O'REILLY®

Рассмотрены iOS 7 и Xcode 5

Программирование для iOS 7

Основы Objective-C, Xcode и Cocoa

Мэтт Нойбург

Программирование для iOS 7

Основы Objective-C, Xcode и Cocoa

Мэтт Нойбург

Москва • Санкт-Петербург • Киев
2014

ББК 32.973.26-018.2.75
Н78
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского *И.В. Берштейна*,
докт. физ.-мат. наук *Д.В. Клюшина*, канд. техн. наук *И.В. Красикова*
Под редакцией докт. физ.-мат. наук *Д.В. Клюшина*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Нойбург, Мэтт.
Н78 Программирование для iOS 7. Основы Objective-C, Xcode и Cocoa. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2014. — 384 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1895-6 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Мэтт Нойбург

Программирование для iOS 7.
Основы Objective-C, Xcode и Cocoa

Литературный редактор *И.А. Попова*
Верстка *Л.В. Чернокозинская*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 12.03.2014. Формат 70x100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 30,96. Уч.-изд. л. 26,87.
Тираж 1500 экз. Заказ № 3084

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1895-6 (рус.)
ISBN 978-1-4919-4557-5 (англ.)

Оглавление

Содержание

Предисловие

После трех изданий моей книги по программированию для операционной системы iOS — *Programming iOS 4* (май 2011), *Programming iOS 5* (март 2012) и *Programming iOS 6* (март 2013) — все выглядит так, будто она разорвана на два куска сразу после части III (главы 13). Теперь это две книги:

- книга, которую вы держите в руках, *iOS 7 Programming Fundamentals*, состоящая из глав 1–13 указанных выше книг,
- и другая книга, *Programming iOS 7*, состоящая из глав 14–40 указанных выше книг.

Это было сделано вовремя. Книга *Programming iOS 6* увеличилась до 1150 страниц и стала неподъемной. На самом деле я задумывался о таком решении еще до того, как была опубликована книга *Programming iOS 4*. Первоначально я предложил издательству O'Reilly Media еще в начале 2010 года книгу *Fundamentals of Cocoa Programming*, которая охватывала бы в основном тот же материал, что и данная книга. Однако это предложение было принято только при условии, что книга в большей степени будет посвящена программированию Cocoa Touch (для системы iOS). Несмотря на мои страстные мольбы осенью 2010 года, я так и не смог убедить издателя разделить рукопись на две книги.

Так что сейчас, можно сказать, исполнилась моя мечта — хотя и не совсем, так как я хотел выпустить одну книгу в двух томах. Моя идея заключалась в том, что обе книги будут иметь одинаковое название и рассматриваться как первый и второй тома, с последовательной нумерацией страниц и глав — т.е. если бы первый том закончился, скажем, на главе 13 страницей 400, то второй том начинался бы со страницы 401 и главы 14. Увы, позиция O'Reilly Media была прямо противоположной.

Таким образом, книга *Programming iOS 7*, хотя и начинается с 1 главы и страницы 1, является продолжением с того места, на котором закончилась книга *iOS 7 Programming Fundamentals*. Эти книги дополняют друг друга. Те, кто хочет получить полную информацию о написании приложений для операционной системы iOS, твердые знания и хорошо понимать предмет, я надеюсь, купят обе книги. В то же время наличие двух книг, как мне представляется, делает объем и область рассмотрения каждой книги более привлекательными для большинства читателей.

Те, кто считают, что уже знают все, что надо знать о языках C и Objective-C, среде Xcode, а также о лингвистических и архитектурных основах среды Cocoa, более не жалуются на наличие 13 "лишних" глав, предшествующих настоящему материалу о написании кода для операционной системы iOS, — потому что эти 13 глав теперь выделены в отдельный том, *iOS 7 Programming Fundamentals*. Вторая книга, *Programming iOS 7*, теперь начинается как "Илиада" Гомера — с средины истории и предназначена для читателя, который знает язык и среду Xcode. Если такой читатель впоследствии меняет свое мнение и решает, что все же знать основы было бы неплохо, то книга *iOS 7 Programming Fundamentals* по-прежнему доступна и ожидает изучения.

Что касается данной книги, *iOS 7 Programming Fundamentals*, то это книга, которую я давно хотел написать, но которая постоянно поглощалась большими книгами — *Programming iOS 4*, *Programming iOS 5* и *Programming iOS 6*. Теперь, наконец-то, она выделена в собственный том и состоит из трех частей, посвященных основам программирования для iOS.

- Часть I представляет собой введение в язык программирования Objective-C и начинается с описания языка программирования C (который используется в практике программирования на Objective-C существенно больше, чем думают многие начинающие). Затем описываются объектно-ориентированные концепции и механизмы работы классов и экземпляров.
- Часть II, немного отходит от языка и обращается к среде Xcode, в которой выполняется вся работа по созданию приложений для операционной системы iOS. Здесь поясняется, что такое проект Xcode и как превратить его в приложение, как комфортно работать в Xcode и как обратиться к документации, как писать и отлаживать код — словом, описаны все стадии разработки приложений вплоть до их размещения в App Store. Здесь также имеется глава, очень важная для понимания работы программы Interface Builder, описывающая выходы и действия, а также механику загрузки nib-файлов; однако ряд специализированных тем, как, например, ограничения автоматического макетирования, перенесены во вторую книгу.
- Часть III возвращается к языку Objective-C, но в этот раз с точки зрения среды Cocoa Touch. Среда Cocoa предоставляет важные базовые классы и добавляет такие лингвистические и архитектурные устройства, как категории, протоколы, делегирование и оповещения, а также освещает вопросы управления памятью. Здесь также рассматриваются шаблоны Key-Value Coding и Key-Value Observing.

