом является щипок двумя пальцами по экрану. Этот жест применяется в целом ряде стандартных приложений, включая Mobile Safari, Mail и Photos. С его помощью можно увеличивать масштаб изображения, разводя пальцы в стороны, или же уменьшать, сводя пальцы вместе. Обнаруживаются щипки довольно просто благодаря классу UIPinchGestureRecognizer. Этот класс относится к числу **распознавателей непрерывных жестов**, поскольку он неоднократно вызывает свой метод действия во время щипка. По ходу выполнения данного жеста распознаватель щипков переходит в целый ряд состояний. Однако нас может заинтересовать лишь одно состояние — UIGestureRecognizerState.began, — в котором оказывается данный распознаватель, когда в первый раз вызывает метод действия после обнаружения факта совершения щипка. В этот момент свойство scale распознавателя щипковых жестов всегда устанавливается равным 1,0, а в остальной части жеста данное числовое значение может как увеличиваться, так и уменьшаться. Именно этим значением свойства scale мы и воспользуемся, чтобы изменить размеры изображения. И в конечном итоге состояние изменится на UIGestureRecognizerState.ended.

Другим распространенным жестом является вращение двумя пальцами. Этот непрерывный жест также обнаруживается распознавателем непрерывных жестов, реализованным в классе UIRotationGestureRecognizer. В нем имеется свойство rotation, которое в момент начала вращения равно 0,0, а затем, когда пользователь вращает пальцами, оно изменяется от 0,0 до 2,0*Pi. Итак, создайте новый проект в среде Xcode, используя снова шаблон Single View Application, и присвойте ему имя PinchMe. Перетащите изображение из файла yosemite-meadow.pgn, находящегося в папке 18 — Image с исходным кодом, прилагаемым к этой книге, или какую-нибудь другую фотографию из своего архива в каталог ресурсов Assets.xcassets данного проекта. Внесите в исходный файл ViewController.swift изменения, приведенные в листинге 18.6.

Листинг 18.6. Изменения файла ViewController.

swift для поддержки приложения PinchMe

```
class ViewController: UIViewController, UIGestureRecognizerDelegate {
    private var imageView: UIImageView!
    private var scale = CGFloat(1)
    private var previousScale = CGFloat(1)
    private var rotation = CGFloat(0)
    private var previousRotation = CGFloat(0)

    override func viewDidLoad() {
        super.viewDidLoad()
        // Дополнительная настройка после загрузки представления,
        // обычно из nib-файла.
```

```

let image = UIImage(named: "yosemite-meadows")
imageView = UIImageView(image: image)
imageView.isUserInteractionEnabled = true
imageView.center = view.center
view.addSubview(imageView)

let pinchGesture = UIPinchGestureRecognizer(target: self,
    action: #selector(ViewController.doPinch(_:)))
pinchGesture.delegate = self
imageView.addGestureRecognizer(pinchGesture)

let rotationGesture = UIRotationGestureRecognizer(target: self,
    action: #selector(ViewController.doRotate(_:)))
rotationGesture.delegate = self
imageView.addGestureRecognizer(rotationGesture)
}

func gestureRecognizer(_ gestureRecognizer: UIGestureRecognizer,
    shouldRecognizeSimultaneouslyWith
    otherGestureRecognizer: UIGestureRecognizer) -> Bool {
    return true
}

func transformImageView() {
    var t = CGAffineTransform(scaleX: scale * previousScale,
        y: scale * previousScale)
    t = t.rotate(rotation + previousRotation)
    imageView.transform = t
}

func doPinch(_ gesture: UIPinchGestureRecognizer) {
    scale = gesture.scale
    transformImageView()
    if gesture.state == .ended {
        previousScale = scale * previousScale
        scale = 1
    }
}

func doRotate(_ gesture: UIRotationGestureRecognizer) {
    rotation = gesture.rotation
    transformImageView()
    if gesture.state == .ended {
        previousRotation = rotation + previousRotation
        rotation = 0
    }
}
}

```

Сначала мы определяем четыре переменные экземпляра для текущего и предыдущего масштаба и вращения, как показано ниже. Предыдущими значениями считаются те, с которых началось и завершилось предыдущее действие распознавателя жестов. Мы должны отслеживать эти значения, потому что распознаватели жестов типа UIPinchGestureRecognizer и UIRotationGestureRecognizer всегда начинают действовать со значений, задаваемых по умолчанию: 1,0 — для масштабирования и 0,0 — для вращения соответственно.

Затем в методе `viewDidLoad()` мы начинаем с создания представления типа `UIImageView` для распознавания щипка и вращения, загрузки в него выбранного изображения и его расположения по центру главного представления, как показано ниже. Следует не забыть предоставить пользователю возможность взаимодействия с представлением изображения, потому что класс `UIImageView` — один из немногих классов из каркаса `UIKit`, в которых взаимодействие с пользователем по умолчанию отключено.

```
let image = UIImage(named: "yosemite-meadows")
imageView = UIImageView(image: image)
imageView.userInteractionEnabled = true
imageView.center = view.center
view.addSubview(imageView)
```

Далее мы настраиваем распознаватели щипков и вращения и даем им команду уведомлять нас об этих жестах с помощью методов `doPinch()` и `doRotation()` соответственно. Кроме того, мы указываем им использовать текущий экземпляр по ссылке `self` в качестве своего делегата следующим образом:

```
let pinchGesture = UIPinchGestureRecognizer(target: self,
    action: #selector(ViewController.doPinch(_:)))
pinchGesture.delegate = self
imageView.addGestureRecognizer(pinchGesture)

let rotationGesture = UIRotationGestureRecognizer(target: self,
    action: #selector(ViewController.doRotate(_:)))
rotationGesture.delegate = self
imageView.addGestureRecognizer(rotationGesture)
```

Единственный метод `gestureRecognizer(_:shouldRecognizeSimultaneouslyWithGestureRecognizer:)` из протокола `UIGestureRecognizerDelegate`, который мы должны реализовать, как показано ниже, всегда возвращает логическое значение `true`, чтобы щипки и вращения распознавались одновременно. В противном случае распознаватель жеста, начавший действовать первым, заблокирует другой распознаватель.

```
func gestureRecognizer(_ gestureRecognizer: UIGestureRecognizer,
    shouldRecognizeSimultaneouslyWith
        otherGestureRecognizer: UIGestureRecognizer) -> Bool {
    return true
}
```

Далее мы реализуем вспомогательный метод, чтобы преобразовать представление изображения в соответствии с текущим коэффициентом масштабирования и углом вращения, заданными распознавателями жестов, как показано ниже. Обратите внимание на то, что предыдущий и текущий коэффициенты масштабирования перемножаются, а углы вращения складываются. Это дает возможность согласовать щипок и вращение для нового жеста, который начинается со значения коэффициента масштабирования и угла вращения, равных по умолчанию 1,0 и 0,0 соответственно.

710 ГЛАВА 18 ❁ НАЖАТИЯ, КАСАНИЯ И ЖЕСТЫ

```
func transformImageView() {
    var t = CGAffineTransform(scaleX: scale * previousScale, y: scale * previousScale)
    t = t.rotate(rotation + previousRotation)
    imageView.transform = t
}
```

В заключение мы реализуем методы действия, получающие аргументы от распознавателей жестов и обновляющие графическое представление. В методах `doPinch()` и `doRotate()` мы сначала извлекаем новый коэффициент масштабирования или угол вращения, а затем выполняем преобразование представления изображения. Если жесты больше не распознаются, т.е. свойство `state` содержит состояние `UIGestureRecognizerState.Ended`, мы сохраняем правильные значения текущего коэффициента масштабирования или угла вращения и восстанавливаем их значения по умолчанию 1,0 и 0,0 соответственно.

```
func doPinch(_ gesture: UIPinchGestureRecognizer) {
    scale = gesture.scale
    transformImageView()
    if gesture.state == .ended {
        previousScale = scale * previousScale
        scale = 1
    }
}

func doRotate(_ gesture: UIRotationGestureRecognizer) {
    rotation = gesture.rotation
    transformImageView()
    if gesture.state == .ended {
        previousRotation = rotation + previousRotation
        rotation = 0
    }
}
```

Это все, что касается распознавания щипка и вращения. Скомпилируйте и запустите данное приложение на выполнение. Сделайте несколько щипков и вращений пальцами, чтобы посмотреть, как изменяется изображение (рис. 18.5). Если пользуетесь симулятором, помните, что щипки имитируются нажатием клавиши <Option>, щелчком в окне симулятора и перетаскиванием указателя мыши.

Резюме

Теперь вам должен быть понятен механизм, применяемый в системе iOS, для уведомления вашего приложения о касаниях, нажатиях и прочих жестах пользователя. В этой главе вы научились обнаруживать жесты, наиболее часто применяемые в приложениях под управлением iOS. Кроме того, вы ознакомились с примерами использования технологии 3D Touch. Более подробную информацию можно найти в документации компании Apple по адресу <https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/Adopting3DTouchOniPhone/>.



Рис. 18.5. Приложение PinchMe, обнаруживающее щипок и жест вращения

Пользовательский интерфейс iOS опирается на жесты, чтобы значительно упростить его применение, поэтому рассмотренные здесь способы распознавания и создания жестов пригодятся вам при разработке большинства собственных приложений под управлением iOS. В следующей главе мы покажем, как определить свои координаты с помощью каркаса Core Location.

ГЛАВА 19



Определение местоположения

Каждое устройство, работающее под управлением системы iOS, обладает способностью определять свое географическое местоположение, используя для этой цели каркас Core Location. В состав iOS входит также каркас Map Kit, позволяющий легко создавать интерактивные карты, показывающие ваше текущее местоположение. В этой главе мы рассмотрим оба упомянутых каркаса. В своей работе каркас Core Location может опираться на следующие три технологии: GPS (Global Positioning System — Глобальная система навигации и определения местоположения), Cell ID Location (определение координат по идентификатору сотовой связи с помощью триангуляции) и WPS (Wi-Fi Positioning Service — Служба определения местоположения средствами беспроводной связи). Самой точной из всех трех считается система GPS, но она недоступна для первого поколения мобильных устройств iPhone, iPod touch и беспроводных iPad. Короче говоря, любое мобильное устройство как минимум третьего поколения содержит модуль GPS. Система GPS принимает сигналы в диапазоне ультракоротких волн от многих спутников для определения текущего местоположения.

ЗАМЕЧАНИЕ. Формально в компании Apple применяется вариант системы GPS под названием Assisted GPS или A-GPS. В системе A-GPS используются сетевые ресурсы, помогающие повысить производительность автономной системы GPS. Основной принцип действия этой системы состоит в том, что поставщик услуг телефонной связи развертывает службы в своей сети, где мобильные устройства автоматически обнаруживают и собирают некоторые данные. Это дает им возможность определить свое исходное местоположение намного быстрее, чем в том случае, если бы они полагались только на спутники системы GPS.

Технология Cell ID Location основана на определении расстояния до сотовой базовой станции, с которой устройство контактирует в данный момент. Поскольку каждая сотовая базовая станция имеет довольно большую зону действия, при

этом возникает довольно большая ошибка. Кроме того, эта технология требует наличия сотовой радиосвязи, поэтому она действует только в мобильных устройствах iPhone (во всех моделях, включая самую первую) и iPad с 3G-соединением. В системе WPS используются MAC-адреса ближайших точек доступа Wi-Fi для приблизительного определения местоположения путем обращения к крупной базе данных известных поставщиков услуг беспроводной связи и зоны их обслуживания. Система WPS не очень точна и может ошибаться на многие километры, определяя местоположение.

Все три описанных выше способа определения местоположения требуют немало энергии заряда батареи питания мобильного устройства, поэтому данное обстоятельство следует иметь в виду, используя Core Location. Ваше приложение не должно опрашивать местоположение чаще, чем в этом есть абсолютная необходимость. Используя Core Location, вы можете указать желательную точность определения местоположения. Тщательно продумав абсолютно необходимый минимум такой точности, вы существенно сэкономите заряд батареи питания своего мобильного устройства. Выбор технологии, от которых зависит нормальное функционирование Core Location, недоступен вашему приложению. Вы не можете предписать каркасу Core Location использовать технологию GPS, триангуляцию или WPS и можете только указать желательную точность определения местоположения, а каркас Core Location сам выберет ту технологию, которая лучше всего подходит для выполнения вашего запроса.

Диспетчер местоположения

С интерфейсом прикладного программирования каркаса Core Location на самом деле очень легко работать. Основным классом, с которым нам чаще всего придется иметь дело, является CLLocationManager, который обычно называется **диспетчером местоположения** (location manager). Для взаимодействия с Core Location нам придется создать экземпляр диспетчера местоположения:

```
let locationManager = CLLocationManager()
```

В этой строке кода автоматически создается экземпляр диспетчера местоположения, но он пока еще не начинает опрос местоположения. Для этого придется создать объект, соответствующий протоколу CLLocationManagerDelegate, и назначить его в качестве делегата диспетчера местоположения. Как только сведения о местоположении станут доступными или претерпят какие-то изменения, диспетчер местоположения вызовет методы этого делегата. Процесс определения местоположения может занять некоторое время — вплоть до нескольких секунд.

Задание требуемой точности

После настройки делегата необходимо также задать требуемую точность определения местоположения. Как упоминалось ранее, степень такой точности не должна превышать абсолютно необходимую величину. Так, если вы разрабатываете приложение, которое должно определять только регион или страну

местонахождения мобильного устройства, вам нет смысла указывать высокую степень точности. Не следует забывать, что чем выше запрашиваемая у Core Location точность, тем больше заряда батареи питания будет израсходовано на выполнение такого запроса. Следует также иметь в виду, что запрашиваемая степень точности определения местоположения совсем не гарантируется. В листинге 19.1 приведен пример кода, в котором задается делегат и запрашивается конкретная степень точности определения местоположения.

Листинг 19.1. Настройка делегата и требуемой точности

```
locationManager.delegate = self
locationManager.desiredAccuracy = kCLLocationAccuracyBest
```

Точность задается с помощью значения типа `CLLocationAccuracy`, определяемого как `Double`. Оно указывается в метрах, поэтому если задать значение типа `CLLocationAccuracy` равным 10, то это будет означать, что от Core Location требуется определить текущее местоположение с точностью до 10 метров, если это, конечно, возможно. Если же указать значение `kCLLocationAccuracyBest`, как в приведенном выше примере кода, то каркас Core Location должен воспользоваться самым точным способом определения текущего местоположения из всех доступных в настоящий момент. Помимо значения константы `kCLLocationAccuracyBest`, можно также указать значения следующих констант: `kCLLocationAccuracyNearestTenMeters`, `kCLLocationAccuracyHundredMeters`, `kCLLocationAccuracyKilometer`, а также `kCLLocationAccuracyThreeKilometers`.

Установка фильтра расстояния

По умолчанию диспетчер местоположения будет уведомлять делегат о любых обнаруженных изменениях в местоположении мобильного устройства. Установив **фильтр расстояния** (*distance filter*), можно предписать диспетчеру местоположения не уведомлять о каждом изменении, а вместо этого уведомлять только в том случае, если местоположение изменится больше чем на определенную величину. Устанавливая фильтр расстояния, можно значительно сократить количество опросов, выполняемых приложением. Фильтры расстояния также задаются в метрах. Так, если задать фильтр расстояния равным 1000, это будет означать, что диспетчер местоположения не должен уведомлять свой делегат до тех пор, пока мобильное устройство iPhone не переместится хотя бы на 1000 метров относительно того места, где раньше сообщалось о местоположении этого мобильного устройства.

```
locationManager.distanceFilter = 1000
```

Если же требуется вернуть диспетчер местоположения в исходное состояние без применения фильтра расстояния, то для этой цели можно воспользоваться константой `kCLDistanceFilterNone`, как показано ниже.

```
locationManager.distanceFilter = kCLDistanceFilterNone
```

Как и при задании требуемой точности определения местоположения, не следует обновлять координаты чаще, чем это необходимо, иначе заряд батареи будет израсходован напрасно. Например, в приложении, имитирующем спидометр, в котором скорость перемещения пользователя рассчитывается по его местоположению, обновления координат должны происходить как можно чаще, а в приложении, показывающем ближайший ресторан быстрого обслуживания, обновлять координаты можно намного реже.

Получение разрешения на пользование службами определения местоположения

Прежде чем воспользоваться службами определения местоположения в своем приложении, вы должны получить на это разрешение от пользователя. В каркасе Core Location предоставляются самые разные службы подобного рода, причем некоторыми из них можно пользоваться даже в том случае, если приложение работает в фоновом режиме. На самом деле можно даже сделать запрос на запуск приложения, если оно еще не выполняется, при наступлении определенного события. В одних случаях достаточно запросить разрешение на доступ к службам определения местоположения в то время, как пользователь работает с приложением, а в других случаях требуется возможность всегда пользоваться такими службами — все зависит от назначения конкретного приложения. Разрабатывая приложение, вы должны решить, какого рода разрешение ему потребуется, и сделать соответствующий запрос, прежде чем обращаться к нужным службам. В процессе создания примера приложения в этой главе будет показано, как это делается.

Запуск диспетчера местоположения

Как только все будет готово для того, чтобы начать процедуру определения координат, при поступлении запроса от пользователя приложение запускает диспетчер местоположения. Как только диспетчер местоположения определит текущее местоположение, он вызывает метод делегата. До тех пор, пока вы не дадите диспетчеру местоположения команду остановиться, он будет продолжать вызовы метода делегата при обнаружении изменений, превышающих текущее значение фильтра расстояния. В приведенной ниже строке кода показано, каким образом запускается диспетчер местоположения.

```
locationManager.startUpdatingLocation()
```

Благоразумное использование диспетчера местоположения

Если нужно лишь один раз определить текущее местоположение, но не за-прашивать его постоянно, то следует использовать метод `requestLocation()`, а не `startUpdatingLocation()`. Этот метод автоматически остановит проце-дуру после определения местоположения пользователя. С другой стороны, если требуется непрерывно определять местоположение, постарайтесь остановить эту процедуру при первой же возможности. Не забывайте, что частые обновления

текущего местоположения, получаемые от диспетчера местоположения, быстро истощают заряд батареи. Для того чтобы дать диспетчеру местоположения команду прекратить передачу новых координат своему делегату, достаточно вызвать метод `stopUpdatingLocation()`:

```
locationManager.stopUpdatingLocation()
```

Если вы используете метод `requestLocation()`, а не `startUpdatingLocation()`, то вызывать метод `stopUpdatingLocation()` не обязательно.

Делегат диспетчера местоположения

Делегат диспетчера местоположения должен соответствовать протоколу `CLLocationManagerDelegate`, в котором определяются несколько дополнительных методов. Один из них вызывается диспетчером местоположения после того, как он определит текущее местоположение или обнаружит изменения в местоположении. Другой метод вызывается, когда диспетчер местоположения обнаружит ошибку. Методы всех этих делегатов будут реализованы в рассматриваемом далее примере приложения.

Обновление координат

Когда диспетчеру местоположения требуется известить своего делегата о текущих координатах, он вызывает метод `locationManager(_ :didUpdateLocations:)`, который принимает два параметра.

- Первый параметр — это сам локальный диспетчер, вызвавший метод.

Второй параметр — это массив объектов типа `CCLocation`, определяющих текущие и предыдущие координаты устройства. Если за короткий период времени произойдет несколько обновлений координат, они могут быть объединены в одно целое при вызове данного метода. Но в любом случае последним элементом этого массива являются координаты самого последнего местоположения.

Определение широты и долготы средствами класса `CCLocation`

Сведения о местоположении передаются от диспетчера местоположения с помощью экземпляра класса `CLLocation`. У этого класса имеются следующие семь свойств, которые могут представлять интерес для приложения.

- `coordinate`
- `horizontalAccuracy`
- `altitude`
- `verticalAccuracy`
- `floor`

- timestamp
- description

В частности, широта и долгота местности хранятся в свойстве coordinate. Для получения широты и долготы местности в градусах достаточно обратиться к данному свойству так, как показано в листинге 19.2.

Листинг 19.2. Определение широты и долготы

```
let latitude = theLocation.coordinate.latitude
let longitude = theLocation.coordinate.longitude
```

Переменные latitude и longitude приводятся к типу CLLocationDegrees. Объект типа CLLocation может также сообщить, насколько надежно диспетчер местоположения рассчитывает широту и долготу местности. Так, его свойство horizontalAccuracy описывает радиус окружности, в центре которой находятся координаты местности из свойства coordinate. Чем больше значение, хранящееся в свойстве horizontalAccuracy, тем менее надежно определяется местоположение в Core Location. Между тем очень малый радиус свидетельствует об определении местоположения с высокой степенью надежности.

Графическое представление свойства horizontalAccuracy можно посмотреть в стандартном приложении Maps (рис. 19.1). Кругом в этом представлении обозначен радиус определения местоположения в приложении Maps с помощью свойства horizontalAccuracy. Исходно диспетчер местоположения предполагает, что пользователь находится в центре этого круга. В противном случае пользователь, вероятнее всего, находится где-то внутри этого круга. Отрицательное значение, хранящееся в свойстве horizontalAccuracy, обозначает, что пользователь данного приложения не может по той или иной причине доверять координатам местности, доступным в свойстве coordinate.

У объекта типа CLLocation имеется также свойство altitude типа CLLocationDistance, позволяющее определить высоту местности над уровнем моря, как показано ниже.

```
let altitude = theLocation.altitude
```

Кроме того, у каждого объекта типа CLLocation имеется свойство verticalAccuracy, обозначающее степень надежности, с которой в каркасе Core Location определяется высота местности над уровнем моря. Значение высоты над уровнем моря может отличаться от истинного на многие метры в зависимости от значения, доступного в свойстве verticalAccuracy, и если значение verticalAccuracy отрицательно, то каркас Core Location сообщает, что он не в состоянии достоверно определить высоту над уровнем моря.

Свойство floor обозначает этаж здания, в котором находится пользователь. Значение этого свойства оказывается достоверным только в тех зданиях, где предоставляются сведения об их этажности. Поэтому полагаться на данное свойство особенно не стоит.



Рис. 19.1. Каркас Core Location применяется в стандартном приложении Maps для определения текущего местоположения пользователя. Внешний круг дает наглядное представление о точности определения местности по горизонтали

Объекты типа `CLLocation` содержат также временную метку, обозначающую момент, в который диспетчер местоположения производил определение местности.

Помимо упомянутых выше свойств, в классе `CLLocation` имеется полезный метод экземпляра, позволяющий определить расстояние между двумя объектами типа `CLLocation`. Этот метод называется `distanceFromLocation()` и возвращает значение типа `CLLocationDistance` (по существу, `Double`). Следовательно, это значение можно использовать в арифметических расчетах, как будет показано в рассматриваемом далее примере приложения. Данный метод вызывается следующим образом:

```
let distance = fromLocation.distanceFromLocation(toLocation)
```

В приведенной выше строке кода возвращается расстояние между объектами `fromLocation` и `toLocation` типа `CLLocation`. Возвращаемое в результате расчета значение переменной `distance` обозначает расстояние между двумя точками по дуге большого круга без учета высоты, а это означает, что обе точки считаются находящимися на уровне моря. Как правило, расчета расстояния по дуге большого круга оказывается более чем достаточно, но если в расчете расстояний требуется принять во внимание и высоту над уровнем моря, то для этой цели придется написать код самостоятельно.

ЗАМЕЧАНИЕ. Если вы не знаете, что означает “расстояние по дуге большого круга” или так называемое “ортодромическое расстояние”, вам придется вспомнить из школьного курса географии понятие “ортодромический маршрут”. Дело в том, что кратчайшее расстояние между двумя точками на поверхности Земли прокладывается по маршруту, проходящему вокруг всей Земли, т.е. по “большому кругу”. При расчете, производимом в классе `CLLocation`, определяется расстояние между двумя точками по такому маршруту с учетом кривизны поверхности Земли. Без учета этой кривизны получается расстояние между двумя точками по прямой линии, от которого мало проку, поскольку прямая линия буквально пронизывает саму Землю.

Уведомления об ошибках

Если каркас Core Location не в состоянии определить текущее местоположение пользователя, то вызывается метод делегата `locationManager(_:didFailWithError:)`. Наиболее вероятной причиной ошибки, возникающей при определении местоположения, является отказ пользователя в доступе к службам определения местоположения, и в этом случае данный метод будет вызван с кодом ошибки `CLError.Denied`. Другая распространенная ошибка имеет код `CLError.LocationUnknown`, который поддерживается в диспетчере местоположения и уведомляет Core Location о невозможности определить местоположение, хотя попытки сделать это будут продолжены. Если код ошибки `CLError.LocationUnknown`, как правило, обозначает затруднение, которое может оказаться временным, то код ошибки `CLError.Denied` и другие ошибки означают, что приложение не сможет получить доступ к механизму Core Location в оставшееся время текущего сеанса связи.

ЗАМЕЧАНИЕ. В симуляторе нельзя определить текущее местоположение, но его можно выбрать (например, местоположение штаб-квартиры компании Apple по умолчанию) или задать самостоятельно, выполнив на симуляторе команду `Debug⇒Location`.

Создание приложения WhereAmI

Создадим небольшое приложение для обнаружения текущего местоположения мобильного устройства и общего расстояния, пройденного за время работы данного приложения. В окончательном виде это приложение будет таким, как на рис. 19.2.

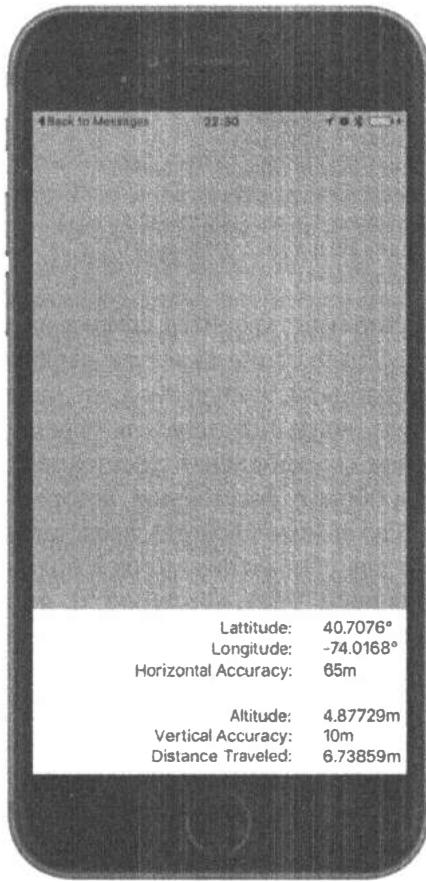


Рис. 19.2. Приложение WhereAmI в действии

Создайте в среде Xcode новый проект, используя шаблон Single View Application, и назовите его WhereAmI. Как только откроется окно данного проекта, выберите исходный файл ViewController.swift и внесите в него следующие изменения.

```
import UIKit
import CoreLocation
import MapKit

class ViewController: UIViewController, CLLocationManagerDelegate {
```

Прежде всего обратите внимание на импорт в данный проект каркаса Core Location. Каркас Core Location не входит в состав каркаса UIKit или Foundation, и поэтому его приходится импортировать вручную. Далее класс ViewController приводится в соответствие с методом делегата CLLocationManagerDelegate, чтобы иметь возможность получать сведения от диспетчера местоположения.

Далее введите объявления следующих свойств:

722 ГЛАВА 19 ■ ОПРЕДЕЛЕНИЕ МЕСТОПОЛОЖЕНИЯ

```
private let locationManager = CLLocationManager()
private var previousPoint: CLLocation?
private var totalMovementDistance = CLLocationDistance(0)

@IBOutlet var latitudeLabel: UILabel!
@IBOutlet var longitudeLabel: UILabel!
@IBOutlet var horizontalAccuracyLabel: UILabel!
@IBOutlet var altitudeLabel: UILabel!
@IBOutlet var verticalAccuracyLabel: UILabel!
@IBOutlet var distanceTraveledLabel: UILabel!
@IBOutlet var mapView: MKMapView!
```

В свойстве `locationManager` хранится ссылка на применяемый диспетчер местоположения типа `CLLocationManager`, а свойство `previousPoint` отслеживает последние координаты, полученное от диспетчера местоположения. Таким образом, всякий раз, когда пользователь переместится настолько, чтобы начать процесс обновления его координат, расстояние, на которое он переместился, будет прибавлено к общему расстоянию, которое хранится в свойстве `totalMovementDistance`. Остальные свойства представляют собой выходы, которые будут использоваться для обновления меток в пользовательском интерфейсе.

Выберите файл раскадровки `Main.storyboard`, чтобы отредактировать графический пользовательский интерфейс. Сначала разверните иерархию контроллеров представлений в окне `Document Outline`, выберите элемент `View`, а затем измените цвет фона на светло-серый в инспекторе атрибутов. Далее перетащите представление типа `UIView` из библиотеки стандартных объектов в существующее представление, изменив его положение и размеры таким образом, чтобы оно покрывало нижнюю половину главного представления, а его нижний, левый и правый края точно совпадали с соответствующими краями серого представления. Ваша цель — создать компоновку, аналогичную приведенной на рис. 19.2, где представление, которое вы только что перетащили, занимает нижнюю половину экрана с белым цветом фона.

Выберите в окне `Document Outline` только что добавленное вами представление, нажмите клавишу `<Control>`, перетащите указатель от него к родительскому представлению и отпустите кнопку мыши. Нажмите клавишу `<Shift>` и выберите команды `Leading Space to Container Margin`, `Trailing Space to Container Margin` и `Vertical Spacing to Bottom Layout Guide` из всплывающего меню. В итоге представление будет закреплено на своем месте, хотя его высота еще не задана. Для того чтобы исправить этот недостаток, снова выберите представление в окне `Document Outline` и щелкните на кнопке `Pin`. В открывшемся окне установите флажок `Height` и высоту, равную 166, а затем выберите вариант `Items of New Constraint` из раскрывающегося списка `Update Frames`, после чего щелкните на кнопке `Add 1 Constraint`, чтобы установить высоту данного представления должным образом.

Создайте столбец меток на правом краю текущего представления, как показано на рис. 19.2. С этой целью перетащите метку из библиотеки стандартных объектов, опустив ее чуть ниже верхнего края белого представления. Установите

ширину этой метки около 80 пикселей и переместите ее поближе к правому краю данного представления. Нажмите клавишу <Option> и перетащите указатель вниз, повторив эту операцию пять раз подряд, чтобы создать копии данной метки (см. рис. 19.2). Теперь нужно зафиксировать размеры и положения всех меток относительно их родительского представления.

Нажмите клавишу <Control> и перетащите указатель в окне Document Outline от самой верхней метки к ее родительскому представлению. Нажмите клавишу <Shift> и выберите команды Top Space to Container Margin и Trailing Space to Container Margin из всплывающего меню, после чего нажмите клавишу <Return>. Для того чтобы задать размеры метки, щелкните на кнопке Pin. Во всплывающем меню Add New Constraints установите флагки Width и Height, введите значения 80 и 21 в полях ширины и высоты соответственно, если эти значения еще не заданы, а затем щелкните на кнопке Add 2 Constraints. В итоге положение и размеры самой верхней метки будут зафиксированы. Повторите эту же процедуру для остальных пяти меток.

Создайте второй столбец меток. С этой целью перетащите метку из библиотеки стандартных объектов, расположив ее слева от самой верхней метки, но оставив небольшой горизонтальный промежуток между ними. Потяните новую метку за левый край, чтобы достичь почти левого края белого представления, а затем выберите в инспекторе атрибутов выравнивание текста метки по правому краю. Сделайте еще пять копий этой метки, нажав клавишу <Option> и перетащив указатель вниз, а затем выровняйте полученные метки по правому краю, как показано на рис. 19.3.

Выберите самую верхнюю метку в левом столбце, нажмите клавишу <Control> и перетащите указатель от ее левого края к левому краю белого представления. Отпустите кнопку мыши и выберите пункт Leading Space to Container Margin из контекстного меню. Затем нажмите еще раз клавишу <Control> и перетащите указатель от этой же метки к соответствующей метке в правом столбце. Отпустите кнопку мыши, чтобы появилось контекстное меню. Нажмите клавишу <Shift>,

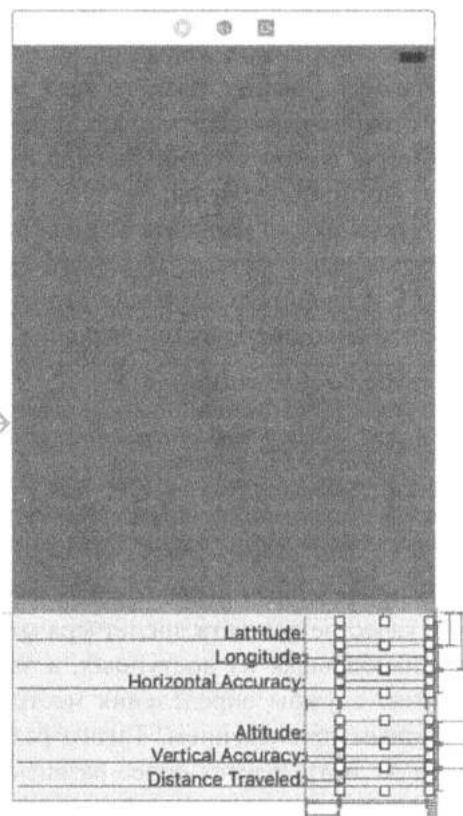


Рис. 19.3. Макет пользовательского интерфейса, который мы хотим создать

выберите команды **Horizontal Spacing** и **Baseline** из этого меню, после чего нажмите клавишу **<Return>**. Повторите эту же процедуру для остальных пяти меток в левом столбце. В заключение выберите пиктограмму **View Controller** в окне **Document Outline**, щелкните на кнопке **Resolve Auto Layout Issues** и выберите команду **Update Frames** из всплывающего меню, если этот режим активизирован.

Нам осталось связать метки в правом столбце с выходами в контроллере представления. Для этого нажмите клавишу **<Control>** и перетащите указатель от желтой пиктограммы контроллера представления в окне **Document Outline** к верхней метке в правом столбце и отпустите кнопку мыши. В появившемся всплывающем меню выберите команду **latitudeLabel**. Снова нажмите клавишу **<Control>** и перетащите указатель от пиктограммы контроллера представления ко второй сверху метке, чтобы связать ее с выходом **longitudeLabel**. Аналогичным образом свяжите третью метку с выходом **horizontalAccuracyLabel**, четвертую — с выходом **altitudeLabel**, пятую — с выходом **verticalAccuracyLabel**, а шестую, самую нижнюю метку — с выходом **distanCeTraveledLabel**. В итоге все шесть меток в правом столбце будут связаны с соответствующими выходами.

Наконец удалите текст из всех меток в правом столбце, а текст меток в левом столбце измените так, как показано на рис. 19.3. В частности, самая верхняя метка в этом столбце должна иметь текст **Latitude:**, следующая метка — текст **Longitude:** и т.д.

Теперь нужно написать код для отображения полезных сведений во всех упомянутых выше метках. Для этого выберите исходный файл **ViewController.swift** и введите следующие строки кода в метод **viewDidLoad()**, чтобы настроить диспетчер местоположения.

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.
    locationManager.delegate = self
    locationManager.desiredAccuracy = kCLLocationAccuracyBest
    locationManager.requestWhenInUseAuthorization()
```

В приведенных выше строках кода экземпляр класса контроллера назначается в качестве делегата диспетчера местоположения, задается требуемая точность как наибольшая из доступных, а затем запрашивается разрешение на использование службы определения местоположения, когда пользователь обращается с данным приложением. Такого разрешения достаточно для данного примера. Если же потребуются более развитые средства **Core Location**, рассмотрение которых выходит за рамки данной книги, вам, скорее всего, придется запросить разрешение на пользование каркасом **Core Location** в любой момент, вызвав в этом случае метод **requestAlwaysAuthorization()**.

ЗАМЕЧАНИЕ. В рассматриваемом здесь простом примере запрос на разрешение делается при запуске приложения, но в компании Apple рекомендуют отложить такой запрос в реальном приложении до тех пор, пока вам действительно понадобятся службы определения местоположения. Дело в том, что пользователь скорее согласится дать разрешение, если необходимость определения местоположения мобильного устройства будет очевидной, чем в том случае, если приложение, которое, вероятно, только что было установлено, запрашивает разрешение сразу же после запуска на выполнение.

При первом запуске данного приложения на выполнение система iOS выведет предупреждение с запросом на разрешение использовать местоположение пользователя. В это предупреждение от системы iOS нужно вставить короткий текст, поясняющий причины, по которым данному приложению нужно знать местоположение пользователя. С этой целью откройте файл Info.plist и введите в нем текст, который требуется вставить в упомянутое предупреждение, ниже ключа NSLocationWhenInUseUsageDescription. Если же требуется сделать запрос на разрешение пользоваться службами определения местоположения, когда приложение не применяется активно, то такой текст следует ввести под ключом NSLocationAlwaysUsageDescription. Для целей данного примера текст может быть следующим: "The application needs to know your location to update your position on a map" ("Приложению необходимо знать ваши координаты, чтобы обновить ваше местоположение на карте").

ПРЕДУПРЕЖДЕНИЕ. В прежних версиях iOS можно было не предоставлять текст, поясняющий причины запроса на разрешение, а в версии iOS 8 это стало обязательным. Если не предоставить такой текст, запрос на разрешение не будет сделан.

Если теперь вы запустите данное приложение на выполнение, то обнаружите, что введенный вами текст используется системой iOS в запросе на разрешение, как показано на рис. 19.4. Если запрос не появился, проверьте, правильно ли вы назвали ключ в файле Info.plist. Поскольку наше приложение пока не завершено, карта пока будет не прорисовываться на фоне, однако рис. 19.4 иллюстрирует цель, к которой мы направляемся.

Такой запрос появляется лишь один раз в течение срока действия приложения. Независимо от того, даст ли пользователь разрешение на пользование службами определения местоположения, этот запрос не будет сделан снова, сколько бы раз приложение ни запускалось на выполнение. Но это совсем не означает, что пользователь не может изменить свое решение. Подробнее об этом речь пойдет далее, в разделе "Изменение разрешений на пользование службами определения местоположения". Что же касается тестирования приложения, то его повторные запуски на выполнение в среде Xcode не оказывают никакого влияния на сохраненный ответ пользователя. Для подготовки приложения к тестированию его придется удалить из симулятора или мобильного устройства.

В таком случае система iOS снова сделает запрос на разрешение после переустановки и перезапуска приложения, а пока выберите кнопку *Allow*, чтобы продолжить разработку нашего приложения.

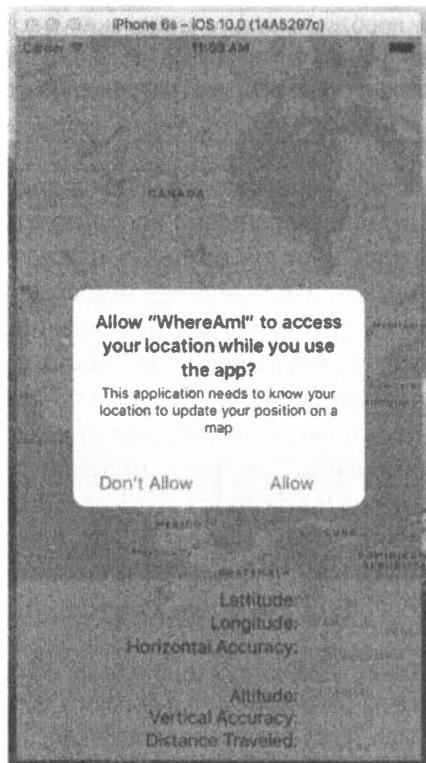


Рис. 19.4. Запрос у пользователя разрешения на пользование службами определения местоположения

Вы, вероятно, обратили внимание на то, что метод `startUpdatingLocation()` не вызывается из диспетчера местоположения в методе `viewDidLoad()` сразу же после вызова метода `requestWhenInUseAuthorization()`. На самом деле этого не стоит делать, потому что процесс авторизации не начинается немедленно. В какой-то момент после возврата управления из метода `viewDidLoad()` метод `locationManager(_:didChangeAuthorizationStatus:)` делегата диспетчера местоположения будет вызываться с состоянием авторизации приложения. Это состояние может быть результатом ответа пользователя на запрос или же последним сохраненным состоянием авторизации, с которым выполнялось приложение. Так или иначе данный метод служит идеальным местом для начала приема новых координат или отправки запроса на определение местоположения при условии, что вы прошли авторизацию. Введите в исходный файл `ViewController.swift` следующую реализацию данного метода:

```

func locationManager(_ manager: CLLocationManager,
    didChangeAuthorization status: CLAuthorizationStatus) {
    print("Authorization status changed to \(status.rawValue)")
    switch status {
        case .authorizedAlways, .authorizedWhenInUse:
            locationManager.startUpdatingLocation()

        default:
            locationManager.stopUpdatingLocation()
    }
}

```

В этом методе прием новых координат начинается, если авторизация прошла успешно, в противном случае он прекращается. Если проверка новых координат не начинается, пока нет авторизации, то какой смысл вызывать метод `stopUpdatingLocation()`? Это хороший вопрос. Рассматриваемый здесь код требуется потому, что пользователь может дать приложению разрешение на пользование каркасом Core Location, а впоследствии отозвать свое разрешение. В таком случае нужно прекратить проверку новых координат. Подробнее об этом речь пойдет ниже, в разделе “Изменение разрешений на пользование службами определения местоположения”.

Если приложение пытается воспользоваться службами определения местоположения, когда у него нет на это разрешения, или же если в какой-то момент возникает ошибка, то диспетчер местоположения вызывает метод `locationManager(_:didFailWithError:)` своего делегата. Введите в контроллер представления следующую реализацию этого метода:

```

func locationManager(_ manager: CLLocationManager,
    didFailWithError error: NSError) {
    let errorType = error.code == CLError.denied.rawValue
        ? "Access Denied": "Error \(error.code)"
    let alertController = UIAlertController(title: "Location Manager Error",
        message: errorType, preferredStyle: .alert)
    let okAction = UIAlertAction(title: "OK", style: .cancel,
        handler: { action in })
    alertController.addAction(okAction)
    present(alertController, animated: true,
        completion: nil)
}

```

В этом примере мы просто предупреждаем пользователя о возникшей ошибке. В реальном приложении придется вывести более содержательное сообщение об ошибке и очистить, если потребуется, состояние приложения.

Использование новых координат в диспетчере местоположения

Итак, разобравшись в получении разрешения на использование местоположения пользователя, организуем обработку этой информации. С этой целью введите в исходном файле `ViewController.swift` следующую реализацию метода `locationManager(_:didUpdateLocation:)` делегата диспетчера местоположения (листинг 19.3).

Листинг 19.3. Метод делегата диспетчера местоположения didUpdateLocation

```

func locationManager(_ manager: CLLocationManager, didUpdateLocations
    locations: [CLLocation]) {
    if let newLocation = locations.last {
        let latitudeString = String(format: "%g\u{00B0}", newLocation.coordinate.latitude)
        latitudeLabel.text = latitudeString
        let longitudeString = String(format: "%g\u{00B0}", newLocation.coordinate.longitude)
        longitudeLabel.text = longitudeString
        let horizontalAccuracyString = String(format: "%gm",
            newLocation.horizontalAccuracy)
        horizontalAccuracyLabel.text = horizontalAccuracyString

        let altitudeString = String(format: "%gm", newLocation.altitude)
        altitudeLabel.text = altitudeString

        let verticalAccuracyString = String(format: "%gm",
            newLocation.verticalAccuracy)
        verticalAccuracyLabel.text = verticalAccuracyString

        if newLocation.horizontalAccuracy < 0 {
            // Неприемлемая точность
            return
        }

        if newLocation.horizontalAccuracy > 100 || newLocation.verticalAccuracy > 50 {
            // Диапазон точности настолько велик,
            // что использовать его не стоит
            return
        }

        if previousPoint == nil {
            totalMovementDistance = 0
        } else {
            print("movement distance: " +
                "\u{2193}newLocation.distance(from: previousPoint!))")
            totalMovementDistance +=
                newLocation.distance(from: previousPoint!)
        }
        previousPoint = newLocation

        let distanceString = String(format: "%gm", totalMovementDistance)
        distanceTraveledLabel.text = distanceString
    }
}

```

Прежде всего, первые пять меток во втором столбце (см. рис. 19.3) обновляются значениями из объектов типа CLLocation, передаваемых данному методу делегата в качестве аргумента locations. Этот массив может содержать не одно, а несколько обновлений местоположения, но пользоваться следует последним элементом этого массива, содержащим последние координаты.

ЗАМЕЧАНИЕ. Широта и долгота местности выводятся на экран в строках форматирования, содержащих загадочного вида символы \u{00B0}. Эти символы обозначают представление знака градуса (°) в unicode. Вводить непосредственно в исходный код символы в другом коде, кроме ASCII, нецелесообразно. Но в то же время в строки кода можно включать шестнадцатеричные значения символов в другом коде, что и было сделано в данном случае.

Затем определяется точность, которую обеспечивает диспетчер местоположения. Большие значения точно указывают на то, что диспетчер местоположения ненадежно определяет местоположение, тогда как отрицательные значения точности — местоположение фактически определено неверно. Но некоторые мобильные устройства не оснащены оборудованием, требующимся для определения их вертикального положения. На таких устройствах и в симуляторе свойство verticalAccuracy всегда будет равно -1, и поэтому мы не исключаем наличие такого значения в сообщениях о положении мобильного устройства. Эти значения точности измеряются в метрах и выражают радиус круга с центром в указанном местоположении, т.е. подразумевается, что вы находитесь где-то внутри этого круга. В рассматриваемом здесь методе проверяется, имеют ли эти значения приемлемую точность. Если же они ее не имеют, то происходит возврат из данного метода вместо обработки некачественных данных, как показано ниже.

```
if newLocation.horizontalAccuracy < 0 {
    // Неприемлемая точность
    return
}

if newLocation.horizontalAccuracy > 100 ||
    newLocation.verticalAccuracy > 50 {
    // Диапазон точности настолько велик,
    // что использовать его не стоит
    return
}
```

Далее проверяется, равно ли nil значение свойства previousPoint. Если оно равно nil, то данное обновление местоположения считается первым достоверным обновлением, полученным от диспетчера местоположения, и поэтому свойство distanceFromStart обнуляется. В противном случае расстояние между самым последним и предыдущим местоположением добавляется к общему расстоянию. В любом случае свойство previousPoint обновляется, чтобы содержать текущее местоположение, как показано ниже.

```
if previousPoint == nil {
    totalMovementDistance = 0
} else {
    print("movement distance: " +
        "\u00d7(newLocation.distance(from: previousPoint!))")
    totalMovementDistance +=
        newLocation.distance(from: previousPoint!)
}
previousPoint = newLocation
```

После этого последняя метка заполняется общим расстоянием, пройденным из точки отправления. Если во время работы данного приложения пользователь движется достаточно долго, чтобы диспетчер местоположения мог обнаружить движение, метка Distance Traveled: будет постоянно обновляться расстоянием, пройденным пользователем с момента запуска данного приложения на выполнение:

```
let distanceString = String(format:@"%gm", totalMovementDistance)
distanceTraveledLabel.text = distanceString
```

Вот, собственно, и все. Как видите, пользоваться каркасом Core Location очень просто. Скомпилируйте и запустите на выполнение данное приложение. Если у вас имеется возможность выполнить его на своем мобильном устройстве iPhone или iPad, попробуйте сделать это во время езды на автомобиле, наблюдая за числовыми значениями, обозначающими изменение вашего местоположения.

Отображение движения на карте

Все, что мы сделали до сих пор, довольно интересно, но было бы неплохо отобразить движение пользователя на карте. К счастью, в состав iOS входит каркас Map Kit, позволяющий это сделать. Каркас Map Kit использует те же самые вспомогательные службы, что и приложение Map компании Apple. Это означает, что он надежный и все время совершенствуется. Каркас Map Kit содержит один главный класс представления для отображения карты, реагирующий на жесты пользователя, как это принято в современных картографических приложениях. Это представление также позволяет вставлять аннотации к любому местоположению, которое требуется отобразить на карте. Эти аннотации по умолчанию изображаются с помощью кнопок, коснувшись которых, можно получить дополнительную информацию. Расширим приложение WhereAmI, чтобы оно могло отображать на экране точку отправления пользователя и его текущее местоположение на карте.

Выберите исходный файл ViewController.swift и введите в нем выделенную ниже полужирным шрифтом строку кода, чтобы импортировать каркас Map Kit.