Прочитавший эту книгу программист приобретет фундаментальные знания и навыки, которыми должен владеть любой хороший программист iOS. Книга сама по себе не рассказывает, как создавать интересные приложения для операционной системы iOS (хотя и подкреплена десятками образцов проектов, которые можно скачать с моего сайта GitHub, `http://github.com/mattneub/Programming-iOS-Book-Examples`), но в ней в качестве иллюстраций постоянно используются мои собственные реальные приложения и ситуации из реального опыта программирования. Прочтя книгу, вы будете готовы приступить к реальному программированию для операционной системы iOS 7.

Версии

Эта книга ориентирована на версии iOS 7 и Xcode 5. Однако, в общем случае, более ранним версиям системы iOS и среды Xcode уделено минимальное внимание. Я не имел намерения охватить в этой книге какие-либо подробности, касающиеся более ранних версий программного обеспечения (в конце концов, они вполне доступны в моих предыдущих книгах). Тем не менее часто можно сказать несколько слов об обратной совместимости, и я регулярно делаю это в книге, отмечая отличия от более ранних версий.

Версия Xcode 5 больше не предлагает пользователю при создании нового проекта приложения из одного из шаблонов проекта указывать, следует ли использовать автоматический подсчет ссылок (ARC). Эта технология управления памятью компилятором намного упрощает жизнь программистов для операционной системы iOS. В среде Xcode 5 механизм ARC просто включен по умолчанию. Таким образом, эта книга с самого начала предполагает, что вы используете механизм ARC. Я довольно часто различаю поведение компилятора с механизмом

ARC от поведения компилятора с отключенным механизмом ARC, но больше не описываю работу без ARC, за исключением главы 12, где я все еще объясняю работу ARC, описывая все, что вам пришлось бы делать, если бы вы отключили эту возможность.

Среда Xcode также больше не предоставляет возможности выбора использования раскадровки. Все проекты (за исключением пустого шаблона) включают основную раскадровку; вариант использования основного `.xib`-файла вместо него отсутствует. Соответственным образом я скорректировали стиль изложения, исходя из того, что первичны именно раскадровки и что вы используете именно их. Я также демонстрирую построение проекта с nib-файлами полностью из `.xib`-файлов. Сейчас это приводит к большему количеству работы, чем ранее, когда можно было просто убрать соответствующий флажок, потому что вы не можете сделать это просто, сняв флажок в диалоговом окне создания шаблона.

Я также осветил — часто без особого ажиотажа — различные другие инновации в системе iOS 7 и среде Xcode 5. Компания Apple ясно заявила, что новое поколение программного обеспечения предназначено для того, чтобы программирование для iOS стало еще легче и приятнее, чем когда-либо (надо сказать, что в целом они добились успеха). Такие нововведения, как модули и автоматическое связывание, каталоги ресурсов и различные панели настроек, сделают вашу работу гораздо более комфортной, и я предполагаю — как само собой разумеющееся, — что вы хотите активно их использовать.

Благодарности

Прежде всего моя благодарность тем сотрудникам O'Reilly Media, которые сделали написание книги таким восхитительно простым. Первыми вспоминаются Рэйчел Румелиотис (Rachel Roumeliotis), Сара Шнайдер (Sarah Schneider), Кристен Браун (Kristen Brown) и Адам Витвер (Adam Witwer). Кроме того, я не могу забыть моего первого редактора, Брайана Джепсона (Brian Jepson), который не принимал участия в данном издании, но влияние которого присутствует во всех моих книгах.

От множества ошибок меня спас острый и преданный глаз моего давнего читателя Петера Ольсена (Peter Olsen), который добросовестно отправлял мне информацию о всех обнаруженных ошибках.

Как и ранее, мне очень помогала масса просто фантастического программного обеспечения, достоинства которого я высоко оценил в процессе написания этой книги. Я хотел бы упомянуть, в частности:

- git (`http://git-scm.com`)
- SourceTree (`http://www.sourcetreeapp.com`)
- TextMate (`http://macromates.com`)
- AsciiDoc (`http://www.methods.co.nz/asciidoc`)
- BBEdit (`http://barebones.com/products/bbedit/`)
- Snapz Pro X (`http://www.ambrosiasw.com`)
- GraphicConverter (`http://www.lemkesoft.com`)
- OmniGraffle (`http://www.omnigroup.com`)

Книга была полностью набрана и отредактирована на моей верной клавиатуре Unicomp Model M (`http://pckeyboard.com`), без которой я никогда не смог так долго и безболезненно работать. Подробнее о моей рабочей среде и ее физических характеристиках рассказано здесь: `http://matt.neuburg.usesthis.com`.

Из предисловия к книге *Programming iOS 4*

Популярность устройства iPhone, с его во многом бесплатными или очень недорогими приложениями, и последующая популярность устройства iPad привлекли и будут привлекать множество программистов, работающих над приложениями для этих устройств, несмотря на то, что они, возможно, не испытывали тех же чувств к операционной системе OS X. Ежегодные конференции Apple WWDC, в которых акцент сместился с системы OS X на систему iOS, также отражают эту тенденцию.

Однако такое всеохватывающее стремление программировать для операционной системы iOS способствовало возникновению неприятной тенденции — начинать программировать, не умея это делать. Система iOS обеспечивает программиста могучими силами, которые могут показаться совершенно безграничными, но которыми нельзя воспользоваться без серьезной подготовки. К сожалению, я часто сталкиваюсь с программистами, которые глубоко погрузились в создание некоторых интересных приложений, но вопросы которых совершенно ясно показывают, что они не знакомы с азами поведения в том мире, в который они так счастливо и беззаботно погрузились.

Именно это положение дел и побудило меня написать книгу, которая предназначена для обучения основам системы iOS. Я люблю среду Cocoa и давно хотел писать о нем, но система iOS и его популярность извиняют мое решение несколько изменить тематику. Здесь я попытался изложить (надеюсь, вполне педагогически, дидактически и логически) то, что, как я надеюсь, станет полезным и поучительным для всех, кого привлекает программирование для системы iOS. Сюда входит хорошее знание основ языка Objective-C (начиная с языка C), понимание объектно-ориентированного программирования, советы по использованию инструментария, достаточно полный рассказ о том, как создаются объекты Cocoa, как они взаимодействуют и как можно управлять их жизненным циклом, и многое другое. Надеюсь, как и в моей предыдущей книге, что вы не только прочтете эту книгу от корки до корки, но и найдете ей место на рабочем столе в качестве удобного справочника.

Эта книга не предназначена для того, чтобы опорочить документацию Apple или их примеры проектов. Это прекрасные ресурсы, и со временем они становятся только лучше; я активно использовал их при подготовке этой книги. Но тем не менее я считаю, что они не могут выполнять ту же функцию, что и разумно упорядоченное изложение фактов. Онлайн-документация вынуждена исходить из предположения о том, что вы уже многое знаете; ведь нельзя гарантировать, что вы будете использовать ее в определенном порядке. Документация в большей степени представляет собой справочник, чем учебник. Как бы хорошо ни был прокомментирован какой-то пример приложения, в нем не так легко разобраться: он демонстрирует, но не учит.

Книга же имеет пронумерованные главы и последовательно идущие страницы. Можно предположить, что вы будете знать язык C до того, как познакомитесь с языком Objective-C, — по той простой причине, что глава 1 предшествует главе 2. Наряду с фактами я хочу передать вам определенный опыт, которым пытаюсь поделиться с вами. В этой книге часто встречаются ссылки на распространенные ошибки новичков; в большинстве случаев это ошибки, которые делал я сам, но есть и ошибки, которые я встречал у других. Я пытаюсь рассказать вам обо всех подводных камнях, на которые натыкался сам, предполагая, что вы будете учиться так же, как я, и наступать на те же грабли. В книге вы встретите много фрагментарных примеров, извлеченных из большего приложения и поясняющих только одну конкретную мысль или метод. Это мгновенный снимок процесса разработки готовой программы, который, как я надеюсь, научит вас мыслить более эффективно. Надеюсь, что главное, что вы получите от чтения этой книги, — именно умение мыслить.

Соглашения, использованные в этой книге

В книге использованы следующие типографические соглашения.

Курсив

- Этим шрифтом выделяются новые термины, а также те или иные важные мысли.

`Моноширинный шрифт`

- Этот шрифт используется для адресов URL, имен файлов, листингов программ, а также для таких элементов программ, как переменные, имена функций, типы данных, ключевые слова и т.п.

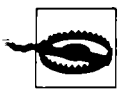
`Полужирный моноширинный шрифт`

- Этот шрифт указывает команды или иной текст, который должен быть введен пользователем буквально.

`Курсивный моноширинный шрифт`

- Этот шрифт указывает то, что должно быть заменено значениями, предоставляемыми пользователем, или значениями, получаемыми из контекста.

Так выглядят подсказки, примечания и советы.



Так выглядят предупреждения или предостережения.

Использование примеров кода

Сопутствующий материал (примеры кода, упражнения и т.п.) доступны для загрузки по адресу `https://github.com/mattneub/Programming-iOS-Book-Examples`.

Эта книга призвана помочь вам справиться со своей работой. В принципе вы можете использовать приведенные в книге примеры в своих программах и документации. Связываться с нами для разрешения использовать код не требуется, если только вы не воспроизводите значительную часть кода. Например, для написания программы, которая использует несколько фрагментов кода из этой книги, разрешение не требуется. Однако продажа или передача CD-ROM с примерами из книг издательства O'Reilly требует разрешения. Ответ на вопрос путем цитирования текста или примера кода из данной книги не требует разрешения. Включение значительного количества примеров кода из этой книги в документацию вашего продукта требует получения разрешения.

Мы были бы благодарны, но не требуем полного указания источника, которое обычно включает в себя название, фамилию автора, издателя и ISBN, например "*iOS 7 Programming Fundamentals* by Matt Neuburg (O'Reilly). Copyright 2014 Matt Neuburg, 978-1-491-94557-5".

Если вы чувствуете, что ваше использование примеров кода выходит за рамки добросовестного использования или разрешений, данных выше, не стесняйтесь связаться с нами по адресу `permissions@oreilly.com`.