```
import UIKit
import CoreLocation
import MapKit
```

Ведите следующее объявление нового свойства представления карты, где будет отображаться местоположение пользователя:

```
@IBOutlet var mapView:MKMapView
```

Выберите файл раскладовки Main.storyboard, чтобы отредактировать представление карты. Перетащите представление карты (Map View) из библиотеки объектов, опустив его на пользовательский интерфейс. Измените размеры этого представления, чтобы оно покрывало экран в целом, включая представление и все его метки, а затем выполните команду меню Editor⇒Arrange⇒Send to Back, чтобы расположить представление Map View позади другого представления.

ЗАМЕЧАНИЕ. Если команда **Send to Back** недоступна, тот же эффект можно получить, перетащив представление Map View в окне Document Outline вверх, чтобы оно появлялось до представления, содержащего метки в своем дочернем списке.

Перейдите в окно Document Outline, нажмите клавишу <Control> и перетащите указатель от представления Map View к его родительскому представлению. Как только появится контекстное меню, нажмите клавишу <Shift> и выберите команды **Leading Space to Container Margin**, **Trailing Space to Container Margin**, **Vertical Spacing to Top Layout Guide** и **Vertical Spacing to Bottom Layout Guide**, а затем нажмите клавишу <Return>.

Итак, представление Map View зафиксировано на месте, но нижняя его часть скрыта. Этот недостаток можно исправить, сделав представление частично прозрачным. С этой целью выберите представление в окне Document Outline, откройте инспектор атрибутов, щелкните на образце цвета фона и выберите во всплывающем меню пункт **Other...**, чтобы открыть селектор цвета. Далее выберите белый цвет фона и переместите ползунок **Opacity** приблизительно к отметке 70%. Нажмите клавишу <Control> и перетащите указатель от пиктограммы контроллера представления к представлению Map View, а затем выберите команду **mapView** во всплывающем меню, которое появится на экране, чтобы позволить вам связать карту с соответствующим выходом.

Выполнив всю эту предварительную работу, напишем небольшой код, чтобы поставить карту себе на службу. Сначала нужно настроить класс модели, представляющей точку отправления. Представление типа MKMapView является составляющей представления в архитектуре MVC (Model–View–Controller) и лучше всего действует, когда имеются отдельные классы для обозначения маркеров на карте. Мы можем передавать объекты модели в представление карты и запрашивать у них координаты, надписи и прочие данные, используя протокол, определенный в каркасе Map Kit.

Нажмите комбинацию клавиш <⌘+N>, чтобы открыть помощник для создания новых файлов, и выберите в разделе iOS шаблон Cocoa Touch Class. Присвойте новому классу имя **Place** и сделайте его подклассом, производным от класса **NSObject**. Затем откройте исходный файл **Place.swift** и внесите в него изменения, приведенные ниже. Эти изменения состоят в том, чтобы импортировать каркас Map Kit, указать протокол, которому должен соответствовать новый класс, а также задать ряд свойств (листинг 19.4).

Листинг 19.4. Новый класс Place в файле **Place.swift**

```
import UIKit
import MapKit

class Place: NSObject, MKAnnotation {
    let title: String?
    let subtitle: String?
    var coordinate: CLLocationCoordinate2D
```

732 ГЛАВА 19 « ОПРЕДЕЛЕНИЕ МЕСТОПОЛОЖЕНИЯ

```
init(title:String, subtitle:String, coordinate:CLLocationCoordinate2D) {  
    self.title = title  
    self.subtitle = subtitle  
    self.coordinate = coordinate  
}  
}
```

По существу, этот “фиктивный” класс действует лишь как место для хранения свойств. В реальном приложении могут существовать классы настоящих моделей, которые должны отображаться на карте в виде аннотации. Возможность добавить эти аннотации в любые классы, не нарушив их иерархии, предоставляет протокол MKAnnotation. Выберите файл ViewController.swift и введите две строки кода, выделенные ниже полужирным шрифтом, в теле метода locationManager(_:didChangeAuthorizationStatus:).

```
func locationManager(_ manager: CLLocationManager,  
                    didChangeAuthorization status: CLAuthorizationStatus) {  
    print("Authorization status changed to \(status.rawValue)")  
    switch status {  
        case .authorizedAlways, .authorizedWhenInUse:  
            locationManager.startUpdatingLocation()  
            mapView.showsUserLocation = true  
  
        default:  
            locationManager.stopUpdatingLocation()  
            mapView.showsUserLocation = false  
    }  
}
```

Свойство showsUserLocation представления Map View делает то, о чем можно только мечтать, избавляя нас от необходимости перемещать маркер вручную по карте по мере передвижения пользователя, поскольку это делается автоматически. Для получения данных о местоположении пользователя происходит обращение к каркасу Core Location, но только в том случае, если приложение авторизовано для этого. Поэтому свойство showsUserLocation включается, если у приложения есть разрешение использовать каркас Core Location, а если разрешение отменено, то данное свойство снова отключается.

Теперь вернемся к методу locationManager(_:didUpdateLocations:), в теле которого уже имеется код, определяющий достоверность полученных данных о первом местоположении пользователя и устанавливающий отправную точку. Помимо этого, нам нужно получить новый экземпляр класса Place и установить в его свойствах координаты, заголовок и подзаголовок маркера, чтобы выводить эти данные на карте. В заключение этот объект следует передать представлению карты. Нам нужно также создать новый объект типа MKCoordinateRegion — специальную структуру, входящую в состав Map Kit и позволяющую сообщать представлению ту часть карты, которую требуется вывести на экран, используя новые координаты и пару расстояний в метрах (100, 100), задающих ширину и высоту данной части карты на экране. Этот объект также

следует передать представлению карты, дав ему команду выполнить анимацию изменений. Все это делается в следующих строках кода.

```

if previousPoint == nil {
    totalMovementDistance = 0
    let start = Place(title:"Start Point",
                      subtitle:"This is where we started",
                      coordinate:newLocation.coordinate)
    mapView.addAnnotation(start)
    let region = MKCoordinateRegionMakeWithDistance(newLocation.coordinate,
                                                    100, 100)
    mapView.setRegion(region, animated: true)
} else {
    print("movement distance: " +
        "\((newLocation.distance(from: previousPoint!)))")
    totalMovementDistance +=
        newLocation.distance(from: previousPoint!)
}

```

Итак, мы сообщили представлению карты, что у нас есть аннотация (т.е. видимая отметка на карте), которую нужно показать пользователю. Но как это сделать? Для этого в представлении карты делается запрос делегата, чтобы выяснить, какое именно представление следует вывести на экран для каждой аннотации. В более сложном приложении это было бы сделано автоматически, но в данном простом примере делегат отсутствует, просто потому что он не нужен. В отличие от объекта типа NSTableView, требующего от своего источника данных предоставить ячейки для отображения на экране, объект типа MKMapView действует иначе: если он не получает представления аннотаций от делегата, то просто выводит на экран представление, заданное по умолчанию. Оно выглядит, как красная кнопка на карте, которая при ее касании выводит на экран дополнительную информацию.

Теперь осталось лишь активизировать применение каркаса Map Kit в данном приложении. С этой целью выберите сначала данный проект в навигаторе проектов, а затем цель проекта WhereAmI. Далее выберите вкладку Capabilities в верхней части области редактирования, перейдите к разделу Maps и переключите селектор справа из состояния OFF в состояние ON. Постройте и запустите данное приложение на выполнение, чтобы посмотреть, как загружается представление карты, а вскоре и координаты вашего местоположения на карте. Вы увидите, как приложение выполняет переход к вашему новому местоположению, устанавливает кнопку в точку отправления и отмечает ваше текущее местоположение светящейся голубой точкой (рис. 19.5). Не так уж и плохо для нескольких десятков строк кода.

ЗАМЕЧАНИЕ. Если вы используете реальное устройство и карта не укрупняет масштаб, чтобы показать ваше текущее местоположение, значит, каркас Core Location не может определить, где вы находитесь, с точностью до 100 метров. Вам необходимо помочь ему, включив службу Wi-Fi, которая иногда позволяет повысить точность каркаса Core Location.



Рис. 19.5. Красная кнопка отмечает отправную точку, а голубая точка показывает текущее местоположение

Изменение разрешений на пользование службами определения местоположения

Когда ваше приложение запускается на выполнение в первый раз, вы надеетесь, что пользователь даст разрешение на пользование службами определения местоположения. Но независимо от того, получите ли вы такое разрешение, вы не можете гарантировать, что это положение не изменится впоследствии. Пользователь может дать или отозвать разрешение на использование данных о своем местоположении через приложение *Settings*. Это можно проверить в симуляторе. Запустите данное приложение на выполнение и дайте разрешение на пользование средствами *Core Location* (если вы не дали раньше такое разрешение, вам придется сначала удалить и переустановить приложение). Ваше текущее местоположение должно появиться на карте. Перейдите к приложению *Settings* и выполните команду *Privacy*⇒*Location Services*. В верхней части экрана появится переключатель, позволяющий включать и отключать службы определения местоположения. Установите этот переключатель в состояние OFF и вернитесь к данному приложению. Как видите, ваше текущее местоположение

больше не отображается на карте, потому что диспетчер местоположения вызвал метод locationManager(_:didChangeAuthorizationStatus:) с кодом получения разрешения `CLAuthorizationStatus.denied`. В ответ на это приложение прекратило получать обновления вашего местоположения, дав каркасу Map Kit команду перестать отслеживать местоположение пользователя. Вернитесь к приложению *Settings* и снова разрешите применение каркаса *Core Location*. Перейдя обратно к данному приложению, вы обнаружите, что оно снова отслеживает ваше местоположение.

Включение и отключение служб определения местоположения — не единственный способ, которым пользователь может изменить возможность пользоваться каркасом *Core Location* в приложении. Вернитесь к приложению *Settings*. Ниже переключателя, разрешающего пользоваться службами определения местоположения, находится список всех приложений, в которых эти службы применяются, включая приложение *WhereAmI* (рис. 19.6, слева). Щелкните на имени данного приложения, чтобы перейти на другую страницу, где вы можете разрешить или запретить ему доступ к службам определения местоположения (рис. 19.6, справа). В настоящий момент данное приложение может

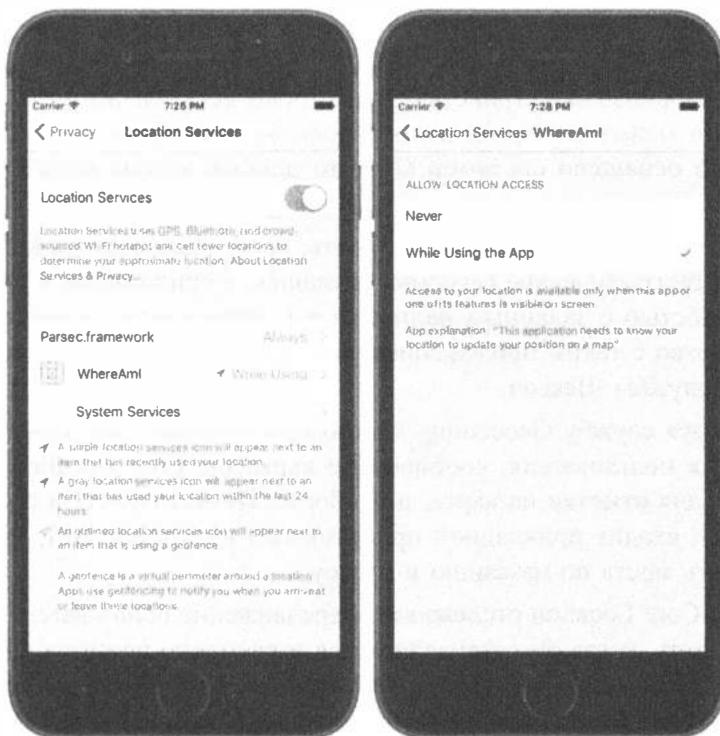


Рис. 19.6. Изменение состояния разрешения на доступ к данным о местоположении пользователя в приложении *WhereAmI*

пользоваться службами определения местоположения, когда пользователь работает с этим приложением. Если вы щелкнете на варианте *Never*, разрешение на пользование этими службами будет отозвано, в чем вы сможете сами убедиться, вернувшись снова к данному приложению. Это наглядно показывает, насколько важно запрограммировать приложение таким образом, чтобы оно могло обнаруживать и надлежащим образом реагировать на изменения в своем состоянии на получение разрешения пользоваться упомянутыми выше службами.

Резюме

На этом введение в каркасы Core Location и Map Kit завершается. У этих каркасов можно обнаружить еще немало других возможностей. Ниже перечислены лишь самые главные их особенности.

- Вместо точного отслеживания местоположения пользователя с помощью метода `startUpdatingLocation()` в тех приложениях, в которых нужна меньшая точность определения местоположения или же меньшая частота его обновления (например, в погодных приложениях наподобие Weather), можно воспользоваться службой Significant Location Updates. Этой службой следует пользоваться, если такая возможность вообще имеется, поскольку она позволяет значительно экономить потребляемую энергию.
- На тех мобильных устройствах, на которых есть магнитометр, каркас Core Location может сообщить азимут пользователя. Если же мобильное устройство оснащено системой GPS, то данный каркас может также сообщить направление, в котором движется пользователь.
- Каркас Core Location может сообщить, когда пользователь входит или покидает географические регионы, указанные в приложении и определяемые окружностью с заданным радиусом и центром, или же когда мобильное устройство с таким приложением находится в радиусе действия радиомаячной службы iBeacon.
- Используя службу Geocoding, можно преобразовать координаты местоположения пользователя, сообщаемые каркасом Core Location, в удобный объект для отметки на карте, и наоборот. Помимо этого, в состав каркаса Map Kit входит прикладной программный интерфейс API, позволяющий находить места по названию и адресу.
- Каркас Core Location отслеживает передвижение пользователя и позволяет определить, когда он останавливается и временно остается на месте. Когда это происходит, считается, что пользователь посещает данное место. Приложение может получить уведомление о том, когда пользователь прибывает или отбывает из посещаемого места.

Наилучшим источником информации обо всех этих возможностях и особенностях каркасов Core Location и Map Kit служит руководство *Location and Maps*

Programming Guide от компании Apple. Несмотря на сложность технологий, положенных в основу этих каркасов, компания Apple предоставила простые интерфейсы, скрывающие большую часть этой сложности и позволяющие очень легко вводить средства определения и нанесения на карту местоположения пользователей в разрабатываемые приложения. Это дает возможность сообщать пользователям, где они находятся, замечать моменты, когда они движутся, и отмечать их местоположение (и любые другие места) на карте. Поскольку речь зашла о передвижении, перейдите сразу к следующей главе, в которой мы исследуем акселерометр, встроенный в мобильное устройство iPhone.

ГЛАВА 20

Распознавание ориентации и перемещения устройства

К числу самых привлекательных аппаратных средств, встроенных в мобильные устройства iPhone, iPad и iPod touch, относится акселерометр — крошечное приспособление, определяющее положение мобильного устройства в руке пользователя, а также распознающее его перемещение. Акселерометр используется в системе iOS для автоматического вращения изображения на экране синхронно с мобильным устройством, а во многих играх — в качестве механизма управления. С его помощью можно также обнаруживать сотрясение и другие неожиданные движения мобильного устройства. Все эти возможности были дополнительно расширены в мобильном телефоне модели iPhone 4, в который встроен также гироскоп, позволяющий определять угол поворота данного устройства относительно каждой оси вращения. Гироскоп и акселерометр теперь являются стандартными средствами во всех новых моделях мобильных устройств iPhone, iPad и iPod touch. В этой главе будет рассмотрено применение каркаса Core Motion для доступа ко всем значениям, определяющим ориентацию мобильного устройства в пространстве с помощью акселерометра и гироскопа.

Физические основы работы акселерометра

Акселерометр измеряет ускорение, а также тяготение, воспринимая величину силы инерции в заданном направлении. Внутри мобильного устройства, работающего под управлением системы iOS, находится трехкомпонентный акселерометр, а это означает, что он способен обнаруживать передвижение или земное притяжение в трехмерном пространстве. Иными словами, акселерометр может быть использован для определения положения мобильного устройства не только в руках пользователя, но и в лежачем положении на столе, причем лицевой стороной вверх или вниз. Показания акселерометров обозначаются как перегрузки в единицах измерения g , где g — сила тяжести. Следовательно, значение 1, 0,

возвращаемое акселерометром, означает, что перегрузка 1 g воспринимается им в определенном направлении, как поясняется в приведенных ниже примерах.

- Если пользователь держит мобильное устройство в руке без движения, оно испытывает силу земного притяжения около 1 g.
- Если пользователь держит мобильное устройство в руке прямо, т.е. в книжной ориентации, акселерометр обнаруживает и сообщает, что данное устройство испытывает силу земного притяжения около 1 g по оси у его вращения.

Если пользователь держит мобильное устройство под углом, то сила земного притяжения около 1 g распределяется по разным осям его вращения в зависимости от положения данного устройства в руке пользователя. Так, если пользователь держит мобильное устройство под углом 45°, сила тяжести 1 g приблизительно равномерно распределяется по двум осям вращения данного устройства.

Резкое движение может быть обнаружено по показаниям акселерометра, значительно превышающим 1 g. В обычных условиях эксплуатации мобильного устройства акселерометр не обнаруживает силу тяжести, значительно превышающую 1 g по любой оси вращения. Но если тряхнуть, уронить или бросить мобильное устройство, то акселерометр обнаружит увеличение силы тяжести по одной оси или более.

На рис. 20.1 приведено графическое представление трех осей, применяемых в акселерометре. Обратите внимание на то, что в акселерометре используется более стандартное условное обозначение координаты y, которая увеличивается вверх, обозначая повышение силы тяжести, в отличие от системы координат в

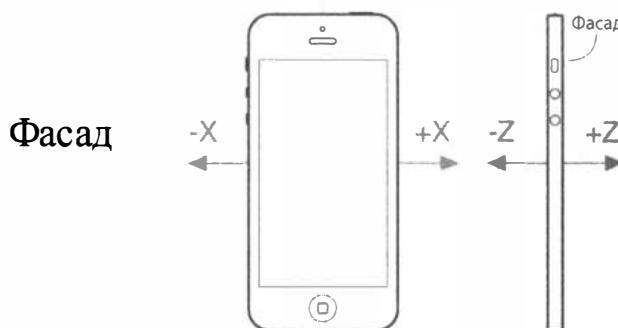


Рис. 20.1. Система трех координат акселерометра, встроенного в мобильное устройство iPhone. Слева показан вид iPhone спереди в плоскости координат x и y, а справа — вид iPhone сбоку в плоскости координат z

библиотеке Quartz 2D, рассматривавшейся в главе 16. Если акселерометр применяется в качестве механизма управления вместе с библиотекой Quartz 2D, то преобразование координаты у требуется. Если выбрать каркас Sprite Kit, в большей степени пригодный для разработки приложений, в которых показания акселерометра используются для управления анимацией, то никакого преобразования координат не потребуется.

Распознавание вращения с помощью гироскопа

Как упоминалось ранее, во все современные модели мобильных устройств от компании Apple встроен также гироскопический датчик, дающий показания, обозначающие углы вращения мобильного устройства вокруг его осей. Для того чтобы отличия в назначении этого датчика от акселерометра стали более понятными, рассмотрим мобильное устройство iPhone в лежачем положении на столе. Если начать поворачивать iPhone по кругу, не поднимая его со стола, показания акселерометра не изменятся, поскольку силы, вызывающие перемещение iPhone, не меняются (в данном случае это сила тяжести, направленная прямо вниз по оси z). (В действительности дело обстоит немного сложнее: движение рукой, приводящее iPhone во вращение, скорее всего, вызовет незначительную реакцию со стороны акселерометра.) Однако во время того же самого движения углы вращения iPhone будут изменяться и, в частности, при вращении относительно оси z. Так, если повернуть iPhone по часовой стрелке, угол его вращения относительно оси z окажется отрицательным. Если повернуть iPhone против часовой стрелки, то угол его вращения относительно оси z окажется положительным. Если же прекратить вращение iPhone, угол его вращения относительно оси z вернется к нулевому значению. Вместо того чтобы регистрировать абсолютное значение угла вращения, гироскопы показывают изменения, происходящие при вращении устройства. Эти особенности работы гироскопа будут более наглядно продемонстрированы на первом же примере, приведенном несколько позже.

Каркас Core Motion и диспетчер движения

Показания акселерометра и гироскопа доступны через каркас Core Motion. Среди прочего, этот каркас предоставляет класс `CMMotionManager`, выполняющий функции своеобразного шлюза для всех значений, описывающих перемещение мобильного устройства пользователем. В разрабатываемом приложении сначала создается экземпляр класса `CMMotionManager`, а затем он применяется в одном из следующих двух режимов работы.

- Автоматическое выполнение некоторого кода всякий раз, когда происходит движение мобильного устройства.
- Постоянное обновление структуры, предоставляющей в любой момент доступ к самым последним значениям, описывающим движение мобильного устройства.

Последний режим работы идеально подходит для игр и других высокоинтегративных приложений, в которых нужен опрос текущего состояния устройства на каждом шаге игрового или рабочего цикла. Ниже будет показано, каким образом реализуются оба упомянутых выше режима работы. Следует, однако, иметь в виду, что класс `CMMotionManager` на самом деле не является синглтоном, хотя в приложении его приходится интерпретировать как синглтон, т.е. в каждом приложении можно создать лишь один объект такого класса. Так, если требуется доступ к диспетчеру движения из нескольких мест в приложении, этот объект, вероятнее всего, придется создать в делегате приложения и оттуда предоставить доступ к нему.

Помимо класса `CMMotionManager`, в каркасе Core Motion предоставляет-
ся ряд других классов, в том числе `CMAccelerometerData` и `CMGyroData`.
Они представляют собой простые контейнеры, через которые приложение мо-
жет получать доступ к исходным данным акселерометра и гироскопа. В клас-
се `CMDeviceMotion` измерения, произведенные акселерометром и гироскопом,
объединяются с информацией о пространственной ориентации, обозначающей,
лежит ли мобильное устройство ровно, приподнято вверх, наклонено влево
и т.д. Класс `CMDeviceMotion` будет использоваться в примерах, представлен-
ных далее в этой главе.

Создание приложения MotionMonitor

Как упоминалось ранее, диспетчер движения может работать в режиме авто-
матического выполнения некоторого кода всякий раз, когда изменяются данные
о движении мобильного устройства. Аналогичные функциональные возможнос-
ти предоставляются в большинстве других классов Cocoa Touch. В частности,
они позволяют связываться с делегатом, получающим сообщение в подходящий
момент, но в каркасе Core Motion это делается немного иначе. Для уведомления
о происходящем вместо вызова методов делегата диспетчеру типа `CMMotion-
Manager` можно передавать замыкание всякий раз, когда происходит движение
мобильного устройства. В примерах, представленных в предыдущих главах, за-
мыкания использовались неоднократно, а теперь мы рассмотрим совсем другое
их применение.

Создайте в среде Xcode новый проект под названием `MotionMonitor` по
шаблону Single View Application. Это будет простое приложение, счита-
вающее показания акселерометра и гироскопа, если таковые имеются, выводя
их на экран.

ЗАМЕЧАНИЕ. Приложения, рассматриваемые в этой главе, не функционируют в симу-
ляторе, поскольку в нем, к сожалению, отсутствуют средства, аналогичные акселero-
метру и гироскопу.

Выберите исходный файл `ViewController.swift` и внесите в него следу-
ющие изменения.

```
class ViewController: UIViewController {
    @IBOutlet var gyroscopeLabel:UILabel!
    @IBOutlet var accelerometerLabel:UILabel!
    @IBOutlet var attitudeLabel:UILabel!
```

В приведенных выше строках кода предоставляются выходы к трем меткам, в которых будут отображаться данные о движении мобильного устройства. Они не требуют особых пояснений, поэтому вам остается лишь сохранить внесенные изменения. Далее откройте файл раскладовки Main.storyboard в программе Interface Builder. Разверните иерархию контроллера представления в окне Document Outline. Перетащите метку (объект типа Label) из библиотеки объектов в данное представление. Измените размеры метки, чтобы она простиралась от левого до правого края экрана, установите ее высоту приблизительно на треть всего представления, а затем выровняйте ее верхний край по верхней голубой направляющей линии. Откройте инспектор атрибутов и замените значение 1 значением 0 в поле Lines. Атрибут Lines служит для обозначения количества текстовых строк, которые могут появиться в метке, ограничивая их жестко заданным верхним пределом. Если же установить нулевое значение в поле этого атрибута, то никакого ограничения на количество текстовых строк не накладывается, а следовательно, метка может состоять из какого угодно количества текстовых строк.

Перетащите вторую метку из библиотеки объектов, опустив ее прямо под первой меткой. Выровняйте ее верхний край по нижнему краю верхней метки, а боковые стороны — по левому и правому краям экрана. Измените размеры этой метки приблизительно на ту же высоту, что и у первой метки. Особая точность при этом не требуется, поскольку окончательная высота меток будет установлена в результате автоматической компоновки. Перетащите третью метку, расположив ее прямо под второй меткой, выровняйте ее верхний край по нижнему краю второй метки, а затем измените ее размеры таким образом, чтобы ее нижний край достиг нижнего края экрана, после чего выровняйте ее боковые стороны по левому и правому краям экрана. Установите нулевое значение в полях атрибута Lines обеих меток.

Уточните расположение и размеры всех трех меток. С этой целью перейдите в окно Document Outline, нажмите клавишу <Control>, перетащите курсор от верхней метки к ее родительскому представлению и отпустите кнопку мыши. Как только откроется контекстное меню, нажмите клавишу <Shift> и выберите команды Leading Space to Container Margin, Vertical Spacing to Top Layout Guide и Trailing Space to Container Margin, после чего нажмите клавишу <Return>. Снова нажмите клавишу <Control> и перетащите курсор от второй метки к ее родительскому представлению. Как только откроется контекстное меню, нажмите клавишу <Shift> и выберите команды Leading Space to Container Margin и Trailing Space to Container Margin, а затем нажмите клавишу <Return>. Еще раз нажмите клавишу <Control> и перетащите курсор от третьей метки к ее родительскому представлению. Нажав клавишу <Shift>, выберите на этот раз команды Leading

Space to Container Margin, Vertical Spacing to Bottom Layout Guide и Trailing Space to Container Margin из всплывающего меню.

Теперь, когда все три метки прикреплены к краям их родительского представления, свяжите их вместе. С этой целью нажмите клавишу <Control> и перетащите курсор от второй метки к первой и выберите пункт Vertical Spacing из всплывающего меню. Снова нажмите клавишу <Control> и перетащите курсор от второй метки к третьей метке и выберите тот же самый пункт из всплывающего меню. В заключение нужно обеспечить одинаковую высоту меток. Для этого нажмите клавишу <Shift> и щелкните по очереди на всех трех метках, чтобы выбрать их. Щелкните на кнопке Pin, установите флагок Equal Heights и щелкните на кнопке Add 2 Constraints в открывшемся окне. Щелкните сначала на кнопке Resolve Auto Layout Issues, а затем на пункте Update All Frames in View Controller. Если же эта команда меню недоступна, выберите пиктограмму View Controller в окне Document Outline и сделайте еще одну попытку.

На этом компоновка главного представления рассматриваемого здесь приложения завершается. Теперь нужно связать все три метки с их выходами. Для этого нажмите клавишу <Control> и перетащите курсор от контроллера представления в окне Document Outline к верхней метке, отпустите кнопку мыши и выберите команду gyroscopeLabel из всплывающего меню, чтобы связать метку с ее выходом Сделайте то же самое со второй меткой, связав ее с выходом accelerometerLabel, а затем и с третьей меткой, которая должна быть связана с выходом attitudeLabel. Дважды щелкните на каждой из меток и удалите имеющийся у них текст. Разработка нашего простого пользовательского интерфейса завершена, поэтому сохраните работу и приготовьтесь к программированию.

Выберите исходный файл ViewController.swift и внесите в него следующие изменения (листинг 20.1).

Листинг 20.1. Новый код в файле ViewController.swift

```
private let motionManager = CMMotionManager()
private let queue = OperationQueue()

override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.

    if motionManager.isDeviceMotionAvailable {
        motionManager.deviceMotionUpdateInterval = 0.1
        motionManager.startDeviceMotionUpdates(to: queue) {
            (motion:CMDeviceMotion?, error:NSError?) -> Void in
            if let motion = motion {
                let rotationRate = motion.rotationRate
                let gravity = motion.gravity
                let userAcc = motion.userAcceleration
                let attitude = motion.attitude
            }
        }
    }
}
```

```
let gyroscopeText =
    String(format: "Rotation Rate:\n-----\n" +
        "x: %+.2f\ny: %+.2f\nz: %+.2f\n",
        rotationRate.x, rotationRate.y, rotationRate.z)
let acceleratorText =
    String(format: "Acceleration:\n-----\n" +
        "Gravity x: %+.2f\tUser x: %+.2f\n" +
        "Gravity y: %+.2f\tUser y: %+.2f\n" +
        "Gravity z: %+.2f\tUser z: %+.2f\n",
        gravity.x, userAcc.x, gravity.y,
        userAcc.y, gravity.z, userAcc.z)
let attitudeText =
    String(format: "Attitude:\n-----\n" +
        "Roll: %+.2f\nPitch: %+.2f\nYaw: %+.2f\n",
        attitude.roll, attitude.pitch, attitude.yaw)

DispatchQueue.main.async {
    self.gyroscopeLabel.text = gyroscopeText
    self.accelerometerLabel.text = acceleratorText
    self.attitudeLabel.text = attitudeText
}
```

Сначала мы импортируем каркас Core Motion и вводим два дополнительных свойства в класс контроллера представления следующим образом:

```
import UIKit
import CoreMotion

class ViewController: UIViewController {
    @IBOutlet var gyroscopeLabel: UILabel!
    @IBOutlet var accelerometerLabel: UILabel!
    @IBOutlet var attitudeLabel: UILabel!
    private let motionManager = CMMotionManager()
    private let queue = OperationQueue()
```

Этот код создает экземпляр класса CMMotionManager, который будет использоваться для отслеживания событий, связанных с движением мобильного устройства. Кроме того, в этих строках кода организуется очередь операций, а по существу, контейнер для операций, которые следует выполнить.

ПРЕДУПРЕЖДЕНИЕ. Для нормальной работы диспетчера движения требуется очередь, куда он будет ставить отдельными частями операции, которые ему предстоит выполнять по мере наступления соответствующих событий, как указано в предоставляемом ему замыканием. Возникает невольное искушение воспользоваться для этой цели стандартной системной очередью, но в документации к классу CMMotionManager дано явное предупреждение ни в коем случае этого не делать. Дело в том, что стандартная системная очередь может оказаться переполненной подобными событиями, затрудняя тем самым обработку самых важных для системы событий.

Далее мы вводим в методе `viewDidLoad()` код для запроса обновлений движения мобильного устройства и соответственно обновляем метки показаниями гироскопа, акселерометра и пространственной ориентации. Сначала нужно проверить, действительно ли мобильное устройство оснащено необходимым оборудованием для предоставления сведений о его движении. Все выпущенные до сих пор мобильные устройства, работающие под управлением системы iOS, оснащены таким оборудованием, но на всякий случай целесообразно проверять на его наличие и все устройства, которые будут выпускаться в дальнейшем. Затем мы задаем временной промежуток между последовательными обновлениями, указываемый в секундах. В данном случае мы запрашиваем временной промежуток 1/10 с, как показано ниже. Впрочем, это не гарантирует, что мы будем получать обновления именно с такой частотой. В действительности это не более чем предел, обозначающий оптимальную частоту обновления, с которой диспетчер движения будет предоставлять нужные нам данные. На практике он может обновлять данные еще реже.

```
if (motionManager.isDeviceMotionAvailable) {
    motionManager.deviceMotionUpdateInterval = 0.1;
```

Затем мы даем диспетчеру движения команду приступить к предоставлению обновленных данных о движении мобильного устройства. С этой целью мы передаем ему замыкание, где определяется операция, которая должна выполняться всякий раз, когда происходит обновление, а также очередь, куда эта операция ставится на выполнение. В данном случае в замыкании получается объект типа `CMDeviceMotion`, содержащий самые последние данные движения мобильного устройства, а возможно, и объект типа `NSError`, если при получении этих данных происходит ошибка, как показано ниже.

```
motionManager.startDeviceMotionUpdates(to: queue) {
    (motion:CMDeviceMotion?, error:NSError?) -> Void in
```

Рассмотрим теперь содержимое самого замыкания. Сначала в нем из текущих данных движения мобильного устройства формируются символьные строки, которые затем выводятся на месте меток. Но в данном случае этого нельзя сделать непосредственно, поскольку такие классы каркаса UIKit, как `UILabel`, как правило, нормально функционируют только в том случае, если они доступны из основного потока выполнения. Поскольку рассматриваемый здесь код будет выполняться в очереди типа `NSOperationQueue`, мы просто не знаем, в каком именно потоке выполнения это произойдет. Поэтому вызываем функцию `Dispatch Queue.main.async` из библиотеки GCD, чтобы передать управление основному потоку выполнения, прежде чем устанавливать свойства, задающие текст меток.

Значения, получаемые от гироскопа, доступны через свойство `rotationRate` объекта типа `CMDeviceMotion`, передаваемого в замыкании. Свойство `rotationRate` относится к типу `RotationRate`, т.е. к простой структуре, содержащей три числовых значения с плавающей точкой, обозначающих скорости

вращения вокруг осей x, y и z. Данные от акселерометра имеют более сложную структуру, поскольку каркас Core Motion сообщает два разных значения: ускорение под действием тяготения и любое дополнительное ускорение, вызванное силами, прилагаемыми пользователем. Эти значения извлекаются из свойств gravity и userAcceleration, относящихся к типу CMAcceleration, — еще одной простой структуры, в которой хранятся данные ускорения по осям x, y и z. В заключение сведения о пространственной ориентации мобильного устройства сообщаются и сохраняются в свойстве attitude, которое относится к типу CMAttitude. Подробнее об этом речь пойдет далее, когда настанет время запускать данное приложение на выполнение.

Прежде чем опробовать данное приложение, нужно сделать еще кое-что. Мы собираемся по-разному перемещать и вращать мобильное устройство, чтобы посмотреть, каким образом значения в структуре CMDeviceMotion согласуются с тем, что происходит с мобильным устройством. Было бы нежелательно, чтобы при этом активизировался режим автоматического вращения. Во избежание этого выберите сначала данный проект в навигаторе проектов, а затем цель проекта MotionMonitor и далее вкладку General. Перейдите сначала к разделу Device Orientation, а затем к разделу Deployment Info и установите флажок Portrait, убедившись, что не выбраны три других вида ориентации. В итоге действие данного приложения будет ограничиваться только книжной ориентацией. Скомпилируйте и запустите на выполнение рассматриваемое здесь приложение на любом доступном вам мобильном устройстве, работающем под управлением системы iOS, чтобы опробовать его (рис. 20.2).

Наклоняя мобильное устройство в разные стороны, вы должны заметить, как меняются показания гироскопа, акселерометра и пространственной ориентации в каждом новом положении устройства. Если вы держите мобильное устройство в устойчивом положении, то эти показания остаются без изменения. Если же мобильное устройство стоит прямо (независимо от его ориентации),

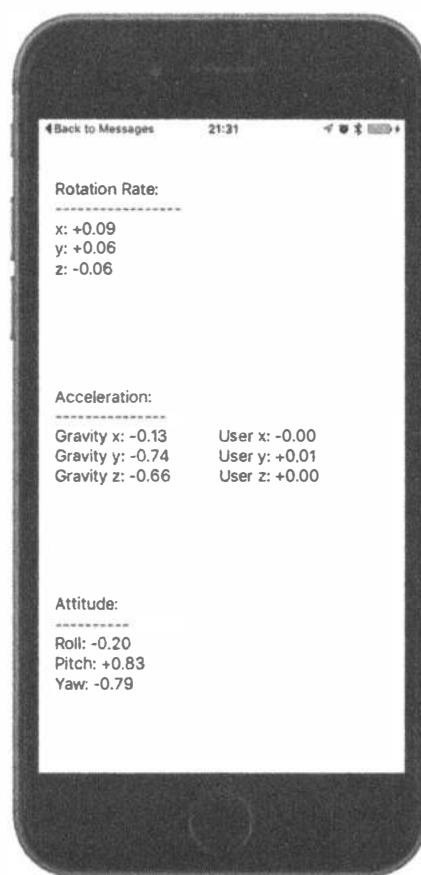


Рис. 20.2. Приложение Motion Monitor, выполняющееся на мобильном устройстве iPhone. К сожалению, попытка выполнить данное приложение в симуляторе приводит к сообщениям об ошибках

показания гироскопа устанавливаются на отметке, близкой к нулевой. Если же вы повернете мобильное устройство, то заметите, как показания гироскопа изменятся в зависимости от того, как он вращается относительно различных осей. Эти показания тотчас возвращаются к нулевой отметке, как только вы прекратите перемещать мобильное устройство. Все эти результаты испытаний мы вскоре проанализируем.

Упреждающий доступ к данным движения

Как пояснялось выше, доступ к данным движения осуществляется путем передачи диспетчеру движения типа `CMMotionManager` замыканий для их вызова при появлении движения. Подобного рода обработка данных управляемого событиями движения вполне пригодна для обычного приложения Сосоа, но иногда ее оказывается недостаточно для приложений с особыми требованиями. Например, интерактивные игры обычно выполняются в непрерывном цикле обработки вводимых пользователем данных, обновления состояния игры и перерисовки экрана. В подобных случаях событийный подход оказывается не вполне пригодным, поскольку предполагает реализацию объекта, ожидающего событий, связанных с движением, запоминающего последние положения, получаемые от датчика по мере их предоставления, а также готового предоставить данные обратно основному игровому циклу по мере надобности. К счастью, в класс `CMMotionManager` встроено специальное решение. Вместо передачи замыканий мы можем просто дать диспетчеру движения данного класса команду активизировать датчики, используя метод `startDeviceMotionUpdates()`. После этого нам остается только считывать показания датчиков в любой удобный момент непосредственно из диспетчера движения.

Внесем изменения в приложение `MotionMonitor`, чтобы применить данный подход и заодно показать на конкретном примере, каким образом он реализуется на практике. Прежде всего сделайте копию папки своего проекта `Motion Monitor`.

ЗАМЕЧАНИЕ. Полная версия данного проекта находится в папке 20 – `MonitorMotion2` с исходным кодом примеров, прилагаемым к этой книге.

Закройте проект, открытый в среде Xcode, и вместо него откройте его новую копию и перейдите непосредственно к исходному файлу `ViewController.swift`. Удалите свойство `queue` и введите новое свойство как указатель на объект типа `NSTimer`, который будет инициировать все обновления отображаемых на экране данных о движении:

```
class ViewController: UIViewController {
    @IBOutlet var gyroscopeLabel: UILabel!
    @IBOutlet var accelerometerLabel: UILabel!
    @IBOutlet var attitudeLabel: UILabel!
    private let motionManager = CMMotionManager()
    private var updateTimer: Timer!
```

Затем удалите метод viewDidLoad(), поскольку он нам больше не понадобится. Мы воспользуемся таймером, чтобы собирать данные движения непосредственно из диспетчера движения через каждую десятую долю секунды, а не доставлять их замыканию. Нам нужно, чтобы таймер (и сам диспетчер движения) был активным только во время очень небольшого промежутка времени, когда фактически отображается текущее представление. Благодаря этому мы сможем свести к самому минимуму использование основного игрового цикла. Этой цели мы можем добиться, реализовав методы viewWillAppeare() и viewDidDisappear(), как показано в листинге 20.2.

Листинг 20.2. Методы viewWillAppeare и viewDidDisappear

```
override func viewWillAppeare(_ animated: Bool) {
    super.viewWillAppeare(animated)
    if motionManager.isDeviceMotionAvailable {
        motionManager.deviceMotionUpdateInterval = 0.1
        motionManager.startDeviceMotionUpdates()
        updateTimer =
            Timer.scheduledTimer(timeInterval: 0.1, target: self,
                selector: #selector(Controller.updateDisplay),
                userInfo: nil, repeats: true)
    }
}

override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
    if motionManager.isDeviceMotionAvailable {
        motionManager.stopDeviceMotionUpdates()
        updateTimer.invalidate()
        updateTimer = nil
    }
}
```

В теле метода viewWillAppeare() сначала вызывается метод startDeviceMotionUpdates() из диспетчера движения, чтобы начать обновление сведений о движении мобильного устройства, а затем создается новый таймер, который устанавливается на срабатывание каждую десятую долю секунды и вызов метода updateDisplay(), который мы еще не создали. Поэтому введите этот метод ниже метода viewDidDisappear(), как показано в листинге 20.3.

Листинг 20.3. Метод updateDisplay в файле ViewController.swift

```
func updateDisplay() {
    if let motion = motionManager.deviceMotion {
        let rotationRate = motion.rotationRate
        let gravity = motion.gravity
        let userAcc = motion.userAcceleration
        let attitude = motion.attitude

        let gyroscopeText =
            String(format: "Rotation Rate:\n-----\n" +
                "x: %+.2f\ny: %+.2f\nz: %+.2f\n",
                rotationRate.x, rotationRate.y, rotationRate.z)
```

```
let acceleratorText =
    String(format: "Acceleration:\n-----\n" +
        "Gravity x: %.2f\tUser x: %.2f\n" +
        "Gravity y: %.2f\tUser y: %.2f\n" +
        "Gravity z: %.2f\tUser z: %.2f\n",
        gravity.x, userAcc.x, gravity.y,
        userAcc.y, gravity.z, userAcc.z)
let attitudeText =
    String(format: "Attitude:\n-----\n" +
        "Roll: %.2f\nPitch: %.2f\nYaw: %.2f\n",
        attitude.roll, attitude.pitch, attitude.yaw)

DispatchQueue.main.async {
    self.gyroscopeLabel.text = gyroscopeText
    self.accelerometerLabel.text = acceleratorText
    self.attitudeLabel.text = attitudeText
}
}
```

Тело этого метода является копией тела замыкания в предыдущем варианте данного примера приложения, за исключением того, что объект типа CMDeviceMotion получается непосредственно из диспетчера движения. Обратите внимание на проверку пустого значения `nil`. Она требуется потому, что таймер может сработать до того, как диспетчер движения получит первую выборку данных. Скомпилируйте и запустите на выполнение рассматриваемое здесь приложение на своем мобильном устройстве. Оно должно действовать точно так же, как и первая его версия. Итак, вы ознакомились с двумя способами доступа к данным движения. Пользуйтесь тем из них, который лучше всего подходит для ваших приложений.

Результаты гирокопических и пространственных измерений

Гирокоп измеряет скорость, с которой мобильное устройство вращается вокруг осей x, y и z. На рис. 20.1 показано, каким образом эти оси связаны с корпусом мобильного устройства. Положите сначала мобильное устройство прямо на стол. Когда оно неподвижно, скорость вращения вокруг всех трех осей близка к нулю, а следовательно, показания крена, тангажа и рыскания также близки к нулю. Теперь поверните понемногу мобильное устройство по часовой стрелке. При этом вы обнаружите, что скорость вращения вокруг оси z становится отрицательной. Чем быстрее вы вращаете мобильное устройство, тем большим становится абсолютное значение скорости вращения вокруг этой оси.

Как только вы прекратите вращение, его скорость станет снова нулевой, но не рыскание, которое обозначает угол, на который мобильное устройство поворачивается вокруг оси z относительно своего исходного положения. Если вы повернете мобильное устройство против часовой стрелки, рыскание будет увеличиваться в отрицательных величинах до тех пор, пока устройство не отклонится на 180° от своего исходного положения, когда рыскание достигнет величины около -3. Если же вы продолжите вращать мобильное устройство по часовой

стрелке, рыскание резко возрастет до величины больше +3, а после вращения в исходное положение уменьшится до нуля. То же самое произойдет, если вы начнете вращать мобильное устройство против часовой стрелки, но на этот раз величина рыскания первоначально окажется положительной. Угол рыскания фактически измеряется в радианах, а не в градусах. Поворот на 180° равнозначен повороту на π радиан, именно поэтому максимальная величина рыскания составляет около 3 (ведь число π чуть больше, чем 3,14).

Снова положив мобильное устройство прямо на стол, приподнимите его за верхний край, не отрывая основания от стола. Такое движение означает вращение вокруг оси x , о чем свидетельствует увеличение скорости вращения вокруг этой оси в положительных величинах до тех пор, пока вы не приведете мобильное устройство в устойчивое положение, и тогда скорость вращения вокруг оси x снова станет нулевой. Теперь обратите внимание на тангаж. Он увеличился на величину, которая зависит от угла, на который вы приподняли верхний край мобильного устройства. Если вы поднимете мобильное устройство в вертикальное положение, тангаж достигнет величины около 1,5. Как и рыскание, тангаж измеряется в радианах, и поэтому мобильное устройство в вертикальном положении окажется повернутым на 90° , или $\pi/2$ радиан, что чуть больше, чем 1,5. Если вы снова положите мобильное устройство ровно на стол, а затем приподнимете его на этот раз за основание, не отрывая верхний край от стола, то, по существу, повернете его против часовой стрелки вокруг оси x и при этом обнаружите, что скорость вращения вокруг этой оси и тангаж станут отрицательными.

В заключение, положив мобильное устройство прямо на стол, приподнимите его левый край, не отрывая правый край от стола. Такое движение означает вращение вокруг оси y , о чем свидетельствует увеличение скорости вращения вокруг этой оси. Общий угол поворота вокруг оси y в любой точке можно получить из величины крена. Она будет составлять около 1,5 (фактически $\pi/2$), когда мобильное устройство стоит прямо на правом боку, и возрастет вплоть до величины числа π , если перевернуть мобильное устройство лицевой стороной вниз, но чтобы увидеть это, потребуется стеклянный стол.

Подводя итог, следует сказать, что по скоростям вращения можно определить, насколько быстро мобильное устройство вращается вокруг каждой из трех осей, а по величинам крена, тангажа и рыскания — получить текущий общий угол поворота вокруг каждой из этих осей относительно исходной ориентации мобильного устройства.

Результаты акселерометрических измерений

Как упоминалось ранее, акселерометр, встроенный в мобильное устройство iPhone, измеряет ускорение по трем осям координат и предоставляет данные этих измерений в структуре типа `CMAcceleration`. Каждая структура типа `CMAcceleration` состоит из трех полей, x , y и z , в которых содержатся значения в

формате с плавающей точкой. Нулевое значение означает, что акселерометр не обнаруживает никакого движения по данной конкретной оси координат, а положительное или отрицательное значение — действие силы в одном направлении. Например, отрицательное значение в поле `u` обозначает восприятие акселерометром силы, тяущей вниз, а это, вероятно, указывает на то, что пользователь держит мобильное устройство прямо, т.е. в книжной ориентации. В то же время положительное значение в поле `u` означает действие некоторой силы в противоположном направлении, а это может указывать на то, что пользователь перемещает мобильное устройство по направлению вниз. Объект типа `CMDeviceMotion` отдельно сообщает об ускорении по каждой из трех осей под действием тяготения и любых других сил, прилагаемых пользователем.

Так, если вы держите мобильное устройство горизонтально, то тяготение приближается к величине -1 по оси `z`, а все составляющие ускорения, сообщаемого пользователем, близки к нулю. Если вы быстро поднимете мобильное устройство вверх, сохранив его горизонтальное положение, то обнаружите, что величины тяготения останутся прежними, тогда как величина ускорения, сообщаемого пользователем по оси `z`, станет положительной. В одних приложениях удобно разделять величины тяготения и ускорения, сообщаемого пользователем, а в других требуется общая величина ускорения, которую можно получить, сложив все составляющие в свойствах `gravity` и `userAcceleration` объекта типа `CMDeviceMotion`.

Руководствуясь в качестве образца схемой, приведенной на рис. 20.1, рассмотрим некоторые результаты измерений, производимых акселерометром (рис. 20.3). Следует, однако, иметь в виду, что в реальной ситуации измеренные акселерометром величины ускорения практически никогда не бывают такими идеальными, поскольку акселерометр — это настолько чувствительный прибор, что, как правило, воспринимает самое малейшее движение, регистрируя малейшее проявление силы по всем трем осям координат. Это практическая, а не школьная физика.

В приложениях независимых разработчиков акселерометр чаще всего находит применение в качестве контроллера для игр. Далее в этой главе будет создана прикладная программа, в которой акселерометр применяется для ввода данных, а до тех пор рассмотрим еще одно типичное применение акселерометра: обнаружение сотрясений.

Распознавание сотрясений

Подобно жесту, сотрясение может быть использовано как форма ввода данных в приложение. Например, прикладная программа рисования `GLPaint`, относящаяся к одному из проектов с примерами кода для системы iOS, дает пользователю возможность стирать рисунки, потряхивая мобильным устройством, как в известной рисовальной игре `Etch A Sketch` (Волшебный экран). Сотрясения

обнаруживаются относительно просто. Для этого достаточно проверить, не превышает ли абсолютная величина ускорения по одной из осей координат заданный порог. В обычных условиях эксплуатации мобильного устройства по одной из трех осей координат нередко регистрируется ускорение около 1,3 g, а для получения намного больших величин ускорения, как правило, требуется намеренно прикладывать силу. По-видимому, акселерометр, встроенный в мобильное устройство, не в состоянии регистрировать ускорение свыше 2,3 g (по крайней мере, так показывает наш опыт). Поэтому не следует устанавливать порог выше этой величины ускорения.

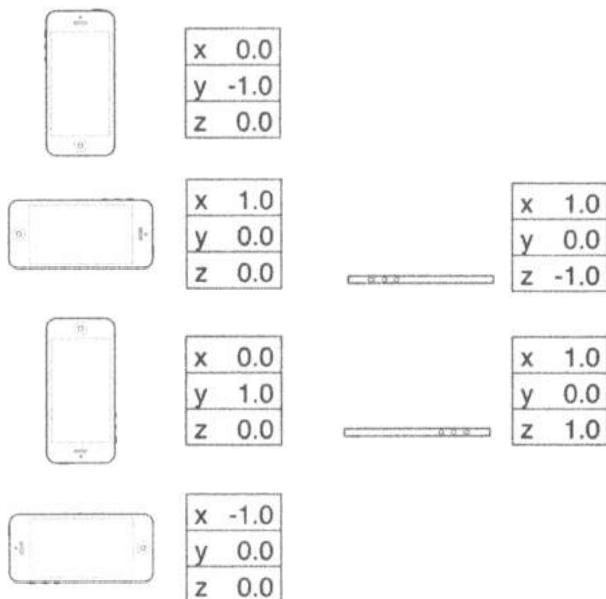


Рис. 20.3. Идеализированные величины ускорения для различных видов ориентации мобильного устройства

Для того чтобы обнаружить сотрясение, достаточно проверить, не превышает ли абсолютная величина ускорения порог 1,5 g при слабом сотрясении и порог 2,0 g при сильном сотрясении. Для этого введите приведенный ниже фрагмент кода в замыкание, передаваемое диспетчеру движения в примере приложения MotionMonitor. В этом фрагменте кода обнаруживается всякое движение по любой оси с ускорением свыше 2,0 g.

```
let userAcc = motion.userAcceleration
if (fabsf(Float(userAcc.x)) > 2.0
    || fabsf(Float(userAcc.y)) > 2.0
    || fabsf(Float(userAcc.z)) > 2.0) {
    // Сделать здесь что-нибудь...
}
```

Встроенные средства обнаружения сотрясений

Сотрясения можно обнаруживать и по-другому: средствами, встроенными в цепочку реагирующих элементов. Напомним, что в главе 18 мы реализовали методы, подобные `touchesBegan(_ :withEvent:)`, для обнаружения касаний. Аналогичные методы предоставляются в iOS и для обнаружения движения. К их числу относятся следующие три метода распознавания.

- Когда движение начинается, метод `motionBegan(_ :withEvent:)` передается первому реагирующему элементу и далее по цепочке реагирующих элементов, как пояснялось в главе 18.
- Когда движение завершается, первому реагирующему элементу передается метод `motionEnded(_ :withEvent:)`.
- Если мобильный телефон звонит или же во время сотрясения происходит какое-нибудь другое прерывающее действие, первому реагирующему элементу передается метод `motionCancelled(_ :withEvent:)`.

В качестве первого аргумента каждому из этих методов передается подтип события, например `UIEventSubtypeMotionShake`. Это означает, что сотрясение можно обнаружить, фактически не обращаясь непосредственно к классу `CMMotionManager`. Для этого достаточно переопределить соответствующие методы, регистрирующие движение в текущем представлении или контроллере представления, и тогда они будут вызываться автоматически, как только пользователь потрясет свой мобильный телефон. Встроенными средствами обнаружения сотрясений можно пользоваться вместо организации их обнаружения вручную, как пояснялось ранее в этой главе, если нет особой необходимости в дополнительных средствах контроля сотрясающего жеста. Тем не менее ручной способ был продемонстрирован в этой главе на тот случай, если дополнительный контроль сотрясений все-таки потребуется. Теперь, когда вы усвоили основной принцип обнаружения сотрясений, посмотрим, удастся ли нам сломать мобильный телефон.

Сотрясение на грани поломки

Мы планируем разработать приложение, обнаруживающее сотрясения и затем приводящее мобильный телефон в такое состояние, в котором он внешне выглядит и издает звуки так, как будто сломался от сотрясения. После запуска на выполнение это приложение выводит на экран изображение, похожее на начальный экран iPhone (рис. 20.4). Однако если тряхнуть мобильный телефон достаточно сильно, он издаст такой звук, который не хотелось бы услышать ни одному пользователю бытовой электронной техники. Более того, его экран будет выглядеть так, как показано на рис. 20.5. Не беспокойтесь. Вы можете легко вернуть свое устройство iPhone в прежнее исправное состояние, просто дотронувшись до его экрана.

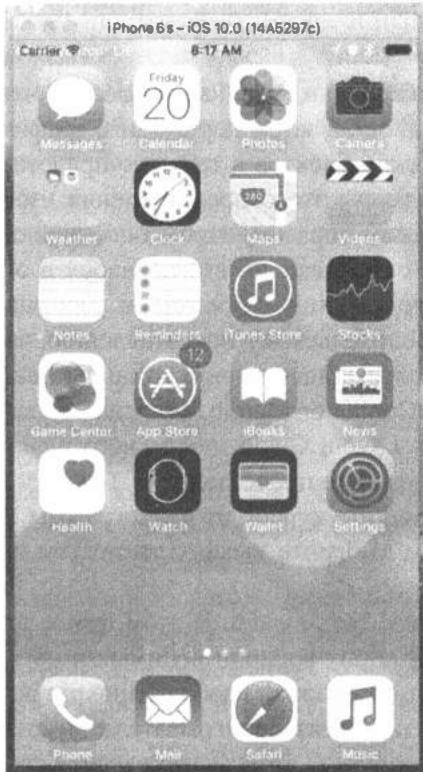


Рис. 20.4. Внешне приложение ShakeAndBreak выглядит довольно безопасно...

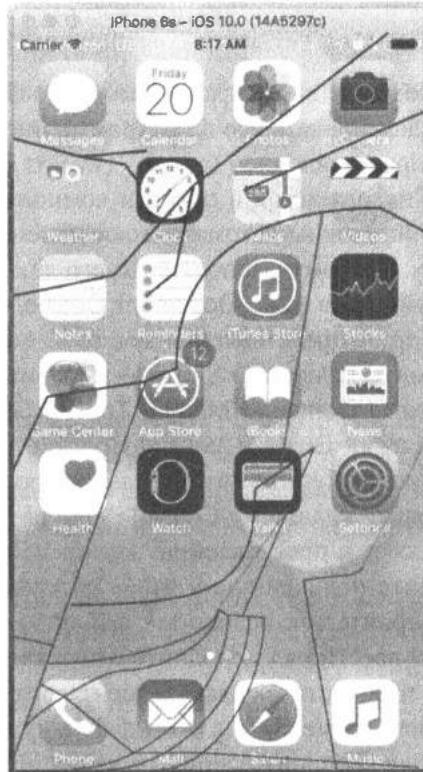


Рис. 20.5. Но если встряхнуть мобильный телефон, то он будет выглядеть так, будто сломался!

Создайте новый проект в среде Xcode, используя шаблон Single View Application. Присвойте новому проекту имя ShakeAndBreak и выберите тип устройства iPhone. В отличие от остальных примеров приложений в этой книге, данное приложение пригодно для работы только на мобильном телефоне iPhone, поскольку применяемые в нем изображения имеют размеры, подходящие для экрана модели iPhone 5. Разумеется, этот пример проекта можно распространить и на мобильное устройство iPad, если сформировать дополнительные приложения. В папке 20 – Images and Sounds с исходным кодом примеров, прилагаемым к этой книге, вы найдете один звуковой файл и два файла изображений, необходимых для построения рассматриваемого здесь приложения. Выберите папку ресурсов Assets.xcassets и перетащите в нее файлы Home.png и Broken Home.png. После этого перетащите в окно навигатора проекта звуковой файл glass.wav.

Начнем с создания контроллера представления. Нам придется создать выход для ссылки на представление изображений, чтобы иметь возможность изменять изображение, отображаемое на экране. Щелкните на исходном файле ViewController.swift и добавьте в него следующее свойство.

756 ГЛАВА 20 ❁ РАСПОЗНАВАНИЕ ОРИЕНТАЦИИ И ПЕРЕМЕЩЕНИЯ УСТРОЙСТВА

```
class ViewController: UIViewController {  
    @IBOutlet var imageView: UIImageView!
```

Сохраните этот файл. Выберите файл раскадровки Main.storyboard, чтобы отредактировать его в программе Interface Builder, а затем перетащите графическое представление (Image View) из библиотеки объектов в текущее представление, находящееся в области компоновки. Размеры этого представления изображений должны автоматически изменяться таким образом, чтобы оно заняло все окно. Поэтому расположите его так, чтобы оно идеально вписывалось в окно. Перейдите в окно Document Outline, нажмите клавишу <Control> и перетащите курсор от представления Image View к его родительскому представлению, затем нажмите клавишу <Shift> и выберите из всплывающего меню команды Leading Space to Container Margin, Trailing Space to Container Margin, Vertical Spacing to Top Layout Guide и Vertical Spacing to Bottom Layout Guide, после чего нажмите клавишу <Return>, чтобы зафиксировать положение представления Image View. Нажмите клавишу <Control> еще раз и перетащите курсор от пиктограммы View Controller к представлению Image View, затем выберите выход imageView и сохраните полученную в итоге раскадровку.

Вернитесь к исходному файлу ViewController.swift. В этом файле предстоит ввести ряд дополнительных свойств для обоих воспроизводимых изображений, чтобы отслеживать момент, когда следует отображать изображение разбитого стекла. Кроме того, в этом файле нужно ввести объект проигрывателя звукозаписей для воспроизведения звука разбиваемого стекла. С этой целью введите строки кода, выделенные ниже полужирным шрифтом, в начало данного файла.

```
import UIKit  
import AVFoundation  
  
class ViewController: UIViewController {  
    @IBOutlet var imageView: UIImageView!  
    private var fixed: UIImage!  
    private var broken: UIImage!  
    private var brokenScreenShowing = false  
    private var crashPlayer: AVAudioPlayer?
```

Ведите в тело метода viewDidLoad() следующий код.

```
if let url = url = Bundle(for: type(of: self)).url(forResource:"glass",  
                                              withExtension:"wav"){  
do {  
    crashPlayer = try AVAudioPlayer(contentsOf: url, fileTypeHint:  
                                         AVFileTypeWAVE)  
} catch let error as NSError {  
    print("Audio error! \(error.localizedDescription)")  
}  
}  
  
fixed = UIImage(named: "Home")  
broken = UIImage(named: "HomeBroken")  
imageView.image = fixed
```

Итак, мы инициализировали переменную `url` для ссылки на звуковой файл, а также экземпляр класса `AVAudioPlayer`, предназначенного для воспроизведения звука. После быстрой проверки на правильность установки проигрывателя звукозаписей мы загрузили нужные изображения и расположили первое из них на месте. Далее введите следующий новый метод:

```
override func motionEnded(_ motion: UIEventSubtype, with event: UIEvent?) {
    if !brokenScreenShowing && motion == .motionShake {
        imageView.image = broken;
        crashPlayer?.play()
        brokenScreenShowing = true;
    }
}
```

Этот метод переопределяет метод `motionEnded(_:withEvent:)` класса `UIResponder` и будет вызываться всякий раз, когда происходит сотрясение. После того как проверено, что разбитый экран еще не отображается и что исключительное событие действительно связано с сотрясением, в данном методе воспроизводятся изображение и звук разбиваемого стекла.

Последний из рассматриваемых здесь методов должен быть уже хорошо вам знаком. Он вызывается при касании экрана. В этом методе осуществляется только возврат к изображению неповрежденного экрана и логическому значению `false` в переменной `brokenScreenShowing`, как показано ниже.

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    imageView.image = fixed
    brokenScreenShowing = false
}
```

Скомпилируйте и запустите данное приложение, а затем испытайте его в действии. Если в вашем распоряжении нет мобильного устройства под управлением системы iOS, вы все равно можете опробовать данное приложение. Симулятор не воспроизводит аппаратные функции акселерометра, но содержит отдельный пункт меню для имитации события, связанного с сотрясением, а следовательно, данное приложение будет работать и в нем. Поэкспериментировав немного с сотрясениями мобильного телефона вплоть до его “поломки”, возвращайтесь к чтению этой главы. Из остальной ее части вы узнаете, как использовать акселерометр в качестве контроллера для игровых и прочих прикладных программ.

Акселерометр в качестве контроллера направления

Зачастую вместо кнопок для управления передвижением персонажа или объекта в играх применяется акселерометр. Например, в игре, имитирующей автогонки, вращением мобильного устройства под управлением системы iOS воспроизводится вращение руля автомашины, наклоном вперед — ускорение, а опрокидыванием назад — замедление движения. Конкретное применение акселерометра в качестве контроллера зависит от особенностей механических движений, воспроизводимых в игре. В простейшем случае можно просто взять

величину ускорения по одной из осей, умножить ее на определенное число, а затем добавить к координатам управляемого объекта. В сложных играх, в которых физические свойства моделируются более правдоподобно, приходится корректировать скорость движения управляемого объекта по значениям, получаемым от акселерометра.

Наибольшая трудность применения акселерометра в качестве контроллера связана с тем, что обратный вызов метода делегата в указанном промежутке времени не гарантируется. Так, если дать диспетчеру движения команду считывать показания акселерометра 60 раз в секунду, то с уверенностью можно утверждать лишь одно: обновление этих показаний не будет происходить чаще, чем 60 раз в секунду. Это означает, что получение 60 обновленных показаний акселерометра через равные промежутки времени каждую секунду совсем не гарантируется. Поэтому, если вы выполняете анимацию на основании входных данных от акселерометра, вам придется отслеживать промежутки времени между последовательными обновлениями показаний акселерометра и учитывать их в выражениях, определяющих расстояния, на которые переместились объекты.

Катание шаров

В следующем приложении мы попробуем перемещать спрайт по экрану iPhone, наклоняя этот мобильный телефон. Это очень простой пример применения акселерометра для получения входных данных. С этой целью мы воспользуемся функциями из библиотеки Quartz 2D для управления анимацией.

ЗАМЕЧАНИЕ. Как правило, для разработки игровых и других прикладных программ, в которых требуется плавная анимация, применяется каркас Sprite Kit, библиотека OpenGL ES или каркас Metal. Однако в рассматриваемом здесь приложении ради простоты и сокращения объема кода, не связанного с применением акселерометра, используется библиотека Quartz 2D.

Если при выполнении данного приложения наклонить мобильный телефон iPhone, шарик будет кататься по его экрану, как по поверхности стола (рис. 20.6). При наклоне влево шарик должен перекатываться влево. Если наклонить iPhone сильнее, шарик станет перекатываться быстрее. Если же опрокинуть iPhone назад, шарик замедлит свое движение, а затем начнет двигаться в другом направлении.

Создайте новый проект, используя шаблон Single View Application. Выберите тип устройства Universal и присвойте данному проекту имя Ball. В папке 20 – Images and Sounds с исходным кодом примеров, прилагаемым к этой книге, вы найдете файл изображения ball.png. Используя навигатор проекта, перетащите файл ball.png в папку ресурсов Assets.xcassets.

Затем выберите проект Ball в навигаторе проектов и закладку General для цели проекта Ball. Перейдите сначала к разделу Device Orientation, а затем к разделу Deployment Info и установите флажок Portrait, убедившись, что остальные флажки сброшены (это было сделано при разработке приложения

MotionMonitor ранее в данной главе). Благодаря этому отменяются стандартные изменения в ориентации пользовательского интерфейса на экране. Ведь нам требуется перекатывать шарик по экрану, а не изменять ориентацию пользовательского интерфейса при перемещении мобильного устройства.

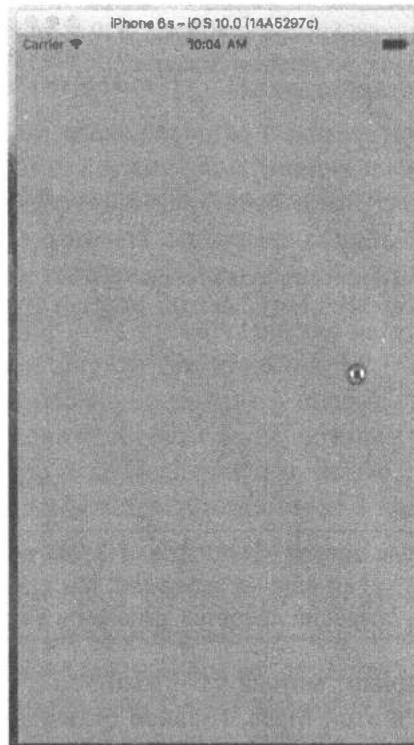


Рис. 20.6. Приложение, позволяющее просто перекатывать шарик по экрану

Щелкните на папке Ball и выберите команду меню **File⇒New⇒File....**. Затем выберите шаблон **Cocoa Touch Class** в разделе **iOS** и щелкните на кнопке **Next**. Сделайте новый класс подклассом, производным от класса **UIView**, назовите его **BallView** и щелкните на кнопке **Create**. Мы еще вернемся к правке этого класса несколько позже. Выберите файл раскладовки **Main.storyboard**, чтобы отредактировать его в программе **Interface Builder**. Щелкните на пиктограмме **View** и воспользуйтесь инспектором идентичности, чтобы изменить класс представления с **UIView** на **BallView**. Перейдите к инспектору атрибутов и измените цвет фона представления на светло-серый, выбрав пункт **Light Gray Color** из раскрывающегося списка **Background**. После этого сохраните полученную в итоге раскладовку.

Теперь самое время отредактировать исходный файл **ViewController.swift**. Введите в самом начале этого файла строки кода из листинга 20.4.

Листинг 20.4. Модифицированный метод viewDidLoad
в файле ViewController.swift

```
import UIKit
import CoreMotion

class ViewController: UIViewController {
    private static let updateInterval = 1.0/60.0
    private let motionManager = CMMotionManager()
    private let queue = OperationQueue()

    override func viewDidLoad() {
        super.viewDidLoad()
        // Дополнительная настройка после загрузки представления,
        // обычно из nib-файла.

        motionManager.startDeviceMotionUpdates(to: queue) {
            (motionData: CMDeviceMotion?, error: NSError?) -> Void in
            let ballView = self.view as! BallView
            ballView.acceleration = motionData!.gravity
            DispatchQueue.main.async {
                ballView.update()
            }
        }
    }
}
```

ЗАМЕЧАНИЕ. Вводя код в теле метода viewDidLoad(), вы, возможно, заметили ошибку, потому что класс BallView еще не завершен. Мы сделаем это в дальнейшем, поскольку большую часть операций придется выполнять в классе BallView.

Приведенный выше вариант метода viewDidLoad() очень похож на тот, который применялся ранее в этой главе. Главное отличие заключается в том, что в данном варианте мы объявляем намного более короткий интервал обновления: 60 раз в секунду. В замыкании, в котором мы даем диспетчеру движения команду на выполнение, когда акселерометр обновляет свои показания, мы передаем объект ускорения текущему представлению, а затем вызываем метод update(), обновляющий положение шарика на экране в зависимости от величины ускорения и времени, прошедшего с момента последнего обновления. Это замыкание может выполняться в любом потоке выполнения. В то же время методы, принадлежащие объектам UIKit (в том числе классу UIView), могут надежно вызываться только из основного потока выполнения. Поэтому и на этот раз мы намеренно вызываем метод update() из основного потока выполнения.

Построение представления для шарика

Выберите исходный файл BallView.swift. В этом файле нужно импортировать каркас Core Motion и ввести свойство, которое будет использоваться в контроллере для передачи величины ускорения и значений пяти других свойств, объявляемых в реализации класса BallView, как выделено ниже полужирным шрифтом.

```
import UIKit
import CoreMotion

class BallView: UIView {
    var acceleration = CMAcceleration(x: 0, y: 0, z: 0)
    private let image = UIImage(named: "ball")!
    private var currentPoint: CGPoint = CGPoint.zero
    private var ballXVelocity = 0.0
    private var ballYVelocity = 0.0
    private var lastUpdateTime = Date()
}
```

Рассмотрим объявленные выше свойства и назначение каждого из них. Первое свойство, `acceleration`, служит для хранения последних величин ускорения, которые контроллер получает из обновленных данных движения. Следующее свойство, `image`, служит для ссылки на спрайт, который нам предстоит перемещать по экрану, как показано ниже.

```
private let image = UIImage(named: "ball")!
```

Свойство `currentPoint` служит для хранения текущего положения шарика, как показано ниже. Мы воспользуемся этим и предыдущим положением шарика, которое предоставляется средствами Swift автоматически, чтобы построить прямоугольник обновления, охватывающий как новые, так и прежние положения шарика, нарисовав его в новом месте и стерев в прежнем.

```
private var currentPoint: CGPoint = CGPoint.zero
```

Кроме того, мы объявляем еще два свойства для отслеживания текущей скорости движения шарика в двух направлениях, как показано ниже. И хотя это будет не очень сложная имитация, нам все же придется перемещать шарик сходным с настоящим шариком образом. Расчет движения шарика будет представлен в следующем разделе. Мы будем получать величину ускорения из акселерометра и отслеживать с помощью этих двух свойств скорость движения шарика по двум осям координат.

```
private var ballXVelocity = 0.0
private var ballYVelocity = 0.0
```

Свойство `lastUpdateTime` устанавливается всякий раз, когда обновляется положение шарика. Мы воспользуемся им для расчета изменений скорости движения шарика в зависимости от промежутка времени между последовательными обновлениями и ускорения шарика.

Теперь напишем код для рисования и перемещения шарика по экрану. Прежде всего введите в исходный файл `BallView.swift` методы, представленные в листинге 20.5.

Листинг 20.5. Методы `init` в файле `ViewController.swift`

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

```

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

private func commonInit() -> Void {
    currentPoint = CGPoint(x: (bounds.size.width / 2.0) +
        (image.size.width / 2.0),
                           y: (bounds.size.height / 2.0) +
        (image.size.height / 2.0))
}

```

В обоих методах, `init?(coder:)` и `init(frame:)`, вызывается метод `commonInit()`. Представление катящегося шарика, созданное в файле раскадровки, инициализируется методом `initWithCoder()`. Поскольку метод `commonInit()` вызывается в обоих методах инициализации, класс данного представления можно благополучно создать из кода, как написанного вручную, так и автоматически сгенерированного в файле раскадровки. И это замечательно, поскольку классы любых представлений, подобных данному, могут использоваться повторно. Удалите комментарии из строк кода, реализующих метод `drawRect()`, введя в его тело строку кода, выделенную ниже полужирным шрифтом и обеспечивающую простую реализацию данного метода.

```

override func draw(_ rect: CGRect) {
    // Код для рисования
    image.draw(at: currentPoint)
}

```

Далее введите метод `update()` в конце класса `BallView`, как показано в листинге 20.6.

Листинг 20.6. Метод `update` класса `BallClass`

```

func update() -> Void {
    let now = Date()
    let secondsSinceLastDraw = now.timeIntervalSince(lastUpdateTime)
    ballXVelocity =
        ballXVelocity + (acceleration.x * secondsSinceLastDraw)
    ballYVelocity =
        ballYVelocity - (acceleration.y * secondsSinceLastDraw)

    let xDelta = secondsSinceLastDraw * ballXVelocity * 500
    let yDelta = secondsSinceLastDraw * ballYVelocity * 500
    currentPoint = CGPoint(x: currentPoint.x + CGFloat(xDelta),
                           y: currentPoint.y + CGFloat(yDelta))
    lastUpdateTime = now
}

```

В заключение введите в свойстве `currentPoint` следующий наблюдатель свойств (листинг 20.7).

Листинг 20.7. Наблюдатель currentProperty

```

var currentPoint : CGPoint = CGPoint.zero {
    didSet {
        var newX = currentPoint.x
        var newY = currentPoint.y
        if newX < 0 {
            newX = 0
            ballXVelocity = 0
        } else if newX > bounds.size.width - image.size.width {
            newX = bounds.size.width - image.size.width
            ballXVelocity = 0
        }
        if newY < 0 {
            newY = 0
            ballYVelocity = 0
        } else if newY > bounds.size.height - image.size.height {
            newY = bounds.size.height - image.size.height
            ballYVelocity = 0
        }
        currentPoint = CGPoint(x: newX, y: newY)

        let currentRect = CGRect(x: newX,
                                width: newX + image.size.width,
                                height: newY + image.size.height)
        let prevRect = CGRect(x: oldValue.x, y: oldValue.y,
                                width: oldValue.x + image.size.width,
                                height: oldValue.y + image.size.height)
        setNeedsDisplay(currentRect.union(prevRect))
    }
}

```

Расчет движения шарика

Метод `drawRect()` вряд ли можно сделать еще проще. В нем просто рисуется изображение в положении, хранящемся в свойстве `currentPoint`. Но совсем другое дело — модифицирующий метод `setCurrentPoint()`. Когда устанавливается новое положение (из рассматриваемого далее `update()`), необходимо проверить, попадает ли шарик на край экрана, и, если попадает, остановить его движение по оси `x` или `y`. Именно для этого мы и реализуем наблюдатель свойств, из которого можно получить доступ к новому значению свойства и видоизменить его, не вызывая наблюдатель свойств снова, что привело бы к бесконечной рекурсии.

Прежде всего мы получаем координаты `x` и `y` положения шарика и проверяем границы. Если любая из координат `x` или `y` положения шарика оказывается меньше нуля или больше ширины или высоты экрана соответственно, учитывая ширину и высоту изображения, то ускорение в данном направлении прекращается, и мы изменяем соответствующую координату положения шарика таким образом, чтобы он оказался на краю экрана, как показано ниже.

```

var newX = currentPoint.x
var newY = currentPoint.y
if newX < 0 {
    newX = 0
    ballXVelocity = 0
} else if newX > bounds.size.width - image.size.width {
    newX = bounds.size.width - image.size.width
    ballXVelocity = 0
}
if newY < 0 {
    newY = 0
    ballYVelocity = 0
} else if newY > bounds.size.height - image.size.height {
    newY = bounds.size.height - image.size.height
    ballYVelocity = 0
}

```

ПОДСКАЗКА. Если нужно, чтобы шарик отскакивал от стенок экрана более естественно, а не просто останавливался, этого совсем нетрудно добиться. Достаточно заменить в методе setCurrentPoint() строку кода self.ballXVelocity = 0; строкой self.ballXVelocity = - (self.ballXVelocity / 2.0);, а строку кода self.ballYVelocity = 0; — строкой self.ballYVelocity = - (self.ballYVelocity / 2.0);. В этих изменениях скорость движения шарика не гасится полностью, а лишь уменьшается наполовину и задается в противоположном направлении. После этого шарик будет двигаться в два раза медленнее в противоположном направлении.

В приведенном выше фрагменте кода координаты положения шарика сохраняются в локальных переменных newX, newY. Как только положение шарика будет видоизменено по мере надобности, значения этих переменных будут использованы для создания и сохранения обновленного положения шарика в свойстве currentPoint:

```
currentPoint = CGPointMake(x: newX, y: newY)
```

После этого рассчитываем два прямоугольных объекта типа CGRect, исходя из размеров изображения. Один из прямоугольников охватывает участок, на котором будет нарисовано новое изображение, а другой — участок, на котором прежнее изображение было нарисовано в последний раз. Второй из этих прямоугольников мы рассчитываем, используя предыдущее положение шарика, которое автоматически сохраняется средствами Swift в константной переменной oldValue. Используя оба эти прямоугольника, мы можем обеспечить стирание прежнего изображения шарика и одновременное рисование нового его изображения, как показано ниже.

```

let currentRect = CGRect(x: newX, y: newY,
                         width: newX + image.size.width,
                         height: newY + image.size.height)
let prevRect = CGRect(x: oldValue.x, y: oldValue.y,
                      width: oldValue.x + image.size.width,
                      height: oldValue.y + image.size.height)

```

Наконец мы создаем новый прямоугольник, объединяющий оба только что рассчитанных прямоугольника, а затем передаем его методу `setNeedsDisplayInRect()`, указывая тем самым часть представления, которую нужно перерисовать:

```
setNeedsDisplay(currentRect.union(prevRect))
```

Последним существенным методом в рассматриваемом здесь классе является метод `update()`, применяемый для точного определения нового местоположения шарика. Этот метод вызывается в методе акселерометра из класса его контроллера после того, как текущему представлению предоставлен новый объект ускорения. Сначала мы рассчитываем, сколько времени прошло с момента последнего вызова данного метода. Текущее время предоставляется экземпляром класса `NSDate`, возвращаемым методом `NSDate()`. Запрашивая у него промежуток времени, прошедшего с момента последнего обновления (`lastUpdateTime`), мы получаем числовое значение, обозначающее количество секунд, прошедших до настоящего времени с момента последнего вызова данного метода:

```
let now = Date()
let secondsSinceLastDraw = now.timeIntervalSinceDate(lastUpdateTime)
```

Далее рассчитываем новую скорость в обоих направлениях, добавляя текущее ускорение к скорости движения шарика в настоящий момент. Величина ускорения умножается на значение переменной `secondsSinceLastDraw`, чтобы согласовать ускорение во времени, как показано ниже. Наклон мобильного телефона под одним и тем же углом будет всегда вызывать одно и то же ускорение.

```
ballXVelocity = ballXVelocity + (acceleration.x * secondsSinceLastDraw);
ballYVelocity = ballYVelocity - (acceleration.y * secondsSinceLastDraw);
```

После этого мы определяем фактическое изменение положения шарика в пикселях, поскольку данный метод вызывался в последний раз в зависимости от скорости движения. Произведение скорости на прошедшее время умножается далее на 500, чтобы сделать движение шарика более естественным, как показано ниже. Если бы мы не выполнили такое умножение на заданную величину, ускорение происходило бы очень медленно, как будто шарик увязает в патоке.

```
let xDelta = secondsSinceLastDraw * ballXVelocity * 500
let yDelta = secondsSinceLastDraw * ballYVelocity * 500
```

Теперь, когда нам известно изменение положения шарика в пикселях, мы задаем новую точку, складывая текущее местоположение шарика с рассчитанным ускорением и присваивая его переменной `currentPoint` следующим образом:

```
currentPoint = CGPointMake(x: currentPoint.x + CGFloat(xDelta),
                           y: currentPoint.y + CGFloat(yDelta))
```

На этом расчеты завершаются, и нам остается лишь обновить переменную `lastDrawTime` текущим временем, как показано ниже.

```
lastUpdateTime = now
```

Теперь соберите и запустите данное приложение на выполнение. Если все прошло удачно, приложение запустится, и вы получите возможность управлять передвижением шарика по экрану, наклоняя свой мобильный телефон. Достигнув края экрана, шарик должен остановиться. Наклоните свой мобильный телефон в противоположную сторону, чтобы он начал катиться в другом направлении.

Резюме

В этой главе у вас была возможность заняться физическими экспериментами, используя замечательные свойства акселерометра и гироскопа, встроенных в мобильные устройства под управлением системы iOS. Возможности приложений, использующих свойства акселерометра и гироскопа, практически безграничны. В следующей главе мы рассмотрим еще один аспект аппаратного обеспечения iOS: встроенную фотокамеру.

ГЛАВА 21



Камера и фотоархив

Теперь уже никого не удивишь тем, что в мобильные устройства iPhone, iPad и iPod touch встроена камера, а искусственное стандартное приложение Photos помогает привести в организованный порядок все замечательные снимки и видеосюжеты, снятые встроенной камерой (рис. 21.1). Но вам, возможно, еще неизвестно, что в своих прикладных программах вы сможете использовать встроенную камеру для получения снимков и предоставлять пользователям своих приложений возможность выбирать снимки среди разной мультимедийной информации, хранящейся на мобильном устройстве. В этой главе мы рассмотрим обе эти возможности.

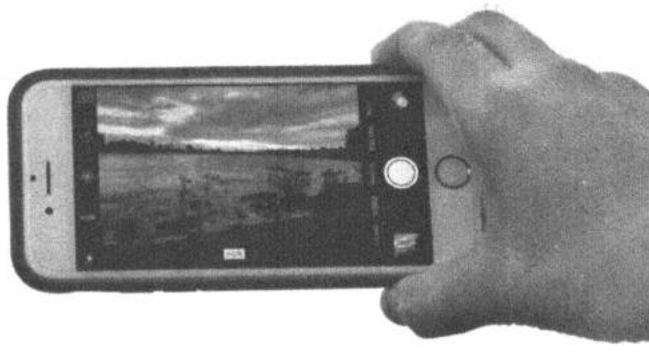


Рис. 21.1. В этой главе мы изучим механизм доступа к камере iPhone и библиотеке фотографий из собственного приложения

Применение селектора изображений и класса UIImagePickerController

Благодаря известной изолированности приложений iOS они не могут получить доступ к фотографиям или другой информации, хранящейся за пределами их изолированных программных сред, называемых “песочницами”. Правда, камера и мультимедийный архив доступны для приложения посредством **селектора изображений** (*image picker*).

Применение контроллера селектора изображений

Как подразумевает название этого средства, селектор изображений представляет собой механизм, позволяющий выбирать изображение из указанного источника. Когда класс селектора изображений впервые появился в системе iOS, он применялся только для изображений. В настоящее время с его помощью можно фиксировать и видеосюжеты. Обычно в селекторе изображений в качестве источника используется перечень изображений или видеосюжетов (рис. 21.2, слева), однако ему можно также дать команду использовать в качестве источника камеру (см. рис. 21.2, справа).



Рис. 21.2. Селектор изображений в действии. Пользователю предоставляется на выбор перечень изображений (слева). Как только изображение выбрано, пользователь получает возможность перемещать и изменять его размеры (справа)

Интерфейс селектора изображений реализуется посредством класса контроллера `UIImagePickerController`. С этой целью вы создаете экземпляр данного класса, указываете делегат (как ни в чем ни бывало), задаете источник изображений и возможность для пользователя выбирать изображение или видеосюжет, а затем предоставляете выбранное пользователю. Селектор изображений берет на себя управление мобильным устройством, чтобы предоставить пользователю возможность выбрать изображение или видеосюжет из имеющегося мультимедийного архива либо снять новую фотографию или видеосюжет встроенной камерой. Как только пользователь сделает свой выбор, вы вольны предоставить ему еще одну возможность: вносить элементарные правки, в том числе изменять масштаб, обрезать изображение или часть видеосюжета. Все эти виды поведения реализуются в классе `UIImagePickerController`, поэтому на вашу долю остается не так уж и много труда.

Если пользователь не нажмет кнопку отмены, выбранные либо полученные им изображения или видеосюжеты будут доставлены указанному вами делегату. Независимо от того, выберет ли пользователь мультимедийный файл или же откажется от выбора, указанный вами делегат берет на себя ответственность за освобождение класса `UIImagePickerController`, чтобы дать пользователю возможность вернуться к вашему приложению.

Объект класса `UIImagePickerController` создается очень просто. Его экземпляр назначается и инициализируется таким же образом, как и в большинстве других классов. Но у данного класса имеется одна каверзная особенность. Далеко не всякое устройство под управлением iOS имеет встроенную камеру. Примером тому могут служить первые модели iPod touch, а также первое поколение устройств iPad, но такие модели мобильных устройств со временем будут сняты компанией Apple с производства. Поэтому, прежде чем создавать экземпляр класса `UIImagePickerController`, вы должны проверить, поддерживается ли источник изображений на том мобильном устройстве, на котором выполняется ваше приложение. Например, перед тем как предоставить пользователю возможность сделать снимок встроенной камерой, вы должны убедиться, что ваше приложение выполняется на мобильном устройстве, имеющем подобную камеру. И такую проверку вы можете выполнить с помощью следующего метода из класса `UIImagePickerController`:

```
if UIImagePickerController.isSourceTypeAvailable(.Camera) {
```

В данном примере методу `isSourceTypeAvailable()` в качестве аргумента передается значение константы `UIImagePickerControllerSourceType.Camera`, указывающей на то, что пользователю нужно предоставить возможность снимать фотографии или видеосюжеты встроенной камерой. Метод `isSourceTypeAvailable()` возвратит логическое значение `true`, если указанный источник доступен в настоящий момент. Помимо константы `UIImagePickerControllerSourceType.camera`, данному методу можно передать две другие константы.

- ◉ `UIImagePickerControllerSourceType.PhotoLibrary` Указывает на то, что пользователь должен выбрать изображение или видеосюжет из имеющегося мультимедийного архива. Выбранное изображение будет возвращено указанному делегату.
- ◉ `UIImagePickerControllerSourceType.SavedPhotosAlbum` Указывает на то, что пользователь будет выбирать изображение из архива имеющихся фотографий, но этот выбор ограничен перечнем последних снимков, сделанных встроенной камерой. Это значение можно выбирать и для приложений, работающих на мобильных устройствах без камеры, хотя пользы от него в этом случае не очень много.

Убедившись в том, что мобильное устройство, на котором выполняется ваше приложение, поддерживает нужный источник изображений, запустите селектор изображений. Это делается относительно просто, как показано в листинге 21.1.

Листинг 21.1. Запуск селектора изображений из контроллера представления

```
let picker = UIImagePickerController()
picker.delegate = self
picker.sourceType = UIImagePickerControllerSourceType.camera
picker.cameraDevice = UIImagePickerControllerCameraDevice.front
self.present(picker, animated:true, completion: nil)
```

После создания и настройки экземпляра класса `UIImagePickerController` для предоставления пользователю селектора изображений вызывается метод `self.present(_:_animated:_completion:)`, наследуемый из класса `UIView`.

ПОДСКАЗКА. На мобильном устройстве с несколькими камерами имеется возможность выбрать одну из них, установив в свойстве `cameraDevice` константу `UIImagePickerControllerCameraDevice.front` или `UIImagePickerControllerCameraDevice.rear`. Чтобы выяснить, какая из камер доступна (фронтальная или задняя), достаточно передать эти же константы методу `isCameraDeviceAvailable()`.

Реализация делегата для контроллера селектора изображений

Для того чтобы выяснить, завершил ли пользователь операции с селектором изображений, следует реализовать протокол `UIImagePickerControllerDelegate`, в котором определяются два метода: `imagePickerController(_:_didFinishPickingMediaWithInfo:)` и `imagePickerControllerDidCancel()`.

Первый метод делегата, `imagePickerController(_:_didFinishPickingMediaWithInfo:)`, вызывается в тот момент, когда пользователь успешно снимает фотографию или выбирает ее из мультимедийного архива. В качестве первого аргумента данному методу передается указатель на созданный ранее объект типа `UIImagePickerController`, а в качестве второго аргумента — словарь, в котором будут храниться выбранная фотография или URL выбранного видеосюжета, а также дополнительные сведения о правке, если последняя активизирована

и пользователь действительно кое-что правил. Словарь будет также содержать исходное, неотредактированное изображение, хранящееся с ключом `UIImagePickerControllerOriginalImage`. В листинге 21.2 приведен пример метода делегата, извлекающего исходное изображение.

Листинг 21.2. Метод делегата для извлечения изображения

```
func imagePickerController(picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [String : AnyObject]) {
    let selectedImage: UIImage? =
        info[UIImagePickerControllerEditedImage] as? UIImage
    let originalImage: UIImage? =
        info[UIImagePickerControllerOriginalImage] as? UIImage

    // Делаем что-нибудь с изображениями selectedImage и originalImage

    picker.dismiss(animated: true, completion:nil)
}
```

Словарь `editingInfo` также сообщает, какая именно часть всего изображения была выбрана во время правки. Для этой цели служит объект типа `NSValue`, хранящийся по ключу `UIImagePickerControllerCropRect`. Этот строковый объект можно преобразовать в объект типа `CGRect` так, как показано в листинге 21.3.

Листинг 21.3. Преобразование `NSValue` в `CGRect`

```
let cropValue: NSValue? = info[UIImagePickerControllerCropRect] as? NSValue
let cropRect: CGRect? = cropValue?.cgRectValue()
```

После этого преобразования объект `cropRect` будет определять ту часть исходного изображения, которая была выбрана в процессе правки. Если же эта информация не нужна, ее можно пренебречь.

ПРЕДУПРЕЖДЕНИЕ. Если изображение, возвращаемое делегату, поступает из камеры, оно не будет сохранено в фотоархиве. Поэтому ответственность за сохранение изображения в фотоархиве, если в этом возникнет необходимость, возлагается на само приложение.

Второй метод делегата, `imagePickerControllerDidCancel()`, вызывается в том случае, если пользователь решает отменить процесс, не снимая и не выбирая фотографии или видеосюжеты. Когда этот метод делегата вызывается из селектора изображений, приложение просто уведомляется о том, что пользователь завершил операции с селектором изображений, так ничего и не выбрав.

Оба упомянутых выше метода из протокола `UIImagePickerControllerDelegate` обозначены как необязательные, хотя они таковыми на самом деле не являются, и вот почему: таким модальным представлениям, как селектор изображений, нужно предписывать освобождаться. Следовательно, даже если вам не нужно производить никаких действий, характерных для приложения, когда

пользователь отменит селектор изображений, обязанность по освобождению этого селектора по-прежнему возлагается на вас. Поэтому для правильного функционирования вашего приложения метод `imagePickerControllerDidCancel()` должен иметь как минимум следующий вид:

```
func imagePickerControllerDidCancel(picker: UIImagePickerController!) {  
    picker.dismiss(true, completion:nil)  
}
```

В этой главе мы разработаем приложение, с помощью которого пользователь сможет сделать снимок или снять видеосюжет встроенной камерой либо сделать выбор из фотоархива и вывести выбранное на экран (рис. 21.3). Если же пользователь выполнит данное приложение на мобильном устройстве без встроенной камеры, то кнопка `New Photo or Video` будет скрыта, но останется возможность выбора из фотоархива.

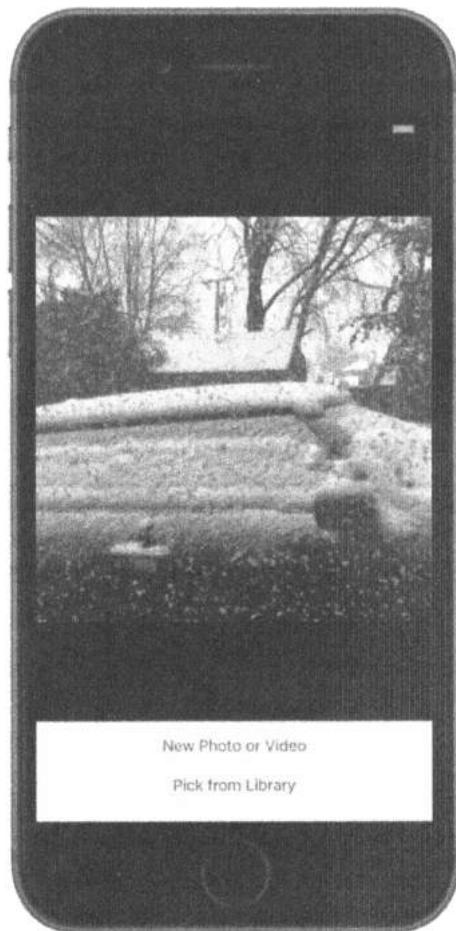


Рис. 21.3. Приложение Camera в действии

Разработка пользовательского интерфейса приложения

Создайте новый проект в среде Xcode, используя шаблон Single View Application. Присвойте новому проекту имя Camera. Прежде всего нам потребуется пара выходов для контроллера представления. Один из них должен указывать на представление, чтобы обновлять его изображением, возвращаемым из селектора изображений, а другой — на кнопку New Photo or Video, чтобы скрыть ее в том случае, если в мобильном устройстве отсутствует камера. Кроме того, нам потребуются два метода действия: один — для функционирования кнопки New Photo or Video, другой — для предоставления пользователю возможности выбрать изображение, имеющееся в фотоархиве.

```
class ViewController: UIViewController,
    UIImagePickerControllerDelegate, UINavigationControllerDelegate {
    @IBOutlet var imageView: UIImageView!
    @IBOutlet var takePictureButton: UIButton!
```

Вероятно, вы заметили, что приведенный выше класс согласован с двумя разными протоколами: UIImagePickerControllerDelegate и UINavigationControllerDelegate. Мы должны согласовать его с обоими этими протоколами, поскольку класс UIImagePickerController является подклассом, производным от класса UINavigationController. Оба метода из протокола UINavigationControllerDelegate являются необязательными, и поэтому ни тот, ни другой нам вообще не нужен, чтобы воспользоваться селектором изображений. Но в то же время мы должны соблюдать протокол, иначе компилятор выдаст предупреждение о несоответствии протоколу.

Вы могли обратить внимание и на то, что мы оперируем экземпляром класса UIImageView для отображения выбранного изображения, тогда как для отображения выбранного видеосюжета ничего подобного нет. Дело в том, что в состав каркаса UIKit не входит общедоступный класс, аналогичный UIImageView и специально предназначенный для отображения видеоконтента. Вместо этого мы воспользуемся объектом типа MPMoviePlayerController, выбрав его свойство view и введя его в иерархию представлений. Это не совсем обычный способ применения контроллера представления, но он фактически одобрен компанией Apple для отображения видеоконтента в иерархии представлений.

Нам предстоит также ввести два метода действия, с которыми требуется соединить упомянутые выше кнопки. Сначала мы создадим лишь пустые их реализации, как показано ниже, чтобы они были доступны в программе Interface Builder, а в дальнейшем наполним их конкретным кодом.

```
@IBAction func shootPictureOrVideo(sender: UIButton) {
}

@IBAction func selectExistingPictureOrVideo(sender: UIButton) {
}
```

Компоновка, которую нам предстоит построить для данного приложения, очень проста: она состоит из графического представления и двух кнопок. В окончательном виде она приведена на рис. 21.4, которым можно пользоваться в качестве образца при разработке данного приложения.

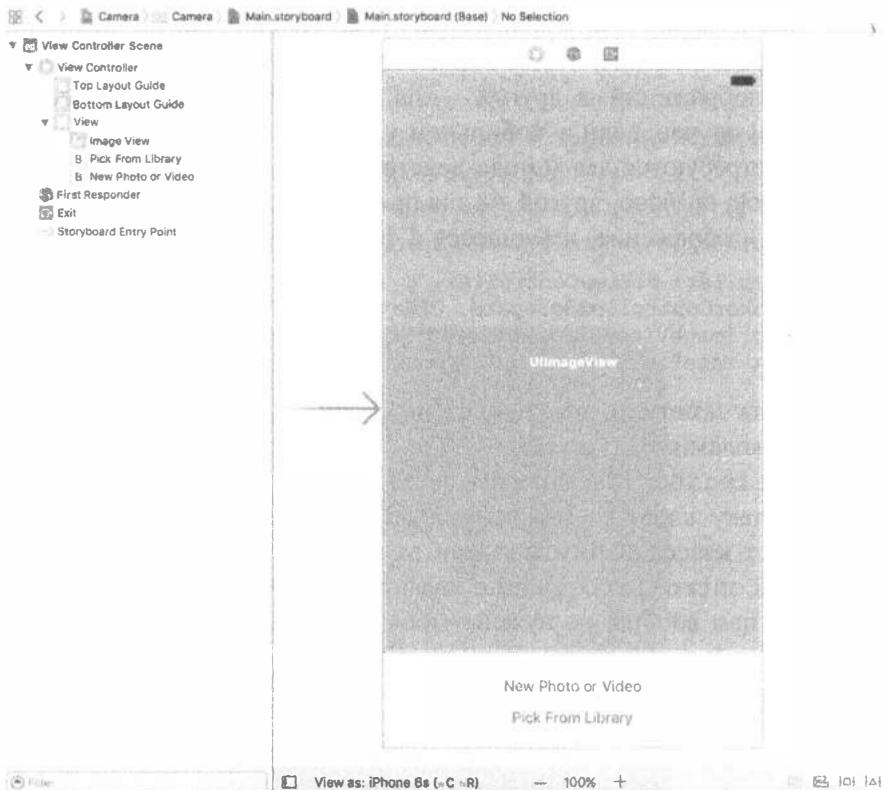


Рис. 21.4. Компоновка раскадровки приложения Camera

Перетащите две кнопки (объекты Button) из библиотеки объектов на представление в раскадровке. Расположите их одну под другой, выровняв нижнюю кнопку по нижней голубой направляющей линии. Дважды щелкните на верхней кнопке и присвойте ей надпись New Photo or Video. Снова дважды щелкните, но на это раз на нижней кнопке и присвойте ей надпись Pick from Library. Затем перетащите графическое представление (объект Image View) из библиотеки объектов, расположив его выше кнопок. Разверните это графическое представление, чтобы оно заняло все пространство выше кнопок, как показано на рис. 21.3. Перейдите к инспектору атрибутов, измените цвет фона данного представления на черный и выберите вариант Aspect Fit из раскрывающегося списка Mode. В итоге размеры изображений будут изменены таким образом, чтобы они вписались в границы данного представления, но в то же время сохранили свои пропорции.

Теперь нажмите клавишу <Control>, перетащите указатель от пиктограммы View Controller к графическому представлению и выберите выход imageView. Снова нажмите клавишу <Control>, перетащите указатель от пиктограммы View Controller к кнопке New Photo or Video и выберите выход takePictureButton. Щелкните на кнопке New Photo or Video и откройте инспектор связей. Перетащите указатель от события Touch Up Inside к пиктограмме View Controller и выберите действие shootPictureOrVideo:. Щелкните на кнопке Pick from Library, перетащите указатель от события Touch Up Inside в инспекторе связей к пиктограмме View Controller и выберите действие selectExistingPictureOrVideo:.

Теперь остается, как обычно, произвести автоматическое наложение ограничений на компоновку. С этой целью разверните сначала иерархию контроллера представления в окне Document Outline и добавьте ограничения на компоновку следующим образом.

1. Перейдите в окно Document Outline, нажмите клавишу <Control> и перетащите указатель от кнопки Pick from Library к родительскому представлению, а затем отпустите кнопку мыши. Как только появится контекстное меню, нажмите клавишу <Shift> и выберите из него команды Center Horizontally in Container и Vertical Spacing to Bottom Layout Guide.
2. Нажмите клавишу <Control>, перетащите указатель от кнопки New Photo or Video к кнопке Pick from Library, отпустите кнопку мыши и выберите пункт Vertical Spacing из контекстного меню.
3. Нажмите клавишу <Control>, перетащите указатель от кнопки New Photo or Video к его родительскому представлению, отпустите кнопку мыши, нажмите клавишу <Shift> и выберите команду Center Horizontally in Container.
4. Нажмите клавишу <Control>, перетащите указатель от кнопки New Photo or Video к графическому представлению. Отпустите кнопку мыши и выберите пункт Vertical Spacing из контекстного меню.
5. Нажмите клавишу <Control>, перетащите указатель от графического представления к его родительскому представлению. Отпустите кнопку мыши, нажмите клавишу <Shift> и выберите команды Leading Space to Container Margin, Trailing Space to Container Margin и Vertical Spacing to Top Layout Guide из контекстного меню.