Об авторе

Мэтт Нойбург (Matt Neuburg) начал программировать в 1968 году в возрасте 14 лет, когда он был членом буквально подпольного школьного клуба, который собирался раз в неделю для работы с компьютерами PDP-10, напоминающими примитивные телетайпные машины. Кроме того, иногда он использовал компьютер IBM-360/67 в Принстонском университете, однажды впав в отчаяние, рассыпав колоду перфокарт. Мэтт специализировался на греческом языке и получил степень доктора философских наук в Корнуэльском университете в 1981 году, написав докторскую диссертацию об Ахилле на мейнфрейме. Он продолжал преподавать антиковедение, литературу и культуру в лучших высших учебных заведениях и публиковать многочисленные академические статьи, которые интересовали немногих. Тем временем он приобрел компьютер Apple IIc и нова влюбился в компьютеры, перейдя в 1990 году на Macintosh. Он написал несколько учебных свободно распространяемых программ, став регулярным автором интернет-журнала TidBITS, а с 1995 года — редактором журнала MacTech. В августе 1996 года он стал вольнонаемным сотрудником. Мэтт Нойбург является автором книг Frontier: The Definitive Guide, REALbasic: The Definitive Guide и AppleScript: The Definitive Guide, а также Programming iOS 7 (все вышли в издательстве O'Reilly & Associates), и Take Control of Using Moutain Lion (издательство TidBITS Publishing).

Колофон

Животное, изображенное на обложке этой книги, — гренландский тюлень (Pagophilus groenkandicus), латинское название которого переводится как "любитель льда из Гренландии". Эти животные водятся в арктических водах Атлантики и большую часть времени проводят в воде, выбираясь на лед только для рождения детенышей и во время линьки. Поскольку, как и у настоящих тюленей, у гренландских тюленей нет ушей, они имеют обтекаемые тела и хорошо плавают. Несмотря на то что ушастые тюлени, например морские львы, являются мощными пловцами, они считаются полуводными, потому что спариваются и отдыхают на суше.

Гренландский тюлень имеет серебристый мех с большими черными пятнами на спине, напоминающими по форме лиру или вилку. Они вырастают до 1,5–2 метров в длину и набирают вес до 130–180 кг. Поскольку эти животные живут в холодном климате, для обогрева они используют толстый слой жира. В рацион гренландского тюленя входят рыба и ракообразные. Он может оставаться под водой до 16 минут, добывая пищу, и может нырять на несколько сотен метров.

Гренландские тюлени рождаются без защитного слоя жира, но сохраняют тепло благодаря своему белому меху, адсорбирующему тепло от солнца. Уход за новорожденным тюленем продолжается 12 дней, после чего они становятся самостоятельными (хотя за это время они могут утроить свой вес благодаря материнскому молоку). В последующие недели, пока молодые тюлени не могут отплывать от льда, они очень уязвимы для хищников и теряют до половины своего веса. Выжившие тюлени достигают зрелости через 4–8 лет (в зависимости от пола) и живут в среднем 35 лет.

Обложка позаимствована из книги *Wood, Animate Creation*.

От издательства

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-сервер 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@dialektika.com`

WWW: `http://www.dialektika.com`

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152

ЧАСТЬ I

Язык

Компания Apple предоставляет широкий выбор инструментальных средств для разработки прикладных программ для операционной системы iOS, которые выглядят и ведут себя так, как вы того хотите. Этот инструментарий представляет собой API (*интерфейс прикладного программирования*). Для того чтобы использовать интерфейс API, вы должны говорить его языком. Этим языком по большей части является Objective-C, который представляет собой надстройку над языком программирования C. Некоторые части API используют сам язык C. Настоящая часть книги посвящена основам вышеупомянутых языков.

- Глава 1 посвящена языку программирования C. В общем случае вам, вероятно, не потребуется знание всех тонкостей этого языка, так что данная глава ограничивается только теми аспектами языка программирования C, которые нужны для того, чтобы пользоваться как API на основе Objective-C, так и API на основе языка программирования C.
- Глава 2 готовит основу для изучения Objective-C. Здесь рассматривается объектно-ориентированное программирование в терминах общей архитектуры, разъясняются некоторые чрезвычайно важные термины, которые будут использоваться во всей книге, а также концепции, лежащие в их основе.
- В главе 3 описывается базовый синтаксис Objective-C.
- В главе 4 продолжается описание Objective-C, рассматривается природа классов Objective-C с упором на создание классов в исходном тексте.
- Глава 5 завершает введение в язык Objective-C. Здесь рассматриваются создание и инициализация экземпляров, а также такие связанные темы, как полиморфизм, переменные экземпляров, функции доступа, ключевые слова `self` и `super`, кодирование ключ–значение и свойства.

В части III мы вернемся к описанию других аспектов языка программирования Objective-C, в частности, каркаса Cocoa.

ГЛАВА 1

Краткое описание языка C

Верите ли вы в C? Верите ли вы в судьбу?

Леонард Бернштейн (Leonard Bernstein)
и Стивен Шварц (Stephen Schwartz), Mass

Для того чтобы создавать приложения для операционной системы iOS, вам надо разговаривать с ней. Все, что вы скажете системе iOS, должно соответствовать ее интерфейсу API (т.е. интерфейсу прикладного программирования, представляющему собой список спецификаций того, что именно вы можете говорить.) Следовательно, вам нужны базовые знания языка программирования C, по крайней мере по следующим причинам.

- Большая часть интерфейса API системы iOS использует язык Objective-C, так что в основном программирование для iOS будет вестись вами на языке Objective-C. Objective-C является надмножеством языка программирования C. Это означает, что Objective-C включает в себя C; все, что верно для C, верно и для Objective-C. Распространенная ошибка заключается в игнорировании того факта, что "Objective-C — это C", и пренебрежении базовыми знаниями языка программирования C.
- Часть интерфейса API системы iOS основана на C, а не на Objective-C. Даже при кодировании на Objective-C часто требуется использовать структуры данных и вызовы функций C. Например, прямоугольник представлен в виде структуры C `CGRect`, а для создания объекта `CGRect` из четырех чисел следует вызвать функцию C `CGRectMake`. Очень часто документация интерфейса API системы iOS содержит выражения на языке программирования C и ожидает от вас их понимания.

Один из лучших способов изучить язык C — прочесть книгу Брайана Кернигана (Brian W. Kernighan) и Денниса Ритчи (Dennis M. Ritchie) *The C Programming Language* (PTR Prentice Hall, 1988)[1], известную также под аббревиатурой *K&R* (Ритчи был создателем языка программирования C). Это одна из лучших когда-либо написанных компьютерных книг: краткая, насыщенная, потрясающе точная и понятная. Книга K&R настолько важна для эффективного программирования для iOS, что она всегда лежит у меня на столе, когда я занимаюсь программированием (советую вам сделать то же самое). Еще одним полезным справочником я бы назвал книгу Майка Банахана (Mike Banahan), Диклана Брэди (Declan Brady) и Марка Дорана (Mark Doran) *The C Book*, доступную по адресу `http://publications.gbdirect.co.uk/c_book/`.

[1] Последний перевод этой книги на русском языке: Керниган Б., Ритчи Д. *Язык программирования C*, 2-е изд. — М.: ООО "И.Д. Вильямс", 2013, ISBN 978-5-8459-1874-1. — *Примеч. пер.*

Совершенно невозможно описать весь язык C в одной главе. Он не слишком большой и не слишком трудный, но имеет ряд "острых мест" и может быть исключительно тонким, мощным и низкоуровневым инструментом. Кроме того, поскольку язык C полно и корректно описан в вышеупомянутых книгах, было бы ошибкой повторять то, что уже было сказано в них лучше, чем мог бы сделать я. Так что данная глава — *не* техническое руководство по языку программирования C.

Для того чтобы эффективно использовать язык Objective-C, не обязательно знать о языке C *все*. Так что моя цель в этой главе — наметить те аспекты языка C, которые важно понять прежде всего, до того как начать пользоваться языком Objective-C для программирования в операционной системе iOS. Эта глава так и называется — "Краткое описание языка C". Она предназначена для комфортного и безопасного начала работы.

Если вы вообще ничего не знаете о языке C, предлагаю вам в дополнение к данной главе прочесть избранные места из книги *K&R*. Вот предлагаемый мною план ее чтения.

- Бегло ознакомиться с главой 1.
- Внимательно прочесть главы 2–4.
- Прочесть первые три раздела главы 5 об указателях и массивах. Оставшуюся часть упомянутой главы можно не читать, так как вам вряд ли придется работать с арифметикой указателей; но вы должны четко понимать, что такое указатель, поскольку в Objective-C почти все представляет собой объекты, а каждая ссылка на объект является указателем. Именно поэтому вы постоянно будете видеть (и использовать) символ `*`.
- Прочтите также первый раздел главы 6 книги *K&R* о структурах; как начинающий, вы, вероятно, не будете определять собственные структуры, но вам придется много и часто их использовать, так что вам надо понимать соответствующую запись (например, как я уже говорил, класс `CGRect` представляет собой структуру).
- Бросьте беглый взгляд на приложение Б в книге K&R[2], в котором рассмотрена стандартная библиотека; возможно, вам придется обращаться к стандартной библиотеке, например вызывать математические функции. Обычная ошибка начинающего программиста — забыть о существовании библиотеки.

Язык программирования C, определенный в книге *K&R*, не является в точности тем языком C, который образует основу для Objective-C. Со времен первого издания *K&R* язык постоянно развивался (появились стандарты ANSI C, C89, C99), да и компилятор Xcode также несколько расширяет стандартный язык программирования C. По умолчанию проекты Xcode рассматриваются как написанные на языке программирования, соответствующем стандарту GNU99 (расширению стандарта C99, хотя при желании вы можете указать и иной стандарт). К счастью, наиболее важные отличия языка из книги *K&R* и среды Xcode невелики и представляют собой улучшения для удобства использования, которые легко запомнить, так что *K&R* остается лучшим и наиболее надежным справочником по C.

Компиляция, инструкции и комментарии

Язык программирования C является компилируемым языком. Вы пишете свою программу в виде текста; для того чтобы ее запустить, следует выполнить два этапа. На первом текст

[2] Ссылка дается по русскому переводу книги *K&R*. — *Примеч. ред.*

компилируется в машинные команды; затем эти машинные команды выполняются. Таким образом, как и в любом компилируемом языке программирования, вы можете делать ошибки двух типов.

- Чисто синтаксические ошибки (означающие, что вы неверно говорите на языке C) отлавливаются компилятором, так что программа даже не пытается выполняться.
- Если ваша программа успешно прошла через компилятор, она будет выполняться, но нет никакой гарантии, что вы не допустили ошибок другого сорта, которые проявляются только в том, что программа ведет себя не так, как предполагалось.

Компилятор C привередлив, но вы должны принимать его замечания с благодарностью. Компилятор — ваш друг, поэтому полюбите его. Он может выдать сообщение, которое выглядит как не относящееся к делу или просто непонятное, но если это произошло, бессмысленно спорить с фактом, что вы сделали что-то неправильно, и компилятор услужливо сообщает об этом. Кроме того, компилятор может предупреждать вас, если что-то покажется ему возможной ошибкой, хотя все находится в рамках допустимого. Эти предупреждения, которые отличаются от сообщений о явной ошибке, также полезны и не должны вами игнорироваться.