Итак, все ограничения на компоновку графического пользовательского интерфейса данного приложения установлены, поэтому сохраните внесенные вами изменения.

Функции защиты конфиденциальности

Поскольку мы планируем использовать камеру, которая частично защищена системой iOS, мы должны запросить у пользователя разрешение на это. Аналогично, поскольку мы планируем снимать видео, нам потребуется доступ к микрофону, если, конечно, вы не собираетесь снимать немое кино. Кроме того, нам

нужен доступ к библиотеке фотографий. К счастью, система iOS решит большинство из этих проблем автоматически, если мы правильно сформулируем свои запросы. Для этого необходимо дописать в файл Info.plist несколько строк.

Поскольку мы уже несколько раз обращались к этому списку свойств, просто добавьте внизу три дополнительные записи, как показано на рис. 21.5. Текст в крайнем справа столбце будет выведен на экран как дополнительная информация о запросе на использование конкретного ресурса. Это позволит четко и ясно сформулировать запрос.

Ключ	Значение	Описание
► Supported interface orientations	Portrait	(3 items)
► Supported interface orientations (i...)	Portrait	(4 items)
Privacy - Camera Usage Description	String	My App Would Like to access the camera
Privacy - Microphone Usage Desc...	String	My App would also like to use the mic
Privacy - Photo Library Usage Des...	String	My App would like to look at your pictures

Рис. 21.5. Добавление функции защиты конфиденциальности для камеры и микрофона в файл Info.plist

Когда нам потребуется доступ к одному из трех ресурсов, пользователь получит сообщение, показанное на рис. 21.6.

Реализация контроллера представления камеры

Выберите исходный файл ViewController.swift, где вносится больше всего изменений. Мы собираемся предоставить пользователям данного приложения дополнительную возможность снимать видеосюжеты, и поэтому нам потребуется свойство для экземпляра класса AVPlayerViewController. Еще два свойства нам потребуются для отслеживания выбранных в последний раз фотографий и видеосюжета, а также символьная строка, чтобы выяснить, что именно было выбрано в последний раз: фотография или видеосюжет. Кроме того, нам придется импортировать дополнительные заголовочные файлы, чтобы привести все это в действие. С этой целью введите в данном файле строки кода, выделенные ниже полужирным шрифтом.

```
import UIKit
import AVKit
import AVFoundation
import MobileCoreServices

class ViewController: UIViewController, UIImagePickerControllerDelegate,
    UINavigationControllerDelegate {
    @IBOutlet var imageView: UIImageView!
    @IBOutlet var takePictureButton: UIButton!
    var avPlayerViewController: AVPlayerViewController!
    var image: UIImage?
    var movieURL: URL?
    var lastChosenMediaType: String?
```

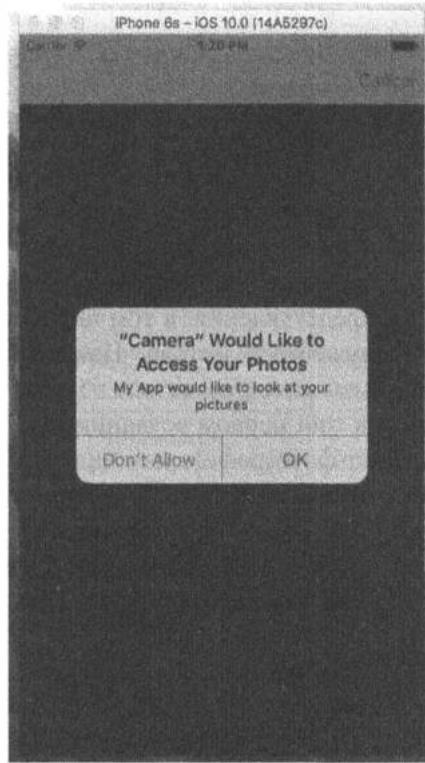


Рис. 21.6. Система iOS автоматически обрабатывает запросы пользователя, но вы должны быть точными, если вашему приложению требуется конкретный ресурс

Теперь расширим метод viewDidLoad() таким образом, чтобы скрывать кнопку New Photo or Video, если на мобильном устройстве, на котором выполняется данное приложение, отсутствует камера. Мы также реализуем метод viewWillAppear(), чтобы вызывать из него метод updateDisplay(), который будет реализован в дальнейшем. Прежде всего внесите в метод viewDidLoad() изменения, приведенные в листинге 21.4.

Листинг 21.4. Модификации методов viewDidLoad и viewWillAppear

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.

    if !UIImagePickerController.isSourceTypeAvailable(
        UIImagePickerControllerSourceType.camera) {
        takePictureButton.isHidden = true
    }
}
```

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

Важно понимать различия между методами `viewDidLoad()` и `viewDidAppear()`. Первый из них вызывается лишь в тот момент, когда представление только загружено в оперативную память, тогда как второй вызывается всякий раз, когда представление отображает, что происходит как во время запуска приложения, так и в ходе возврата к текущему контроллеру после отображения очередного полноэкранного представления, в том числе селектора изображений.

Рассмотрим далее три служебных метода. Начнем с метода `updateDisplay()`. Напомним, что он вызывается из метода `viewDidAppear()`, который, в свою очередь, вызывается как при первом создании представления, так и после того, как пользователь выберет фотографию или видеосюжет и оставит селектор изображений. В силу этого двойного назначения данного метода в нем приходится выполнять несколько проверок, чтобы выяснить ситуацию и соответственно настроить графический интерфейс пользователя. Итак, введите код, приведенный в листинге 21.5, в конце исходного файла `ViewController.swift`.

Листинг 21.5. Метод `updateDisplay`

```
func updateDisplay() {
    if let mediaType = lastChosenMediaType {
        if mediaType == kUTTypeImage as NSString {
            imageView.image = image!
            imageView.isHidden = false
            if avPlayerViewController != nil {
                avPlayerViewController!.view.isHidden = true
            }
        } else if mediaType == kUTTypeMovie as NSString {
            if avPlayerViewController == nil {
                avPlayerViewController = AVPlayerViewController()
                let avPlayerView = avPlayerViewController!.view
                avPlayerView?.frame = imageView.frame
                avPlayerView?.clipsToBounds = true
                view.addSubview(avPlayerView!)
                setAVPlayerViewLayoutConstraints()
            }
            if let url = movieURL {
                imageView.isHidden = true
                avPlayerViewController.player = AVPlayer(url: url)
                avPlayerViewController!.view.isHidden = false
                avPlayerViewController!.player!.play()
            }
        }
    }
}
```

Этот метод отображает подходящее представление в зависимости от типа мультимедийного содержимого, выбранного пользователем: графическое

представление для фотографии или проигрыватель видеозаписей для видеосюжета или фильма. Графическое представление присутствует всегда, тогда как проигрыватель видеозаписей создается и вводится в пользовательский интерфейс только в том случае, если пользователь выберет в первый раз видеосюжет.

Вводя проигрыватель видеозаписей, мы должны добиться того, чтобы он занимал столько же места на экране, сколько и графическое представление, а для этого нам придется наложить ограничения на компоновку, оставив место на тот случай, если мобильное устройство поворачивается. Подобные ограничения накладываются на компоновку в методе, приведенном в листинге 21.6.

Листинг 21.6. Наложение ограничений на компоновку AV-плеера в файле ViewController.swift

```
func setAVPlayerViewLayoutConstraints() {
    let avPlayerView = avPlayerViewController!.view
    avPlayerView?.translatesAutoresizingMaskIntoConstraints = false
    let views = ["avPlayerView": avPlayerView!,
                 "takePictureButton": takePictureButton!]
    view.addConstraints(NSLayoutConstraint.constraints(
        withVisualFormat: "H:[avPlayerView]|",
        options: .alignAllLeft,
        metrics:nil, views:views))
    view.addConstraints(NSLayoutConstraint.constraints(
        withVisualFormat: "V:[avPlayerView]-0-[takePictureButton]",
        options: .alignAllLeft, metrics:nil, views:views))
}
```

Ограничения, накладываемые по горизонтали, привязывают проигрыватель видеозаписей к левой и правой боковым сторонам главного представления, а ограничения, накладываемые по вертикали, — к верхним краям главного представления и кнопки *New Photo or Video*.

Последний служебный метод, `pickMediaSource()`, вызывается из обоих методов действия. Сам по себе он довольно прост. В нем лишь создается и настраивается селектор изображений, для чего используется передаваемое ему в качестве аргумента свойство `sourceType`, по которому определяется, следуя ли активировать камеру или фотоархив. Введите в конце исходного файла `ViewController.swift` код, представленный в листинге 21.7.

Листинг 21.7. Метод `pickMediaFromSource` в файле ViewController.swift

```
func pickMediaFromSource(_ sourceType: UIImagePickerControllerSourceType) {
    let mediaTypes =
        UIImagePickerController.availableMediaTypes(for: sourceType)!
    if UIImagePickerController.isSourceTypeAvailable(sourceType)
        && mediaTypes.count > 0 {
        let picker = UIImagePickerController()
        picker.mediaTypes = mediaTypes
        picker.delegate = self
        picker.allowsEditing = true
        picker.sourceType = sourceType
    }
}
```

```

        present(picker, animated: true, completion: nil)
    } else {
        let alertController = UIAlertController(title: "Error accessing media",
            message: "Unsupported media source.",
            preferredStyle: UIAlertControllerStyle.alert)
        let okAction = UIAlertAction(title: "OK",
            style: UIAlertActionStyle.cancel, handler: nil)
        alertController.addAction(okAction)
        present(alertController, animated: true, completion: nil)
    }
}
}

```

Далее реализуйте следующие методы действия, связанные с кнопками (листинг 21.8).

Листинг 21.8. Методы делегата представления селектора

```

func imagePickerController(_ picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [String : AnyObject]) {
    lastChosenMediaType = info[UIImagePickerControllerMediaType] as? String
    if let mediaType = lastChosenMediaType {
        if mediaType == kUTTypeImage as NSString {
            image = info[UIImagePickerControllerEditedImage] as? UIImage
        } else if mediaType == kUTTypeMovie as NSString {
            movieURL = info[UIImagePickerControllerMediaURL] as? URL
        }
    }
    picker.dismiss(animated: true, completion: nil)
}

func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
    picker.dismiss(animated: true, completion: nil)
}

```

Первый метод делегата проверяет, что именно выбрано (фотография или видеосюжет), уточняет выбор, а затем освобождает модальный селектор изображения. Если фотография оказывается больше доступного места на экране, то ее размеры соответственно изменяются при отображении в графическом представлении, поскольку для содержимого последнего при его создании был установлен режим *Aspect Fit*. Второй метод просто освобождает селектор изображений.

Вот, собственно, и все, что нужно сделать в данном приложении. Скомпилируйте и запустите его на выполнение. Если вы выполняете данное приложение в симуляторе, у вас не будет возможности сделать новый снимок. Если у вас имеется возможность выполнить данное приложение на настоящем мобильном устройстве, опробуйте его в реальных условиях эксплуатации. В этом случае вы сможете сделать новый снимок, а также увеличить или уменьшить его масштаб, используя жесты, имитирующие щипки. В первый раз приложение запросит доступ к фотографиям пользователя в системе iOS и попросит у него разрешения. Это новое средство защиты конфиденциальности впервые появилось в системе iOS 6, чтобы приложения не крали незаметно фотографии без ведома пользователя.

Если после съемки или выбора фотографии вы измените масштаб или панорамируете ее перед тем, как нажать кнопку *Use Photo*, то изображение, возвращаемое данному приложению в методе делегата, окажется обрезанным.

Резюме

Хотите — верьте, хотите — нет, но всего описанного выше достаточно для того, чтобы дать пользователям возможность делать снимки встроенной в мобильное устройство камерой, а затем использовать их в вашем приложении. По желанию можете даже предоставить пользователям возможность немного поправить изображение.

В следующей главе будут рассмотрены вопросы расширения потенциального круга пользователей приложений для системы iOS. Для этого они должны легко переводиться в ходе локализации на другие языки.

ГЛАВА 22



Локализация приложений

На момент написания этой книги мобильные устройства под управлением системы iOS имелись в продаже более чем в половине стран мира, и это число со временем возрастет (рис. 22.1). Теперь мобильный телефон iPhone можно приобрести и использовать на всех континентах, кроме Антарктиды. Мобильные устройства iPad и iPod touch не так распространены, как iPhone, хотя и продаются во многих странах мира. Если вы собираетесь выпускать и распространять свои приложения через интернет-магазин App Store, то ваш потенциальный рынок окажется намного больше рынка вашей страны, в которой живут люди, говорящие на том же языке, что и вы. Правда, в системе iOS имеется надежная архитектура локализации, позволяющая легко перевести приложение самостоятельно (или предоставить это другим) не только на многие языки мира, но даже на несколько диалектов одного и того же языка. Так, если для англоязычных пользователей в Великобритании нужна не такая терминология, как в США, то внедрить ее в свое приложение не составит большого труда.



Рис. 22.1. Устройства iOS доступны в половине стран мира, поэтому разработка приложений на многих языках предоставит вам больше возможностей для успеха

Локализация не вызывает особых трудностей, если код приложения написан корректно. Однако настроить приложение на поддержку локализации намного труднее, чем написать его корректно с самого начала. В этой главе сначала будет показано, как написать код приложения, чтобы его можно было легко локализовать, а затем особенности локализации приложений будут продемонстрированы на конкретном примере.

Архитектура локализации

Когда выполняется нелокализованное приложение, весь его текст представлен на родном языке его разработчиков, называемом также **базовым языком разработки**. Когда же разработчики решают локализовать свое приложение, они создают подкаталог в пакете приложения для каждого поддерживающего языка. В подкаталоге каждого языка содержится подмножество ресурсов приложения, переведенных на этот язык. Каждый такой подкаталог называется **проектом локализации или папкой локализации**. Имена папок локализации всегда оканчиваются расширением `.lproj`.

В стандартном для системы iOS приложении `Settings` пользователю предоставляется возможность выбрать язык и региональный формат. Так, если родным языком пользователя является английский, к числу доступных для него регионов могут быть отнесены Соединенные Штаты, Австралия или Гонконг, т.е. все регионы, где говорят по-английски.

Когда локализованному приложению требуется загрузить ресурс, например изображение, список свойств или `pib`-файл, оно проверяет выбранный пользователем язык и регион и осуществляет поиск папки локализации, соответствующей данной настройке. Если приложение найдет такую папку, оно загрузит локализованную версию ресурса вместо базовой версии.

Так, если пользователь выбрал французский в качестве языка iOS и Швейцарию в качестве региона, приложение будет сначала искать папку локализации `fr-CH.lproj`. Две первые буквы в имени этой папки соответствуют коду страны по стандарту ISO, обозначающему французский язык. Еще две буквы после дефиса соответствуют двузначному коду по стандарту ISO, обозначающему Швейцарию.

Если приложение не в состоянии найти соответствие по двузначному коду, оно пытается найти соответствие по трехзначному коду, обозначающему язык по стандарту ISO. Так, если в приведенном выше примере приложению не удастся найти папку `fr-CH.lproj`, оно попытается найти папку локализации `fre-CH` или `fra-CH`.

Все языки обозначаются по меньшей мере одним трехзначным кодом, а некоторые — даже двумя трехзначными кодами: одним — для английского написания языка, другим — для местного написания. Если же язык обозначается как двух-, так и трехзначным кодом, то предпочтение отдается первому.

ЗАМЕЧАНИЕ. Перечень применяемых в настоящее время кодов стран по стандарту ISO можно найти на веб-сайте ISO — Международной организации по стандартизации по адресу http://www.iso.org/iso/country_codes.htm. Двух- и трехзначные коды определяются в стандарте ISO 3166 как составная его часть.

Если приложению не удастся найти папку, точно соответствующую искомому языку, оно попытается найти в пакете приложения папку локализации, которая соответствует только коду искомого языка, но без кода региона. Поэтому если продолжить приведенный выше пример с франкоязычным пользователем из Швейцарии, то приложение будет далее искать проект локализации под названием `fr.lproj`. Если ему не удастся найти проект под таким названием, оно попытается сначала найти проект `fre.lproj`, а затем `fra.lproj`. Если же ему не удастся вообще ничего найти, оно выбирает проект `French.lproj`. Последняя конструкция существует лишь для поддержки устаревших приложений под управлением Mac OS X, и, вообще говоря, ее применения следует избегать.

Если же приложению не удастся найти проект локализации, соответствующий заданной комбинации языка и региона или только выбранному языку, оно воспользуется ресурсами из базового языка разработки. Если оно все же найдет подходящий проект локализации, то впредь будет искать любые необходимые ему ресурсы только в этом месте. Например, если объект типа `UIImage` загружается методом `imageNamed()`, приложение будет сначала искать изображение по указанному имени в проекте локализации. Если приложение найдет такое изображение, то воспользуется им, а иначе обратится к ресурсу базового языка.

При наличии в приложении нескольких проектов локализации, соответствующих критериям поиска локализованных ресурсов, например одного проекта под названием `fr-CH.lproj` и другого — под названием `fr.lproj`, оно будет сначала искать ресурс, соответствующий более конкретному критерию поиска (в данном случае это проект `fr-CH.lproj`, если пользователь из Швейцарии выбрал французский язык). Если нужный ресурс не будет найден в данном месте, приложение попытается найти его в проекте `fr.lproj`. Это дает возможность предоставлять в одном проекте локализации ресурсы, являющиеся общими для всех говорящих на выбранном языке, локализуя только те ресурсы, на которые оказывают влияние различия в диалекте и географическом регионе.

Локализовать следует только те ресурсы, на которые оказывает влияние язык или страна. Так, если изображение, используемое в приложении, не содержит слов 'и' имеет универсальный смысл, то локализовать такое изображение нет никакой необходимости.

Файлы символьных строк

Что же делать со строковыми литералами и константами в исходном коде? Рассмотрим для примера исходный код из главы 19, показанный в листинге 22.1.

Листинг 22.1. Пример строковых литералов и констант

```
let alertController = UIAlertController(title: "Location Manager Error",
                                       message: errorType,
                                       preferredStyle: .alert)
let okAction = UIAlertAction(title: "OK",
                             style: .cancel,
                             handler: nil)
alertController.addAction(okAction)
self.presentViewController(alertController, animated: true,
                           completion: nil)
```

Если вы потратили немало усилий на локализацию своего приложения для конкретного круга пользователей, то вам, безусловно, не хотелось бы, чтобы предупреждающие сообщения появлялись на базовом языке разработки. Выходом из этого затруднительного положения может стать хранение символьных строк подобных предупреждающих сообщений в специальных текстовых файлах, называемых **файлами символьных строк**.

Содержимое файла символьных строк

Файлы символьных строк представляют собой ни что иное, как текстовые файлы, набранные в уникоде и содержащие список, состоящий из пар символьных строк, каждая из которых снабжена комментариями. В листинге 22.2 приведен пример того, как может выглядеть содержимое файла символьных строк в приложении.

Листинг 22.2. Пример файла символьных строк

```
/* Используется для запроса у пользователя его имени */
"LABEL_FIRST_NAME" = "First Name";

/* Используется для запроса у пользователя его фамилии */
"LABEL_LAST_NAME" = "Last Name";

/* Используется для запроса у пользователя даты его рождения */
"LABEL_BIRTHDAY" = "Birthday";
```

Все, что находится между знаками `/*` и `*/`, служит только в качестве комментария для переводчика. Эти комментарии не используются в приложении, поэтому их можно опустить, хотя польза от них несомненна, ведь они задают контекст, показывая, каким образом конкретная символьная строка используется в приложении. Как видите, в каждой строке из приведенного выше примера одна и та же символьная строка повторяется дважды. Символьная строка слева от знака равенства служит в качестве ключа и всегда содержит одно и то же значение независимо от выбранного языка, а символьная строка справа от знака равенства — перевод этого значения на местный язык. Таким образом, приведенное выше содержимое файла символьных строк после локализации на русский язык может выглядеть так, как показано в листинге 22.3.

Листинг 22.3. Французская версия файла символьных строк

```
/* Используется для запроса у пользователя его имени */
"LABEL_FIRST_NAME" = "Prénom";

/* Используется для запроса у пользователя его фамилии */
"LABEL_LAST_NAME" = "Nom de famille";

/* Используется для запроса у пользователя даты его рождения */
"LABEL_BIRTHDAY" = "Anniversaire";
```

Функция для локализации символьных строк

Требуемые локализованные версии символьных строк можно получить с помощью макроса `NSLocalizedString()` на этапе выполнения приложения. Как только исходный код вашего приложения будет написан и готов к локализации, в среде Xcode будет произведен поиск во всех файлах исходного кода вхождений данной функции, извлечение из них однозначных символьных строк и последующая их вставка в файл, который можно передать переводчику (или можно перевести его содержимое самостоятельно). Как только это будет сделано, вы даете среде Xcode команду импортировать обновленный файл и использовать его содержимое для создания файлов из символьных строк, локализованных на тех языках, на которых вы предоставили переводы. Рассмотрим подробнее первую стадию данного процесса. Приведем сначала традиционное объявление символьной строки:

```
let myString = "First Name"
```

Для того чтобы локализовать эту символьную строку, нужно сделать следующее:

```
let myString = NSLocalizedString("LABEL_FIRST_NAME",
    comment: "Используется для запроса у пользователя его имени")
```

Макрос `NSLocalizedString()` принимает пять параметров, но три из них имеют стандартные значения, которых, как правило, оказывается достаточно. Поэтому ему можно передать два следующих параметра.

- В качестве первого параметра указывается ключ для поиска локализованной символьной строки. Если по этому ключу отсутствует локализованный текст, то в приложении он будет использован в качестве локализованного текста.
- В качестве второго параметра указывается комментарий, в котором поясняется, как пользоваться текстом. Этот комментарий появится в файле, передаваемом переводчику, а также в файле с локализованными символьными строками после его импорта.

Макрос `NSLocalizedString()` осуществляет поиск файла с локализуемыми символьными строками `localizable.strings` в пакете приложения в соответствующем проекте локализации. Если он не находит такой файл, то

возвращает свой первый параметр, т.е. ключ к запрашиваемому локализованному тексту. Если макрос `NSLocalizedString()` найдет файл с локализуемыми символьными строками, то он осуществит в нем поиск строки, совпадающей с первым ее параметром. В приведенном выше примере данная функция будет искать строку "LABEL_FIRST_NAME" в файле символьных строк. Если же этот макрос не найдет символьную строку, совпадающую с указанной, в проекте локализации, соответствующем языковым настройкам пользователя, она продолжит поиск в файле символьных строк на базовом языке разработки и воспользуется найденным там значением. При отсутствии файла с локализуемыми символьными строками функция `NSLocalizedString()` просто воспользуется первым переданным ей параметром.

В качестве ключа, передаваемого макросу `NSLocalizedString()`, можно воспользоваться текстом на базовом языке, поскольку этот макрос возвращает переданный ей ключ, если она не обнаружит совпадающий с ним локализованный текст. В таком случае приведенный выше пример кода примет следующий вид:

```
let myString = NSLocalizedString("First Name",
    comment: "Used to ask the user his/her first name")
```

Тем не менее такой подход не рекомендуется по двум причинам. Во-первых, маловероятно получить с первой же попытки идеальный текст для приложения, а постоянно изменять все ключи в файлах с локализуемыми символьными строками неудобно и чревато ошибками. Это означает большую вероятность того, что ключи вообще не будут совпадать с текстом, используемым в приложении. И во-вторых, набирая ключи прописными буквами, можно сразу же обнаружить, что, глядя только на любой текст, когда приложение выполняется, можно просто забыть его локализовать.

Теперь, когда вам стал понятен принцип действия архитектуры локализации и файлов символьных строк, рассмотрим локализацию приложения на конкретном практическом примере.

Реализация приложения LocalizeMe

Итак, разработаем небольшое приложение, отображающее текущие региональные параметры пользователя. **Региональные параметры** (экземпляр класса `NSLocale`) представляют не только язык, но и регион пользователя. Они используются в системе с целью определить язык для последующего взаимодействия с пользователем, а также порядок отображения дат, денежных единиц, времени и прочей информации с учетом местной специфики. Сначала разработаем приложение, а затем локализуем его на других языках. По ходу дела вы научитесь выполнять локализацию файлов раскадровки, файлов символьных строк, изображений и даже отображаемого наименования вашего приложения.

Рассматриваемое здесь приложение должно внешне выглядеть так, как показано на рис. 22.2. Его название, отображаемое в верхней части экрана, взято из региональных параметров пользователя. Слова, обозначающие простые числа

и перечисленные вдоль левого края экрана, будут заданы в результате локализации файла раскладовки. Слова, перечисленные вдоль правого края экрана, а также изображение флага в нижней части экрана будут выбираться программно во время выполнения данного приложения, исходя из предпочтетемого пользователем языка. Перейдем непосредственно к разработке приложения.

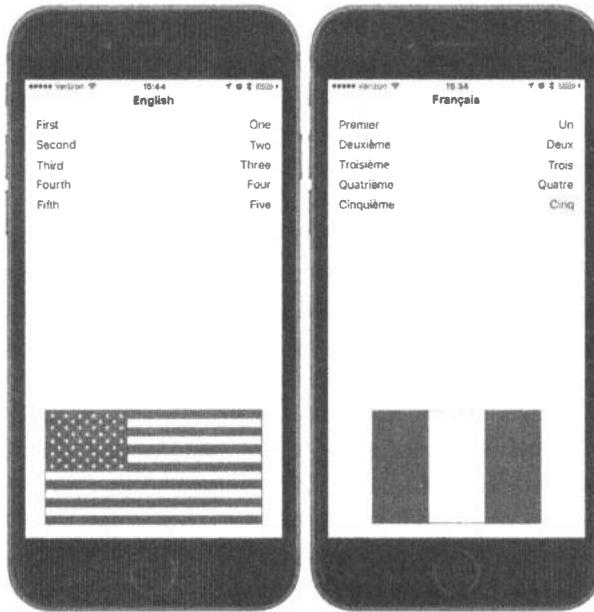


Рис. 22.2. Приложение LocalizeMe с настройками на двух языках

Создайте в среде Xcode новый проект, используя шаблон Single View Application. Присвойте новому проекту имя LocalizeMe. Найдите в папке 22 – Images с исходным кодом примеров, прилагаемым к этой книге, два файла изображений: flag_usa.png и flag_france.png. Выберите в среде Xcode папку ресурсов Assets.xcassets и перетащите в нее оба файла, flag_usa.png и flag_france.png. Теперь нужно создать выходы к некоторым меткам контроллера представления в данном проекте. В частности, один выход — для голубой метки у верхнего края представления, другой — для представления изображений, в котором воспроизводится флаг, и коллекцию выходов — для всех слов вдоль правого края экрана (см. рис. 22.1). С этой целью выберите исходный файл ViewController.swift и внесите в него изменения, выделенные ниже полужирным шрифтом.

```
class ViewController: UIViewController {
    @IBOutlet var localeLabel : UILabel!
    @IBOutlet var flagImageView : UIImageView!
    @IBOutlet var labels : [UILabel]!
```

Выберите файл раскадровки Main.storyboard, чтобы отредактировать графический пользовательский интерфейса данного приложения в среде Interface Builder. Перетащите метку из библиотеки объектов, опустив ее в верхней части главного представления и выровняв по верхней голубой направляющей линии. Измените размеры метки таким образом, чтобы расположить ее по всей ширине от одного до другого края данного представления. Если метка выделена, откройте инспектор атрибутов. Найдите элемент управления Font и щелкните на содержащейся в нем маленькой пиктограмме T, чтобы открыть небольшое всплывающее меню для выбора шрифта. Щелкните на селекторе Style и измените его значение на Bold, чтобы метка заглавия немного отличалась от остальных. Затем установите в инспекторе атрибутов выравнивание текста по центру. При желании можете воспользоваться селектором шрифтов, чтобы подобрать более крупный шрифт для текста метки. Если же в инспекторе атрибутов метки выбран вариант Minimum Font Size из раскрывающегося списка Autoshrink, то размеры текста автоматически изменятся, чтобы подогнать его по длине. Расположив метку в нужном месте, нажмите клавишу <Control>, перетащите указатель от пиктограммы View Controller к этой новой метке и выберите выход localeLabel.

Перетащите еще пять меток из библиотеки объектов и расположите их одну под другой вдоль левого края представления, используя соответствующую голубую направляющую линию (см. рис. 22.2). Дважды щелкните на самой верхней из этих меток и измените текст ее надписи с Label на First. Повторите эту процедуру для четырех остальных меток, изменив текст их надписей на Second, Third, Fourth и Fifth соответственно.

Перетащите из библиотеки объектов еще пять меток, расположив их на этот раз вдоль правого края текущего представления. Выровняйте текст надписей этих меток по правому краю, используя инспектор атрибутов. Нажмите клавишу <Control> и перетащите указатель от пиктограммы View Controller к каждой из пяти новых меток в отдельности, связав каждую из них с коллекцией выходов меток в правильном порядке сверху вниз.

Далее перетащите графическое представление из библиотеки объектов, расположив его таким образом, чтобы оно касалось нижней и левой голубых направляющих линий. Выберите в инспекторе атрибутов файл значение flag_usa для атрибута Image данного представления, а затем измените размеры изображения таким образом, чтобы оно простипалось по ширине от одной голубой направляющей линии до другой, а по высоте — приблизительно на треть пользовательского интерфейса. Кроме того, измените в инспекторе атрибутов текущее значение атрибута Mode на Aspect Fit. Дело в том, что не все национальные флаги имеют одинаковые пропорции, и поэтому мы должны обеспечить правильный внешний вид изображения национального флага в локализованных версиях данного приложения. Благодаря установке этого значения атрибута Mode любые изображения, размещаемые в графическом представлении, будут подогнаны по его размерам, но с сохранением правильных пропорций, т.е. соотношения ширины и высоты. Нажмите клавишу <Control>, перетащите указатель

от контроллера представления к графическому представлению и выберите выход `flagImageView`.

Для того чтобы завершить построение пользовательского интерфейса, нужно произвести автоматическое наложение ограничений на его компоновку. Начиная с верхней метки, нажмите клавишу `<Control>`, перетащите указатель от нее к ее родительскому представлению в окне `Document Outline`, нажмите клавишу `<Shift>`, выберите команды `Leading Space to Container Margin`, `Trailing Space to Container Margin` и `Vertical Spacing to Top Layout Guide` из контекстного меню и нажмите клавишу `<Return>`.

Далее нужно уточнить положение каждого из пяти рядов меток. С этой целью нажмите клавишу `<Control>` и перетащите указатель от метки с текстом `First` к ее родительскому представлению в окне `Document Outline`, нажмите клавишу `<Shift>` и выберите команды `Leading Space to Container Margin` и `Vertical Spacing to Top Layout Guide` из контекстного меню и нажмите клавишу `<Return>`. Снова нажмите клавишу `<Control>`, перетащите указатель от одной метки к другой в том же самом ряду и выберите команду `Baseline` из контекстного меню, а затем еще раз нажмите клавишу `<Control>`, перетащите указатель по горизонтали от правой метки к родительскому представлению в окне `Document Outline` и выберите команду `Trailing Space to Container Margin` из контекстного меню.

Итак, вы расположили верхний ряд меток. Сделайте то же самое с остальными рядами меток. Выберите все пять меток на правом краю, нажимая клавишу `<Shift>` и щелкая на них по очереди, а затем выберите команду меню `Editor⇒Size to Fit Content`. После этого можете удалить исходный текст каждой метки, поскольку он будет задан в дальнейшем программно.

Для того чтобы зафиксировать положение флага, нажмите клавишу `<Control>`, перетащите указатель от метки флага к родительскому представлению в окне `Document Outline`, нажмите клавишу `<Shift>` и выберите команды `Leading Space to Container Margin`, `Trailing Space to Container Margin` и `Vertical Spacing to Bottom Layout Guide` из всплывающего меню, а затем нажмите клавишу `<Return>`. Выберите метку флага, щелкните на кнопке `Pin`, установите флажок `Height` и щелкните на кнопке `Add 1 Constraint` в открывшемся окне. Итак, все нужные ограничения на компоновку графического пользовательского интерфейса данного приложения автоматически наложены.

Сохраните изменения, внесенные вами в раскладовку, а затем перейдите к исходному файлу `ViewController.swift` и введите в теле метода `viewDidLoad()` строки кода, приведенные в листинге 22.4.

Листинг 22.4. Модификация метода viewDidLoad

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Дополнительная настройка после загрузки представления,
    // обычно из nib-файла.
    let locale = Locale.current
    let currentLangID = Locale.preferredLanguages[0]
```

```

let displayLang = locale.displayName(forKey: Locale.Key.languageCode,
                                    value: currentLangID)
let capitalized = displayLang?.capitalized(with: locale)
localeLabel.text = capitalized

labels[0].text = NSLocalizedString("LABEL_ONE", comment: "The number 1")
labels[1].text = NSLocalizedString("LABEL_TWO", comment: "The number 2")
labels[2].text = NSLocalizedString("LABEL_THREE", comment: "The number 3")
labels[3].text = NSLocalizedString("LABEL_FOUR", comment: "The number 4")
labels[4].text = NSLocalizedString("LABEL_FIVE", comment: "The number 5")
let flagFile = NSLocalizedString("FLAG_FILE", comment: "Name of the flag")
flagImageView.image = UIImage(named: flagFile)
}

```

В этом методе мы получаем сначала экземпляр класса `NSLocale`, представляющего текущие региональные параметры пользователя, из которых можно узнать языковые и региональные предпочтения пользователя, указанные им в стандартном для мобильного устройства приложении `Settings`, как показано ниже.

```
let locale = Locale.current
```

Далее мы извлекаем предпочитаемый пользователем язык. В итоге получаем двухзначный код языка (например, `en` или `fr`) или же символьную строку вроде `"fr_CH"` для региональной разновидности языка:

```
let currentLangID = Locale.preferredLanguages[0]
```

Следующая строка кода из рассматриваемого здесь метода требует более подробного пояснения.

```
let displayLang = locale.displayName(forKey: Locale.Key.languageCode,
                                    value: currentLangID)
```

Объект типа `NSLocale` действует аналогично словарю. Он может предоставить всю необходимую информацию о текущих региональных предпочтениях пользователя, включая наименование используемой денежной единицы и предполагаемый формат даты. С полным перечнем информации, предоставляемой объектом типа `NSLocale`, можно ознакомиться в справочном руководстве по прикладному программному интерфейсу API класса `NSLocale`. В этой строке кода вызывается метод `displayName(forKey value:)` с целью извлечь конкретное название выбранного языка, переведенное на язык текущих региональных параметров пользователя. Назначение этого метода — возвратить значение запрашиваемого элемента на конкретном языке.

Так, название французского языка по-французски должно выводиться на экран как `français`, а по-английски — как `French`. Данный метод дает возможность извлекать сведения о любых региональных параметрах, чтобы отображать информацию в подходящем для пользователя виде. В данном случае мы получаем отображаемое название региона на языке этого региона, для чего передаем рассматриваемому здесь методу идентификатор `currentLangID` в качестве второго аргумента, как показано ниже. Этот идентификатор представляет собой

символьную строку, состоящую из двухзначного кода языка в формате, которым мы пользовались ранее, при разработке языковых проектов. Для пользователя, говорящего на американском английском языке, этот идентификатор будет иметь значение `en`, а для франкоязычного пользователя из Франции — значение `fr`.

Получаемое в итоге название языка должно быть чем-то вроде "English" или "français", причем оно пишется с прописной буквой только в том случае, если это всегда делается в названиях языков, предпочитаемых пользователем, что характерно для английского, но не французского языка. Тем не менее нам нужно написать название языка с прописной буквы для его отображения в заголовке данного приложения. К счастью, в классе `NSString` имеются специальные методы для написания символьных строк с прописных букв, в том числе те, которые делают это по правилам, установленным в заданных региональных параметрах! Так, в следующей строке кода название французского языка "français" преобразуется в "Français":

```
let capitalized = displayLang?.capitalized(with: locale)
```

В данном случае используется тот факт, что классы `NSString` языка Objective-C и `String` языка Swift прозрачно состыкованы для вызова метода `capitalizedStringWithLocale()` из класса `NSString` для экземпляра класса `String`. Получив отображаемое название регионального языка, мы пользуемся им для установки верхней метки в текущем представлении следующим образом:

```
localeLabel.text = capitalized
```

Далее мы устанавливаем пять других меток в соответствии с номерами от одного до пяти, написанными словами на базовом языке разработки данного приложения. Для получения текста этих меток вызывается метод `NSLocalizedString()`, которому передаются ключ и комментарии к каждому слову, как показано ниже. Если слова очевидны, как в данном случае, вместо комментариев в качестве второго аргумента данному методу можно передать пустую строку. Но любая символьная строка, передаваемая в качестве второго аргумента, будет все равно превращена в комментарий в файле с символьными строками, поэтому такой комментарий может оказаться полезным для работы в более тесном контакте с переводчиком.

```
labels[0].text = NSLocalizedString("LABEL_ONE", comment: "The number 1")
labels[1].text = NSLocalizedString("LABEL_TWO", comment: "The number 2")
labels[2].text = NSLocalizedString("LABEL_THREE", comment: "The number 3")
labels[3].text = NSLocalizedString("LABEL_FOUR", comment: "The number 4")
labels[4].text = NSLocalizedString("LABEL_FIVE", comment: "The number 5")
```

Наконец мы осуществляем еще один поиск символьных строк, чтобы найти название изображения флага и заполнить именованным изображением соответствующее представление изображений, как показано ниже.

```
let flagFile = NSLocalizedString("FLAG_FILE", comment: "Name of the flag")
flagImageView.image = UIImage(named: flagFile)
```

Теперь перейдем к компиляции и запуску данного приложения на выполнение. После запуска приложение LocalizeMe будет выглядеть так, как показано на рис. 22.3.

Воспользовавшись функцией `NSLocalizedString()` вместо статических символьных строк, мы подготовили рассматриваемое здесь приложение к локализации, но пока еще не локализовали его, о чем свидетельствуют названия меток прописными буквами справа и отсутствие флага внизу экрана. Если вы воспользуетесь стандартным приложением `Settings` в симуляторе или на своем мобильном устройстве под управлением системы iOS, чтобы выбрать другой язык или регион, то выполнение приложения LocalizeMe приведет к тем же самым результатам, за исключением метки у верхнего края представления (рис. 22.4).

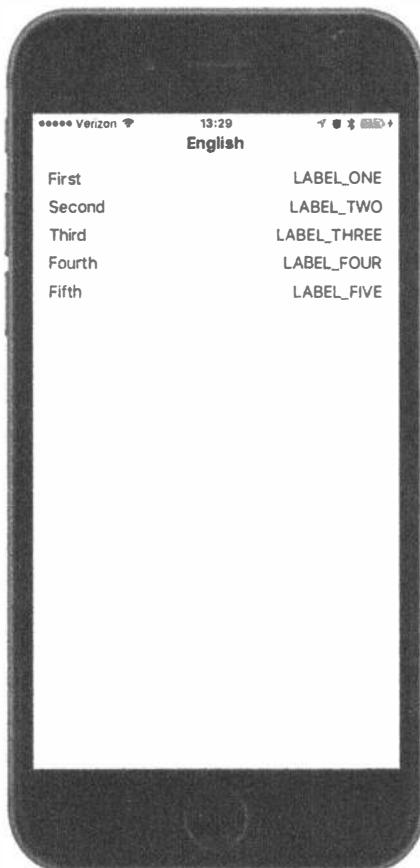


Рис. 22.3. Нелокализованное приложение LocalizeMe, выполняющееся на мобильном устройстве iPhone и настроенное для применения на французском языке

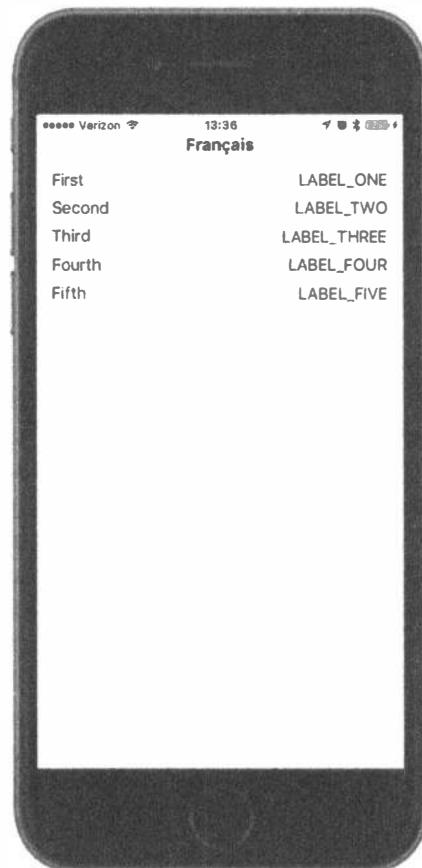


Рис. 22.4. Нелокализованное приложение, выполняемое на устройстве iPhone и настроенное на французский язык

Локализация проекта

Теперь локализуем проект данного приложения. Перейдите в среду Xcode, щелкните на пиктограмме LocalizeMe в окне навигатора проекта и выберите вкладку Info для данного проекта. Посмотрите на раздел Localizations вкладки Info. Как видите, в нем показан только один вариант локализации: English. Такая локализация обычно называется **базовой** и вводится автоматически при создании проекта в среде Xcode. Нам требуется локализация на французском языке, поэтому щелкните на кнопке со знаком +, расположенной внизу раздела Localizations, и выберите вариант French (fr) из раскрывающегося списка (рис. 22.5).

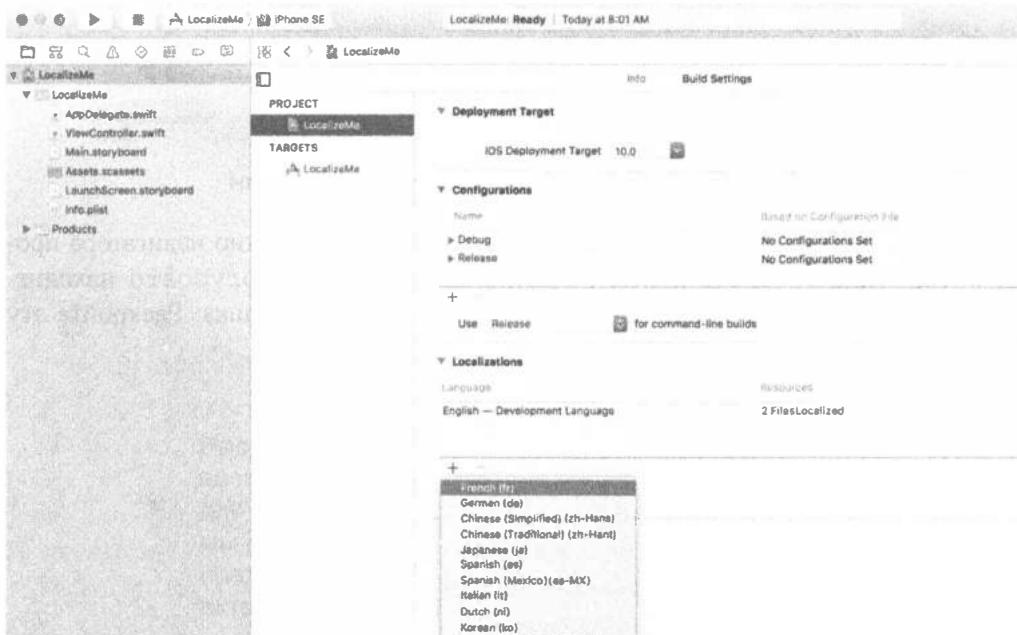


Рис. 22.5. Настройки локализации проекта и другая информация

Вам будет предложено выбрать все существующие локализуемые файлы и базовый язык, с которого должна начинаться французская локализация (рис. 22.6). Иногда начинать выгодно с выбора файлов для нового языка, опираясь на те файлы другого языка, на котором уже осуществлена локализация. Например, чтобы выполнить локализацию на французском языке для пользователей из Швейцарии в проекте, который уже переведен на французский язык, что и будет сделано далее в этой главе, лучше выбрать уже имеющуюся локализацию на французском языке в качестве отправной точки вместо базовой локализации. Для этого достаточно выбрать вариант French из раскрывающегося списка Reference Language. Но в настоящий момент для локализации имеются лишь два файла и один вариант выбора языка в качестве отправной точки (т.е. базового языка), поэтому оставьте все как есть и щелкните на кнопке Finish.



Рис. 22.6. Выбор файлов для локализации

Введя локализацию на французском языке, вернитесь в окно навигатора проекта. Как видите, теперь рядом с именем файла Main.storyboard находится раскрывающий треугольник, как будто это группа или папка. Раскройте эту группу и просмотрите ее содержимое (рис. 22.7).

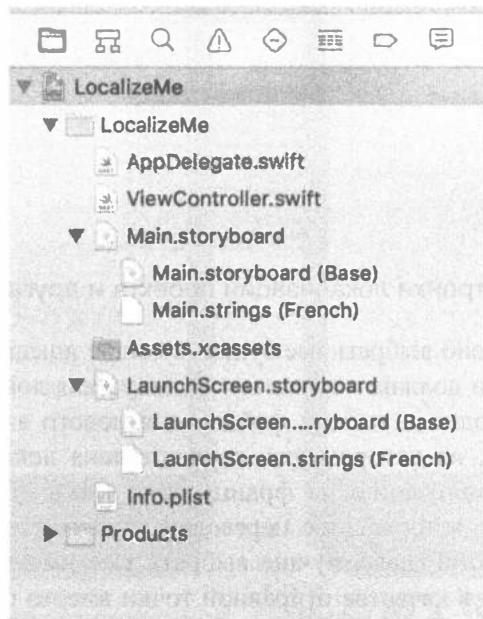


Рис. 22.7. Рядом с именами локализованных файлов расположен треугольник раскрытия и значок потомка для каждого добавляемого языка и региона

В рассматриваемом здесь проекте файл Main.storyboard представлен в виде группы, имеющей два потомка. Первый из них называется Main.strings и помечен как French. Второй помечен как Base, формируется автоматически при создании проекта и обозначает базовый язык разработки приложений. То же самое относится и к файлу LaunchScreen.xib. Каждый из этих файлов находится в отдельной папке: Base.lproj и fr.lproj. Перейдите в окно Finder и откройте папку LocalizeMe в папке проекта LozalizeMe. Помимо остальных файлов данного проекта, вы должны обнаружить в ней папки Base.lproj и fr.lproj (рис. 22.8).

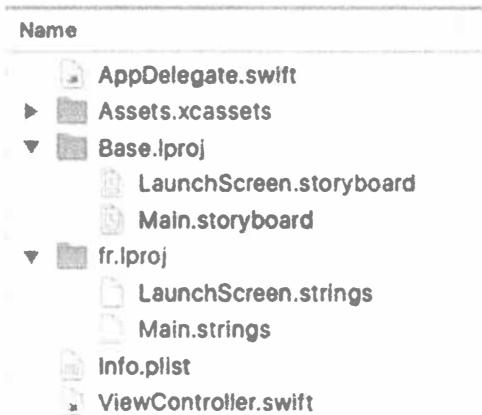


Рис. 22.8. Изначально в наш проект, разрабатываемый в среде Xcode, включена папка проекта для базового языка (Base.lproj). После того как мы сделали файл раскадровки локализуемым, в среде Xcode была создана еще одна папка проекта для выбранного языка (fr.lproj)

Обратите внимание на то, что папка Base.proj находится там с самого начала и содержит копию файла Main.storyboard. Если в среде Xcode обнаруживается ресурс, требующий только одного варианта локализации, то он отображается в виде отдельного элемента. Если же ресурс имеет несколько вариантов локализации, то он отображается как группа. Когда среде Xcode была дана команда произвести локализацию на французском языке, в папке текущего проекта была создана новая папка локализации fr.lproj, а в ней размещены файлы с символьными строками, содержащими значения, извлеченные из файлов Base.lproj/Main.storyboard и Base.lproj/LaunchScreen.storyboard. Вместо дублирования обоих файлов средствами Xcode из них просто извлекается каждая текстовая строка, и затем из этих строк создаются файлы, готовые для локализации. Когда приложение компилируется и запускается на выполнение, значения из файлов с локализованными символьными строками извлекаются для замены значений в файлах раскадровки и начального экрана.