Я уже говорил, что запуск программы требует выполнения предыдущего этапа компиляции. Но на самом деле есть еще один этап, который предшествует компиляции, — предварительная обработка (препроцессинг) исходного текста. Предварительная обработка изменяет исходный текст программы, так что когда обработанный текст передается компилятору, он не идентичен написанному вами. Предварительная обработка может показаться сложной и ненужной, но на самом она выполняется в соответствии с данными вами в исходном тексте командами и позволяет писать более ясный и компактный текст программы.

Среда Xcode позволяет просматривать влияние предварительной обработки на текст программы (выберите команду Product⇨Perform Action⇨Preprocess [Имя файла]), так что если вы думаете, что сделали ошибку в директивах препроцессора, то можете отследить его работу. Позже я расскажу о некоторых командах, которые можно отдать препроцессору.

C — язык инструкций. Каждая инструкция заканчивается точкой с запятой. (Отсутствие точки с запятой — распространенная ошибка новичков.) Для удобочитаемости программы пишутся преимущественно с одной инструкцией в строке, но это ни в коем случае не строгое правило: длинные инструкции (которые, к сожалению, возникают очень часто из-за многословности Objective-C) обычно разнесены по нескольким строкам, а очень короткие инструкции иногда пишутся по две-три в одной строке. Однако разбить строку где угодно просто так нельзя; например, строковый литерал не может содержать символ "возврата каретки". Отступы не несут никакого языкового смысла и являются вопросом соглашения о стиле программирования (о чем программисты на C часто спорят с почти фанатичным пылом); среда Xcode "разумно" помогает программисту, автоматически добавляя отступы, и этим можно воспользоваться, чтобы легко получить удобочитаемый код, и убедиться, что вы не сделали никаких грубых синтаксических ошибок.

Комментарии в языке C из книги *K&R* имеют вид `/*...*/`; сам комментарий между косыми чертами со звездочками может занимать несколько строк (*K&R* 1.2). В современных версиях C комментарий может располагаться после двух косых черт (`//`); правило применения такого комментария гласит, что все, следующее за двумя косыми чертами, до конца строки игнорируется:

```
int lower = 0; // Нижний предел температурной таблицы
```

Такие комментарии иногда называют комментариями в стиле C++, и для кратких пояснений и примечаний они существенно удобнее синтаксиса комментариев из книги *K&R*.

Язык программирования C (и, следовательно, Objective-C) чувствителен к регистру символов. Все имена чувствительны к регистру. Например, не существует такого типа, как `Int`, имеется тип `int`. Если вы объявите переменную `lower` типа `int`, а затем попытаетесь говорить о ней, как о `Lower`, компилятор пожалуется на то, что такая переменная ему неизвестна. По соглашению имена переменных начинаются со строчной буквы.

История компилятора

Изначально компилятором Xcode был свободный компилятор с открытым кодом GCC (`http://gcc.gnu.org`). Однако в конце концов Apple разработала собственный свободный компилятор с открытым кодом LLVM (`http://llvm.org`), известный также как Clang, тем самым сделав доступными все усовершенствования, которые были невозможны в рамках GCC. Изменение компилятора — слишком крутой поворот, который Apple выполняла поэтапно.

- В версии Xcode 3 наряду с компиляторами LLVM и GCC компания Apple предложила гибридный компилятор LLVM-GCC, который предоставлял преимущества компиляции LLVM, в то время как синтаксический анализ кода для максимальной обратной совместимости выполнялся с помощью GCC, при этом последний не является компилятором по умолчанию.
- В версии Xcode 4 компилятор LLVM-GCC стал компилятором по умолчанию, но GCC оставался доступным.
- В версии Xcode 4.2 компилятором по умолчанию стал LLVM 3.0, а чистый GCC был изъят.
- В версии Xcode 4.6 произошло обновление компилятора до LLVM 4.2.
- В версии Xcode 5 был изъят LLVM-GCC; текущая версия компилятора — LLVM 5.0, и на этом переход от GCC к LLVM полностью завершен.

Объявление, инициализация и типы данных переменных

Язык программирования C — строго типизированный. Каждая переменная до ее использования должна быть объявлена, и должен быть указан ее тип данных. Объявление может также включать явную инициализацию, придающую переменной значение; объявленная, но явно не инициализированная переменная имеет неопределенное значение (и должна рассматриваться как источник опасностей, пока она *не* инициализирована). В книге *K&R* объявления C должны предшествовать всем прочим инструкциям, но в современных версиях C это правило ослаблено, так что вы не обязаны объявлять переменную до самого начала ее использования:

```
int height = 2;
int width = height * 2;
height = height + 1;
int area = height * width;
```

Все фундаментальные встроенные типы данных в языке C являются числовыми: `char` (один байт), `int` (4 байта), `float` и `double` (числа с плавающей точкой) и такие разновидности, как `short` (короткое целое число), `long` (длинное целое), `unsigned short` и т.д. По желанию тип числового литерала может выразить с помощью буквы (или букв) суффикса:

например, 4 имеет тип `int`, но `4UL` имеет тип `unsigned long`; 4.0 представляет собой `double`, но `4.0f` относится к типу `float`. Язык Objective-C позволяет использовать некоторые другие числовые типы, производные от числовых типов языка C (с помощью инструкции `typedef`, см. *K&R* 6.7), разработанных для 64-битового процессора; наиболее важными из них являются `NSInteger` (наряду с `NSUInteger`) и `CGFloat`. Вам не нужно использовать их, если только интерфейс API явно не требует этого, и даже когда вы их используете, просто думайте об объекте типа `NSInteger` как о числе типа `int`, а об объекте типа `CGFloat` как о числе типа `float`, и все будет хорошо.

Для *приведения типа* значения переменной явным образом к другому типу перед именем переменной указывается имя этого другого типа в скобках:

```
int height = 2;
float fheight = (float)height;
```

В этом конкретном примере явное приведение не является необходимым, поскольку целое значение преобразуется в значение с плавающей точкой неявно при присваивании переменной соответствующего типа и дано в иллюстративных целях. В языке Objective-C вы встретите совсем мало приведений типов, в основном применяемых для того, чтобы компилятор не выдавал излишних предупреждений (с примерами вы ознакомитесь в главе 3).

Еще одним видом числовой инициализации являются перечисления, или `enum` (*K&R* 2.3). Это способ назначить имена последовательных числовых значений, и он полезен, когда значение представляет собой одно из нескольких возможных значений. Интерфейс API каркаса Cocoa широко применяет этот метод. Например, три возможных типа анимации строки состояния можно определить следующим образом:

```
typedef enum { UIStatusBarAnimationNone,
               UIStatusBarAnimationFade,
               UIStatusBarAnimationSlide,
             } UIStatusBarAnimation;
```

Это определение назначает значение 0 имени `UIStatusBarAnimationNone`, значение 1 имени `UIStatusBarAnimationFade` и значение 2 имени `UIStatusBarAnimationSlide`. При этом вы можете использовать значимые имена, не заботясь о числовых значениях (и даже не зная их), которые эти имена представляют. Это полезная идиома, и у вас может быть множество причин для определения перечислений в собственном коде.

Это определение также назначает общее имя `UIStatusBarAnimation` всего перечисления. Именованное перечисление не является типом данных, но вы можете притвориться, что это так, а компилятор может предупредить вас о смешивании типов перечислений. Предположим, например, что вы пишете следующий код:

```
UIStatusBarAnimation anim = UIInterfaceOrientationPortrait;
```

Это вполне разрешенное действие; `UIInterfaceOrientationPortrait` представляет собой не что иное, как другое имя для 0, так же, как если бы вы указали `UIStatusBarAnimationNone`. Однако оно взято из перечисления с другим именем, а именно `UIInterfaceOrientation`. Компилятор это обнаруживает и выдает соответствующее предупреждение. Как и в случае реальных типов данных, вы можете подавить вывод таких предупреждений с помощью приведения типов.

В системе iOS 7 анимация строки состояния определена следующим образом:

```
typedef NS_ENUM(NSInteger,UIStatusBarAnimation) {
    UIStatusBarAnimationNone,
    UIStatusBarAnimationFade,
    UIStatusBarAnimationSlide, };
```

Такая запись введена в компиляторе LLVM в версии 4.0, дебют которой состоялся в версии Xcode 4.4. NS_ENUM представляет собой макрос, вид текстовой подстановки препроцессором, который рассматривается в конце этой главы; при выполнении подстановки указанный код превращается в следующий:

```
typedef enum UIStatusBarAnimation :
            NSInteger UIStatusBarAnimation;
enum UIStatusBarAnimation : NSInteger {
            UIStatusBarAnimationNone,
            UIStatusBarAnimationFade,
            UIStatusBarAnimationSlide, };
```

Этот код выглядит почти так же, как и старое выражение того же перечисления, но новый способ включает некоторые обозначения, не являющиеся частью стандартного C, говорящие компьютеру о том, какие именно целочисленные значения здесь использованы (в данном случае NSInteger). Это делает UIStatusBarAnimation еще более похожим на истинный тип данных; кроме того, новая запись перечисления позволяет среде Xcode еще более разумно помочь вам при использовании автодополнения, описанного в главе 9. Еще один макрос, NS_OPTIONS, вычисляется в Objective-C как синоним NS_ENUM (они различны только в коде C++, который в данной книге не рассматривается).

Строка кажется фундаментальным текстовым типом в C, но на самом деле это иллюзия; за кулисами она выглядит как массив элементов типа char, завершающийся нулевым символом. Например, в C можно написать строковый литерал как

```
"string"
```

На самом деле это семь 7 байт, которые представляют собой числовые (ASCII) эквиваленты каждой буквы, последний из которых — байт с числовым значением нуль, обозначающий конец строки. Эта структура данных, именуемая C-строкой, редко встречается при программировании для iOS. В общем случае при работе со строками вы будете использовать тип объекта Objective-C, который называется NSString. Он полностью отличается от C-строки. Однако Objective-C позволяет записывать литералы NSString способом, очень похожим на строковый литерал C:

```
@"string"
```

Обратите внимание на символ @! Это выражение в действительности представляет собой директиву компилятору Objective-C о том, что надо создать объект класса NSString. Распространенная ошибка новичков — отсутствующий символ @, в результате чего выражение интерпретируется как C-строка, которая на самом деле является совершенно другой сущностью.

Поскольку запись литералов NSString моделируется записью C-строк, о них стоит знать еще кое-что (с чем вы, впрочем, можете и не встретиться). Так, в книге *K&R* перечислен ряд управляющих символов (*K&R* 2.3), которые можно использовать в литерале NSString, в том числе следующие.

- \n — Символ новой строки в Unix.
- \t — Символ табуляции.
- \" — Двойная кавычка (предваряется управляющим символом, чтобы указать, что это не конец строкового литерала).
- \\ — Обратная косая черта.