Локализация раскадровки

Перейдите в среду Xcode и выберите в навигаторе проекта элемент Main.strings (French), чтобы открыть файл с символьными строками на французском языке, которые будут вставлены в раскадровку, показываемую франкоязычным пользователям. В этом файле вы обнаружите текст, аналогичный следующему:

```
/* Class = "UILabel"; text = "Fifth"; ObjectID = "5tN-09-txB"; */
"5tN-09-txB.text" = "Fifth";

/* Class = "UILabel"; text = "Third"; ObjectID = "G05-hd-zou"; */
"G05-hd-zou.text" = "Third";

/* Class = "UILabel"; text = "Second"; ObjectID = "NCJ-hT-XgS"; */
"NCJ-hT-XgS.text" = "Second";

/* Class = "UILabel"; text = "Fourth"; ObjectID = "Z6w-b0-U06"; */
"Z6w-b0-U06.text" = "Fourth";

/* Class = "UILabel"; text = "First"; ObjectID = "kS9-Wx-xgy"; */
"kS9-Wx-xgy.text" = "First";

/* Class = "UILabel"; text = "Label"; ObjectID = "yGf-tY-SVz"; */
"yGf-tY-SVz.text" = "Label";
```

Каждая пара строк кода обозначает символьную строку, обнаруженную в раскадровке. В комментариях указываются класс объекта, содержащего символьную строку, сама символьная строка и однозначный идентификатор каждого объекта, который, скорее всего, будет другим в копии данного файла. По существу, изменить требуется значение из правой части операции присваивания в строке кода, следующей после каждого комментария. Как видите, некоторые слова, обозначающие простые числа (например, First), взяты из меток, показанных на рис. 22.4, слева, и получивших свои имена в раскадровке. Запись под именем Label сделана для метки заглавия, которое устанавливается программно и поэтому не требует локализации.

До версии iOS 8 локализация раскадровки, как правило, заключалась в непосредственном редактировании данного файла. По желанию это можно по-прежнему сделать и в версии iOS 8, но если предполагается привлечь профессионального переводчика, то удобнее, чтобы он перевел сразу текст в раскадровке и символьные строки в прикладном коде. Именно поэтому компания Apple предоставила возможность собирать все символьные строки, требующие перевода на каждый язык, в одном файле, который можно передать переводчику. Если вы собираетесь пойти именно по такому пути, оставьте файл с символьными строками раскадровки и перейдите к следующей стадии данного процесса, описываемой в следующем разделе. Но все-таки файл с символьными строками раскадровки можно видоизменить, и даже в этом случае внесенные изменения не будут потеряны, если потребуется внести корректиды в перевод или

локализовать дополнительный текст. Именно на этот случай мы и локализуем символьные строки раскадровки старым способом. С этой целью найдите текст меток First, Second, Third, Fourth и Fifth и замените его текстом Premier, Deuxième, Troisième, Quatrième и Cinquième соответственно в правой части каждой операции присваивания символьной строки. По завершении сохраните данный файл.

```
/* Class = "UILabel"; text = "Fifth"; ObjectID = "5tN-09-txB"; */
"5tN-09-txB.text" = "Cinquième";

/* Class = "UILabel"; text = "Third"; ObjectID = "G05-hd-zou"; */
"G05-hd-zou.text" = "Troisième";

/* Class = "UILabel"; text = "Second"; ObjectID = "NCJ-hT-XgS"; */
"NCJ-hT-XgS.text" = "Deuxième";

/* Class = "UILabel"; text = "Fourth"; ObjectID = "Z6w-b0-U06"; */
"Z6w-b0-U06.text" = "Quatrième";

/* Class = "UILabel"; text = "First"; ObjectID = "kS9-Wx-xgy"; */
"kS9-Wx-xgy.text" = "Premier";

/* Class = "UILabel"; text = "Label"; ObjectID = "yGf-tY-SVz"; */
"yGf-tY-SVz.text" = "Label";
```

Сохраните этот файл. Теперь наша раскадровка локализована на французском языке. Есть три способа увидеть локализации в своем приложении — предварительный просмотр в среде Xcode, специальная схема запуска или изменение активного языка на симуляторе или реальном устройстве. Рассмотрим эти возможности по очереди, начиная с предварительного просмотра.

Использование помощника редактора для просмотра локализации

Выберите файл Main.storyboard в окне навигатора проекта и откройте окно помощника редактора. На панели быстрых переходов в окне помощника редактора выберите команду Preview⇒Main.storyboard. На экране появится приложение, настроенное на базовый язык (рис. 22.9).

В нижней части окна помощника редактора вы видите текущий язык приложения (English). Щелкните на этой кнопке, чтобы открыть всплывающий список всех возможных локализаций вашего проекта, и выберите пункт French. Механизм предварительного просмотра обновит представление приложения для французских пользователей, как показано на рис. 22.9, слева. Однако национальный флаг отображается неправильно. Это объясняется тем, что механизм предварительного просмотра учитывает только локализацию раскадровки, а изображение флага задается в коде. Если локализация устанавливается с помощью кода, необходимо правильно выбрать установки, чтобы получить правильное представление.

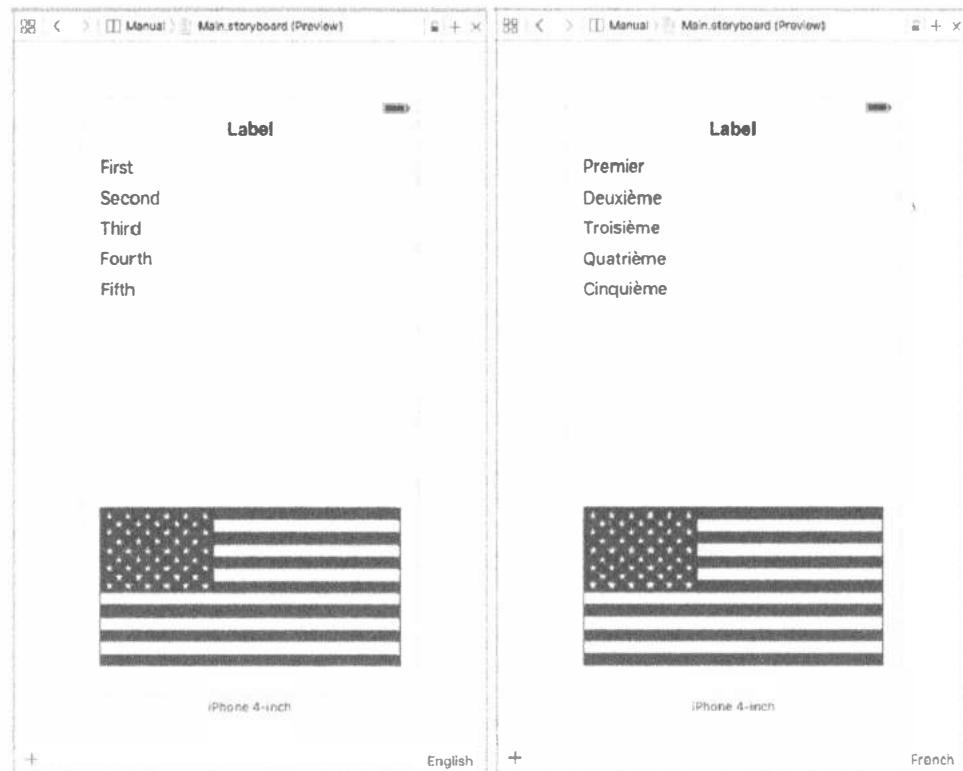


Рис. 22.9. Предварительный просмотр приложения на базовом языке и на французском языке

Использование настроек для изменения языка и региона

Создание специализированной схемы позволяет быстро увидеть локализованную версию приложения, выполняемого на симуляторе или реальном устройстве. В отличие от предварительного просмотра, специализированная схема позволяет увидеть локализацию, реализованную как в коде, так и в раскладовке. Щелкните на левой части селектора Scheme в среде Xcode. Этот селектор расположен на верхней панели, следом за кнопками Run и Stop. В данный момент селектор должен показывать на экране текст LocalizeMe, представляющий собой имя текущей схемы, а также выбранное устройство или симулятор. После щелчка на тексте LocalizeMe среда Xcode открывает контекстное меню с несколькими командами. Выберите команду Manage Schemes..., чтобы открыть диалоговое окно Scheme, как показано на рис. 22.10.

В данный момент у нас есть только одна схема. Щелкните на пиктограмме + под списком схем, чтобы открыть другое окно, которое позволит выбрать название новой схемы. Назовите ее LocalizeMe_fr и нажмите кнопку OK. Вернитесь в диалоговое окно Scheme, выберите только что созданную схему и щелкните на кнопке Edit..., чтобы открыть окно редактора схем (рис. 22.11).



Рис. 22.10. Диалог Scheme позволяет видеть, добавлять и удалять схемы



Рис. 22.11. Окно редактора схем, в котором в списке Application Language выбран пункт French, а в списке Application Region — France

Убедитесь, что в левом столбце выбран пункт Run, а затем обратите внимание на списки Application Language и Application Region в основном окне редактора. Здесь можно выбрать язык и регион, на которые настроено приложение в соответствии со специальной схемой. Выберите язык French и регион France, а затем щелкните на кнопке Close. Вернитесь в главное окно Xcode, в котором вы увидите, что новая схема теперь является выбранной. Скомпилируйте и выполните приложение, и увидите, что французская локализация была активизирована (рис. 22.12).



Рис. 22.12. Просмотр текущего состояния приложения на французском языке

Как видите, изображение флага отсутствует. Конечно, это объясняется тем, что флаг задается в коде, а мы пока завершили французскую локализацию. Для того чтобы вернуться к базовой локализации, просто переключитесь на исходную схему LocalizeMe и запустите приложение снова.

Переключение настроек языка и региона на устройстве или симуляторе

Последний способ увидеть, как ваше приложение выглядит на иностранном языке или после настроек для другого региона, — переключение этих настроек на симуляторе или устройстве. Этот вариант потребует немного больше времени, чем первый или второй вариант, поэтому лучше всего делать это только в конце цикла тестирования, когда вы твердо уверены, что все в порядке. Ниже мы покажем, как сделать французский язык базовым на вашем устройстве (или симуляторе).

Запустите приложение **Settings**, выберите строку **General**, а затем выберите строку с именем **Language and Region**. Здесь вы можете изменить настройки языка и региона (рис. 22.13)

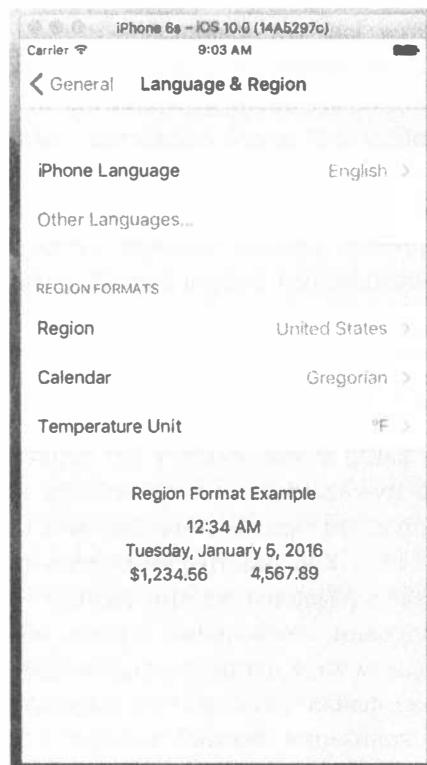


Рис. 22.13. Изменение настроек языка и региона

Коснитесь пальцем ссылки **iPhone Language**, чтобы открыть список языков, на которых локализована система iOS, а затем найдите и выберите в нем элемент **French**, который по-французски обозначен как **Français**. Нажмите кнопку **Done** и подтвердите свое требование сменить язык на данном мобильном устройстве. В течение нескольких секунд произойдет частичная перезагрузка мобильного устройства. Запустив данное приложение снова на выполнение, вы обнаружите, что на месте меток с левой стороны экрана теперь отображается текст, локализованный на французском языке (рис. 22.12). Однако национальный флаг и текст в правом столбце по-прежнему отображаются неправильно. Этот недостаток будет исправлен в следующем разделе.

Формирование и локализация файла с символьными строками

Как следует из рис. 22.12, вместо текста, обозначающего простые числа по-французски, с правой стороны экрана по-прежнему появляются названия меток прописными буквами, поскольку они еще не переведены. Они используются как

ключи в функции `NSLocalizedString()` для поиска локализованных текстовых строк. Для того чтобы локализовать их, придется сначала извлечь ключи и строки комментариев из кода. Правда, в версии Xcode 6 совсем не трудно извлечь локализуемый текст из проекта и разместить его в одном файле на каждый язык. Ниже поясняется, как это делается.

Выберите сначала рассматриваемый здесь проект в окне навигатора проекта, затем — этот проект или его цели в редакторе и далее — команду меню `Editor⇒Export for Localization....`. В итоге откроется диалоговое окно, в котором выбираются те языки, на которые требуется локализовать данный проект, а также место для записи файлов на каждом языке. Выберите место для записи файла (например, корневой каталог данного проекта), затем вариант `Existing Translations` из раскрывающегося списка `Include`, установите флагок `French` в разделе `Languages` и нажмите кнопку `Save`. В выбранном вами месте средствами Xcode будет создан файл `fr.xliff`. Если вы собираетесь воспользоваться услугами сторонней организации для перевода текста своего приложения и если там работают с файлами формата XLIFF, что вполне вероятно, вам останется только передать им этот файл, а они обновят его переведенными символьными строками, после чего его нужно снова импортировать в среде Xcode. Однако в рассматриваемом здесь простом примере мы сделаем перевод самостоятельно.

Откройте файл `fr.xliff`. Как видите, он содержит немало кода разметки в формате XML. Этот файл разделен на три разные части, содержащие символьные строки из раскадровки, символьные строки, обнаруженные средствами программы Xcode в исходном коде данного приложения, а также целый ряд локализованных значений из файла `Info.plist` текущего проекта. О причинах, по которым требуется локализация записей в файле `Info.plist`, речь пойдет далее в этой главе, а пока переведем текст, извлеченный из исходного кода данного приложения. Просмотрите содержимое файла `fr.xliff` и найдите в нем следующий текст, встроенный в XML-разметку:

```
<file original="LocalizeMe/Localizable.strings"
      source-language="en" datatype="plaintext"
      target-language="fr">
<header>
  <tool tool-id="com.apple.dt.xcode" tool-name="Xcode"
        tool-version="n.n.n" build-num="nnnnn"/>
</header>
<body>
  <trans-unit id="FLAG_FILE">
    <source>FLAG_FILE</source>
    <note>Name of the flag</note>
  </trans-unit>
  <trans-unit id="LABEL_FIVE">
    <source>LABEL_FIVE</source>
    <note>The number 5</note>
  </trans-unit>
  <trans-unit id="LABEL_FOUR">
    <source>LABEL_FOUR</source>
```

```

<note>The number 4</note>
</trans-unit>
<trans-unit id="LABEL_ONE">
  <source>LABEL_ONE</source>
  <note>The number 1</note>
</trans-unit>
<trans-unit id="LABEL_THREE">
  <source>LABEL_THREE</source>
  <note>The number 3</note>
</trans-unit>
<trans-unit id="LABEL_TWO">
  <source>LABEL_TWO</source>
  <note>The number 2</note>
</trans-unit>
</body>
</file>

```

Как видите, для каждой символьной строки, которую требуется перевести на французский язык, имеется элемент разметки `<trans-unit>`, который, в свою очередь, содержит элементы разметки `<source>` и `<note>` с оригинальным текстом и комментариями, извлеченными из исходного кода функцией `NSLocalizedString()`. В распоряжении профессиональных переводчиков должны быть программные средства, предоставляющие содержимое данного файла и позволяющие им вводить в него переведенный текст. Однако вам придется вручную ввести в него элементы разметки `<target>`, содержащие текст на французском языке.

```

<file original="LocalizeMe/Localizable.strings"
      source-language="en" datatype="plaintext"
      target-language="fr">
<header>
  <tool tool-id="com.apple.dt.xcode" tool-name="Xcode"
        tool-version="n.n.n" build-num="nnnnn"/>
</header>
<body>
  <trans-unit id="FLAG_FILE">
    <source>FLAG_FILE</source>
    <note>Name of the flag</note>
    <target>flag_france</target>
  </trans-unit>
  <trans-unit id="LABEL_FIVE">
    <source>LABEL_FIVE</source>
    <note>The number 5</note>
    <target>Cinq</target>
  </trans-unit>
  <trans-unit id="LABEL_FOUR">
    <source>LABEL_FOUR</source>
    <note>The number 4</note>
    <target>Quatre</target>
  </trans-unit>
  <trans-unit id="LABEL_ONE">
    <source>LABEL_ONE</source>
    <note>The number 1</note>
  </trans-unit>

```

```

<target>Un</target>
</trans-unit>
<trans-unit id="LABEL_THREE">
    <source>LABEL THREE</source>
    <note>The number 3</note>
    <target>Trois</target>
</trans-unit>
<trans-unit id="LABEL_TWO">
    <source>LABEL TWO</source>
    <note>The number 2</note>
    <target>Deux</target>
</trans-unit>
</body>
</file>

```

Если вы еще не перевели символьные строки из раскадровки, можете сделать это теперь, найдя их в отдельном блоке элементов разметки `<trans-unit>`. Их нетрудно обнаружить по комментариям, включающим в себя ссылки на метки, из которых извлечен локализуемый текст. Если же вы уже перевели строки из раскадровки, то обнаружите их перевод в выделенных ниже полужирным шрифтом строках XML-разметки, автоматически включенных в файл `fr.xliff` средствами Xcode.

```

<trans-unit id="GO5-hd-zou.text">
    <source>Third</source>
    <target>Troisième</target>
    <note>Class = "IBUILabel"; text = "Third"; ObjectID = "GO5-hd-zou";
    </note>
</trans-unit>
<trans-unit id="NCJ-hT-XgS.text">
    <source>Fourth</source>
    <target>Quatrième</target>
    <note>Class = "IBUILabel"; text = "Fourth"; ObjectID = "NCJ-hT-XgS";
    </note>
</trans-unit>

```

Сохраните результаты перевода, чтобы импортировать их снова в среде Xcode. С этой целью выполните команду `Editor⇒Import Localizations`, найдите файл `fr.xliff` и откройте его. В среде Xcode будет показан список ключей, для которых еще не предоставлен перевод. Эти ключи взяты из файла `Info.plist`, и локализация их значений будет произведена далее, а пока щелкните на кнопке `Import`, чтобы завершить процесс импортирования. В итоге вы обнаружите два файла, `InfoPlist.strings` и `Localizable.strings`, добавленных под заголовком `Supporting Files` в окне навигатора проекта. Открыв файл `Localizable.strings`, вы найдете в нем следующие символьные строки, извлеченные средствами Xcode из исходного файла `ViewController.swift` и переведенные на французский язык:

```

/* Имя флага */
"FLAG_FILE" = "flag_france";

```

```
/* Число 5 */
"LABEL_FIVE" = "Cinq";

/* Число 4 */
"LABEL_FOUR" = "Quatre";

/* Число 1 */
"LABEL_ONE" = "Un";

/* Число 3 */
"LABEL_THREE" = "Trois";

/* Число 2 */
"LABEL_TWO" = "Deux";
```

Теперь скомпилируйте и запустите данное приложение на выполнение. Метки с правой стороны экрана должны появиться в переводе на французский язык, а в нижней части экрана — французский флаг (рис. 22.14).

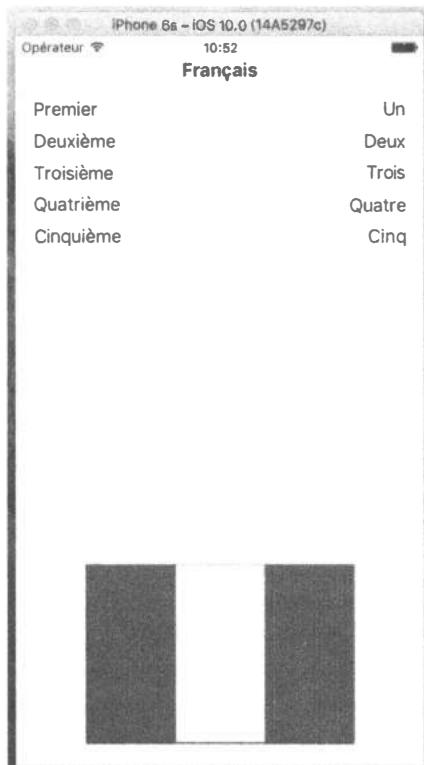


Рис. 22.14. Правильно локализованное приложение на французском языке, включая правильное изображение флага

Итак, локализацию можно считать завершенной? Не совсем. Перейдите к приложению *Settings*, переключитесь обратно на английский язык и снова запустите данное приложение на выполнение. В итоге появится его нелокализованная версия (см. рис. 22.3). Для того чтобы данное приложение нормально работало на английском языке, придется локализовать его на этом языке. С этой целью выполните сначала команду меню *Editor⇒Export for Localization...*, но на этот раз выберите пункт *Development Language Only* в раскрывающемся списке *Include*, а затем щелкните на кнопке *Save*. В итоге будет создан файл *en.xliff*, в котором и предстоит произвести локализацию на английском языке. Отредактируйте этот файл, включив в него следующие изменения.

```
<file original="LocalizeMe/Localizable.strings"
      source-language="en" datatype="plaintext">
<header>
  <tool tool-id="com.apple.dt.xcode" tool-name="Xcode"
        tool-version="n.n.n" build-num="nnnnn"/>
</header>
<body>
  <trans-unit id="FLAG_FILE">
    <source>FLAG_FILE</source>
    <note>Name of the flag</note>
    <target>flag_usa</target>
  </trans-unit>
  <trans-unit id="LABEL_FIVE">
    <source>LABEL_FIVE</source>
    <note>The number 5</note>
    <target>Five</target>
  </trans-unit>
  <trans-unit id="LABEL_FOUR">
    <source>LABEL_FOUR</source>
    <note>The number 4</note>
    <target>Four</target>
  </trans-unit>
  <trans-unit id="LABEL_ONE">
    <source>LABEL_ONE</source>
    <note>The number 1</note>
    <target>One</target>
  </trans-unit>
  <trans-unit id="LABEL_THREE">
    <source>LABEL_THREE</source>
    <note>The number 3</note>
    <target>Three</target>
  </trans-unit>
  <trans-unit id="LABEL_TWO">
    <source>LABEL_TWO</source>
    <note>The number 2</note>
    <target>Two</target>
  </trans-unit>
</body>
</file>
```

Импортируйте файл *en.xliff* с внесенными выше изменениями обратно в Xcode по команде меню *Editor⇒Import Localizations*.

Среда Xcode создаст папку en.lproj, а в ней — файлы InfoPlist.strings, Localizable.strings и Main.strings, содержащие символьные строки, локализованные на английском языке. По существу, мы ввели ссылки на файл изображения американского флага и текст, заменяемый по ключам при вызове функции NSLocalizedString() в коде данного приложения. Скомпилировав, запустив данное приложение на выполнение и выбрав английский язык, вы увидите на экране текст меток на этом языке, а под ними — американский флаг.

Требование предоставить файл изображения флага и текстовые строки для базовой локализации возникает потому, что мы решили не пользоваться локализованным текстом в качестве ключей при вызове функции NSLocalizedString(). Если бы мы сделали нечто вроде приведенного ниже, текст на английском языке появился бы в пользовательском интерфейсе при выборе любого языка, локализация на котором отсутствует, даже если бы мы и не предоставили базовую локализацию.

```
labels[0].text = NSLocalizedString("One", comment: "The number 1")
labels[1].text = NSLocalizedString("Two", comment: "The number 2")
labels[2].text = NSLocalizedString("Three", comment: "The number 3")
labels[3].text = NSLocalizedString("Four", comment: "The number 4")
labels[4].text = NSLocalizedString("Five", comment: "The number 5")
let flagFile = NSLocalizedString("flag_usa", comment: "Name of the flag")
```

И хотя такой способ вполне допустим, его недостаток заключается в следующем: если потребуется изменить любую из текстовых строк на английском языке, то придется изменить и ключ, используемый для поиска символьных строк на всех остальных языках. Следовательно, обновить вручную придется все файлы localized.strings, чтобы использовать в них новый ключ.

Локализация отображаемого названия приложения

В заключение покажем, каким образом решается еще одна часто возникающая задача локализации, а именно: локализация названия приложения, отображаемого на начальном экране или где-нибудь еще. Подобная локализация осуществляется в целом ряде стандартных приложений от компании Apple. Возможно, и вы пожелаете сделать то же самое. Название приложения обычно хранится в его файле Info.plist, который можно обнаружить в группе Supporting Files, перейдя в окно навигатора проекта. Выберите этот файл для редактирования и обратите внимание на то, что один из элементов в этом файле — Bundle display name — в настоящий момент имеет значение \${PRODUCT_NAME}. В синтаксисе, используемом в файлах Info.plist, все, что начинается со знака денежной единицы (\$), подлежит замене в переменной. В данном случае это означает, что, когда приложение компилируется в Xcode, значение такого элемента заменяется названием программного продукта из текущего проекта в Xcode, т.е. названием самого приложения. Именно здесь нам и требуется осуществить локализацию, заменив значение \${PRODUCT_NAME} локализованным на каждом языке названием. Но на практике сделать это не так просто, как кажется на первый взгляд.

Файл Info.plist — это частный случай, не предполагающий локализации. Поэтому вместо локализации содержимого файла Info.plist придется создать его локализованные версии под названием InfoPlist.strings. Однако, прежде чем сделать это, нужно создать базовую версию данного файла. Если вы придерживались процедуры, описанной в предыдущем разделе для локализации приложения, то, скорее всего, уже имеете в своем распоряжении английскую и французскую пустые версии этого файла. В противном случае выполните следующие действия, чтобы получить эти файлы.

1. Выберите сначала команду меню **File⇒New⇒File...**, затем раздел **iOS**, подраздел **Resource** и шаблон **Strings File**. Щелкните на кнопке **Next**, присвойте новому файлу имя **InfoPlist.strings**, назначьте его для группы **Supporting Files** в проекте **LocalizeMe** и создайте его.
2. Выберите новый файл и щелкните на кнопке **Localize** в инспекторе файлов. В открывшемся диалоговом окне перенесите этот файл в папку локализации на английском языке, а по возвращении в инспектор файлов установите флажок **French** в разделе **Localization**. В итоге копии этого файла должны появиться в группе **Supporting Files** в окне навигатора проекта для локализации как на французском, так и на английском языках.



Рис. 22.15. Пункт Strings File для выбора типа файла iOS Recource

В каждую локализованную копию данного файла нужно ввести строку кода, чтобы определить отображаемое название приложения. Как пояснялось выше, в

файле Info.plist отображаемое название приложения связано со словарным ключом Bundle display name, но оказывается, что это не настоящее имя ключа, а всего лишь изящный способ сделать это имя более удобочитаемым в среде Xcode. Настоящее имя данного ключа — CFBundleName, в чем вы можете сами убедиться, выбрав файл Info.plist, щелкнув правой кнопкой мыши в любом месте текущего представления и выбрав команду Show Raw Keys/Values из контекстного меню. В итоге будут представлены подлинные имена используемых ключей. Итак, выберите локализацию файла InfoPlist.strings на английском языке и введите или модифицируйте следующую строку кода:

```
"CFBundleName" = "Localize Me";
```

Этот ключ может уже существовать, если вы следовали описанной ранее процедуре локализации рассматриваемого здесь приложения на английском языке, поскольку он уже был введен в процесс импорта файла формата XLIFF. На самом деле еще один способ локализовать название приложения состоит в том, чтобы ввести его перевод в файл формата XLIFF таким же образом, как это было проделано ранее с текстовыми строками, которые требовалось перевести. Для этого достаточно найти запись CFBundleDisplayName и ввести элемент разметки <trans> с переведенным названием приложения. Аналогично выберите локализацию файла InfoPlist.strings на французском языке и отредактируйте его, чтобы присвоить приложению соответствующее название на французском языке, как показано ниже.

```
"CFBundleName" = "Localisez Moi";
```

Сберите и запустите данное приложение на выполнение, а затем нажмите кнопку Home, чтобы вернуться к начальному экрану. И разумеется, переключите мобильное устройство или симулятор на французский язык, если рабочим в настоящий момент выбран английский язык. Локализованное название приложения должно появиться под его пиктограммой, но иногда это происходит не сразу. По-видимому, система iOS кеширует эту информацию при вводе нового приложения, но совсем не обязательно изменяет ее, когда уже имеющееся приложение заменяется новой его версией, по крайней мере не при замене, выполняемой в среде Xcode. Поэтому, если вы выполняете симуляцию на французском языке, но не видите нового названия приложения, не отчаявайтесь, а просто удалите данное приложение из симулятора, вернитесь в среду Xcode и еще раз скомпилируйте и запустите приложение на выполнение. Теперь рассматриваемое здесь приложение можно считать полностью локализованным на французский язык.

ПРЕДУПРЕЖДЕНИЕ. Вы не увидите имя локализованного приложения, если запустите его в рамках специальной схемы. Единственный способ увидеть его — переключить устройство или симулятор на французский язык.

Теперь наше приложение полностью локализовано для французского и английского языков.

Добавление еще одной локализации

В завершение рассматриваемого здесь примера приложения введем в него еще одну локализацию. На этот раз оно будет локализовано на французском языке для пользователей из Швейцарии, где этот язык имеет свои местные отличия и обозначается кодом fr-SH.

Основной принцип локализации остается прежним. На самом деле, опираясь на уже имеющуюся локализацию, выполнить ее на этот раз будет намного проще. Итак, выберите сначала данный проект в окне навигатора проекта, а затем — в окне редактора и вкладке Info. Щелкните на кнопке со знаком + в разделе Localizations, чтобы добавить новый язык. Во всплывающем меню вы не обнаружите пункт Swiss French, поэтому выберите пункт Other в самом конце этого меню. В итоге откроется подменю с довольно длинным списком языков; правда, они перечислены в алфавитном порядке. Выберите в этом списке пункт French (Switzerland), а затем пункт French в списке Reference Language для всех файлов, перечисленных в открывшемся диалоговом окне, которое выглядит так, как показано на рис. 22.6. После этого щелкните на кнопке Finish. В навигаторе проекта вы теперь обнаружите версии файлов для локализации раскадровки и символьных строк на французском языке для пользователей из Швейцарии, а также файлы InfoPlist.strings. Для того чтобы убедиться в отличиях этой локализации от той, которая сделана на французском языке для пользователей из Франции, откройте версию файла InfoPlist.strings для пользователей из Швейцарии и замените имя пакета следующим:

```
"CFBundleName" = "Swiss Localisez Moi";
```

Постройте и запустите данное приложение на выполнение. Перейдите к приложению Settings, а в нем — к экрану Language and Region. Как пояснялось ранее, в списке языков, поддерживаемых мобильным устройством, отсутствует швейцарский диалект французского. Поэтому щелкните на ссылке Other Languages и выполните прокрутку вниз (или поиск) до тех пор, пока не найдете вариант French (Switzerland). Выберите его и нажмите кнопку Done. В итоге появится лист действий, на котором вам будет предложено сделать выбор между французским языком для пользователей из Швейцарии и текущим языком. Выберите вариант Swiss French и предоставьте системе iOS возможность перезагрузиться. Перейдите на главный экран, и обнаружите, что данное приложение теперь называется Swiss Localisez Moi (на самом деле вы не увидите его название полностью, поскольку оно слишком длинное, но вы, конечно, догадаетесь). К сожалению, на экране отображается французский, а не швейцарский флаг. Однако теперь вы знаете, как исправить этот недостаток и отредактировать файлы швейцарской локализации. В качестве упражнения попробуйте загрузить изображение швейцарского флага из Интернета и внедрить его в свое приложение.

Резюме

Если вы хотите максимально расширить рынок сбыта своих приложений для системы iOS, локализуйте их как можно больше. Правда, архитектура локализации в iOS значительно упрощает труд по поддержке многих языков и даже нескольких диалектов одного и того же языка в приложении. Как было показано в этой главе, практически любой тип файла, добавляемого в приложение, может быть локализован, если в этом есть необходимость.

Даже если вы не собираетесь локализовать свое приложение, возьмите себе за правило пользоваться в своем коде функцией `NSLocalizedString()` вместо статических символьных строк. Благодаря средству Code Sense для распознавания кода, вводимого в Xcode, время набора вызовов функции `NSLocalizedString()` увеличивается едва заметно по сравнению с символьными строками, но впоследствии это намного облегчает труд, когда возникает потребность в локализации приложения. А возвращаться впоследствии к проекту, чтобы найти в нем все текстовые строки, которые следует локализовать, неудобно и чревато ошибками, которых можно избежать, заранее приложив немного усилий.

Резюме книги

Язык программирования и каркасы, с которыми вы ознакомились в этой книге, стали результатом более чем 25-летней эволюции. Разработчики из компании Apple продолжают неустанно трудиться над следующими замечательными технологическими новшествами на платформе iOS. Она находится лишь в начале своего расцвета, который обещает быть бурным. Проработав материал этой книги, вы заложили прочное основание и получили солидные знания о языке Swift, каркасе Cocoa Touch и тех инструментальных средствах, которые объединяют эти технологии для создания замечательных новых приложений, предназначенных для мобильных устройств iPhone, iPod touch и iPad. Теперь вы знаете программную архитектуру iOS — шаблоны проектирования, приводящие механизм Cocoa Touch в действие. Короче говоря, теперь вы, читатель, готовы пойти своим путем, чем мы, авторы, можем только гордиться. Удачи!

ПРИЛОЖЕНИЕ



Введение в язык Swift

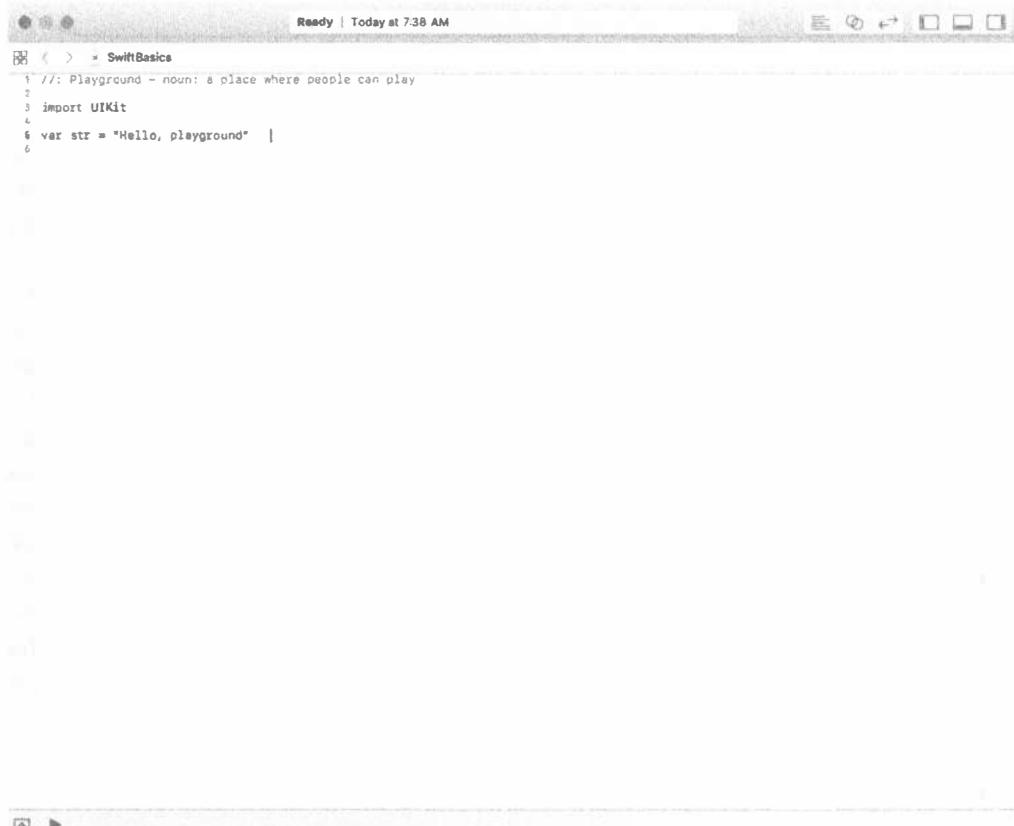
Еще совсем недавно разработка приложения для мобильного устройства iPhone или iPad означала программирование на языке Objective-C. В силу необычного синтаксиса отношение к языку Objective-C сложилось полярное: его можно или любить, или ненавидеть. Это положение изменилось в 2014 году на Всемирной конференции разработчиков, на которой компания представила новый альтернативный язык под названием “Swift”. Синтаксис этого языка сделан более привычным для тех, кто привык программировать на таких распространенных объектно-ориентированных языках, как C++ и Java, что упрощает задачу разработки приложений под управлением iOS (и для компьютеров Mac, на которых язык Swift полностью поддерживается для этих целей в системе macOS). В этом приложении рассматриваются те части языка Swift, которые нужно знать, чтобы понимать примеры кода, представленные в главах данной книги. При этом предполагается, что у читателей имеется некоторый опыт программирования и они знают, что такое переменные, функции, методы и классы. Материал этого приложения не может служить ни справочным, ни тем более исчерпывающим руководством по языку Swift. Для этой цели имеется немало других источников и ресурсов, и некоторые из них перечислены в главе 1.

Основы языка Swift

К числу самых примечательных функциональных возможностей среды Xcode 6, помимо Swift, относится **игровая площадка** (playground). Как подразумевает название этого средства, оно служит местом, где можно экспериментировать с прикладным кодом, не создавая среду, в которой он должен выполняться. Для этого достаточно открыть игровую площадку, ввести немного кода и просмотреть полученные результаты. Игровые площадки служат отличным местом для изучения нового языка программирования, и поэтому они будут использоваться в этом приложении.

Начнем с создания новой игровой площадки. С этой целью запустите Xcode на выполнение и выполните команду меню **File⇒New⇒Playground...**. В открывшемся

диалоговом окне присвойте игровой площадке имя (например, SwiftBasics), непременно выберите вариант iOS из раскрывающегося списка Platform и нажмите кнопку Next. Затем выберите папку, в которой будет храниться игровая площадка, и нажмите кнопку Create. В среде Xcode будет создана игровая площадка, которая откроется в новом окне, как показано на рис. 1. Прорабатывая примеры из этого приложения, можете экспериментировать с ними, вводя свой код на игровой площадке или видоизменяя их код, чтобы посмотреть, что из этого получится.



The screenshot shows the Xcode interface with a playground window titled "SwiftBasics". The code area contains the following Swift code:

```
1 //: Playground - noun: a place where people can play
2
3 import UIKit
4
5 var str = "Hello, playground" |
```

Рис. 1. Вновь созданная игровая площадка

Игровые площадки, комментарии, переменные и константы

Бросим беглый взгляд на созданную нами игровую площадку, чтобы посмотреть, что же на ней есть. Как видите, она разделена на две области: кода слева и результатов справа. По мере ввода исходного кода компилятор языка Swift компилирует и выполняет его, практически сразу отображая результаты. Так, в коде, приведенном на рис. 1, объявляется новая переменная str, которая инициализируется символьной строкой "Hello, playground". Эту строку можно

видеть в столбце результатов справа. Попробуйте изменить значение данной переменной, и вы сразу же заметите, что результат обновится, как только вы прекратите вводить код.

Код в строке кода 1 на рис. 1 служит комментарием. Все, что следует после знаков // до конца строки, игнорируется компилятором. В данном случае комментарий занимает всю строку кода, хотя это и не обязательно. Ввести комментарий можно и в конце строки кода, как показано ниже.

```
var str = "Hello, world" // Комментарий
```

Для того чтобы ввести комментарий длиннее одной строки, начните его со знаков /*, а завершите знаками */.

```
/*
Этот комментарий занимает
больше одной строки.
*/
```

Ввести комментарий можно и по-другому. Одни предпочитают явно указать, что строка кода является частью комментария, начав ее со знака *, как показано ниже.

```
/*
* Этот комментарий занимает
* больше одной строки.
*/
```

Другим нравится вводить комментарии следующим образом:

```
/* Этот еще один способ написания одностороннего комментария. */
```

Оператор import в строке кода 3 делает доступным каркас UIKit от компании Apple на игровой площадке, как показано ниже.

```
import UIKit
```

В системе iOS имеется немало каркасов, и некоторые из них рассматриваются в отдельных главах данной книги. В частности, UIKit служит каркасом для пользовательского интерфейса и применяется во всех примерах кода из данной книги. Не менее часто применяется каркас Foundation, который содержит классы, предоставляющие основные функциональные возможности, в том числе для обработки даты и времени, ведения коллекций, управления файлами, работы в сети и многое другое. Для доступа к этому каркасу нужно импортировать его следующим образом:

```
import Foundation
```

Тем не менее Foundation автоматически импортируется из UIKit, и поэтому на любой игровой площадке, где импортируется каркас UIKit, автоматически становится доступным и каркас Foundation. Следовательно, импортировать его явным образом необязательно.

818 ПРИЛОЖЕНИЕ * ВВЕДЕНИЕ В ЯЗЫК SWIFT

Приведенная ниже строка 5 является первой (и единственной) строкой выполняемого кода на данной игровой площадке.

```
var str = "Hello, playground"
```

Ключевое слово `var` служит для объявления новой переменной с заданным именем. В данном случае переменная объявляется под именем `str`, которое вполне подходит для строковой переменной. Язык Swift весьма либерален к именам переменных: в их именах можно использовать практически любой символ, за исключением первого символа, на который накладываются определенные ограничения. Подробнее о правилах именования переменных можно узнать в документации компании Apple на Swift по адресу https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language.

После объявления переменной следует выражение, в котором ей присваивается начальное значение. При объявлении переменной инициализировать ее совсем не обязательно, но это следует непременно сделать перед ее употреблением (т.е. перед выполнением любого кода, в котором читается ее значение). Но если инициализировать переменную, ее тип будет выведен автоматически, избавляя от необходимости указывать его явно. В данном случае тип переменной `str` выводится как строковый, поскольку она инициализирована строковым литералом. Если же не инициализировать переменную (возможно, для того чтобы не фиксировать ее начальное значение), то, объявляя переменную, обязательно следует указать тип после ее имени, отделив его двоеточием, как показано ниже.

```
var str2: String // Неинициализированная переменная
```

Попробуйте изменить код в строке 4 игровой площадки на следующий:

```
var str: String  
str = "Hello, playground"
```

Этот фрагмент кода совершенно равнозначен первоначальному коду, но теперь в нем пришлось явно указать, что переменная `str` является строковой (тип `String` представляет в языке Swift символьные строки). Как правило, объявление переменной удобнее объединить с ее инициализацией, предоставив компилятору языка Swift автоматически вывести тип переменной. Такая возможность используется в следующем примере кода:

```
var count = 2
```

В данном примере компилятор выводит целочисленный тип переменной `count`, а по существу — тип `Int` (подробнее о числовых типах, предоставляемых в языке Swift, речь пойдет в следующем разделе). А как убедиться в том, что тип переменной выводится правильно? Очень просто. Предоставьте компилятору языка Swift самому сообщить об этом! С этой целью введите приведенную выше строку кода на игровой площадке, наведите указатель мыши на имя переменной `count` и нажмите клавишу `<Option>`. Указатель примет вид знака

вопроса. Щелкните кнопкой мыши, и компилятор языка Swift покажет вам выведенный тип данной переменной во всплывающей подсказке (рис. 2).

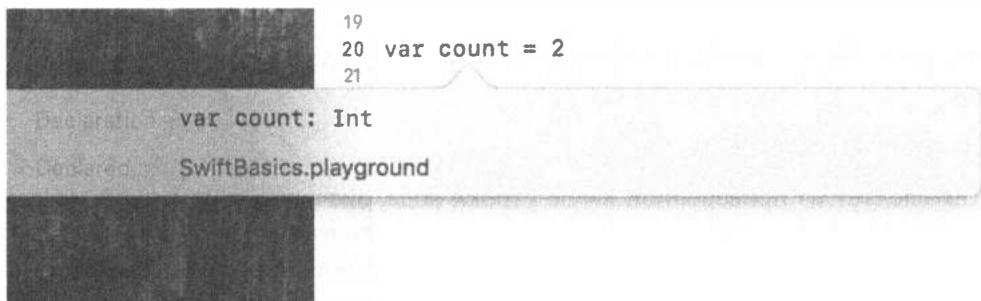


Рис. 2. Получение выведенного типа переменной

Однако, если не указать тип переменной в ее объявлении, это совсем не означает, что у нее нет никакого типа или что с ней можно обращаться легко-мысленно. Тип назначается переменной при ее объявлении, и после этого его нужно строго придерживаться. В отличие от таких динамических языков программирования, как JavaScript, в языке Swift нельзя изменить тип переменной одним только присваиванием ей нового значения. Попробуйте, например, ввести следующий фрагмент кода:

```
var count = 2
count = "Two"
```

Попытка присвоить строковое значение целочисленной переменной приведет к ошибке. При этом на полях левой области игровой площадки появится красная метка. Щелкните на ней, и на экран будет выведено сообщение, поясняющее причину ошибки (рис. 3).



Рис. 3. Язык Swift не является динамическим. В нем нельзя изменить тип переменной

Вы, возможно, обратили внимание на то, что в конце операторов вообще отсутствуют точки с запятой. Это одна из примечательных особенностей языка Swift, которая состоит в том, что указывать точку с запятой в конце оператора требуется крайне редко. Если у вас имеется опыт программирования на C, C++, Objective-C или Java, такая особенность Swift может поначалу показаться вам не совсем обычной, но со временем вы к ней привыкнете. Разумеется, указывать точку с запятой в конце оператора *можно*, если есть на то желание, но чаще всего делать это не нужно. Точку с запятой *следует* указывать лишь в

том случае, если в одной строке требуется написать два оператора подряд. Так, следующая строка кода неверная:

```
var count = 2 count = 3
```

Ведите точку с запятой, как показано ниже, чтобы удовлетворить компилятор.

```
var count = 2; count = 3
```

Как следует из приведенной выше строки кода, значение переменной можно изменить. Именно поэтому она и называется **переменной**. Но что если требуется присвоить фиксированное значение, иными словами, создать константу и задать ее значение? Для этой цели в языке Swift предоставляется оператор `let`. Синтаксически оператор `let` действует аналогично оператору `var`, за исключением того, что в нем нужно указать начальное значение, как показано ниже.

```
let pi = 3.14159265
```

Тип константы, как и переменной, можно вывести (или явно указать при объявлении). Но повторно присваивать значение константе нельзя. Ведь именно для этого она и предназначена. Так, следующая строка кода неверная:

```
pi = 42 // ОШИБКА — присвоить повторно значение константе pi нельзя
```

Естественно, что значение переменной можно инициализировать из константы, как показано ниже.

```
let pi = 3.14159265
var value = pi
```

Как было показано ранее, на игровой площадке результаты выполнения операторов в языке Swift выводятся справа от соответствующих строк кода. Однако их можно вывести и на консоль с помощью функции `print()` из стандартной библиотеки Swift. Попробуйте, например, ввести следующую строку кода:

```
print("Hello, world")
```

Символьная строка "Hello, world", за которой следует символ перехода на новую строку, появится, как обычно, в области результатов на игровой площадке, а также будет выведена на консоль. Для просмотра вывода на консоль наведите указатель мыши на область результатов. На экране появится пара круглых элементов управления. Щелкните на правом элементе управления — и результаты появятся под инструкцией `print()` (рис. 4). Обратите внимание на нижнюю часть области отладки, в которой также выводятся результаты. Для того чтобы открыть или закрыть область отладки, в которой будет выводится результат, используется треугольник раскрытия, расположенный в левом нижнем углу игровой площадки.

```

24 25 26
26 print("Hello, world")

```

Hello, world

26

Hello, world

Рис. 4. Просмотр результатов вывода на консоль игровой площадки

Если вы не хотите, чтобы к строке автоматически добавлялся символ перехода на новую строку, удалите его или замените другой строкой, используя другой вариант функции `print()`, получающей дополнительный аргумент.

```
print("Hello, world", terminator: "")
```

Этот код заменяет символ перехода на новую строку пустой строкой. Если проверить область результатов, то можно увидеть, что переход на новую строку больше не происходит.

ПОДСКАЗКА. Функция `print()` входит в стандартную библиотеку языка Swift. Документацию к этой библиотеке можно найти по адресу <https://developer.apple.com/library/ios/documentation/General/reference/SwiftStandardLibraryReference>. С другой стороны, посмотреть содержимое стандартной библиотеки можно, введя строку кода `import Swift` на игровой площадке, а затем нажав клавишу `<Control>` и щелкнув на слове `Swift`. Игровая площадка переключится на вывод содержимого стандартной библиотеки. Вы получите немало полезной информации, из которой можно узнать о языке Swift и лучше понять его, но это стоит делать, чтобы понять, что именно можно найти в документации.

Предопределенные типы, операции и управляемые операторы

В языке Swift предопределен ряд основных типов. В последующих разделах будет показано, что эти типы можно вводить в определения собственных классов, структур и перечислений. Существующий тип можно даже наделить функциональными возможностями, создав его расширение. В языке Swift предусмотрены также операции и управляемые операторы, которые, без сомнения, известны вам из других языков программирования. Сделаем, прежде всего, краткий обзор основных типов данных в языке Swift.

Числовые типы

В языке Swift имеются четыре основных числовых типа — `Int`, `UInt`, `Float` и `Double` — и целый ряд более специализированных целочисленных типов. Все целочисленные типы перечислены в табл. 1 вместе с их разрядностью (в битах) и пределами допустимых значений.

Таблица 1. Целочисленные типы в языке Swift

Тип	Разрядность (в битах)	Максимальное значение	Минимальное значение
Int	32 или 64	Как у типа Int32 или Int64	Как у типа Int32 или Int64
UInt	32 или 64	Как у типа UInt32 или UInt64	0
Int64	64	9,223,372,036,854,775,807	-9,223,372,036,854,775,808
UInt64	64	18,446,744,073,709,551,615	0
Int32	32	2,147,483,647	-2,147,483,648
UInt32	32	4,294,967,295	0
Int16	16	32767	-32768
UInt16	16	65535	0
Int8	8	127	-128
UInt8	8	255	0

Тип `Int` и его производные обозначают целочисленные значения со знаком, тогда как тип `UInt` и его производные — целочисленные значения без знака. По умолчанию для целочисленных значений выбирается тип `Int` (т.е. он выводится при написании выражения наподобие `var count = 3`). Этот тип рекомендуется применять в первую очередь, если только нет особых оснований использовать другие целочисленные типы данных.

Как следует из табл. 1, пределы допустимых значений, которые могут быть представлены типами `Int` и `UInt`, зависят от конкретной платформы. В 32-разрядных системах (например, в некоторых моделях iPad и всех моделях iPhone, выпущенных после iPhone 4s и iPhone 5c) эти типы обозначают 32-разрядные целочисленные значения, тогда как в 64-разрядных системах — 64-разрядные целочисленные значения. Если требуются целочисленные значения, которые, определенно, должны быть 32- или 64-разрядными, то следует использовать типы `Int32` и `Int64`. Типы `Int8` и `UInt8` служат для обозначения байтов.

Максимально и минимально допустимые значения для каждого из типов данных можно определить программно, используя их свойства `max` и `min` соответственно. Например, попробуйте ввести следующие строки кода на игровой площадке (без комментариев, показывающих результаты):

```
print(Int8.max)    // 127
print(Int8.min)    // -128
print(Int32.max)   // 2,147,483,647
print(Int32.min)   // -2,147,483,648
print(UInt32.max)  // 4,294,967,295
```

Целочисленные литералы могут быть указаны в десятичной, шестнадцатеричной, двоичной или восьмеричной системе счисления. В качестве примера введите следующие строки кода:

```
let decimal = 123      // Значение равно 123
let octal = 0o77       // Восьмеричное значение 77 равно десятичному 63
let hex = 0x1234        // Шестнадцатеричное 1234 равно десятичному 4660
let binary = 0b1010     // Двоичное значение 1010 равно десятичному 10
```

Префикс 0o обозначает восьмеричные значения, префикс 0x — шестнадцатеричные, а префикс, а 0b — двоичные. Для большей удобочитаемости исходного кода группы чисел можно также разделять знаком подчеркивания (_), как показано ниже.

```
let v = -1_234      // То же, что и -1234
let w = 12_34_56    // То же, что и 123456
```

Типы `Float` и `Double` обозначают 32- и 64-разрядные числовые значения с плавающей точкой соответственно. Числовое значение с плавающей точкой присваивается переменной с помощью литерала с плавающей точкой. В языке Swift выводится тип `Double`, если не указано иное, следующим образом:

```
let a = 1.23          // Тип этого значения выводится как Double
let b: Float = 1.23   // Тип этого значения принудительно
                      // обозначается как Float
```

Крупные числа удобно обозначать в экспоненциальном (или научном) представлении, как показано ниже.

```
let c = 1.23e2        // Вычисляется как 123.0
let d = 1.23e-1       // Вычисляется как 0.123
let e = 1.23E-1       // То же, что и 1.23e-1
```

По своему характеру числа с плавающей точкой не совершенно точные. Это, в частности, объясняется тем, что дробные значения не могут быть точно представлены в двоичной форме с плавающей точкой. В этом можно убедиться, если ввести следующие строки кода на игровой площадке (как и прежде, комментарии к ним обозначают результаты):

```
let f:Float = 0.123456789123    // 0.1234568
let g:Double = 0.123456789123   // 0.123456789123
```

Представление приведенного выше числа в формате `Float` оказывается менее точным, чем в формате `Double`. Если же сделать дробную часть числа длиннее, то можно превысить точность представления чисел и в формате `Double`, как показано ниже.

```
let g:Double = 0.12345678912345678 // 0.1234567891234568
```

Числа с плавающей точкой теряют свою точность, когда их значения становятся большими, как в следующих примерах:

824 ПРИЛОЖЕНИЕ & ВВЕДЕНИЕ В ЯЗЫК SWIFT

```
let f: Float = 123456789123456      // Неточно: 1.234568e+14
let g: Double = 123456789123456     // Точно: 123,456,789,123,456.0
let h: Double = 1234567891234567    // Неточно: 1.234567891234568e+17
```

В отличие от других языков программирования, в языке Swift не предусмотрено неявное преобразование типов, когда переменной (или выражению) одного числового типа присваивается значение переменной другого числового типа. Так, следующий фрагмент кода не подлежит компиляции (рис. 5):

```
let a = 123
let b = 0.456
let c = a + b
```



```
48 let a = 123
49 let b = 0.456
50 let c = a + b
51
52
53
```

Рис. 5. В языке Swift не допускается присваивание разнотипных переменных

В приведенном выше фрагменте кода переменная `a` относится к типу `Int`, а переменная `b` — к типу `Double`. И хотя числовое значение типа `Int` можно было бы преобразовать в тип `Double`, в языке Swift это не предусмотрено. Такое преобразование приходится выполнять вручную следующим образом:

```
let a = 123
let b = 0.456
let c = Double(a) + b
```

В выражении `Double(a)` вызывается инициализатор типа `Double` с целочисленным аргументом. Для подобного рода преобразований все числовые типы предоставляют свои инициализаторы.

Еще одним примером может служить часто употребляемый тип `CGFloat` чисел с плавающей точкой, определенный в каркасе Core Graphics. Он служит для обозначения, среди прочего, координат и размеров графических объектов. В зависимости от того, где именно выполняется приложение (на 32- или 64-разрядной платформе), этот тип равнозначен типу `Float` или `Double`. Для того чтобы выполнить операции над данными типа `CGFloat` и других типов, придется явно преобразовать один тип в другой. Например, в следующем фрагменте кода складываются числовые значения типа `Double` и `CGFloat`, а в результате преобразования типа `Double` в тип `CGFloat` получается суммарное значение типа `CGFloat`:

```
let a: CGFloat = 123
let b: Double = 456
let c = a + CGFloat(b) // Результат типа CGFloat
```

На 32-разрядной платформе тип `CGFloat` оказывается менее точным, чем тип `Double`, и поэтому в приведенной выше операции сложения может произойти потеря точности, а значит, информации. Но это неизбежно, если требуется

получить результат типа `CGFloat`. Если же нужно получить результат типа `Double`, то тип `CGFloat` можно преобразовать в тип `Double` без потери точности, как показано ниже.

```
let a: CGFloat = 123
let b: Double = 456
let c = Double(a) + b // Результат типа Double
```

Следует заметить, что в языке Swift все же допускается сочетание разных числовых типов, если все значения считаются литералами, как в следующем примере:

```
1 + 0.5 // Вычисляется как 1.5
```

В языке Swift поддерживаются все обычные операции двоичной арифметики. В этих операциях можно сочетать числа одного и того же типа или же разных типов, при условии явного преобразования одного из операндов, как было показано в предыдущих примерах кода. Доступные в языке Swift арифметические операции перечислены в табл. 2 в порядке их предшествования.

Таблица 2. Операции двоичной арифметики, предопределенные в языке Swift

Операция	Назначение
<code><<</code>	Поразрядный сдвиг влево
<code>>></code>	Поразрядный сдвиг вправо
<code>*, &*</code>	Умножение
<code>/, &/</code>	Деление
<code>%, &%</code>	Получение остатка от деления
<code>&</code>	Поразрядная логическая операция И
<code>+, &+</code>	Сложение
<code>-, &-</code>	Вычитание
<code>:</code>	Поразрядная логическая операция ИЛИ
<code>^</code>	Поразрядная логическая операция "исключающее ИЛИ"

В арифметических операциях `+`, `-`, `*`, `/` и `%` автоматически обнаруживается переполнение. Если же требуется пренебречь переполнением, следует воспользоваться операциями `&+`, `&-`, `&*`, `&/` или `&%`. В качестве примера введите следующие строки кода на игровой площадке:

```
let a = Int.max
let b = 1
let c = Int.max + b
```

В данном примере значение 1 складывается с максимально допустимым целочисленным значением, что должно привести к переполнению. Поэтому результат этого сложения, присваиваемый переменной `c` в последней строке кода,

826 ПРИЛОЖЕНИЕ * ВВЕДЕНИЕ В ЯЗЫК SWIFT

не выводится на экран. Для того чтобы продолжить выполнение, пренебрегая переполнением, следует воспользоваться операцией &+, как показано ниже.

```
let a = Int.max  
let b = 1  
let c = a &+ b
```

Операции << и >> выполняют сдвиг левого операнда влево или вправо соответственно на величину, обозначаемую в их правом операнде. Это равнозначно умножению или делению на величину в степени 2, как показано в следующем примере кода:

```
let a = 4  
let b = a << 2    // Результат равен 16  
let c = b >> 1    // Результат равен 8
```

Если левый операнд отрицательный, знак результата сохраняется, как показано ниже.

```
let a = -4  
let b = a << 2    // Результат равен -16  
let c = b >> 1    // Результат равен -8
```

Операции &, | и ^ выполняют над своими operandами поразрядные логические операции И, ИЛИ и исключающее ИЛИ соответственно. Их не следует путать с логическими операциями && и ||, дающими в итоге логическое значение, как поясняется далее, в разделе “Логические значения”. Ниже приведены некоторые примеры выполнения поразрядных логических операций.

```
let a = 7      // Значение равно 0b111  
let b = 3      // Значение равно 0b011  
let c = a & b  // Результат равен 0b011 = 3  
  
let a = 7      // Значение равно 0b111  
let b = 3      // Значение равно 0b011  
let c = a | b  // Результат равен 0b111 = 7  
  
let a = 7      // Значение равно 0b111  
let b = 3      // Значение равно 0b011  
let c = a ^ b  // Результат равен 0b100 = 4
```

Имеются также составные варианты арифметических и логических операций, которые сочетаются с последующим присваиванием. В этом случае адресат присваивания действует в качестве левого операнда составной операции. Ниже приведены некоторые примеры выполнения составных арифметических и логических операций.

```
var a = 10  
a += 20      // Сокращенный вариант операции a = a + 20, результат = 30  
var b = 7  
b &= 3      // Сокращенный вариант операции b = b & 3, результат = 3
```

ЗАМЕЧАНИЕ. В языке Swift 3 унарные операторы `++` и `--` больше недоступны. Для их замены необходимо использовать синтаксические конструкции наподобие `"a++"` и `"a+=1"`. Это также значит, что следует внимательно следить за порядком следования этих операторов (например `++a` и `a++` и т.д.).

Операция `~` выполняет поразрядное инвертирование своего целочисленного операнда, как показано ниже.

```
let a = 0b1001
let b = ~a
```

В результате выполнения этой операции получается 32-разрядное двоичное значение `0b1111111111111111111111110110`, равнозначное десятичному значению `-10`.

Строки

В языке Swift символьные строки представлены типом `String` в виде последовательности символов в уникоде. Благодаря этому можно разрабатывать приложения, поддерживающие самые разные наборы символов, не прибегая к написанию специального кода. В то же время это доставляет некоторые сложности, отчасти упоминаемые здесь. Исчерпывающее обсуждение последствий применения уникода приведено в руководстве *The Swift Programming Language*, доступном на сайте компании Apple для разработчиков по адресу, упоминавшемуся в начале этого приложения, или же на сайте iBooks.

Строковый литерал представляет собой последовательность символов, заключаемую в двойные кавычки, как в следующей строке кода, которая уже приводилась раньше:

```
let str = "Hello, playground"
```

Если в символьную строку требуется включить знак `"`, его следует предварить (т.е. экранировать) знаком `\`, как показано ниже.

```
let quotes = "Contains \"quotes\""  
// Результат равен Contains "quotes"
```

Для того чтобы включить в символьную строку знак `\`, его следует экранировать еще одним знаком `\`, как показано ниже.

```
let backslash = "\\\" // Результат равен \\"
```

Любой символ в уникоде можно вставить в символьную строку, задав его шестнадцатеричное значение (или так называемую *кодовую точку*) в фигурных скобках выражения `\u{}`. Например, у знака `@` имеется кодовая точка `0x40`, и поэтому в следующей строке кода показаны два способа обозначения этого значения в символьной строке на языке Swift:

```
let atSigns = "@\u{40}"  
// Результат равен @"
```

Некоторые символы имеют специальное экранированное представление в виде управляющих последовательностей. Например, управляющие последовательности `\n` и `\t` обозначают символы новой строки и табуляции соответственно. Ниже показано, каким образом они вводятся в символьную строку.

```
let specialChars = "Line1\nLine2\tTabbed"
```

Символьные строки обладают полезной способностью интерполировать выражения, заключенные в управляющую последовательность `\()`, как показано в следующем примере кода:

```
print("The value of pi is \(\M_PI)") // Выводит символьную строку
                                         // "The value of pi is 3.14159265358979\n"
```

В приведенной выше строке кода интерполируется значение предопределенной константы `M_PI`, которое вставляется в символьную строку, а полученный результат выводится на экран. В качестве интерполируемого значения можно также указывать выражение, причем не одно, как показано ниже.

```
// В этом фрагменте кода выводится символьная строка:
// "Area of a circle of radius 3.0 is 28.2743338823081\n"
let r = 3.0
print("Area of a circle of radius \(\r) is \(\M_PI * \r * \r)")
```

С помощью операции `+` символьные строки можно сцеплять. Это единственный способ соединить символьные строки, чтобы разбить длинный текст на несколько строк кода в исходном файле, как показано ниже.

```
let s = "That's one small step for man, " +
        "one giant leap for mankind"
print(s) // "That's one small step for man, one giant leap for mankind"
```

В операциях `==` и `!=` можно сравнивать две символьные строки. Эти операции, по существу, выполняют посимвольное сравнение своих операндов, как поясняется далее, в разделе “Логические значения”. Ниже приведены некоторые примеры сравнения символьных строк.

```
let s1 = "String one"
let s2 = "String two"
let s3 = "String " + "one"
s1 == s2 // False: строки не одинаковые
s1 != s2 // True: строки разные
s1 == s3 // True: строки состоят из одинаковых символов
```

Простая, на первый взгляд, задача получения длины символьной строки усложняется тем, что такие строки состоят из символов в unicode. Ведь не все символы представлены в unicode одной кодовой точкой. Не вдаваясь здесь в подробности, отметим лишь некоторые особенности решения данной задачи, которые следует иметь в виду. Прежде всего, для получения точного количества символов в строке следует использовать свойство символов `count`.

```
s3.characters.count // 10
```

Если же заранее известно, что строка содержит только символы, представленные одной кодовой точкой в unicode, то можно воспользоваться свойством `utf16Count`, чтобы ускорить данную операцию, как показано ниже.

```
s3.utf16.count // 10
```

В самом типе `String` предусмотрено несколько удобных строковых операций, описание которых можно найти в документации. Зачастую для обработки символьных строк проще всего воспользоваться тем обстоятельством, что тип `String` в языке Swift стыкуется с классом `NSString` из каркаса Foundation. Это означает, что методы, определенные в классе `NSString`, можно вызывать так, как будто они определены в самом типе `String`.

Для хранения отдельных символов, извлекаемых из строки, можно воспользоваться типом `Character`. Это дает возможность перебрать отдельные символы строки в цикле следующим образом:

```
let s = "Hello"
for c in s.characters {
    print(c)
}
```

Тело цикла в приведенном выше примере кода выполняется для вывода каждого символа из строки, причем символ присваивается переменной цикла `c`, тип которой выводится как `Character` (подробнее синтаксис циклов обсуждается далее в этом приложении). Результат выполнения данного цикла нельзя увидеть в правом столбце игровой площадки, где отображается лишь тот факт, что цикл был выполнен пять раз подряд. Вместо этого необходимо щелкнуть на левом элементе управления (его символ напоминает глаз), удерживая указатель мыши над столбцом результатов. В итоге откроется всплывающее меню, в котором необходимо щелкнуть правой кнопкой и выбрать команду `Value History`. Благодаря этому вы увидите все символы, как показано на рис. 6. Впрочем, на момент написания книги щелчок на любых полосах прокрутки не приводил к ожидаемому результату и нам приходилось использовать клавиши навигации.

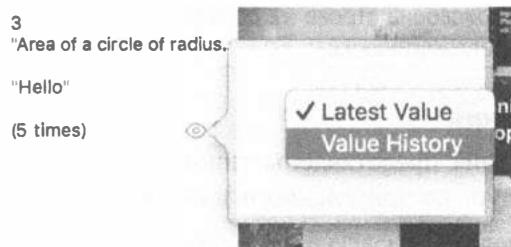


Рис. 6. Всплывающее меню `Value History`

ЗАМЕЧАНИЕ. Если вы не видите результатов, наведите указатель мыши на правый край столбца результатов и удерживайте его, пока не откроется панель для изменения ширины столбца, а затем увеличьте ширину столбца, чтобы увидеть результаты.

Переменную или константу типа `Character` можно создать и инициализировать таким же образом, как и аналогичные элементы кода типа `String`. Но при этом нужно явно указать тип, чтобы избежать автоматического выведения типа `String`, а инициализатор должен состоять только из одного символа, как показано ниже.

```
let c: Character = "s"
```

С символьными значениями типа `Character` мало что можно сделать, кроме сравнения с другими значениями типа `Character`. В частности, их нельзя соединять вместе или присоединять к объектам типа `String`. Для этого придется вызвать метод `append()` для объекта типа `String`, как показано ниже.

```
let c: Character = "s"
var s = "Book"           // Эта переменная объявляется как var,
                        // поскольку она будет видоизменена
s += c                  // ОШИБКА – недопустимо!
s.append(c)             // "Books"
```

Символьные строки в языке `Swift` не являются неизменяемыми, но являются **объектами-значениями**. Это означает, что всякий раз, когда символьная строка присваивается переменной, указывается в качестве аргумента функции илиозвращается из нее, она копируется. Изменения, вносимые в копию символьной строки, не оказывают никакого влияния на ее оригинал, как показано ниже.

```
var s1 = "Book"
var s2 = s1           // Страна s2 теперь содержит копию строки s1
s2 += "s"            // Символ "s" присоединяется к строке s2,
                    // а строка s1 не изменяется
s1                   // "Book"
s2                   // "Books"
```

ЗАМЕЧАНИЕ. Для повышения эффективности реальная копия содержания строки не создается сразу после присваивания. В предыдущем примере строки `s1` и `s2` после присваивания `s1=s2` будут содержать одни и те же копии символов, пока не будет выполнена инструкция `s2 += s`. Только тогда будет создана копия общего содержания, которая будет присвоена строке `s2` до того, как к ней будет добавлена строка `s`. Все это происходит автоматически.

Логические значения

Логические значения представлены типом `Bool` и могут быть равны `true` или `false`, как следует из приведенного ниже примера кода.

```
var b = true // Выводится значение типа Bool
var b1: Bool
b1 = false
```

В языке `Swift` предусмотрены обычные операции сравнения (`==`, `!=`, `>`, `<`, `>=`, `<=`), возвращающие логическое значение, когда они выполняются над числовыми значениями. Ниже приведены некоторые примеры выполнения таких операций.

```

var a = 100
var b = 200
var c = a

a == c      // true
a == b      // false
a != b      // true
a > b      // false
a < b      // true
a >= c     // true

```

Эти операции можно выполнять и над символьными строками, как показано ниже.

```

let a = "AB"
let b = "C"
let c = "CA"
let d = "AB"
a == b      // false: строки имеют разное содержимое
a == d      // true: строки имеют одинаковое содержимое
a != c      // true: строки имеют разное содержимое
a > b      // false: используется порядок сортировки
a < c      // true: обе строки начинаются с символа C,
            // но строка с длиннее строки a

```

Логическое значение можно обратить с помощью унарного оператора ! следующим образом:

```

var a = 100
var b = 200
a == b      // false
!(a == b)    // !false == true

```

Логические выражения можно объединять с помощью операторов ==, !=, && и |||. Операторы && и ||| являются укороченными. Это означает, что их второй operand вычисляется лишь в том случае, если результат оператора нельзя определить по ее первому operandу. В частности, вычислять второй operand в логическом операторе ||| не требуется, если первый его operand принимает логическое значение true, а в операторе && — логическое значение false. Ниже приведены некоторые примеры выполнения этих операторов.

```

var a = 100
var b = 200
var c = 300
'
a < b && c > b      // true: вычисляются оба выражения
a < b ||| c > b     // true: второе выражение не вычисляется
a > b && c > b     // false: второе выражение не вычисляется

```

Перечисления

Перечисления позволяют присваивать содержательные имена значениям разных типов, если эти значения фиксированы и заранее известны. Для того чтобы

определить перечисление, ему следует присвоить имя и указать перечень всех возможных его значений в виде вариантов выбора case, как показано ниже.

```
enum DaysOfWeek {
    case Sunday, Monday, Tuesday, Wednesday,
        Thursday, Friday, Saturday
}
```

Каждый вариант выбора в перечислении можно также определить отдельно следующим образом:

```
enum DaysOfWeek {
    case Sunday
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
    case Saturday
}
```

Для того чтобы обратиться к перечислимому значению, следует указать имена перечисления и варианта выбора, разделив их точкой, как показано ниже.

```
var day = DaysOfWeek.Sunday // Тип переменной day выводится
                            // как перечислимый тип DaysOfWeek
```

Имя перечисления можно опустить, если оно очевидно из контекста. В приведенной ниже строке кода компилятору уже известно, что переменная day относится к перечислимому типу DaysOfWeek. Поэтому не нужно указывать явно имя перечисления, когда его значение присваивается данной переменной.

```
day = .Friday // Указывать точку "." обязательно!
```

В языке Swift можно также присвоить ассоциативное значение варианту выбора из перечисления. Попробуйте, например, ввести на игровой площадке следующий фрагмент кода:

```
enum Status {
    case OK
    case ERROR(String)
}

let status = Status.OK
let failed = Status.ERROR("That does not compute")
```

В данном фрагменте кода вариант выбора ERROR имеет ассоциативное значение, описывающее ошибку. Восстановить ассоциативное значение можно в операторе выбора switch, как будет показано далее, в разделе "Управляющие операторы".

Иногда оказывается удобно назначить само имя варианта выбора для значения, называемого **исходным**. Для этого следует указать тип базового значения наряду с именем перечисления, а затем присвоить отдельные значения каждому

варианту выбора при его определении. Ниже приведен вариант перечисления DaysOfWeek, в котором целочисленное значение присваивается каждому варианту выбора.

```
enum DaysOfWeek : Int {
    case Sunday = 0
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
    case Saturday
}
```

Тип исходного значения указывается при объявлении перечисления. В данном случае это тип Int, как показано ниже.

```
enum DaysOfWeek : Int {
```

Исходными значениями могут быть символьные строки или величины любого из целочисленных типов. Если исходное значение относится к целочисленному типу, присваивать его явно каждому варианту выбора из перечисления не нужно. В этом случае неприсвоенные значения автоматически выводятся добавлением 1 к предыдущему исходному значению. Так, в приведенном выше примере перечисления варианту выбора Sunday было присвоено нулевое исходное значение. Следовательно, варианту выбора Monday будет присвоено исходное значение 1, варианту выбора Tuesday — исходное значение 2 и т.д. Впрочем, автоматическое присваивание исходных значений можно отменить, при условии, что у каждого варианта выбора имеется свое особое значение, как показано ниже.

```
enum DaysOfWeek : Int {
    case Sunday = 0
    case Monday           // 1
    case Tuesday          // 2
    case Wednesday         // 3
    case Thursday          // 4
    case Friday = 20        // 20
    case Saturday          // 21
}
```

Получить исходное значение варианта выбора можно также, обратившись к его свойству rawValue следующим образом:

```
var day = DaysOfWeek.Saturday
let rawValue = day.rawValue // Равно 21; верна также и следующая ссылка:
                           // DaysOfWeek.Saturday.rawValue
```

Ниже приведен еще один пример перечисления, в котором присваиваются исходные значения типа String.

```
enum ResultType : String {
    case SUCCESS = "Success"
```

834 ПРИЛОЖЕНИЕ * ВВЕДЕНИЕ В ЯЗЫК SWIFT

```
case WARNING = "Warning"  
case ERROR = "Error"  
}  
let s = ResultType.WARNING.rawValue // s = "Warning"
```

По заданному исходному значению можно сформировать значение соответствующего варианта выбора из перечисления, передав его инициализатору, как показано ниже.

```
let result = ResultType(rawValue: "Error")
```

В данном примере кода переменная `result` относится не к типу `ResultType`, а к **необязательному** типу `ResultType?`. Инициализатору вполне возможно передать неверное исходное значение, и поэтому нужно каким-то образом указать, что достоверного варианта выбора с таким значением в перечислении не существует. Для этой цели в языке Swift предусмотрено возвращаемое специальное значение `nil`, но его можно присвоить переменной только необязательного, а не обычного типа. Необязательные типы обозначаются завершающим знаком `?`, а манипулировать ими следует очень аккуратно. Более подробно необязательные типы рассматриваются далее в этом приложении.

В языке Objective-C перечисление часто используется для определения битовых масок. Каждое конкретное перечисление представляет собой маску, содержащую один бит. Эти значения можно комбинировать, применяя операцию OR к двум или более перечислениям. Такой пример приведен в главе 14, где рассматриваются методы класса `NSFileManager` из каркаса Foundation, позволяющие найти стандартные каталоги, такие как `Documents`. Один из аргументов этих методов задает домен или домены, в которых следует выполнить поиск. В языке Objective-C домены можно определить с помощью перечисления, как показано ниже:

```
enum {  
    NSUserDomainMask = 1,  
    NSLocalDomainMask = 2,  
    NSNetworkDomainMask = 4,  
    NSSystemDomainMask = 8,  
    NSAllDomainsMask = 0xffff,  
};  
typedef NSUInteger NSSearchPathDomainMask;
```

Легко видеть, что отдельные значения представляют собой степени двойки. Это позволяет комбинировать их с помощью операции OR, не теряя возможности использовать оригинальные значения.

Массивы, диапазоны и словари

В языке Swift предусмотрены три основные разновидности коллекций: массивы, словари и множества, а также синтаксис диапазонов, предоставляющий удобный способ обозначить (возможно, большой) диапазон значений. Диапазоны особенно удобны для доступа к массивам.

Массивы и диапазоны

В языке Swift поддерживается создание массива значений с помощью синтаксиса [тип], где тип — это тип значения в массиве. В следующем примере кода создаются и инициализируются массивы целочисленных значений и символьных строк:

```
var integers = [1, 2, 3]
var days = ["Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"]
```

Разумеется, объявление массива можно отделить от его инициализации, при условии, что он инициализируется перед его применением. В этом случае нужно явно указать тип массива следующим образом:

```
var integers: [Int] // [Int] означает массив значений типа Int

integers = [1, 2, 3]
```

Чтобы инициализировать пустой массив, достаточно указать квадратные скобки [], как показано ниже.

```
var empty: [String] = []
```

Для доступа к элементам массива служит числовой индекс. Первый элемент массива доступен по нулевому индексу, как показано ниже.

```
integers[0]      // 1
integers[2]      // 3
days[3]          // "Wednesday"
```

Этот синтаксис используется и для присваивания нового значения элементу массива, как в следующих строках кода:

```
integers[0] = 4           // [4, 2, 3]
days[3] = "WEDNESDAY"     // Заменяет "Wednesday" на "WEDNESDAY"
```

Для извлечения или видоизменения части массива служит предусмотренный в языке Swift **синтаксис диапазонов**. Он позволяет вносить изменения в целый ряд элементов массива, как показано ниже.

```
var integers = [1, 2, 3]
integers[1..<3]           // Получить 1- и 2-й элементы массива.
                           // Результат равен [2, 3]
integers[1..<3] = [4]       // Заменить 1- и 2-й элементы массива
                           // 4-м элементом. Результат равен [1, 4]
integers = [1, 2, 3]
integers[0...1] = [5, 4]     // Заменить нулевой и 1-й элементы массива
                           // элементами [5, 4]. Результат равен [5, 4, 3]
```

ЗАМЕЧАНИЕ. Если вы посмотрите на столбец результатов, то не увидите правильный ответ; вам необходимо развернуть результат, как было показано выше, или просмотреть его в нижней части окна отладки.

836 ПРИЛОЖЕНИЕ 4 ВВЕДЕНИЕ В ЯЗЫК SWIFT

Синтаксис диапазона `a..` означает все значения в пределах от `a` до `b`, исключая `b`, а следовательно, `1..5` равнозначно `1, 2, 3, 4`. Синтаксис `a...b` означает включение `b` в указанный диапазон, и поэтому `1...5` означает `1, 2, 3, 4, 5`. В заключение синтаксис `a..` всегда означает пустой диапазон `a...a`, содержащий один элемент (т.е. себя самого). Значение `b` должно быть больше или равно значению `a` с неявным приращением на 1.

Для получения количества элементов в массиве служит свойство `count`, как показано ниже.

```
var integers = [1, 2, 3]
integers.count           // 3
integers[1..3] = [4]
integers.count           // 2
```

Для ввода элемента в массив вызывается метод `append()` или метод `insert(_:_atIndex:)` следующим образом:

```
var integers = [1, 2, 3]
integers.append(4)          // Результат равен [1, 2, 3, 4]
integers.insert(-1, atIndex: 0) // Результат равен [-1, 2, 3, 4]
```

Для удаления всех или только некоторых элементов из массива вызываются методы `removeAll()`, `removeAtIndex()` и `removeRange()`, как показано ниже.

```
var days = ["Sunday", "Monday", "Tuesday", "Wednesday",
           "Thursday", "Friday", "Saturday"]
days.remove(at: 3)           // Удаляет строку "Wednesday" и
                            // возвращает ее вызывающему коду
days.removeSubrange(0..4)    // Оставляет в массиве только
                            // элементы ["Friday", "Saturday"]
days.removeAll(keepCapacity: false) // Оставляет массив пустым
```

Аргумент `keepCapacity`, передаваемый методу `removeAll()`, обозначает, будет ли место, выделенное для элементов массива, сохранено (логическое значение `true`) или освобождено (логическое значение `false`). Перебрать можно весь массив или только его часть, используя оператор цикла `for`, рассматриваемый далее, в разделе “Управляющие операторы”.

Содержимое массива, создаваемого с помощью оператора `let`, ни коим образом не может быть изменено, как показано ниже.

```
let integers = [1, 2, 3]    // Массив констант
integers = [4, 5, 6]        // ОШИБКА: заменить массив нельзя!
integers[0] = 2             // ОШИБКА: переназначить элемент массива нельзя!
integers.removeAll(keepCapacity: false) // Ошибка: невозможно модифицировать
                                         // содержимое
```

Как и символьные строки, массивы являются объектами-значениями. Следовательно, когда они присваиваются или передаются функциям, то создаются их копии, как показано ниже.

```
var integers = [1, 2, 3]
var integersCopy = integers      // Создается копия массива integers
integersCopy[0] = 4             // Массив integers не изменяется
integers                      // [1, 2, 3]
integersCopy                   // [4, 2, 3]
```

Для проверки, содержит ли массив заданный элемент, можно использовать метод `contains()`.

```
let integers = [1, 2, 3]
integers.contains(2)    // true
integers.contains(4)    // false
```

Для получения индекса элемента в массиве используется метод `index(of:)`.

```
let integers = [1, 2, 3]
integers.index(of: 3)      // Результат равен 2
```

Если элемент не найден в массиве, то результат равен `nil`:

```
let integers = [1, 2, 3]
integers.index(of: 5)      // Результат равен nil
```

Словари

Используя синтаксис, аналогичный применяемому для массивов, можно создать словарь — структуру данных, отображающую экземпляры ключей на соответствующие значения. В следующем примере кода создается словарь, в котором ключами служат символьные строки, а значениями — целые числа:

```
var dict = ["Red": 0,
           "Green": 1,
           "Blue": 2]
```

Формально словарь в данном примере относится к типу `[String: Int]`. Используя синтаксис **обобщений**, который в этом приложении не рассматривается, этот словарь можно также обозначить как `Dictionary<String, Int>`. Аналогично массив типа `[Int]` обозначается как `Array<Int>`.

Если инициализатор не применяется, тип словаря можно указать явно при его объявлении следующим образом:

```
var dict: [String: Int];
dict = ["Red": 0, "Green": 1, "Blue": 2]
```

Для получения значения из словарной статьи по заданному ключу служит индексное обозначение, как показано ниже.

```
let value = dict["Red"] // Результат равен 0, значение отображается в ключ "Red"
```

Внести изменения в словарь можно таким же образом, как и в массив:

```
dict["Yellow"] = 3      // Вводит новое значение по ключу "Yellow"
dict["Red"] = 4         // Обновляет значение по ключу "Red"
```

Для того чтобы удалить отдельный элемент из словаря, достаточно вызвать метод `removeValue(forKey:)`, а чтобы удалить все элементы — метод `removeAll()`, как показано ниже.

```
var dict = ["Red": 0, "Green": 1, "Blue": 2]
dict.removeValue(forKey: "Red")      // Удаляет значение по ключу "Red"
dict.removeAll()                   // Опустошает словарь
```

Словари, создаваемые с помощью оператора `let`, становятся неизменяемыми, как показано в следующем примере кода:

```
let fixedDict = ["Red": 0, "Green": 1, "Blue": 2]
fixedDict["Yellow"] = 3            // Недопустимо
fixedDict["Red"] = 4              // Недопустимо
fixedDict = ["Blue": 7]           // Недопустимо
fixedDict.removeValue(forKey: "Red") // Недопустимо
```

Перебрать словарь по ключам можно в цикле с помощью оператора `for`, поясняемого далее, в разделе “Управляющие операторы”. Для получения количества пар “ключ–значение”, хранящихся в словаре, служит свойство `count`, как показано в следующем примере кода:

```
var dict = ["Red": 0, "Green": 1, "Blue": 2]
dict.count           // 3
```

Подобно массивам, словари состоят из значений определенных типов и копируются при присваивании, передаче функциям в качестве аргумента или возврате из них. Изменения в копии словаря не оказывает никакого влияния на его оригинал, как показано ниже.

```
var dict = ["Red": 0, "Green": 1, "Blue": 2]
var dictCopy = dict
dictCopy["Red"] = 4               // Не оказывает влияния на исходный словарь
                                  // в переменной dict
dict                         // "Red":0, "Green": 1, "Blue": 2
dictCopy                      // "Red":4, "Green": 1, "Blue": 2
```

Множества

Третья, и последняя, разновидность коллекции в языке Swift — класс `Set`. Он представляет собой коллекцию элементов, не имеющих определенного порядка и дубликатов. Добавление нового экземпляра элемента, который уже содержится в множестве, не влияет на ее содержимое. В большинстве остальных аспектов множество очень похоже на массив.

Синтаксис инициализации множества совпадает с синтаксисом инициализации массива. Для того чтобы избежать неоднозначности, необходимо явно указать, что вы создаете экземпляр класса `Set`. Ниже приведены два (эквивалентных) варианта инициализации.

```
let s1 = Set([1, 2, 3])
let s2: Set<Int> = [1, 2, 3]
```

Метод `contains()` возвращает значение типа `Bool`, которое содержит информацию о том, есть ли заданный элемент в множестве, и свойство `count`, в котором хранится количество элементов в множестве.

```
s1.contains(1) // true
s2.contains(4) // false
s1.count // 3
```

Как будет показано позднее, элементы множества можно обойти с помощью цикла.

Для того чтобы добавить элемент в множество или удалить из него, используются методы `insert()` и `remove()`.

```
var s1 = Set([1, 2, 3]) // [2, 3, 1] (порядок значения не имеет)
s1.insert(4) // [2, 3, 1, 4]
s1.remove(1) // [2, 3, 4]
s1.removeAll() // [] (empty set)
```

Классы `NSArray`, `NSDictionary` и `NSSet`

В каркасе Foundation имеются свои классы `NSArray`, `NSDictionary` и `NSSet` представляющие на языке Objective-C массивы, словари и множества. Взаимосвязь этих классов с их аналогами в языке Swift аналогична взаимосвязи классов `NSString` и `String`. В общем, массив в языке Swift можно трактовать как объект типа `NSArray`, словарь — как объект типа `NSDictionary`, а множество — как объект класса `NSSet`. Допустим, что имеется следующая строка кода:

```
let s: NSString = "Red,Green,Blue"
```

Как разделить содержимое этой переменной на три символьные строки с отдельными названиями цветов? В классе `NSString` имеется метод `components(separatedBy:)`, делающий именно то, что в данном случае нужно. Такой же метод есть и в методе `String`, но в данном случае мы работаем с классом `NSString`. В частности, этот метод принимает в качестве аргумента разделитель символьных строк и возвращает массив типа `NSArray`, содержащий отдельные составляющие исходной строки.

```
let s: NSString = "Red,Green,Blue"
let components = s.components(separatedBy: ",") // Вызывает метод класса NSString
components // ["Red", "Green", "Blue"]
```

Несмотря на то что данный метод возвращает массив типа `NSArray`, он автоматически преобразуется языковыми средствами Swift в соответствующую коллекцию (массив объектов типа `String`), а его элементы выводятся к типу `[String]`. Можете убедиться в этом сами, установив указатель на имя переменной, нажав клавишу `<Option>` и щелкнув кнопкой мыши.

Экземпляры классов `NSDictionary` и `NSArray` можно создать и непосредственно в коде Swift следующим образом:

```
let d = NSDictionary()
```

Разумеется, для константы `d` в данном случае выводится тип `NSDictionary`. Привести тип `NSDictionary` явным образом к типу словаря в языке Swift можно с помощью рассматриваемого далее ключевого слова `as` независимо от того, создается или получается словарь из метода, предоставляемого в каркасе Foundation или другом каркасе. Ниже показано, как это делается.

```
let e = d as Dictionary
```

Если теперь нажать клавишу `<Option>` и щелкнуть на имени константы `e`, можно обнаружить, что для нее языковыми средствами Swift выводится тип `Dictionary<NSObject, AnyObject>`, и это еще один способ обозначить ее тип как `[NSObject: AnyObject]`. Что это за типы ключей и значений и откуда они взялись? В отличие от словарей в языке Swift, типы объектов, содержащихся в словаре типа `NSDictionary`, неизвестны. То же самое относится и к массиву типа `NSArray`. Поэтому в языке Swift обычно нельзя выяснить типы ключей и значений из словаря типа `NSDictionary`. Вместо них подставляются более общие типы. В частности, `NSObject` является базовым классом для всех объектов в каркасе Foundation, тогда как `AnyObject` — рассматриваемым далее протоколом, соответствующим практически любому типу в языке Swift. Сигнатура типа `[NSObject: AnyObject]` обозначает, что в языке Swift ничего неизвестно о содержимом словаря. Когда элементы извлекаются средствами языка Swift из словаря типа `NSDictionary` или `NSArray`, то для этой цели приходится пользоваться операцией `as!` для их явного приведения к нужным типам данных или приведения самого типа `NSDictionary` к тому типу данных, к которому этот словарь должен принадлежать. Так, если заранее известно, что в словаре типа `NSDictionary` одни символьные строки отображаются на другие, то можно выполнить следующее:

```
let d = NSDictionary()
let e = d as! [String: String]
```

В данном случае тип `NSDictionary` можно привести к типу `[String: String]`. Такой код оказывается вполне работоспособным, даже если словарь фактически содержит объекты типа `NSString`, а не `String`, что вполне возможно, поскольку в языке Swift выполняется автоматическое взаимное преобразование объектов `String` и `NSString`. Следует, однако, иметь в виду, что если типы ключей и значений из словаря типа `NSDictionary` не соответствуют тому, что требуется в коде приложения, оно завершается аварийно. Возможно, вы обратили внимание на то, что приведение типов в коде выполнено с помощью оператора `as!`, а не `as`. Разницу между этими двумя операторами мы объясним немного позднее.

Подобное затруднение можно разрешить более надежным способом, как поясняется в следующем разделе.

Необязательные типы данных

Вернемся к следующему примеру кода из предыдущего раздела, посвященного словарям:

```
var dict: [String: Int];
dict = ["Red": 0, "Green": 1, "Blue": 2]
let value = dict["Red"]
```

Несмотря на то что все значения в словаре относятся к типу `Int`, их тип на самом деле выводится не как `Int`, а как `Int?`, т.е. **необязательный целочисленный тип**. Знак `?` обозначает необязательный характер данного типа. Что же означает необязательный тип и почему он здесь применяется? Для того чтобы ответить на эти вопросы, проанализируем, что произойдет, если получить доступ к словарю по ключу, для которого в нем отсутствует значение, как показано ниже.

```
let yellowValue = dict["Yellow"]
```

Какое же значение следует присвоить константе `yellowValue`? В большинстве языков программирования этот вопрос разрешается с помощью различного значения, которое обычно обозначает “отсутствие значения”. Так, в языке Objective-C это значение обозначается как пустое `nil` и, по существу, является переопределением нулевого значения; в С и C++ — как `NULL` (в этих языках оно также служит переопределением нулевого значения); а в языке Java — как `null`. Но дело в том, что пользоваться значением `nil` или другим равнозначным ему значением небезопасно. Например, применение пустой ссылки `null` в языке Java приведет к исключению, а в С и C++ — скорее всего, к аварийному завершению приложения. Хуже того, выяснить, может ли переменная содержать пустое значение `null` (`nil` или `NULL`), никак нельзя. Подобное затруднение изящно разрешается в языке Swift, в котором предусмотрено пустое значение `nil`, но это значение может быть установлено в переменной или константе только в том случае, если оно объявляется как необязательное или его тип выводится как необязательный. В итоге по типу переменной или константы можно сразу же выяснить, может ли она быть пустой (`nil`). К тому же данное обстоятельство в языке Swift требуется принимать во внимание явным образом всякий раз, когда используется значение необязательного типа.

Для того чтобы все сказанное выше стало понятнее, рассмотрим некоторые примеры применения необязательных типов. Допустим, что переменная `color` определяется следующим образом:

```
var color = "Red"
```

Если переменная `color` объявляется именно таким образом, ее тип выводится как `String`, т.е. он не является необязательным. Следовательно, пытаться присвоить этой переменной пустое значение `nil` недопустимо, как показано ниже.

842 ПРИЛОЖЕНИЕ ■ ВВЕДЕНИЕ В ЯЗЫК SWIFT

```
color = nil      // Недопустимо: переменная color не относится
                // к необязательному типу!
```

Вследствие этого можно вообще не беспокоиться, содержит ли переменная color пустое значение nil. Это также означает, что в данной переменной нельзя хранить значение, возвращаемое из словаря, даже если заранее известно, что это значение не будет пустым (nil), как показано ниже.

```
let dict = [0: "Red", 1: "Green", 2: "Blue"]
color = dict[0]      // Недопустимо: значение статьи словаря dict[0]
                    // относится к необязательному строковому типу,
                    // тогда как значение переменной color не
                    // является необязательным!
```

Для того чтобы приведенное выше присваивание стало допустимым, заменим тип String переменной color типом String? следующим образом:

```
let dict = [0: "Red", 1: "Green", 2: "Blue"]
var color: String?      // Здесь "String?" означает необязательный
                        // строковый тип
color = dict[0]          // Допустимо!
print(color)            // Что же выводится на экран?
```

Как же воспользоваться значением, присвоенным переменной color? Введите приведенный выше фрагмент кода на игровой площадке. В итоге вы обнаружите, что на экран выводится результат Optional("Red"), а не "Red". При доступе к словарю вместо конкретного значения возвращается значение, заключенное в оболочку необязательного типа. Для того чтобы получить конкретное строковое значение, его придется извлечь из оболочки необязательного типа с помощью операции ! следующим образом:

```
let actualColor = color!      // Здесь "color!" означает извлечение
                            // конкретного значения из оболочки
                            // необязательного типа
```

В данном случае сначала выводится тип String константы actualColor, а затем ей присваивается строковое значение "Red". Из этого следует, что всякий раз, когда значение извлекается из словаря, оно должно относиться к необязательному типу, из оболочки которого придется извлечь нужное значение. Не следует, однако, забывать об упомянутой выше опасности пустых ссылок nil. В языке это означает, что извлекать конкретные значения из необязательных типов может быть небезопасно. Замените код на игровой площадке следующим фрагментом:

```
let dict = [0: "Red", 1: "Green", 2: "Blue"]
let color = dict[4]
let actualColor = color!
```

Тип переменной color правильно приводится языковыми средствами Swift к необязательному типу String?, но когда из его оболочки извлекается результат доступа к словарю, чтобы присвоить его константе actualColor, то возникает

ошибке. Об этой ошибке сообщается на игровой панели, а в приложении она приведет к аварийному сбою. Почему эта ошибка происходит? Дело в том, что в словаре отсутствует статья по ключу 4, и поэтому переменной `color` присваивается пустое значение `nil`. Если же попытаться извлечь это значение из оболочки необязательного типа, возникнет аварийный сбой. В этой связи может возникнуть суждение, что необязательные типы никак не улучшают положение в языке Swift по сравнению с другими языками программирования, но это совсем не так. Во-первых, тот факт, что переменная `color` относится к необязательному типу, означает, что в ней следует предполагать наличие пустого значения `nil`, и наоборот, что не менее важно. Во-вторых, в языке Swift предоставляется возможность разрешать ситуации, когда необязательное значение оказывается пустым (`nil`). На самом деле это можно сделать тремя способами.

Прежде всего, можно проверить, получено ли пустое значение из словаря. Если оно не получено, только тогда извлекать его из оболочки необязательного типа, как показано ниже.

```
if color != nil {
    let actualColor = color!
}
```

Приведенная выше конструкция настолько распространена, что в языке Swift предусмотрен ее укороченный вариант. Ниже приведен второй способ извлечения значения из оболочки необязательного типа.

```
if let actualColor = color {
    // Выполняется только в том случае, если переменная color
    // не содержит пустое значение nil
    print(actualColor)
}
```

Блок кода, связанный с условным оператором `if`, выполняется только в том случае, если переменная `color` не содержит пустое значение `nil`, а константе `actualColor` присваивается значение, извлекаемое из оболочки необязательного типа. Более того, константу `actualColor` можно даже сделать переменной, заменив оператор `let` ключевым словом `var` следующим образом:

```
if var actualColor = color {
    // Выполняется только в том случае, если переменная color
    // не содержит пустое значение nil. В переменную actualColor
    // можно внести изменения
    print(actualColor)
}
```

В рассмотренных выше примерах мы определили новую переменную `actualColor` для хранения нераспакованного значения, которому мы должны дать новое имя. Иногда кажется неестественным использовать разные имена для того, что в действительности представляет собой одно и то же, и на самом деле это совсем необязательно — новой переменной можно дать старое имя, которое имела нераспакованная переменная необязательного типа.

```
if var color = color {
    // Выполняется, только если значение исходной
    // переменной color цвета не равно nil
    print(color) // Ссылка на новую переменную,
                  // содержащую распакованное значение
}
```

Дело в том, что новая переменная `color` не связана с существующей и вообще имеет другой тип (`String`, а не `String?`). В теле инструкции `if` имя `color` относится к новой, распакованной переменной, а не к исходной, упакованной.

```
let dict = [0: "Red", 1: "Green", 2: "Blue"]
let color = dict[0]
if var color = color {
    // Выполняется, только если переменная color не равна nil
    print(color) // "Red"
    color = "Green" // Повторное присваивание локальной переменной
} // Новая переменная color выходит из области видимости
color // Ссылка на исходное значение: "Red"
```

А что если требуется воспользоваться значением ключа по умолчанию, отсутствующим в словаре? Для разрешения этого вопроса в языке Swift предусмотрено удобное языковое средство. Ниже приведен третий способ извлечения значения из оболочки необязательного типа.

```
let dict = [0: "Red", 1: "Green", 2: "Blue"]
let color = dict[4]
let actualColor = color ?? "Blue"
```

Левый operand в операции `??`, называемой **объединяющей**, извлекается из оболочки необязательного типа, если он не является пустым (`nil`). В итоге возвращается значение первого операнда, а иначе — второго операнда объединяющей операции. В данном случае предполагается, что константа `color` содержит пустое значение `nil`, и поэтому константе `actualColor` присваивается строковое значение `"Blue"`. Разумеется, приведенный выше фрагмент кода можно сократить до двух операторов, чтобы сделать его более удобочитаемым, как показано ниже.

```
let dict = [0: "Red", 1: "Green", 2: "Blue"]
let actualColor = dict[4] ?? "Blue"
```

Словари служат далеко не единственным контекстом, в котором применяются необязательные типы. Если обратиться к документации к интерфейсу прикладного программирования Swift, то можно обнаружить немало методов, возвращающих значения необязательных типов. Имеется даже возможность возвратить значение необязательного типа из инициализатора, как уже показывалось в одном из приведенных ранее примеров. Так, если попытаться инициализировать экземпляр перечисления недостоверным исходным значением, то в итоге будет получено пустое значение `nil`, как показано ниже.

```
enum ResultType : String {
```

```

case SUCCESS = "Success"
case WARNING = "Warning"
case ERROR = "Error"
}
let result = ResultType(rawValue: "Invalid")

```

Для получаемого результата выводится необязательный тип `ResultType?`. В данном случае это пустое значение `nil`, поскольку в перечислении отсутствует вариант выбора с исходным строковым значением `"Invalid"`.

Выше мы предупреждали, что извлекать значения из оболочки необязательных типов следует осторожно, но иногда это может быть неудобно, да и не нужно. Выполнив примеры из данной книги, вы обнаружите, что в классе нередко определяется переменная, которую нельзя инициализировать при инициализации самого класса. Тем не менее она будет иметь достоверное значение до ее применения в прикладном коде. В подобных случаях переменную придется определить как относящуюся к необязательному типу, извлекая ее из оболочки необязательного типа всякий раз, когда ею нужно будет воспользоваться. Это вполне удовлетворяет компилятор, но в то же время означает, что везде, где требуется, придется ввести операторы `!` или заключить доступ к переменной в конструкцию `if let`, даже если заранее известно, что исход извлечения конкретного значения из оболочки необязательного типа всегда будет удачным.

К счастью, в языке Swift этого можно избежать, сообщив лишь, что автоматическое извлечение из оболочки необязательного типа следует выполнять только при доступе к значению, заключенному в оболочку этого типа. Ниже показано, как это сделать на примере словаря.

```

let dict = [0: "Red", 1: "Green", 2: "Blue"]

var color: String!      // Обратите внимание на операцию !
color = dict[0]          // Присваивает Optional("Red"), т.е. значение,
                        // заключенное в оболочку необязательного типа
print(color)            // Автоматически извлекает значение из оболочки
                        // необязательного типа

```

В данном примере кода переменная `color` объявлена как относящаяся к типу `String!`, а не `String?`. Переменные, объявленные подобным образом, считаются **неявно извлекаемыми из оболочки необязательных типов**. Операция `!` означает, что переменная всегда будет иметь непустое значение, когда она применяется, а поэтому ее следует извлечь из оболочки необязательного типа. Это означает, что вместо вызова `print(color!)` можно сделать вызов `print(color!).` И хотя в данном случае ничего не сохраняется, используя это языковое средство, в конечном счете придется набирать меньше операторов, если потребуется неоднократный доступ к получаемым результатам.

Следует, однако, иметь в виду, что в данном примере компилятору языка Swift было сообщено, что переменная вообще не должна иметь пустое значение `nil`. Если же окажется, что она содержит пустое значение `nil` при попытке

воспользоваться ею, то ее значение будет извлечено из оболочки необязательного типа, что приведет к аварийному сбою в приложении.

Необязательные типы данных позволяют разрешить еще одно затруднение, связанное с обработкой значений, получаемых из словаря типа `NSDictionary`. Рассмотрим это затруднение, создав словарь типа `NSDictionary` с некоторыми исходными значениями, как показано ниже.

```
let d = NSDictionary(objects: ["Red", "Green", "Blue"],
                     forKeys: [0 as NSCopying, 1 as NSCopying, 2 as NSCopying])
```

В первой строке приведенного выше фрагмента кода словарь типа `NSDictionary` инициализируется с преобразованием его статей из типа `Int` в тип `String`, а во второй строке кода его тип приводится к соответствующему типу словаря в языке Swift с помощью операции `as`. На практике создавать и приводить словарь типа `NSDictionary` к соответствующему типу подобным образом вряд ли придется, поскольку для этого достаточно создать словарь в языке Swift. Но допустим, что словарь получен в результате вызова метода из каркаса Foundation, чтобы, например, заполнить его содержимым файла. При условии, что статьи такого словаря действительно преобразуются из типа `Int` в тип `String`, приведенная ниже строка кода оказывается вполне работоспособной, а следовательно, содержимое словаря оказывается доступным обычным образом.

```
let color = d[1] // Получает строковое значение "Green", заключенное
                  // в оболочку необязательного типа
```

Следует, однако, иметь в виду, что компилятору языка Swift на самом деле ничего не известно о типе значения, получаемого из словаря. Поэтому оно приводится к необязательному типу `AnyObject?`, в оболочку которого заключается некоторый объект, о чем выдается соответствующее предупреждение. Во избежание этого предупреждения получаемый результат можно явным образом привести к нужному типу, как показано ниже.

```
let color = d[1] as! String
```

Но что если полученный тип словаря оказывается неверным? Возможно, кто-нибудь ввел неверные данные в файл, в результате чего получен его тип `[Int : Int]`. Для того чтобы посмотреть, что при этом произойдет, измените числовой тип ключей на строковый в первой строке кода следующим образом:

```
let d = NSDictionary(objects: [ 0, 1, 2 ],
                     forKeys: [0 as NSCopying, 1 as NSCopying,
                               2 as NSCopying])
let value = d[1] as! String
```

При попытке привести тип словаря к типу `String` в данном фрагменте кода возникнет аварийный сбой. Поэтому нужно каким-то образом обнаружить подобное условие и предпринять корректирующее действие, чтобы дело не дошло до аварийного сбоя. Для этого можно, в частности, воспользоваться оператором

`as?` вместо оператора `as`. В результате выполнения оператора `as` возвращается значение необязательного типа. Если первый его operand не относится к типу, задаваемому вторым operandом, то вместо аварийного сбоя возвращается пустое значение `nil`. Таким образом, можно написать код, аналогичный следующему:

```
let d = NSDictionary(objects: [ 0, 1, 2],
                     forKeys: [0 as NSCopying, 1 as NSCopying,
                               2 as NSCopying])
if let value = d[1] as? String { // Оператор as? возвращает nil
    // если d не имеет тип String?
    print("OK")
} else {
    print("Incorrect types") // Выполняется, если d не имеет тип [Int: String]
}
```

Добиться того же самого результата можно и по-другому, воспользовавшись ключевым словом `is`, чтобы проверить, относится ли словарь к предполагаемому типу, прежде чем приводить к нему словарь, как показано ниже.

```
if d is [Int: String] { // Вычисляется логическое значение true, если
    // статьи в словаре d преобразуются из типа Int в тип String
    print("Is [Int: String]")
} else {
    print("Incorrect types")
}
```

Возможно, вы заметили, что при приведении типов мы использовали как оператор `as`, так и оператор `as!`. В чем же заключается разница между ними? Грубо говоря, оператор `as` используется, когда приведение является гарантировано корректным, а `as!` — когда это условие не выполняется. Знак `!` выражает тот факт, что вы заставляете компилятор принять ошибочный код, например содержащий понижающее приведение. Ниже приведен пример безопасного приведения, в котором используется оператор `as`.

```
let s = "Fred"
let n = s as NSString
```

Тип, выведенный из переменной `s`, — это класс `String`, который мы пытались привести к типу `NSString`. Язык Swift выполняет эту операцию автоматически, поэтому мы использовали оператор `as`. Аналогично с помощью оператора `as` можно выполнить любое повышающее приведение.

```
let label = UILabel()
let view = label as UIView
```

Здесь переменная `label` имеет тип `UILabel`, представляющий собой класс каркаса `UIKit`, представляющий метку — компонент графического пользовательского интерфейса, демонстрирующий текст. Каждый компонент графического пользовательского интерфейса в каркасе `UIKit` представляет собой подкласс класса `UIView`, поэтому можно свободно выполнить повышающее приведение

метки к ее базовому классу `UIView` с помощью оператора `as`. Однако обратное утверждение неверно.

```
let view: UIView = UILabel()
let label = view as! UILabel // Требуется понижающее приведение!
```

На этот раз мы создали объект класса `UILabel`, но присвоили их переменной типа `UIView`. Для того чтобы привести эту переменную обратно к типу `UILabel`, необходимо выполнить оператор `as!`, потому что нет гарантии, что переменная типа `UIView` ссылается на объект класса `UILabel`, — она может ссылаться на объект другого типа из каркаса `UIKit`, например `UIButton`.

Рассмотрим еще один пример, который мы уже видели.

```
let d = NSDictionary(objects: ["Red", "Green", "Blue"],
                     forKeys: [0 as NSCopying, 1 as NSCopying,
                               2 as NSCopying])
let color = d[1] as! String
```

Как мы уже видели, когда мы извлекаем значение из объекта класса `NSDictionary`, его тип выводится как `AnyObject?` и может быть любым типом языка Swift. Для того чтобы привести этот тип к типу `String`, необходимо использовать оператор `as!`.

На этом временно остановим рассмотрение необязательных типов. Мы еще вернемся к ним, когда дело дойдет до обсуждения методов в последующих разделах, “Свойства” и “Методы”.

Управляющие операторы

Итак, представив типы данных в языке Swift, их инициализацию и выполняемые над ними операции, перейдем к рассмотрению управляющих операторов. В языке Swift поддерживаются все управляющие операторы, обнаруживаемые в других языках программирования, и в то же время они наделены рядом новых свойств.

Условный оператор `if`

В языке Swift условный оператор `if` действует таким же образом, как и в большинстве других языков программирования, проверяя условное выражение и выполняя некоторый код только в том случае, если вычисление этого выражения дает истинный результат (`true`). Условный оператор `if` можно дополнить блоком оператора `else`, который выполняется, если вычисление условного выражения дает ложный результат (`false`), как показано в следующем примере:

```
let random = arc4random_uniform(10)
if random < 5 {
    print("Hi")
} else {
    print("Ho")
}
```

Функция `arc4random_uniform()` возвращает случайное число в пределах от 0 (включительно) до значения ее аргумента (исключительно), поэтому в данном примере возвращается любое целое число пределах от 0 до 9. Затем это значение проверяется, и если оно меньше 5, то выводится символьная строка "Hi", а иначе — символьная строка "Ho". Следует заметить, что в языке Swift не требуется заключать условное выражение в круглые скобки, и поэтому любое из следующих выражений оказывается работоспособным:

```
if random < 5 { // Круглые скобки не требуются
}

if (random < 5) { // Но могут быть использованы
}
```

В отличие от других языков программирования, в языке Swift выполняемый код должен быть заключен в фигурные скобки, даже если он состоит из единственной строки. Это означает, что приведенный ниже код написан неверно.

```
if random < 5
    print("Hi") // Неверно: должно быть в фигурных скобках!
else
    print("Ho") // Неверно: должно быть в фигурных скобках!
```

В языке Swift предусмотрена также тернарная операция `? :`, которая, возможно, знакома вам из других языков программирования. В этой операции сначала вычисляется логическое условие, предшествующее знаку `?`, а затем выполняется оператор, указываемый между знаками `?` и `:`, если вычисление этого выражения дает истинный результат, а иначе — оператор, следующий после знака `..`. Используя эту операцию, предыдущий пример кода можно написать следующим образом:

```
let random = arc4random_uniform(10)
random < 5 ? print("Hi") : print("Ho")
```

В данном случае код можно сделать еще более кратким, если требуется лишь вывести символьную строку, как показано ниже.

```
let random = arc4random_uniform(10)
print(random < 5 ? "Hi" : "Ho")
```

Такой код выглядит более или менее ясным и удобочитаемым в зависимости от точки зрения конкретного программиста.

Мы уже встречали специальную форму оператора `if`, упрощающую обработку значений необязательных типов:

```
let dict = [0: "Red", 1: "Green", 2: "Blue"]
let color = dict[0]
if let color = color {
    // Выполняется, только если значение color не равно nil
    print(color) // "Red"
}
```

На самом деле можно одновременно распаковать несколько значений необязательных типов в одной инструкции `if let` или `if var`:

```
let dict = [0: "Red", 1: "Green", 2: "Blue", 3: "Green", 4: "Yellow"]
let color1 = dict[Int(arc4random_uniform(6))]
let color2 = dict[Int(arc4random_uniform(6))]
if let color1 = color1, color2 = color2 {
    // Выполняется, только если обе переменные,
    // color1 и color2, не равны nil
    print("color1: \(color1), color2: \(color2)")
}
```

Этот код генерирует пару случайных ключей в диапазоне от 0 до 5 и использует строки, обозначающие цвет, из словаря. Поскольку записи с ключом 5 нет, возможно, что одно или оба значения равны `nil`. Сложная форма оператора `if` позволяет безопасно распаковать оба значения и выполнить соответствующие инструкции, только если оба значения не равны `nil`. Кроме того, можно вставить тестовое условие, добавив раздел `where`:

```
let dict = [0: "Red", 1: "Green", 2: "Blue", 3: "Green", 4: "Yellow"]
let color1 = dict[Int(arc4random_uniform(6))]
let color2 = dict[Int(arc4random_uniform(6))]
if let color1 = color1, color2 = color2 where color1 == color2 {
    // Выполняется, только если значения одинаковы
    print("color1: \(color1), color2: \(color2)")
}
```

Проверку можно вставить даже перед распаковкой значений необязательного типа:

```
let dict = [0: "Red", 1: "Green", 2: "Blue", 3: "Green", 4: "Yellow"]
let color1 = dict[Int(arc4random_uniform(6))]
let color2 = dict[Int(arc4random_uniform(6))]
if dict.count > 3, let color1 = color1, color2 = color2 where color1 == color2 {
    print("color1: \(color1), color2: \(color2)")
}
```

Оператор `for`

В языке Swift предусмотрены два варианта оператора `for`. Первый из них хорошо знаком всем, у кого имеется опыт программирования на С-подобном языке, как показано в следующем примере кода. Этот вариант в языке Swift 3 объявлен устаревшим и больше не считается корректным.

```
for var i = 0; i < 10; i+=1 {
    print(i)
}
```

Второй, допустимый вариант оператора `for` позволяет перебирать последовательность значений, например, в заданном диапазоне или в коллекции. В следующем примере кода достигается тот же самый результат, что и в предыдущем примере:

```
for i in 0..<10 {
    print(i)
}
```

Обратите внимание на то, что указывать переменную цикла в данной форме оператора `for` можно и без ключевого слова `var`. Используя функцию `stride()` из стандартной библиотеки Swift, можно организовать перебор и более общего диапазона значений. Так, в следующем примере кода выводятся целые числа от 10 до 0 включительно:

```
for i in stride(from: 10, through: 0, by: -2) {
    print(i)
}
```

Циклический перебор элементов массива организуется очень просто, а назначение прикладного кода становится более очевидным, чем в том случае, если организовать такой перебор по индексу массива, как показано ниже.

```
let strings = ["A", "B", "C"]
for string in strings {
    print(string)
}
```

Для перебора ключей в словаре можно воспользоваться его свойством `keys`, как показано ниже, хотя порядок итерации не определен, поскольку словари никоим образом не обеспечивают упорядочение своего содержимого.

```
let d = [ 0: "Red", 1: "Green", 2: "Blue" ]
for key in d.keys {
    print("\(key) -> \(d[key])")
}
```

Того же самого результата можно добиться и более прямым путем, перебирая словарь как множество пар “ключ–значение”, как в следующем примере кода:

```
for (key, value) in d {
    print("\(key) -> \(value)")
}
```

Каждая пара “ключ–значение” возвращается в виде **кортежа**, состоящего из двух элементов, где имя `key` связано с первым элементом кортежа, а имя `value` — со вторым его элементом. Подробнее о кортежах читайте в руководстве *The Swift Programming Language*.

Операторы `repeat` и `while`

В языке Swift предусмотрены такие же операторы цикла `repeat` и `while`, как и в языках C, C++, Java и Objective-C. Оба эти оператора выполняют код в теле цикла до тех пор, пока удовлетворяется заданное условие цикла. Они отличаются тем, что это условие в операторе `while` проверяется перед началом каждого шага цикла, тогда как в операторе `repeat` — в конце каждого

шага цикла. В приведенных ниже примерах кода демонстрируется применение операторов `repeat` и `while`.

```
var i = 10
while i > 0 {
    print(i)
    i -= 1
}

var j = 10
repeat {
    print(j)
    j -= 1
} while j > 0
```

Оператор `switch`

В языке Swift имеется весьма эффективный оператор `switch`. Здесь недостаточно места, чтобы описать все его свойства, поэтому рекомендуется подробнее ознакомиться с ним в руководстве *The Swift Programming Language*. Ниже на конкретных примерах будут проиллюстрированы лишь наиболее важные свойства этого оператора.

С помощью оператора `switch` можно организовать выбор ветви выполнения кода по одному из нескольких возможных значений переменной или выражения. Так, в следующем примере кода выводятся разные результаты в зависимости от значения переменной:

```
let value = 11
switch value {
case 2, 3, 5, 7, 11, 13, 17, 19:
    print("Count is prime and less than 20")
case 20...30:
    print("Count is between 20 and 30")
default:
    print("Greater than 30")
}
```

В данном примере кода обращают на себя внимание несколько особенностей, присущих оператору `switch`. Во-первых, в ветвях `case` может быть указано несколько возможных значений, разделяемых запятыми. Код в ветви `case` выполняется в том случае, если выражение, указываемое в операторе `switch`, принимает одно из значений, перечисленных в этой ветви. Во-вторых, в ветви `case` может быть указан диапазон значений. На самом деле в операторе `switch` можно организовать сопоставление с шаблоном (подробнее об этом — в документации к данному оператору). И в-третьих, управление не передается от одной ветви `case` к другой, как это происходит в большинстве других языков программирования. Это означает, что в приведенном выше примере будет выполнен код только в одной ветви `case`, а следовательно, результат будет выведен только один раз. Вводить оператор `break` в каждой ветви оператора совсем

необязательно `switch`. Если действительно требуется продолжить выполнение кода из одной ветви `case` в следующей ветви, достаточно ввести оператор `fallthrough` в конце первой из этих ветвей.

Список ветвей `case` должен быть исчерпывающим. Так, если исключить ветвь `default` из рассматриваемого здесь примера кода, то компилятор сообщит об ошибке. Более того, в каждой ветви `case` должна быть по крайней мере одна строка выполняемого кода. Это означает, что следующий фрагмент кода недопустим и вводит в заблуждение:

```
switch (value) {
    case 2:
    case 3: // Недопустимо, так как предыдущая ветвь case пустая!
        print("Value is 2 or 3")
    default:
        print("Value is neither 2 nor 3")
}
```

Для того чтобы исправить данный код, достаточно ввести оба значения 2 и 3 в одну и ту же ветвь `case` следующим образом:

```
switch (value) {
    case 2, 3: // Верно, поскольку в этой ветви case
        // перехватываются значения 2 и 3
        print("Value is 2 or 3")
    default:
        print("Value is neither 2 nor 3")
}
```

В качестве альтернативы можно использовать оператор `fallthrough`:

```
switch (value) {
    case 2: fallthrough
    case 3: // Неправильно – предыдущая ветвь пустая.
        print("Value is 2 or 3")
    default:
        print("Value is neither 2 nor 3")
}
```

Выражение в операторе `switch` не обязательно должно быть числовым. Так, в приведенном ниже примере выбор выполняемого кода в операторе `switch` осуществляется на основании значения в указанной символьной строке.

```
let s = "Hello"
switch s {
    case "Hello":
        print("Hello to you, too")
    case "Goodbye":
        print("See you tomorrow")
    default:
        print("I don't understand")
}
```

854 ПРИЛОЖЕНИЕ 8 ВВЕДЕНИЕ В ЯЗЫК SWIFT

В следующем примере кода демонстрируются применение определенного ранее перечисления Status, а также выбор той ветви case в операторе switch, где указано ассоциативное значение из данного перечисления:

```
enum Status {
    case OK
    case ERROR(String)
}
let result = Status.ERROR("Network connection rejected")
switch (result) {
case .OK:
    print("Success!")
case .ERROR(let message):
    print("Ooops: \(message)")
}
```

Компилятору известно, что выражение в операторе switch относится к типу Status, и поэтому полностью уточнять значения в ветвях case не нужно, а достаточно указать .OK вместо Status.OK. Компилятору также известно, что в перечислении Status определены только два возможных значения, а поскольку оба эти значения перечислены в ветвях case, потребность в дополнительной ветви default отпадает. Обратите внимание на форму выражения .ERROR(let message) во второй ветви case. Такая форма вынуждает компилятор извлечь ассоциативное значение ERROR и присвоить его константе message, которая достоверна только в области действия самой ветви case. Если же в перечислении определено больше одного ассоциативного значения, то для получения всех этих значений достаточно указать через запятую по одному оператору let на каждое из них.

ФУНКЦИИ И ЗАМЫКАНИЯ

В отличие от многих других языков программирования, для обозначения функции в языке Swift служит ключевое слово func. Для того чтобы определить функцию, нужно указать ее имя, список ее аргументов и тип возвращаемого значения. Так, в следующей функции площадь прямоугольника рассчитывается и возвращается по заданной ширине и высоте:

```
func areaOfRectangle(width: Double, height: Double) -> Double {
    return width * height
}
```

Аргументы функции заключаются в круглые скобки, перечисляются через запятую и указываются таким же образом, как и при объявлении переменных, т.е. сначала — имя, а затем — тип аргумента. Но в отличие от объявления переменных, указывать тип аргументов необязательно. Если же у функции отсутствуют аргументы, то круглые скобки должны быть пустыми, как показано ниже.

```
func hello() {
    print("Hello, world")
}
```

Если функция возвращает значение, то его тип должен быть указан после знака `->`. В приведенном выше примере функции `areaOfRectangle()` оба аргумента, а также возвращаемое значение относятся к типу `Double`. Если же функция ничего не возвращает, то тип возвращаемого значения можно опустить, как показано выше, в объявлении функции `hello()`, или же указать его как `-> Void`. В следующем примере кода определяется функция, которая ничего не делает или выводит свой аргумент в зависимости от значения переменной `debug`, но ничего не возвращает.

```
var debug = true // Разрешает отладку
func debugPrint(value: Double) {
    if debug {
        print(value)
    }
}
```

Определение этой функции можно также записать более пространно следующим образом:

```
func debugPrint(value: Double) -> Void {
// Указывать тип "-> Void" возвращаемого значения необязательно
```

Для того чтобы вызвать функцию, достаточно указать ее имя и предоставить соответствующие аргументы в круглых скобках, как показано ниже:

```
let area = areaOfRectangle(width: 20, height: 10)
```

До появления версии Swift 3 можно было пропускать имя первого аргумента, как показано ниже:

```
let area = areaOfRectangle(20, height: 10) // В Swift 3 это больше не допускается
```

Одна из необычных особенностей, присущих только языку Swift, состоит в том, что аргументы функции могут иметь свои имена: **внешнее и внутреннее**. Указывать внешнее имя совсем необязательно. Если внешнее имя аргумента не предоставляется, то оно считается таким же, как и внутреннее имя аргумента. Ниже показан еще один способ объявления функции `areaOfRectangle()`. На этот раз указываются как внешние, так и внутренние имена аргументов.

```
func areaOfRectangle(width w: Double, height h: Double) -> Double {
    return w * h
}
```

Если в объявлении функции указывается как внешнее, так и внутреннее имя аргумента, то первым следует внешнее имя. Если указать только одно имя аргумента, то оно будет считаться его внутренним именем, а его внешнее имя будет отсутствовать. Обратите внимание на то, что в теле данной функции используются внутренние (`w` и `h`), а не внешние (`width` и `height`) имена аргументов. Так зачем же нужны внешние имена аргументов? Отчасти это объясняется условными обозначениями имен, принятыми в методах языка Objective-C, поскольку имена этих методов должны быть сопоставлены компилятором Swift таким образом,

856 ПРИЛОЖЕНИЕ 2: ВВЕДЕНИЕ В ЯЗЫК SWIFT

чтобы их можно было вызывать из кода, написанного на языке Swift. Мы еще вернемся к данному вопросу, когда будем рассматривать классы Swift далее в этом приложении. Единственное преимущество, которое в данном примере дает дополнительное указание внешних имен аргументов, состоит в следующем: если эти имена существуют, то они должны быть предоставлены при вызове функции, и благодаря этому код оказывается более удобочитаемым, как показано ниже.

```
let area = areaOfRectangle(width: 20, height: 10)
// Теперь указывать внешние имена аргументов "width" и "height" обязательно
```

Для того чтобы предоставить значение аргумента по умолчанию, его достаточно включить в список аргументов функции. В следующей функции исходная символьная строка разбивается на составляющие, разграничиваемые заданным разделителем (по умолчанию — пробелом):

```
func separateWords(str: String, delimiter: String = " ") -> [String] {
    return str.components(separatedBy: delimiter)
}
```

Эту функцию можно вызвать, не предоставляя разделитель явным образом. И в этом случае в качестве разделителя употребляется пробел, как показано ниже.

```
let result = separateWords(str: "One small step")
print(result) // [One, small, step]
```

В языке Swift для аргумента функции со значением по умолчанию автоматически предоставляется внешнее имя, которое оказывается таким же, как и внутреннее имя, хотя это положение можно изменить, предоставив свое внешнее имя. Это означает, что имя аргумента должно быть указано, если отсутствует его значение по умолчанию, как показано ниже.

```
let result = separateWords(str: "One. Two. Three",
                           delimiter: ". ") // Нужен разделитель
print(result) // [One, Two, Three]
```

Если требуется вообще избавить пользователя от необходимости предоставлять имя аргумента, то внешнее имя аргумента можно обозначить знаком `_`, как показано ниже. Хотя делать это не рекомендуется, чтобы не затруднять чтение исходного кода.

```
func separateWords(str: String, _ delimiter: String = " ") -> [String] {
    return str.components(separatedBy: delimiter)
}
let result = separateWords(str: "One. Two. Three", ". ") // Нужен разделитель
print(result) // [One, Two, Three]
```

Эту возможность можно использовать, чтобы изменить функцию `areaOfRectangle()` так, чтобы ей не требовались имена никаких аргументов:

```
func areaOfRectangle(_ w: Double, _ h: Double) -> Double {
    return w * h
}
let area = areaOfRectangle(20, 10)
```

Обратите внимание на то, что перед первым аргументом должен стоять символ `_`, если вы сначала хотите использовать неименованный аргумент.

```
func areaOfRectangle(_ w: Double, _ h: Double) -> Double {
    return w * h
}
```

В языке Swift функции относятся к типам данных, а следовательно, можно создать переменную типа функции, присвоить ей ссылку на функцию и воспользоваться этой переменной, чтобы вызвать функцию. Аналогично одну функцию можно передать в качестве аргумента другой функции или возвратить одну функцию из другой.

Для того чтобы объявить переменную типа функции, в качестве ее типа следует указать сигнатуру данной функции. Такая сигнатура состоит из типов аргументов функции (указываемых в круглых скобках, если их несколько, а иначе круглые скобки можно опустить), знака `->` и возвращаемого типа. В следующем примере кода создается переменная, в которой можно хранить ссылку на функцию, выполняющую неуказанный операцию над числовым значением типа `Double` и возвращающую другое числовое значение типа `Double`:

```
var operation: (Double) -> Double
```

В данном случае круглые скобки можно опустить, поскольку функция имеет единственный аргумент, как показано ниже.

```
var operation: Double -> Double // В Swift 3 больше не допускается
```

Остается лишь написать функции, оперирующие числовыми значениями типа `Double`, а затем присвоить любую из этих функций переменной, как показано в следующем примере кода:

```
func double(number: Double) -> Double {
    return 2 * number
}
operation = double
```

Теперь функцию можно вызвать с помощью переменной типа этой функции, предоставив аргумент таким же образом, как и при непосредственном вызове функции:

```
operation(2) // Результат равен 4
```

Используя то же самое вызывающее выражение, можно сделать и так, чтобы переменная `operation` ссылалась на другую функцию и выполняла иную операцию, как показано ниже.

```
func quadrupleMe(number: Double) -> Double {
    return 4 * number
}
operation = quadrupleMe
operation(2) // Результат равен 8
```

Возможность передавать функции подобным способом оказывается очень удобной. Так, в стандартной библиотеке Swift имеется несколько функций, принимающих аргументы типа функции. К их числу относится метод `sorted()`, сортирующий упорядоченную коллекцию в соответствии с порядком, заданным в качестве аргумента, полученного от другой функции. Функция, задающая критерий сортировки, должна принимать в качестве аргументов два элемента сортируемой коллекции и возвращать логическое значение `true`, если первый аргумент оказывается меньше второго аргумента, а иначе — логическое значение `false`. Ниже приведен пример такой функции, сравнивающей два числовых значения типа `Int`.

```
func compareInts(_ first: Int, _ second: Int) -> Bool {
    return first < second
}
```

Теперь можно создать массив числовых значений типа `Int` и вызвать метод `sorted()` и `compareInts()` для его сортировки. Для этого достаточно передать функцию `compareInts()` в качестве аргумента метода `sorted()`, как показано ниже.

```
var values = [12, 3, 5, -4, 16, 18]
let sortedValues = values.sorted(isOrderedBefore: compareInts)
sortedValues // Результат: [-4, 3, 5, 12, 16, 18]
```

Функция `sorted()` возвращает отсортированную копию массива. Она аналогична функции `sort()`, упорядочивающей исходную коллекцию в порядке возрастания.

```
let values = [12, 3, 5, -4, 16, 18]
values.sort(compareInts)
values // Результат: [-4, 3, 5, 12, 16, 18]
```

В языке Swift существует возможность определять функцию сравнения непосредственно в списке аргументов метода `sorted()`. Это значит, что нет необходимости отдельно определять такую функцию и присваивать ей имя, которое иначе не используется. В приведенном ниже примере кода показано, как отсортировать подобным способом список значений.

```
var values = [12, 3, 5, -4, 16, 18]
let sortedValues = values.sorted(isOrderedBefore:
    {(first: Int, second: Int) -> Bool in
        return first < second
})
```

На первый взгляд, такой синтаксис кажется довольно сложным, поэтому рассмотрим его по частям. Функция сравнения должна быть заключена в фигурные скобки, где сначала указывается список аргументов данной функции и возвращаемое ею значение, а затем — ключевое слово `in`, отделяющее их от тела функции, как показано ниже.

```
{(first: Int, second: Int) -> Bool in
```

Далее следует тело функции, которое остается таким же, как и в отдельно приведенном ранее определении функции `compareInts()`, а после этого — фигурная скобка, закрывающая определение функции. В заключение следует круглая скобка, закрывающая список аргументов метода `sorted()`, как показано ниже.

```
return first < second
})
```

Такая определяемая на месте анонимная функция называется **замыканием**. Освоившись с замыканиями, вы, вероятнее всего, будете часто ими пользоваться. Если замыкание оказывается последним аргументом функции, то синтаксис определения функции можно немного упростить, вынеся замыкание за пределы списка аргументов функции следующим образом:

```
let sortedValues = values.sorted() {
    // Замыкание вынесено за скобки
    (first: Int, second: Int) -> Bool in
    return first < second
}
```

Тем не менее код по-прежнему остается довольно громоздким. Правда, его можно немного упростить. Компилятор языка Swift в состоянии определить, что замыканию требуются два аргумента типа `Int` и оно должно возвратить логическое значение. Об этом компилятору становится известно по тому, как функция `sorted()` определяется в стандартной библиотеке. И благодаря этому можно опустить типы аргументов и круглые скобки, в которые заключаются имена аргументов и возвращаемый тип, оставив только следующее:

```
let sortedValues = values.sorted() {
    first, second in      // Типы аргументов и возвращаемый тип
    // выводятся в языке Swift автоматически!
    return first < second
}
```

Теперь код выглядит намного лучше. Но его можно упростить еще больше, опустив также имена аргументов. Компилятору языка Swift известно о двух аргументах функции, и, если не указать их имена, они будут называться `$0` и `$1` (`$2`, `$3` и т.д.). Благодаря этому определение замыкания можно свести к единственной строке кода, как показано ниже.

```
let sorted = values.sorted() { return $0 < $1 }
```

Но и это еще не все. В языке Swift можно опускать ключевое слово `return`. В конечном итоге получается следующая строка кода:

```
let sortedValues = values.sorted() { $0 < $1 }
```

Согласитесь, что такое определение замыкания оказывается намного более выразительным, чем первоначальное, по крайней мере оно таким представляется тем, кто привык читать исходный код, написанный кратким синтаксисом.

Поэкспериментируйте с синтаксисом замыканий на игровой площадке, опробовав разные его варианты, чтобы выяснить, какие из них работоспособны и какие неработоспособны.

Замыкания называются так потому, что они “замыкают” переменные, создаваемые в области их действия. Это означает, что в замыканиях можно выполнять чтение и запись данных в таких переменных. Оценить, насколько это будет удобно без подходящего контекста, не так-то просто. В главах данной книги приведено немало примеров, иллюстрирующих это положение. В приведенном ниже примере демонстрируется замыкание, в котором применяется значение, определенное за пределами этого замыкания.

```
func getInterestCalculator(rate: Double) -> (Double, Int) -> Double {
    let calculator = {
        (amount: Double, years: Int) ->
            Double in rate * amount * Double(years)
    }
    return calculator
}
```

Как следует из сигнатуры функции `getInterestCalculator()`, ей требуется аргумент типа `Double`. Она возвращает другую функцию с двумя аргументами типа `Double` и `Int` и возвращаемым значением типа `Double`.

В данном примере функции `getInterestCalculator()` передается величина процентной ставки и возвращается функция, вычисляющая простейший процентный доход по заданной ставке.

В теле функции `getInterestCalculator()` создается замыкание, которое присваивается переменной `calculator` следующим образом:

```
let calculator = {
    (amount: Double, years: Int) ->
        Double in rate * amount * Double(years)
}
```

Как видите, функции замыкания требуется сумма в виде аргумента `amount` типа `Double`, а также период, в течение которого будет рассчитываться процентный доход, в виде аргумента `years` типа `Int`. Обратите внимание на то, что в теле замыкания используется аргумент `rate`, передаваемый функции `getInterestCalculator()`. В заключение замыкание возвращается коду, вызывающему функцию `getInterestCalculator()`.

Ниже приведен пример кода, в котором сначала вызывается функция `getInterestCalculator()`, а затем возвращаемая ею функция.

```
let calculator = getInterestCalculator(rate: 0.05)
calculator(100.0, 2)      // Результат равен 10: процентный доход при
                        // ставке 5% от суммы 100 долларов за 2 года
```

Возвращаемая функция сначала присваивается константе `calculator`, а затем вызывается для расчета процентного дохода от суммы 100 долларов за два года при процентной ставке 5%. Рассмотрим, что же интересного при этом

происходит. Возвращаемое замыкание ссылается на значение аргумента `rate` функции `getInterestCalculator()`, но это делается после возврата из данной функции. Как же это возможно? Это возможно потому, что аргумент функции *фиксируется* в замыкании по ссылке на него. Когда переменная фиксируется в замыкании, как только оно создается, выбирается копия ее значения, которая затем используется при выполнении замыкания. Именно таким образом величина процентной ставки, передаваемая в виде аргумента функции `getInterestCalculator()`, оказывается доступной замыканию после того, как сам аргумент перестает существовать.

Обработка ошибок

Просматривая страницу документации среды Xcode, посвященную классу `NSString`, вы увидите объявления нескольких методов, содержащие слово `throws`:

```
init(contentsOfFile path: String, encoding enc: UInt) throws
```

Это значит, что данный метод может либо проигнорировать выбранную вами ошибку, либо обработать ее. В первую очередь, в документации описывается, как проигнорировать возникшую ошибку, а затем показано, как ее перехватить и обработать. Попробуем проигнорировать ошибку, не обращая внимания на слово `throws`.

```
let s = String(contentsOfFile: "XX", encoding: String.Encoding.utf8)
```

Если выполнить эту команду на игровой площадке, то вы увидите сообщение об ошибке “Call can throw but is not marked with ‘try’” (“Вызов метода может генерировать исключение, но не добавлен в раздел `try`”). Существуют три варианта раздела `try`, два из которых позволяют игнорировать ошибку, а третий используется, когда вы собираетесь ее перехватывать. Продолжим игнорировать потенциальную ошибку.

```
let s = try? String(contentsOfFile: "XX", encoding: String.Encoding.utf8)
```

Это приводит к ошибке компиляции. Игровая площадка выполняет эту инструкцию и показывает, что результат равен `nil`. Используя инструкцию `try?`, вы преобразуете тип возвращаемого методом объекта в необязательный тип. Поскольку этот метод, по существу, является инициализатором класса `String` (который класс `String` на самом деле получает от класса `NSString`), тип возвращаемого значения преобразуется в тип `String?`. Именно такой тип получает переменная `s`. В данном случае ее значение равно `nil`, потому что файла `XX` на самом деле нет.

Раздел `try?` можно использовать в операторе `if`, чтобы делать что-то только при условии, что файл `XX` существует и его содержимое можно успешно загрузить.

```
if let s = try? String(contentsOfFile: "XX", encoding: String.Encoding.utf8) {
    print("Content loaded")
```

```

} else {
    print("Failed to load contents of file")
}

```

Если вы уверены, что файл, который вы пытаетесь прочитать, действительно существует и его содержимое можно прочитать, то можно заменить инструкцию `try?` инструкцией `try!`:

```
let s = try! String(contentsOfFile: "XX", encoding: String.Encoding.utf8)
```

Несмотря на то что это вполне допустимая конструкция, на практике применять ее не рекомендуется, потому что ваше приложение может потерпеть крах, если что-то пойдет неправильно, — вы можете увидеть это, попытавшись выполнить эту инструкцию на игровой площадке.

Перехват ошибок

Если использовать конструкцию `try?` как часть инструкции `let`, то можно увидеть, правильно ли была выполнена операция, и предпринять определенные меры, если операция была выполнена неправильно. Инструкция `try?` неявно сообщает, что возникла ошибка, которая заключается в том, что возвращаемое значение равно `nil`, но при этом нет никаких средств для того, чтобы сообщить об этом. Для этого необходимо применить инструкцию `try` и включить ее в блок `do-catch`:

```

do {
let s = try String(contentsOfFile: "XX", encoding: String.Encoding.utf8)
    print("Loaded content \(s)")
} catch {
    print(error)
}

```

Инструкции внутри блока `do` выполняются до конца или до момента, когда будет сгенерирована ошибка. В этот момент управление передается блоку `catch`. Инициализатор `init(contentsOfFile:encoding:)` генерирует значение типа `NSError`. Этот класс относится к обобщенным типам ошибок в каркасе Foundation. Как показано в предыдущем примере, внутри блока `block` можно выяснить значение переменной `error`.

Генерирование ошибок

Ошибка может генерировать любая функция или метод, при условии, что определения этой функции или метода содержат ключевое слово `throws`. Функция или метод может генерировать значение любого типа, который соответствует протоколу `ErrorType`, как, например, класс `NSError`: фактически, если проверить выведенный тип переменной `error` в предыдущем примере, то окажется, что она имеет тип `ErrorType`. Ошибки, генерируемые методами из каркаса Foundation (и других каркасов компании Apple), имеют тип `NSError`, но пользователь может определять свои типы ошибок в собственном коде. Обычно

для этого используются перечисления, потому что они удобны для представления ошибок и даже позволяют включать дополнительную информацию об ошибках.

Например, напишем функцию, вычисляющую длину третьей стороны треугольника, по длинам остальных двух сторон и углу между ними. Мы покажем, что значения аргументов, соответствующие сторонам треугольника, должны быть положительными, а угол должен быть от 0 и до π радиан. Сначала напишем функцию, не предусматривающую проверку ошибок.

```
func calcThirdSide(_ side1: Double, side2: Double, angle: Double) -> Double {
    return sqrt(side1 * side1 + side2 * side2 - 2 * side1 * side2 * cos(angle))
}
```

Для того чтобы вычислить длину третьей стороны, мы используем правило косинуса. Убедимся, что эта реализация является правильной, попробовав вычислить длину стороны прямоугольного треугольника со сторонами, длины которых равны 3 и 4:

```
let side3 = calcThirdSide(3, side2: 4, angle: M_PI/2)
print(side3)
```

По теореме Пифагора длина третьей стороны должна быть равной 5, и именно этот результат мы видим на экране, выполнив этот код. Добавим проверку ошибок. Мы должны предусмотреть два вида условий — для длин и для угла. Для этого естественно применить перечисления:

```
enum TriangleError : ErrorProtocol {
    case SideInvalid(reason: String)
    case AngleInvalid(reason: String)
}
```

Поскольку мы собираемся генерировать экземпляры этого перечисления, оно должно подчиняться протоколу `ErrorProtocol`. Мы определили класс для каждого типа ошибки, предусмотрев соответствующее сообщение, поэтому с каждым условием связана определенная строка. Перед тем как добавить проверку ошибок в нашу функцию, мы должны сообщить компилятору, что она может генерировать ошибку. Для этого добавим в определение функции ключевое слово `throws`. Это ключевое слово должно стоять после списка аргументов и перед типом возвращаемого значения (если оно есть):

```
func calcThirdSide(_ side1: Double, side2: Double, angle: Double) throws ->
    Double {
    return sqrt(side1 * side1 + side2 * side2 - 2 * side1 * side2 * cos(angle))
}
```

Поскольку мы добавили ключевое слово `throws`, компилятор сообщит нам, что мы не включили вызов функции в раздел `try`. Исправим этот недостаток, добавив код для перехвата ошибки:

864 ПРИЛОЖЕНИЕ # ВВЕДЕНИЕ В ЯЗЫК SWIFT

```
do {
    let side3 = try calcThirdSide( 3, side2: 4, angle: M_PI/2)
    print(side3)
} catch {
    print(error)
}
```

Теперь можно начинать добавлять проверки ошибок в функцию `calcThirdSide(side1:side2:angle:)`. Изменим определение функции следующим образом:

```
func calcThirdSide(_ side1: Double, side2: Double, angle: Double) throws ->
    Double { if side1 <= 0 {
        throw TriangleError.SideInvalid(reason: "Side 1 must be >= 0,
                                                not \(side1)")
    }
    return sqrt(side1 * side1 + side2 * side2 - 2 * side1 * side2 * cos(angle))
}
```

Инструкция `if` проверяет, равно ли значение аргумента `side1` нулю или оно меньше нуля. Если да, она использует ключевое слово `throw`, чтобы сгенерировать экземпляр перечисления `TriangleError` с соответствующим сообщением. Для того чтобы проверить, как это работает, изменим значение аргумента `side1` в тестируемом коде с 3 на -1:

```
let side3 = try calcThirdSide(-1, side2: 4, angle: M_PI/2)
```

Мы должны увидеть в области результатов игровой площадки текст `Side Invalid("Side 1 must be >= 0, not -1.0")`, который свидетельствует о том, что была сгенерирована и перехвачена соответствующая ошибка.

Теперь добавим в функцию `calcThirdSide(side1:side2:angle:)` остальные три проверки ошибок:

```
func calcThirdSide(_ side1: Double, side2: Double, angle: Double) throws ->
    Double { if side1 <= 0 {
        throw TriangleError.SideInvalid(reason: "Side 1 must be >= 0,
                                                not \(side1)")
    }
    if side2 <= 0 {
        throw TriangleError.SideInvalid(reason: "Side 2 must be >= 0,
                                                not \(side2)")
    }
    if angle < 0 {
        throw TriangleError.AngleInvalid(reason: "Angle must be >= 0,
                                                not \(angle)")
    }
    if angle >= M_PI {
        throw TriangleError.AngleInvalid(reason: "Angle must be <= π,
                                                not \(angle)")
    }
    return sqrt(side1 * side1 + side2 * side2 - 2 * side1 * side2 * cos(angle))
}
```

Проверьте, что при передаче функции calcThirdSide(side1:side2:angle:) неправильных аргументов генерируемые и перехват ошибок выполняются корректно.

Инструкция *guard*

Довольно часто проверку аргументов функции выполняют в самом ее начале. Язык Swift предоставляет еще одну возможность для проверки ошибок — вместо инструкции *if* можно использовать инструкцию *guard*. Перепишем функцию calcThirdSide(side1:side2:angle:), используя ключевое слово *guard* вместо *if*:

```
func calcThirdSide(_ side1: Double, side2: Double, angle: Double)
    throws -> Double {
    guard side1 > 0 else {
        throw TriangleError.SideInvalid(reason: "Side 1 must be >= 0,
                                                not \(side1)")
    }

    guard side2 > 0 else {
        throw TriangleError.SideInvalid(reason: "Side 2 must be >= 0,
                                                not \(side2)")
    }

    guard angle >= 0 else {
        throw TriangleError.AngleInvalid(reason: "Angle must be >= 0,
                                                not \(angle)")
    }

    guard angle < M_PI else {
        throw TriangleError.AngleInvalid(reason: "Angle must be <= p,
                                                not \(angle)")
    }

    return sqrt(side1 * side1 + side2 * side2 - 2 * side1 * side2 * cos(angle))
}
```

Тело инструкции *guard* должно предшествовать ключевое слово *else*. Это тело выполняется, только если проверяемое условие нарушается. Инструкцию *guard* можно интерпретировать так: “Если условие не выполняется, то выполнить тело инструкции”. По этой причине смысл проверки должен быть противоположным условию инструкции *if*; например, условие, проверяемое в начале инструкции *if*, выглядело так: *side1 <= 0*, а условие соответствующей инструкции *guard* так: *side1 > 0*. Сначала это может показаться странным, но потом вы убедитесь, что использование инструкции *guard* делает код яснее, поскольку он начинается с проверки условий, которые должны выполняться до его выполнения, а не наоборот.

Общая форма инструкции *guard* выглядит следующим образом:

```
guard условное_выражение else {
// Инструкции тела guard
// Передача управления за пределы видимости инструкции guard
}
```

```
// Если тело инструкции guard было выполнено, то
// управление сюда попасть не должно
```

При необходимости тело инструкции `guard` может содержать несколько разных инструкций, если выполнено одно условие: в конце блока, содержащегося в теле `guard`, управление должно передаваться за пределы ближайшей области видимости инструкции `guard`. В предыдущем примере инструкция `guard` находилась в области видимости функции `calcThirdSide(side1:s side2:angle:)`, поэтому в конце блока `guard` управление должно передаваться за пределы этой функции.

Итак, мы умеем генерировать ошибки, но мы можем также возвращать значения из функции. Если блок `guard` содержится в теле инструкции `for`, то ее ближайшей областью видимости является инструкция `for`, так что для передачи управления можно использовать либо инструкцию `continue`, либо инструкцию `break`. Это правило относится и к инструкциям `repeat` и `while`. Если позволить потоку выполнения программы достичь конца блока `guard` и перейти к следующим инструкциям, то возникнет ошибка компиляции.

Дополнительная информация о перехвате ошибок

До сих пор мы использовали отдельный обобщенный блок `catch` для перехвата ошибок и выдачи сообщений из функции `calcThirdSide(side1:side2:angle:)`. Однако инструкция `do-catch` может иметь несколько блоков `catch`, причем каждый блок `catch` может содержать выражение, соответствующее перехватываемой ошибке. Если ошибке не соответствует ни один блок `catch`, то выполняется блок без выражений; если такого блока нет, то ошибка передается в точку вызова функции (или метода), содержащей инструкцию `do-catch`, и тогда эта функция (или метод) должна определять, что должно быть выполнено, включая соответствующий раздел `throws` в свое определение. Рассмотрим несколько примеров.

Изменим блок `do-catch` в игровой площадке следующим образом:

```
do {
    let side3 = try calcThirdSide( -1, side2: 4, angle: M_PI/2)
    print(side3)
} catch let e as TriangleError {
    print(e)
}
```

Мы добавили выражение в блок `catch`, которое означает, что этот блок перехватывает исключительно ошибки типа `TriangleError`. Сама ошибка присвоена переменной `e`, область видимости которой ограничена только блоком `catch`. Поскольку функция `calcThirdSide(side1:side2:angle:)` генерирует только ошибки указанного типа, нам не нужны другие блоки `catch`. Если бы мы захотели обрабатывать конкретные типы ошибок по-разному, то мы просто добавили бы больше разделов `catch`. Рассмотрим пример.

```

do {
    let side3 = try calcThirdSide( -1, side2: 4, angle: M_PI/2)
    print(side3)
} catch TriangleError.SideInvalid(let reason) {
    print("Caught invalid side: \(reason)")
} catch {
    print("Caught \(error)")
}

```

Первый блок `catch` обрабатывает ошибки типа `SideInvalid` и присваивает сообщение об ошибке из перечисления переменной `reason`, чтобы использовать в теле блока `catch`. Второй блок `catch` не имеет выражения, поэтому перехватывает любые ошибки, которые не были перехвачены предыдущими блоками. В данном случае он перехватывает ошибку типа `AngleInvalid`. Если блок `catch` перехватывает все ошибки, как в данном случае, то он должен стоять последним. Таким образом, следующий код является неправильным:

```

do {
    let side3 = try calcThirdSide(-1, side2: 4, angle: M_PI/2)
    print(side3)
} catch { // Invalid - this must be the last catch block
    print("Caught: \(error)")
} catch TriangleError.SideInvalid(let reason) {
    print("Caught invalid side: \(reason)")
}

```

Для того чтобы обработать каждую ошибку по-отдельности, необходимо предусмотреть для каждой из них свой блок `catch`:

```

do {
    let side3 = try calcThirdSide(-1, side2: 4, angle: -M_PI/2)
    print(side3)
} catch TriangleError.AngleInvalid(let reason) {
    print("Caught invalid angle: \(reason)")
} catch TriangleError.SideInvalid(let reason) {
    print("Caught invalid side: \(reason)")
}

```

Поскольку мы явно обработали оба типа возможных ошибок, которые может генерировать функция `calcThirdSide(side1:side2:angle:)`, в блоке `catch`, перехватывающем все ошибки, нет надобности.

Классы и структуры

В языке Swift предоставляются два разных способа создания специальных типов данных в виде классов и структур, которые могут содержать свойства, инициализаторы и методы. Свойство представляет собой переменную, определенную в теле класса или структуры; метод — функцию, определяемую в классе или структуре; а инициализатор — разновидность метода, специально предназначенного для установки в первоначальное состояние создаваемого экземпляра класса или структуры.

Структуры

Ниже приведен пример структуры, представляющей окружность заданного радиуса. У этой структурой имеется одно свойство для хранения радиуса окружности, а также два метода, возвращающих площадь и длину окружности.

```
struct CircleStruct {
    var radius: Double

    func getArea() -> Double {
        return M_PI * radius * radius
    }

    func getCircumference() -> Double {
        return 2 * M_PI * radius
    }
}
```

Для структуры в языке Swift автоматически создается инициализатор, присваивающий значения ее свойствам по передаваемым ему аргументам. В приведенном ниже примере демонстрируется применение такого инициализатора для создания экземпляра структуры `CircleStruct`.

```
var circleStruct = CircleStruct(radius: 10)
```

Сформированный инициализатор структуры наделяется одним аргументом на каждое свойство. Имена аргументов совпадают с именами свойств и указываются в том порядке, в каком свойства определены в самой структуре. Следует иметь в виду, что имена аргументов требуются при использовании инициализатора, поскольку у аргументов инициализатора имеются как внешние, так и внутренние имена.

Как только будет создан экземпляр структуры, появится возможность читать и видоизменять значения свойств, обращаясь к ним по их именам. Так, в следующем примере кода читается и удваивается радиус окружности:

```
var circleStruct = CircleStruct(radius: 10)
let r = circleStruct.radius      // Читает значение из свойства radius -
                                // результат равен 10
circleStruct.radius = 2 * r      // Удваивает радиус
```

Структуры представляют собой объекты-значения, поэтому в приведенном ниже примере кода сначала создается копия исходного объекта типа `CircleStruct`, а затем она присваивается переменной `newCircleStruct`. Изменения, вносимые в свойство `radius` по ссылке из переменной `newCircleStruct`, не оказывают никакого влияния на оригинал.

```
var newCircleStruct = circleStruct      // Копирует структуру
newCircleStruct.radius = 32              // Воздействует только на копию
newCircleStruct.radius                  // Новое значение: 32
circleStruct.radius                     // Прежнее значение: 20
```

Если присвоить структуру константе, все ее свойства становятся доступными только для чтения, как показано ниже.

```
let constantCircleStruct = CircleStruct(radius: 5)
constantCircleStruct.radius = 10      // Неверно: constantCircleStruct
                                         // является константой!
```

В языке Swift требуется, чтобы все свойства структуры (или класса) были инициализированы прежде, чем инициализатор завершит свое выполнение. Установить значения свойств можно в самом инициализаторе или при их определении. Ниже приведена немного другая реализация структуры `CircleStruct`, инициализирующая радиус окружности по умолчанию значением 1, когда она определяется.

```
struct CircleStruct {
    var radius: Double = 1

    init() {}

    init(radius: Double) {
        self.radius = radius
    }

    func getArea() -> Double {
        return M_PI * radius * radius
    }

    func getCircumference() -> Double {
        return 2 * M_PI * radius
    }
}
```

Как видите, у этой структуры теперь имеются два инициализатора (они всегда называются `init`): один не принимает аргументов, другой принимает радиус в качестве своего аргумента. Первый инициализатор позволяет создать экземпляр структуры, используя радиус, устанавливаемый при определении свойства `radius`, как показано ниже.

```
let circleStructDefault = CircleStruct()
circleStructDefault.radius      // Результат равен 1.0
```

Второй инициализатор присваивает значение своего аргумента свойству `radius`, как демонстрируется в следующем примере:

```
init(radius: Double) {
    self.radius = radius
}
```

Вводить инициализатор без аргументов не требуется, поскольку это делается в языке Swift автоматически. Но если инициализаторы вводятся вручную, то придется также ввести инициализатор без аргументов. Обратите внимание на

то, что для присваивания значения свойству используется форма `self.radius`. Переменная `self` представляет экземпляр инициализируемой структуры. При вызове метода она представляет тот экземпляр, для которого вызывается метод. Как правило, уточнять доступ к свойству по ссылке `self` не нужно, но в данном случае это приходится делать, поскольку имя аргумента инициализатора оказывается таким же, как имя свойства.

Как упоминалось ранее, по умолчанию в языке Swift требуется указывать имя аргумента инициализатора, когда создается экземпляр структуры (то же самое относится и к классам, как будет показано ниже). Если определяется собственный инициализатор, то внешнее имя аргумента можно обозначить знаком `_`, как показано в следующем видоизмененном варианте второго инициализатора структуры `CircleStruct`:

```
init(_ radius: Double) {
    self.radius = radius
}
```

С учетом этого изменения инициализатор нужно использовать следующим образом:

```
var circleStruct = CircleStruct(10) // Имя аргумента должно быть пропущено
```

В реализациях методов `getArea()` и `getCircumference()` не внедряется ничего нового. В них просто выполняются простые расчеты, в которых используется значение свойства `radius`. Следует, однако, иметь в виду, что в данном случае чтение значения из свойства `radius` не осуществляется по ссылке `self`, хотя это и можно было бы сделать. Следующая версия метода `getArea()` равнозначна предыдущей:

```
func getArea() -> Double {
    return M_PI * self.radius * self.radius
    // Явное обращение к свойству по ссылке self -
    // не рекомендуется
}
```

Эти методы вызываются аналогично любой функции, за исключением того, что для их вызова требуется экземпляр структуры, как показано ниже.

```
let circleStructDefault = CircleStruct()
circleStructDefault.getArea()           // Возвращает площадь круга
circleStructDefault.getCircumference() // Возвращает длину окружности
```

Классы

Создание класса синтаксически очень похоже на создание структуры, но все же имеет свои отличия, о которых следует знать. В качестве примера ниже приведена реализация окружности в виде класса.

```
class CircleClass {
    var radius: Double = 1
```

```

init() {
}

init(radius: Double) {
    self.radius = radius
}

func getArea() -> Double {
    return M_PI * radius * radius
}

func getCircumference() -> Double {
    return 2 * M_PI * radius
}
}

```

Единственное отличие этой реализации окружности от ее варианта в виде структуры состоит в применении ключевого слова `class` вместо `struct`. Однако в интерпретации инициализаторов классов и структур средствами Swift имеются некоторые отличия, которые состоят в следующем.

- Для установки исходных значений свойств класса в языке Swift создается не специальный, а пустой инициализатор `init()`.
- Если инициализаторы вводятся вручную, то инициализатор `init()` недоступен автоматически, как это имеет место и для структур.

Что касается класса `CircleClass`, то присутствие в нем инициализатора `init(radius: Double)` означает, что его пустой вариант `init()` не будет сформирован средствами Swift, и поэтому его пришлось ввести вручную.

Для создания экземпляра класса `CircleClass` в зависимости от того, требуется ли установить радиус окружности, можно воспользоваться любым из этих инициализаторов.

```

var circleClassDefault = CircleClass()           // Установка радиуса
                                                // по умолчанию
circleClassDefault.radius                      // Результат равен 1
var circleClass = CircleClass(radius: 10)        // Явная установка радиуса
circleClass.radius                            // Результат равен 10

```

Классы не являются объектами-значениями, и поэтому присваивание экземпляра класса переменной или его передача функции не влечет за собой создание копий этого экземпляра, как демонстрируется в следующем примере кода:

```

var newCircleClass = circleClass    // Без копирования экземпляра
newCircleClass.radius = 32          // Только одна копия, поэтому
                                    // такое изменение доступно по
newCircleClass.radius              // обеим ссылкам. Результат равен 32
circleClass.radius                // Результат равен 32

```

Свойства

Свойство `radius` окружности называется *сохраняемым*, поскольку его значение сохраняется средствами класса или структуры. Допускаются также *вычисляемые свойства*, которые не сохраняются. Вместо этого значение вычисляемого свойства рассчитывается всякий раз, когда оно читается. Так, площадь круга и длина окружности могут считаться свойствами, и поэтому их было бы благородно реализовать как вычисляемые свойства. Ниже показано, как это делается в классе `CircleClass` (аналогичным образом они реализуются и в структуре `CircleStruct`).

```
class CircleClass {
    var radius: Double = 1
    var area: Double {
        return M_PI * radius * radius
    }

    var circumference: Double {
        return 2 * M_PI * radius
    }

    init() {}

    init(radius: Double) {
        self.radius = radius
    }
}
```

Синтаксис для создания свойств предельно прост: достаточно указать имя свойства и его тип, что обязательно для вычисляемых свойств, а затем блок кода, требующегося для расчета значения свойства. Если площадь круга и длина окружности реализованы как свойства, то их значения можно получить более простым способом, чем вызов метода, как показано ниже.

```
let circleClass = CircleClass(radius: 10)
circleClass.area
circleClass.circumference
```

По существу, это сокращенная реализация рассматриваемых здесь свойств. Полная форма, например, свойства `area` будет выглядеть следующим образом:

```
var area: Double {
    get {
        return M_PI * radius * radius
    }
}
```

Эта форма отличается ключевым словом `get` и дополнительными фигурными скобками, в которые заключается код для расчета значения свойства. Поскольку свойство `area` доступно только для чтения, для него достаточно реализовать только метод получения, опустив реализацию метода установки.

Вполне возможно счесть свойство `area` устанавливаемым, хотя при установке новой площади круга можно рассчитать и сохранить соответствующий радиус окружности. То же самое относится и к длине окружности. Для того чтобы сделать свойство устанавливаемым, нужно ввести блок кода `set` и восстановить блок кода `get`, если он опущен. Ниже показано, каким образом реализуется метод установки свойств `area` и `circumference`.

```
var area: Double {
    get {
        return M_PI * radius * radius
    }

    set {
        radius = sqrt(newValue/M_PI)
    }
}

var circumference: Double {
    get {
        return 2 * M_PI * radius
    }

    set {
        radius = newValue/(2 * M_PI)
    }
}
```

В реализации методов установки используется значение, устанавливаемое в переменной `newValue`. Если это неподходящее имя, его можно заменить другим, указав его после ключевого слова `set`, как показано ниже.

```
set (value) {
    radius = value/(2 * M_PI)
}
```

Теперь, когда рассматриваемые здесь свойства сделаны устанавливаемыми, с их помощью можно определить радиус по длине окружности или площади круга следующим образом:

```
circleClass.area = 314
circleClass.radius           // Если площадь круга окружности – 314, то
                            // радиус окружности равен 9.997
circleClass.circumference = 100
circleClass.radius           // Если длина окружности равна 100, то
                            // радиус окружности равен 15.915
```

В языке Swift предусмотрена возможность вводить код для наблюдения за изменениями в сохраняемых (но не вычисляемых) свойствах. Для этого достаточно ввести блок кода `willSet` или `didSet` (или же оба вместе) в определение свойства. Эти блоки кода вызываются всякий раз, когда устанавливается значение свойства, даже если новые и прежние значения одинаковы, кроме тех случаев, когда свойство инициализируется. В частности, блок кода `willSet`

вызывается перед присваиванием свойству нового значения, а блок кода `didSet` — после такого присваивания. Как правило, наблюдатель свойств служит для гарантии того, что свойствам могут быть присвоены только достоверные значения. Ниже показано, как видоизменить свойство `radius` в классе `CircleClass`, чтобы ему присваивались только положительные значения.

```
class CircleClass {
    var radius: Double = 1 {
        didSet {
            if (radius < 0) {
                radius = oldValue
            }
        }
    }
}
```

Когда вызывается блок кода `didSet`, новое значение уже хранится в свойстве `radius`. Если оно отрицательно, то восстанавливается предыдущее значение, которое можно получить из переменной `oldValue`. Как и в блоках установки свойств, имя этой переменной можно при желании изменить. Установка значения свойства в блоке `didSet` не приводит к повторному вызову блоков наблюдателей свойств. С учетом этих изменений любая попытка установить отрицательный радиус окружности будет проигнорирована, как показано ниже.

```
circleClass.radius = 10      // Верно: радиус установлен равным 10
circleClass.radius          // Результат: 10.0
circleClass.radius = -1     // Неверно: перехвачено в блоке кода didSet
circleClass.radius          // Результат: 10.0
```

Методы

Как было показано выше, методы из классов и структур очень похожи на функции, за исключением того, что они имеют доступ к неявно определенному значению `self`, которое ссылается на класс (или структуру), из которого вызывается метод. Методы, определенные в классе, могут изменять значения своих свойств, а методы, определенные в структуре, — нет. Покажем, как это происходит на практике, введя в класс `CircleClass` новый метод `adjustRadiusByAmount(_:_times)` для коррекции радиуса окружности на заданную величину в указанное количество раз. Ниже приведено определение этого метода.

```
func adjustRadiusByAmount(amount: Double, times: Int = 1) {
    radius += amount * Double(times)
}
```

При таком определении данный метод можно вызывать следующим образом:

```
var circleClass = CircleClass(radius: 10)
circleClass.radius          // Результат: 10
circleClass.adjustRadiusByAmount(5, times: 3)
circleClass.radius          // Результат = 10 + 3 * 5 = 25
circleClass.adjustRadiusByAmount(5)      // По умолчанию множитель равен 1
circleClass.radius          // Результат = 30
```

Теперь попробуем включить этот же метод в структуру CircleStruct.

```
struct CircleStruct {
    var radius: Double = 1

    init() {}

    init(radius: Double) {
        self.radius = radius
    }

    func getArea() -> Double {
        return M_PI * radius * radius
    }

    func getCircumference() -> Double {
        return 2 * M_PI * radius
    }

    func adjustRadiusBy(amount: Double, times: Int = 1) {
        radius += amount * Double(times)
    }
}
```

К сожалению, приведенный выше фрагмент кода не подлежит компиляции. По умолчанию методы из структуры не могут изменять ее состояние, поскольку объекты-значения, к которым относятся структуры, рекомендуется при всякой возможности делать неизменяемыми. Если в структуру все же требуется ввести модифицирующий метод, его объявление следует предварить ключевым словом `mutating`.

```
mutating func adjustRadiusBy(amount: Double, times: Int = 1) {
    radius += amount * Double(times)
}
```

Связывание необязательных типов в цепочку

Как же вызвать метод или обратиться к свойству необязательного типа? Прежде всего, его придется извлечь из оболочки необязательного типа, но делать это нужно очень осторожно. Ведь, как вам уже известно, попытка извлечь пустое значение `nil` из необязательного типа приведет к аварийному завершению приложения. Допустим, что имеется следующая переменная и требуется получить радиус любой окружности, на которую она указывает:

```
var optionalCircle: CircleClass?
```

Значение переменной `optionalCircle` может быть и пустым (`nil`, как в данном случае), и поэтому следующее обращение к свойству `radius` недопустимо:

```
optionalCircle!.radius // АВАРИЙНЫЙ СБОЙ!
```

Вместо этого придется проверить, содержит ли переменная optionalCircle пустое значение nil, прежде чем извлекать его из оболочки необязательного типа, как показано ниже.

```
if optionalCircle != nil {
    optionalCircle!.radius
}
```

Такой код безопасный, хотя избыточный. Но имеется и более совершенный способ, называемый **связыванием необязательных типов в цепочку**, который состоит в применении знака ! вместо знака ? для извлечения значений из оболочки необязательных типов. Используя связывание необязательных типов в цепочку, приведенный выше фрагмент кода можно сократить до одной строки, как показано ниже.

```
var optionalCircle: CircleClass?
var radius = optionalCircle?.radius
```

Если используется связывание необязательных типов в цепочку, то свойство оказывается доступным только в том случае, если значение переменной optionalCircle не является пустым (nil). Поскольку значение для присваивания переменной radius может отсутствовать, его тип выводится средствами Swift как Double?, а не Double. Обычно эта возможность используется в конструкции if let.

```
var optionalCircle: CircleClass?
if let radius = optionalCircle?.radius {
    print("radius = \(radius)")
}
```

Эффективность этого способа по-настоящему проявляется в ситуациях, когда одни объекты вложены в другие, в которых ссылки на вложенные объекты имеют необязательный тип. Для того чтобы проследить цепочку объектов до целевого значения, необходимо проверить все ссылки на равенство со значением nil. При связывании необязательных типов в цепочку компилятор Swift выполняет эту работу автоматически.

```
outerObject?.innerObject?.property
```

Связывание необязательных типов в цепочку распространяется и на вызовы методов, как показано ниже.

```
var optionalCircle: CircleClass?
optionalCircle?.adjustRadiusBy(5, times: 3)
```

Вызов метода происходит лишь в том случае, если значение переменной optionalCircle не является пустым (nil), а иначе этот вызов пропускается.

Создание производных классов и наследование

Итак, мы получили очень полезный класс, представляющий окружность. Но что если требуется наделить его какими-то новыми функциональными возможностями? Допустим, что в этот класс требуется ввести свойство цвета (color). Это свойство можно было бы просто ввести в уже имеющееся определение класса. Но что если это не наш собственный, а принадлежащий стороннему каркасу класс CircleClass, исходный код которого нельзя видоизменять? Во многих подобных случаях класс можно расширить. Благодаря расширению функциональные возможности можно ввести в любой класс (даже в сторонние классы), и зачастую это самый верный способ добиться нужного результата. Но в данном случае это не самый верный способ, поскольку требуется ввести сохраняемое свойство, чего нельзя сделать с помощью расширения, как станет ясно при обсуждении расширений в конце данного приложения. Вместо этого мы расширим класс, создав его подкласс. Подклассы наследуют свойства и методы из того класса, из которого они происходят, т.е. из базового класса. Если то, что предоставляется в подклассе, не совсем верно, такое поведение можно переопределить, сохранив то, что требуется, но изменив то, что необходимо, хотя это, конечно, относится к предмету надлежащей разработки базового класса.

В этом разделе мы собираемся расширить класс CircleClass, создав производный от него класс ColoredCircleClass. Он будет обладать теми же свойствами, что и класс CircleClass, т.е. свойствами radius, area и circumference, а кроме того, новым свойством color. Но прежде введем ряд новых средств в класс CircleClass, чтобы стало ясно, как их переопределить в производном классе.

Итак, очистите полностью игровую площадку и введите на ней следующее видоизмененное определение класса CircleClass:

```
class CircleClass {
    var radius: Double = 1 {
        didSet {
            if (radius < 0) {
                radius = oldValue
            }
        }
    }

    var area: Double {
        get {
            return M_PI * radius * radius
        }
        set {
            radius = sqrt(newValue/M_PI)
        }
    }
}
```

```

var circumference: Double {
    get {
        return 2 * M_PI * radius
    }

    set {
        radius = newValue/(2 * M_PI)
    }
}

var description: String {
    return "Circle of radius \(radius)"
}

required init() {

}

init(radius: Double) {
    self.radius = radius
}

func adjustRadiusBy(amount: Double, times: Int = 1) {
    radius += amount * Double(times)
}

func reset() {
    radius = 1;
}
}

```

Прежде всего, в данный класс было введено вычисляемое свойство `description`, просто возвращающее описание этого класса в строковой форме. Это свойство является вычисляемым, поскольку нам требуется возвратить значение, содержащее текущий радиус, который может измениться. В новом классе `ColoredCircleClass` нам потребуется ввести в это описание цвет окружности, для чего свойство `description` будет определено.

Далее мы предварили ключевым словом `required` объявление инициализатора без аргументов. Иметь в классе такой инициализатор очень удобно, поскольку он позволяет установить значение по умолчанию в каждом свойстве. Для того чтобы этот инициализатор предоставлялся также в подклассе, производном от данного класса, в его объявление введено ключевое слово `required`. Этим компилятор языка Swift гарантирует, что инициализатор `init()` должен быть непременно реализован не только в подклассах, но и в производных от них подклассах. Так, в версию этого инициализатора в классе `ColoredCircleClass` будет введен цвет, выбираемый по умолчанию.

В заключение мы ввели в новое определение рассматриваемого здесь класса метод `reset()`, восстанавливающий значение по умолчанию в свойстве `radius`. Этот метод придется переопределить и в подклассе, чтобы восстановить значение по умолчанию в новом свойстве `color`, определяющем цвет окружности.

Теперь, когда стало ясно, что нам нужно, приступим к делу. Итак, следуя приведенным ниже инструкциям, введите новый код на игровой площадке. По ходу дела вы обнаружите сообщения об ошибках и предупреждения от компилятора языка Swift. Не обращайте на них особого внимания, поскольку они будут в дальнейшем устранены.

Сначала нужно создать подкласс, сообщив компилятору языка Swift, что CircleClass является его базовым классом.

```
class ColoredCircleClass : CircleClass {  
}
```

Это обычное начало определения класса, в которой компилятору языка Swift сообщается, что у класса ColoredCircleClass имеется базовый класс Circle Class. У любого класса может быть лишь один базовый класс, который может быть объявлен подобным образом. Как будет показано далее, для создаваемого класса может потребоваться соответствие одному или нескольким протоколам, которые также перечисляются в его определении.

Далее в рассматриваемый здесь класс вводится обычным образом новое сохраняемое свойство.

```
class ColoredCircleClass : CircleClass {  
    var color: UIColor = UIColor.black  
}
```

Новое свойство color относится к типу UIColor — классу, представляющему цвет в каркасе UIKit. Это свойство инициализируется экземпляром класса UIColor, представляющим черный цвет. Теперь переопределим свойство description, чтобы ввести цвет в возвращаемую символьную строку с описанием данного класса.

```
class ColoredCircleClass : CircleClass {  
    var color: UIColor = UIColor.blackColor()  
  
    override var description: String {  
        return super.description + ", color \(color)"  
    }  
}
```

Обратите, прежде всего, внимание на ключевое слово override, сообщающее компилятору языка Swift, что нам уже известно об определении свойства description в базовом классе и что мы намеренно переопределяем его. В некоторых языках программирования очень легко создать неумышленно метод или свойство с таким же именем, как и в базовом классе, не реализовав его. Это может повлечь за собой программные ошибки, которые трудно выявить впоследствии.

В новой реализации свойства description сначала происходит обращение к его базовой версии с помощью выражения super.description. Ключевое слово super сообщает компилятору языка Swift, что нам требуется ссылка на свойство description из суперкласса, а не на одноименное свойство из производного от него класса. К содержимому этого свойства затем присоединяется

символьная строка, описывающая цвет в свойстве `description`, переопределяемом в производном классе.

Далее нам требуется переопределить инициализатор `init()`, объявленный как `required`. Ниже показано, как это делается.

```
required init() {
    super.init()
}
```

От каждого инициализатора требуется непременно инициализировать все свойства сначала в его собственном классе, а затем в его базовом классе. В данном случае имеется единственное свойство `color`, которое инициализируется в момент его объявления. Поэтому нам остается только предоставить базовому классу возможность инициализировать свое состояние, для чего и делается вызов инициализатора `super.init()`. Если бы имелись другие свойства, которые требовалось бы инициализировать, компилятор языка Swift потребовал бы от нас установить их значения перед вызовом инициализатора `super.init()`. Обратите также внимание на ключевое слово `required`, которое должно быть непременно указано, поскольку этим словом отмечен также инициализатор `init()` в базовом классе.

ЗАМЕЧАНИЕ. Ранее утверждалось, что пометка инициализатора базового класса `init()` ключевым словом `required` вынуждает все подклассы реализовывать такой же инициализатор. Наш класс `ColoredCircleClass` не имеет явного инициализатора `init()`, но это не ошибка. Почему? Дело в том, что, поскольку класс `ColoredCircleClass` не имеет других инициализаторов, компилятор автоматически создает инициализатор `init()`, удовлетворяя требования базового класса.

Далее нужно определить инициализатор, устанавливающий нестандартные значения свойств `radius` и `color`. С этой целью введите следующий фрагмент кода в рассматриваемый здесь класс:

```
init(radius: Double, color: UIColor) {
    self.color = color
    super.init(radius: radius)
}
```

Сначала в этом фрагменте кода устанавливается значение свойства `color` из аргумента инициализатора, а затем значение свойства `radius` передается инициализатору базового класса. Вот, собственно, и все, что требуется сделать в данном инициализаторе.

Последнее, что нужно сделать, — переопределить метод `reset()`, как показано ниже, чтобы восстановить черный цвет в свойстве `color`. Теперь вы уже знаете, что требуется для этого сделать.

```
override func reset() {
    super.reset()
    color = UIColor.black
}
```

Как и прежде, ключевое слово `override` сообщает компилятору языка Swift, что мы переопределяем метод из базового класса. Между прочим, если ошибочно набрать имя этого метода, например `resets`, компилятор языка Swift выдаст предупреждение, что такого метода не существует в базовом классе. Именно в этом и заключается еще одно преимущество, извлекаемое из ключевого слова `override`. Остальная часть данного метода следует уже известному, представленному ранее образцу. Сделайте то, что требуется от вас, и предоставьте базовому классу сделать то, что требуется от него, вызвав его метод `reset()` с помощью ключевого слова `super`. В данном случае не так важно, что именно сделать в первую очередь: восстановить исходный черный цвет или вызвать метод `reset()` из базового класса.

```
override func reset() {
    color = UIColor.black
    super.reset()
}
```

Вот и все. Теперь опробуйте инициализатор по умолчанию, введя следующий фрагмент кода:

```
var coloredCircle = ColoredCircleClass()
coloredCircle.radius          // Результат: 1
coloredCircle.color           // Результат: black
coloredCircle.description     // Результат: "Circle of radius 1.0,
                             // color UIDeviceWhiteColorSpace 0 1"
```

Далее опробуйте другой инициализатор, введя следующий фрагмент кода:

```
coloredCircle = ColoredCircleClass(radius: 20,
                                    color: UIColor.redColor())
coloredCircle.radius          // Результат: 20
coloredCircle.color           // Результат: red
coloredCircle.description     // Результат: "Circle of radius 20.0,
                             // color UIDeviceRGBColorSpace 1 0 0 1"
```

Создание производных классов является очень полезной методикой программирования. Едва ли не в каждом примере кода, представленном в данной книге, применяется хотя бы один подкласс. Поэтому стоит потратить некоторое время на изучение разделов руководства *The Swift Programming Language*, посвященных инициализации и наследованию, в которых эти вопросы обсуждаются более подробно, чем здесь.

Протоколы

Протокол представляет собой объявление группы методов, инициализаторов и свойств, которым должны соответствовать класс, структура и перечисление, предоставляя их реализации. Как демонстрируется на протяжении всей данной книги, протоколы нередко служат для определения объектов, называемых **делегатами** и предоставляемыми **перехватчики**, позволяющие специально

настраивать поведение библиотечных или каркасных классов, реагировать на события и выполнять прочие действия.

В качестве примера ниже приведено определение протокола Resizable.

```
protocol Resizable {
    var width: Float { get set }
    var height: Float { get set }

    init(width: Float, height: Float)
    func resizeBy(wFactor: Float, hFactor: Float) -> Void
}
```

В этом объявлении протокола Resizable требуется, чтобы в любом типе данных, соответствующем этому протоколу, предоставлялись два свойства, один инициализатор и одна функция. Обратите внимание на то, что в самом протоколе ничего не определяется, а только предписывается то, что должно быть непременно реализовано в соответствующем ему типе данных. В качестве примера ниже приведен класс Rectangle, соответствующий протоколу Resizable.

```
class Rectangle : Resizable, Printable {
    var width: Float
    var height: Float
    var description: String {
        return "Rectangle, width \$(width), height \$(height)"
    }

    required init(width: Float, height: Float) {
        self.width = width
        self.height = height
    }

    func resizeBy(wFactor: Float, hFactor: Float) -> Void {
        width *= wFactor
        height *= hFactor
    }
}
```

Тип, указывающий в объявлении класса на его соответствие конкретному протоколу, состоит из имени этого протокола, после которого обычно следует имя базового класса, если таковой существует. Приведенный выше класс соответствует протоколам Resizable и CustomStringConvertible, определенным в стандартной библиотеке Swift.

Как видите, в данном классе предоставляется конкретная реализация средств, требующихся по протоколу. Так, если в протоколе требуется реализовать инициализатор, то такой инициализатор должен быть обозначен ключевым словом `required`, как показано выше. Экземпляры типов данных, соответствующих протоколу, могут присваиваться переменным типа этого протокола, как показано ниже.

```
let r: Resizable = Rectangle(width: 10, height: 20)
```

Для соответствия протоколу `Printable` требуется, чтобы в типе данных было реализовано свойство `description`, предназначенное для представления в удобочитаемом виде описания этого типа данных, если таковое потребуется. В частности, данное свойство используется в том случае, если экземпляр типа данных, соответствующего протоколу, передается в качестве аргумента функции `print()`, как показано ниже.

```
let rect = Rectangle(width: 10, height: 20)
print(rect) // Выводит строку "Rectangle, width 10.0, height 20.0"
```

Соответствие протоколу `CustomStringConvertible` требует от типа наличия свойства типа `String`, которое используется, когда необходимо представление типа, доступное для чтения человеком. В частности, это свойство используется, когда экземпляр соответствующего типа передается как аргумент функции `print()`:

```
let rect = Rectangle(width: 10, height: 20)
print(rect) // Выводит строку "Rectangle, width 10.0, height 20.0"
```

Расширения

Расширения являются эффективным языковым средством, общим для Swift и Objective-C. Расширение позволяет вводить дополнительные функциональные возможности в любой класс, структуру или перечисление, в том числе из стандартной библиотеки Swift или системных каркасов. Синтаксис определения расширений такой же, как и синтаксис классов и структур. Расширение внедряется с помощью ключевого слова `extension`, после которого следуют имя расширяемого типа данных и само расширение. В расширяемый тип данных можно вводить вычисляемые (но не сохраняемые) свойства, методы, инициализаторы, вложенные типы данных и соответствие протоколам.

В главе 16 демонстрируется пример применения расширения для ввода метода, возвращающего произвольно выбираемый цвет, в класс `UIColor` из каркаса UIKit. Ниже показано, каким образом определяется такое расширение.

```
extension UIColor {
    class func randomColor() -> UIColor {
        let red = CGFloat(Double(arc4random() % 256))/255
        let green = CGFloat(Double(arc4random() % 256))/255
        let blue = CGFloat(Double(arc4random() % 256))/255
        return UIColor(red: red, green: green, blue: blue, alpha:1.0)
    }
}
```

Метод из этого расширения применяется следующим образом:

```
let randomColor = UIColor.randomColor()
```

Резюме

На этом краткое введение в наиболее важные языковые средства языка Swift завершено. Безусловно, овладения этими средствами явно недостаточно для профессионального программирования на языке Swift. Поэтому для восполнения пробелов в своих знаниях языка Swift вам придется обратиться к официальной документации к нему или к литературе, перечисленной в главе 1 данной книги. Между тем, вернитесь к материалу главы 3 или той части книги, которую вы оставили, чтобы изучить основы программирования на языке Swift в этом приложении, и приступайте к написанию на языке Swift своего первого приложения под управлением iOS.

Предметный указатель

А

Автодополнение 175
Акселерометр 739
Альфа-канал 130, 613
Анимация
 неявная 592
Аргумент
 sender 83
Архивирование 491, 494
Атрибут 510
 Background 130
 Mode 128
 Semantic 128
 Tag 128
 Tint 130

Б

Базовый язык 784
Библиотека 59
 Quartz 2D 607
 мультимедиа 60
 объектов 60
 сниппетов кода 60
 файловых шаблонов 60
Блокировка 572
Блок
 анимации 243

В

Выход 82

Г

Граф сцены 635
Группа 53

Д

Действие 83
Делегат 231, 251, 881
 приложения. 111
Диапазон 834
Директива
 MARK 271
Диспетчерская группа 579

Диспетчер

 движения 741

 местоположения 714

Дисплей

 Retina 169

 мультисенсорный 682

Долговременное хранение

 нескольких файлов 482

 одного файла 482

Е

Единица работы 573

Ж

Жест 682

З

Замыкание 859

 параллельное 579

Запрос

 на извлечение 513

И

Игровая площадка 815

Индикатор раскрытия 351, 353

Инспектор

 размеров 178

Источник данных 251

К

Каркас 28

 AppKit 60

 Cocoa 28

 Cocoa Touch 27

 Core Animation 635

 Core Data 508

 Core Foundation 94

 Core Graphics 609

 Core Location 714

 Core Motion 741

 Foundation 817

 Grand Central Dispatch 568

 Map Kit 713

- Sprite Kit 635
- UIKit 60
- Xcode 35
- Касание 682
- Каталог
 - Documents 479
 - Library 479
 - tmp 479
 - ресурсов 67
- Класс 867
 - Array 483
 - CellIdentifier 323
 - CLLocation 717
 - CLLocationDegrees 718
 - CMAccelerometerData 742
 - CMDeviceMotion 742
 - CMGyroData 742
 - CMMotionManager 742
 - Date 483
 - DetailViewController 404, 411
 - Dictionary 280, 483
 - MasterViewController 404
 - MPMoviePlayerController 776
 - NSArray 275, 483, 839
 - NSBundle 280
 - NSData; 483
 - NSDate 483
 - NSDictionary 275, 483, 839
 - NSEntityDescription 513
 - NSFontAttributeName 111
 - NSLayoutConstraint 99
 - NSLocale 788
 - NSManagedObjectContext 512
 - NSMutableArray 483
 - NSMutableAttributedString 110
 - NSMutableData 483
 - NSMutableDictionary 483
 - NSMutableString 483
 - NSNotificationCenter 471
 - NSNumber 483
 - NSSet 839
 - NSString 111, 483
 - NSThread 570
 - NSUserDefaults 431, 563
 - NSUndoManager 550
 - NSUserDefaults 431, 563
 - SKLabelNode 643
 - SKNode 639
 - SKPhysicsWorld 662
 - SKScene 639
 - SKSprite 635
 - SKView 637
 - String 111, 483
 - StringsTable 440
 - UIAlertAction 161
 - UIApplication 111, 469
 - UIButton 157
 - UICollectionView 387
 - UICollectionViewFlowLayout 389
 - UIControl 121
 - UIDocument 530
 - UIGestureRecognizer 697
 - UIGestureRecognizerStateBegan 707
 - UIImage 287
 - UIImagePickerController 769
 - UILabel 62
 - UINavigationController 221, 223, 348, 406
 - UIPinchGestureRecognizer 708
 - UIResponder 144
 - UIRotationGestureRecognizer 708
 - UISplitViewController 406
 - UISwipeGestureRecognizer 699
 - UITabBarController 221, 223, 254
 - UITableView 422
 - UITableViewCell 294
 - UITableViewDataSource 294
 - UITableViewDelegate 294
 - UITapGestureRecognizer 702
 - UITextViewDelegate 138
 - UITouch 688
 - UIView 58, 144
 - UIViewController 81, 223
 - Кнопка раскрытия 352
 - Кодирование
 - ключ–значение 491, 511
 - Комплект
 - SceneKit 636
 - Компонент
 - селектора 249
 - зависимый 249
 - Контекст 608
 - графический 608
 - управляемых объектов 512

Контроллер 78
 всплывающего меню 421
 контейнерного представления 237
 корневого представления 349
 корневой 223, 252, 349
 навигации 347
 представления 56
 Контур 609
 Кортеж 851
 Кривая анимации 244

Л

Ленивая загрузка 234, 237
 Локализация 784
 базовая 795
 раскадровки 798
 Лямбда-функция 574

М

Массив 834
 Менеджер
 отмены 512
 Метод 874
 Механизм
 Auto Layout 173
 Core Data 482
 View Debugging 343
 автоматического поворота 167
 распознавания жестов 144
 Модель 78
 Grayscale 614
 художника 607
 цветовая
 CMYK 614
 HSL 614
 HSV 614
 RGB 612
 RYB 613
 Мьютекс 572

Н

Навигатор
 отладки 46
 отчетов 49
 поиска 45
 проблем 45
 проекта 44
 символов 44

точек прерывания 49
 Нажатие 682
 Начальный пробел 179

О

Область
 вспомогательная 52
 редактирования 43
 Объект-значение 830
 Объектно-реляционное отображение 498
 Объект

пользовательский 484
 сериализованный 483
 управляемый 510
 Ограничение 173
 Окно
 всплывающее 162
 проекта 41
 Оператор
 let 820
 switch 852
 управляющий 848
 условный 848
 цикла 850

Операция
 арифметическая 825
 двоичной арифметики 825
 объединяющая 844
 Опциональная последовательность 302
 Ориентация
 интерфейса 175
 устройства 175
 Отступ
 вертикальный 191
 верхний 179
 горизонтальный 190
 Очередь 574

П

Пакет 279
 настроек 431
 Панель
 библиотеки 59
 быстрого перехода 50
 вкладок 217
 инструментальная 41, 220
 навигационная 217

- Папка
 Classes 53
 Products 54
 локализации 784
- Параметр
 atomically 484
 региональный 788
- Переключатель 118
- Переменная 820
 связанная 500
- Песочница 29
- Пиктограмма
 First Responder 57
 View 58
 View Controller 57
 вспомогательная 351
- Подконтроллер 349
- Поле
 многозначное 449
 редактирования 118
- Ползунок
 Alpha 129
- Помощник редактора 43
- Потокобезопасный код 572
- Поток 568
 выполнения 570
 главный 541
 основной 573
 фоновый 541
- Представление 56, 78, 215
 главное 29, 62
 графическое 118
 действий 43
 дочернее 62
 коллекции 387
 контейнерное 144
 контроля версий 43
 модальное 160
 навигатора 44
 разделенное 219, 401
 селектора 247
 содержимого 215
 стека 399
 табличное 293
 индексированное 295
 сгруппированное 295
 простое 295
- Предупреждение 159
- Приложение
 использующее панель вкладок 216
 служебное 216
 с несколькими представлениями 216
- Проект
 Cross-platform 38
 iOS 38
 macOS 38
 tvOS 38
 watchOS 38
 локализации 784
- Протокол 881
 CLLocationManagerDelegate 714
 UIImagePickerControllerDelegate 771
 MKAnnotation 732
 NSCoder 491
 NSCoding 491
 NSCopying 491
 Printable 882
 Resizable 882
 UIApplicationDelegate 112
 UIImagePickerControllerDelegate 770, 773
 UIPickerViewDataSource 264
 UIPickerViewDelegate 264
 UITableViewDataSource 294, 300
 UITableViewDelegate 294
- P**
- Раздел 296
- Размер
 действительный 104
 естественный 104
- Распознаватель
 вращения 707
 жестов 683
 непрерывных жестов 707
 скольжения 698
 шипов 707
- Расширение 883
- Редактор
 моделей данных 510
- Рефакторинг 224
- C**
- Свойство 872
 извлекаемое 511

Связь 510
Селектор 247
 даты 248
 изображений 768
 многокомпонентный 249
 однокомпонентный 249
 с зависимыми компонентами 249
 с изображениями 250
Сериализация
 последовательная 483
Симулятор 65
Синтаксис
 диапазонов 835
 обобщения 837
Словарь 834, 837
Служба
 Geocoding 736
 iCloud 529
 iMessage 589
 Significant Location Updates 736
Событие 682
Состояние
 приложения 582
 активное 583
 неактивное 583
 невыполнения 583
 фоновое 583
 приостановки 583
 элемента управления
 выбранное 157
 недоступное 157
 обычное 156
 подсвеченное 157
 сфокусированное 156
Список
 действий 159
 свойств 483
Стек 348
Строка состояния 169
Структура 867
Сущность 510
Схема 41

Т

Технология
 Core Animation 244
 Grand Central Dispatch 567

Тип
 CGContext 608
 Double 821
 Float 821
 Int 821, 822
 Int16 822
 Int32 822
 Int64 822
 Int8 822
 UInt 821
 UInt 822
 UInt16 822
 UInt32 822
 UInt64 822
 UInt8 822
 необязательный 834

У

Уведомление 469
 Уровень отступа 309

Ф

Файл
 .nib 55
 .png 67
 символьных строк 786
 сцены 638

Фильтр
 расстояния 715

Флажок
 Drawing
 Autoresize Subviews 131
 Clears Graphics Context 130
 Clip Subviews 131
 Hidden 130
 Opaque 130
 Interaction 129
 Multiple Touch 129
 User Interaction Enabled 129

Text Field
 Adjust to Fit 137
 Auto-enable Return Key 137
 Clear When Editing Begins 137
 Secure Text Entry 137

Фоновая работа 581
 Фрейм стека 47
 Функция 854

890 ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Х

Хранилище
в оперативной памяти 512
постоянное 512
резервное 512

Ц

Цветовая гамма 612
Цвет
заливки 609
обводки 609
Цель сборки 54
Центр уведомлений 469
Цепочка реагирующих элементов 683
Цикл удержания 423

Ш

Шаблон
Single View Application 79
проекта 40

Э

Элемент
изображения
оконечный 155
реагирующий 683
управления
активный 120
пассивный 121
сегментированный 118

Я

Язык
Objective-C 94
SQL 498
Swift 815

SWIFT

КАРМАННЫЙ СПРАВОЧНИК

ЯЗЫК ПРОГРАММИРОВАНИЯ ДЛЯ IOS И MAC OS X

2-Е ИЗДАНИЕ

Энтони Грей



www.williamspublishing.com

Этот краткий справочник карманного типа составлен таким образом, чтобы читатель мог быстро найти ответы на вопросы, возникающие во время разработки и отладки прикладных программ на языке программирования Swift версии 2.1. Справочник удобен для изучения современных языковых средств Swift, включая типовую безопасность, обобщения, определение типов, замыкания, кортежи, автоматическое управление памятью, классы, структуры данных, протоколы, пользовательские и встроенные функции, поддержку необязательных типов и Юникода. Справочник рассчитан на широкий круг читателей, интересующихся программированием на Swift и разработкой программного обеспечения на платформах iOS и Mac OS X.

ISBN 978-5-8459-2088-1 **в продаже**

ОПТИМИЗАЦИЯ ПРОГРАММ НА С++

Курт Гантерот



С++ язык динамично развивающийся, так что методы оптимизации программ на нем тоже постоянно развиваются и совершенствуются. То, что когда-то было передовым способом оптимизации, теперь зачастую делает вместо вас компилятор. Однако в настоящем программировании всегда есть возможность усовершенствовать имеющийся код. Как это сделать? Об этом вы и узнаете из книги, написанной профессионалом для профессионалов и достойной занять свое место рядом с клавиатурой каждого серьезного программиста на С++.

www.williamspublishing.com

ISBN 978-5-9908910-6-7 в продаже

ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ: САМОЕ ОСНОВНОЕ

Вон Вернон



www.dialektika.com

В книге кратко описаны реализация методов предметно-ориентированного проектирования (DDD) и специализированные подходы к реализации систем на основе современной архитектуры, а также показана важность ориентации на предметную область с учетом технических ограничений. Автор дает рекомендации, как разделять модели предметной области с помощью мощного шаблона ОГРАНИЧЕННЫЙ КОНТЕКСТ, как разработать ЕДИНЫЙ ЯЗЫК с четкими границами и как обеспечить совместную работу бизнес-экспертов и разработчиков над созданием такого языка. Он показывает, как с помощью ПОДОБЛАСТЕЙ выполнить интеграцию унаследованных систем и объединить несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ, используя отношения внутри группы и технические механизмы. Книга представляет собой краткий справочник по методам предметно-ориентированного проектирования и предназначена для программистов и пользователей всех уровней.

ISBN 978-5-9908463-8-8 в продаже

ЯЗЫК ПРОГРАММИРОВАНИЯ GO

Алан А. А. Донован
Брайан У. Керниган



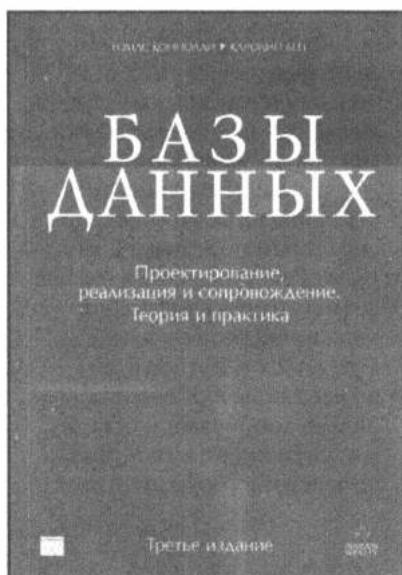
www.williamspublishing.com

Вы краем уха слышали о новом языке программирования Go, но не знаете, что он представляет собой на самом деле? Ответы на все ваши вопросы вы получите в этой книге. Она поможет вам познакомиться с языком Go поближе, узнать о его предназначении и преимуществах, и о том, как писать программы на этом языке (причем это будут программы не начинающего, но профессионального программиста — эффективные и идеоматичные). Книга написана двумя профессионалами — как в области программирования, так и в области написания книг на программистские темы. Так что если вы хотите быть в курсе последних достижений в программировании — не сомневайтесь, взяв эту книгу, вы свернули на верный путь к профессиональному овладению языком Go. Книга предназначена в первую очередь для программистов, уже уверенно владеющих каким-либо языком программирования.

ISBN 978-5-8459-2051-5 в продаже

БАЗЫ ДАННЫХ: ПРОЕКТИРОВАНИЕ, РЕАЛИЗАЦИЯ И СОПРОВОЖДЕНИЕ. ТЕОРИЯ И ПРАКТИКА. 3-Е ИЗДАНИЕ

**Томас Коннолли,
Каролин Бегг**



www.williamspublishing.com

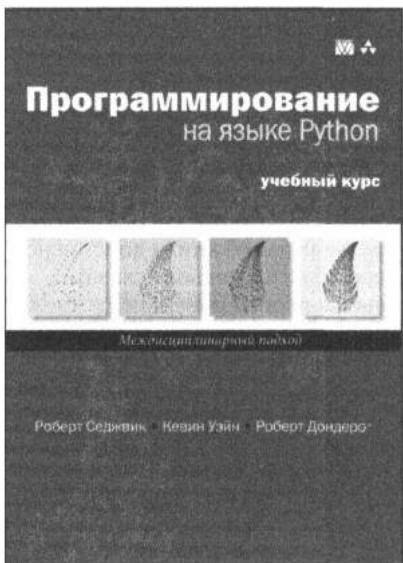
в продаже

Авторы этой книги сконденсировали на ее страницах весь свой опыт разработки баз данных для нужд промышленности, бизнеса и науки, а также обучения студентов в университете Пейсли, Шотландия. Результатом их труда стало беспрецедентно полное справочное руководство по проектированию, реализации и сопровождению баз данных. Ясное изложение теоретического и практического материала, включающего детально разработанную методологию проектирования и реализации баз данных, а также подробное рассмотрение существующих языков и стандартов, делает эту книгу доступной и полезной как студентам, так и опытным профессионалам. Третье издание книги дополнено несколькими новыми главами, освещающими новейшие технологии в этой области — объектные базы данных, объектно-реляционные базы данных, использование СУБД в Web, использование хранилищ данных и средств комплексного анализа (OLAP), а также большим количеством новых примеров и переработанных упражнений. Ясное и четкое изложение материала, наличие двух полномасштабных учебных примеров и множества контрольных вопросов и упражнений, позволяет использовать эту книгу не только при самостоятельном обучении, но и как основу для разработки курсов обучения любых уровней сложности — от студентов младших курсов, до аспирантов, а также как исчерпывающее справочное руководство для профессионалов.

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON

учебный курс

Роберт Седжвик
Кевин Уэйн
Роберт Дондеро



www.dialektika.com

Авторы книги сосредоточиваются на самых полезных и важных средствах языка Python и не стремятся к его абсолютно полному охвату. Весь код этой книги был отработан и проверен на совместимость как с языком Python 2, так и Python 3, что делает его подходящим для каждого программиста и любого курса на много лет вперед.

Особенности книги:

- всеобъемлющий, основанный на приложениях подход: изучение языка Python на примерах из области науки, математики, техники и коммерческой деятельности;
- основное внимание главному: самым полезным и важным средствам языка Python;
- совместимость примеров кода проверена на языках Python 2.x и Python 3.x;
- во все главы включены разделы с вопросами и ответами, упражнениями и практическими упражнениями.

ISBN 978-5-9908462-1-0 в продаже

Swift 3: разработка приложений в среде Xcode для iPhone и iPad с использованием iOS SDK

Не требуя от читателей предварительных знаний о языке программирования Swift, авторы предлагают доступный и полный курс программирования для устройств iPhone и iPad. Изложение начинается с основных сведений, загрузки и инсталляции программы Xcode и комплекта iOS 10 SDK, а также создания первого простого приложения.

В третьем издании этого бестселлера описывается процесс интеграции всех популярных элементов пользовательского интерфейса iOS: кнопок, переключателей, селекторов, инструментальных панелей и ползунков. Прочитав учебник, читатели освоят множество проектных шаблонов — от простого отдельного представления до сложных иерархических детализированных представлений. Авторы раскрывают секреты создания табличных представлений и сохранения данных с помощью файловой системы iPhone. Читатели научатся сохранять и извлекать данные с помощью многочисленных механизмов долговременного хранения, включая Core Data и SQLite. И это еще не все!

В книге описаны как новые технологии, так и существенные модификации старых технологий. Читатели найдут в ней все, что необходимо для создания приложений, работающих под управлением современных версий системы iOS. Все примеры, включенные в книгу, разработаны с использованием возможностей последней версии программы Xcode и самых современных проектных шаблонов, предназначенных для системы iOS 10.

НА ВЕБ-САЙТЕ

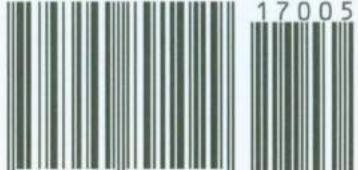
Исходные коды всех примеров, рассмотренных в книге, можно загрузить с веб-сайта издательства по адресу:
[http://www.williamspublishing.com/
Books/978-5-9908910-2-9.html](http://www.williamspublishing.com/Books/978-5-9908910-2-9.html)

Категория: языки программирования

Предмет рассмотрения: Swift 3

Уровень: для программистов средней и высокой квалификации

ISBN 978-5-9908910-2-9



17005
9 785990 891029



www.dialektika.com

Apress®

www.apress.com