Объекты класса NSString изначально ориентированы на использование колировки Unicode, так что вполне можно использовать символы, не являющиеся ASCII-символами, непосредственно в литералах NSString; предупреждения об обратном являются устаревшими, и вы должны игнорировать их. Неплохо знать о наличии управляющих последовательностей \x и \u, но они вам вряд ли понадобятся.

В книге *K&R* также упоминается запись конкатенации строковых литералов, при которой несколько строковых литералов, разделенных только пробельными символами, автоматически объединяются и рассматриваются как один строковый литерал. Эта запись полезна при разбиении длинных символьных литералов на несколько строк для удобочитаемости, и Objective-C также использует это соглашение для литералов NSString, с тем отличием, что вы должны помнить о символе @:

```
@"Это большая длинная строка символов, "
@"которую я разделин на две строки исходного текста.";
```

Структуры

Язык C предлагает только несколько простых собственных типов данных. Каким же образом создавать более сложные типы данных? Существуют три способа: структуры, указатели и массивы. Структуры и указатели играют огромную роль при программировании для iOS. Массивы языка программирования C требуются гораздо реже, потому что Objective-C имеет свой собственный объектный тип NSArray.

Структура в языке C (*K&R* 6.1) представляет собой составной тип данных: она объединяет несколько типов данных в один, который рассматривается как единая сущность. Элементы, составляющие структуру, имеют имена, и к ним можно обращаться по этим именам с помощью записи с точкой. В интерфейсе API системы iOS предусмотрено множество структур, обычно вместе с функциями для работы с этими структурами.

Например, в документации iOS говорится, что структура CGPoint определена следующим образом:

```
struct CGPoint {
    CGFloat x;
    CGFloat y;
};
typedef struct CGPoint CGPoint;
```

Вспомним, что объект класса CGFloat представляет собой не что иное, как число с плавающей точкой, так что данный составной тип объединяет два простых встроенных типа данных; по сути, структура CGPoint имеет две части типа CGFloat с именами x и y. (Странно выглядящая последняя строка фрагмента кода просто позволяет использовать краткое имя типа данных CGPoint вместо более длинного struct CGPoint.) Таким образом, мы можем записать

```
CGPoint myPoint;
myPoint.x = 4.3;
myPoint.y = 7.1;
```

Так же как мы можем выполнить присваивание myPoint.x, чтобы *установить* значение этой части структуры, мы можем использовать имя myPoint.x, чтобы *получить* значение этой части структуры. Все выглядит так, как если бы myPoint.x было именем переменной.

Более того, элемент структуры сам может быть структурой, и запись с точкой может образовывать цепочки. Для иллюстрации рассмотрим еще одну структуру iOS, CGSize:

```
struct CGSize {
    CGFloat width;
    CGFloat height;
};
typedef struct CGSize CGSize;
```

Объединим структуры CGPoint и CGSize и получим структуру CGRect:

```
struct CGRect {
    CGPoint origin;
    CGSize size;
};
typedef struct CGRect CGRect;
```

Теперь предположим, что у нас есть уже инициализированная переменная типа CGRect с именем myRect. Тогда myRect.origin представляет собой структуру CGPoint, а myRect.origin.x — структуру CGFloat. Аналогично myRect.size представляет собой структуру CGSize, а myRect.size.width — структуру CGFloat. Можно непосредственно изменить одну только часть width структуры CGRect, записав код наподобие

```
myRect.size.width = 8.6;
```

Вместо инициализации структуры путем присваивания значения каждому из ее элементов можно инициализировать ее во время объявления, указав одновременно значения для всех ее элементов в фигурных скобках и разделив их запятыми:

```
CGPoint myPoint = { 4.3, 7.1 };
CGRect myRect = { myPoint, {10, 20} };
```

Вы не обязаны использовать инициализатор структуры только с помощью присваивания переменной соответствующего типа. Инициализатор можно использовать везде, где ожидается данная структура; но от вас может потребоваться использовать явное приведение к требуемому типу, чтобы пояснить компилятору, что означает выражение в фигурных скобках:

```
CGContextFillRect(con, (CGRect){myPoint, {10, 20}});
```

В этом примере CGContextFillRect — это функция. О функциях я расскажу несколько позже в данной главе, а пока главное в примере — что то, что идет после первой запятой, должно быть CGRect, так что здесь можно разместить инициализатор для типа CGRect, сопроводив его явным приведением к типу CGRect.

Указатели

Другим способом расширения набора типов данных в языке программирования C являются указатели (*K&R* 5.1). Указатель представляет собой целое число (того или иного размера), обозначающее адрес в памяти, где находятся реальные данные. Знание структуры реальных данных и принципы работы с ними, а также выделение блока памяти требуемого размера заранее и его освобождение, когда он больше не нужен, — это достаточно сложное дело. К счастью, язык Objective-C берет практически всю работу на себя и заботится о программисте настолько, что все, что вам действительно нужно знать, чтобы использовать указатели, — это то, что они существуют и какие обозначения нужны для обращения к ним.

Начнем с простого объявления. Если мы хотим объявить в языке программирования C целое число, мы пишем

```
int i;
```

Эта строка гласит: "i является целым числом". Теперь объявим *указатель* на целое число:

```
int* intPtr;
```

Смысл этой строки — "intPtr является указателем на целое число". Не имеет значения, откуда мы знаем, что по адресу, указываемому данным указателем, находится целое число, — здесь я только показываю обозначения, используемые при работе с указателями. Звездочку в объявлении можно ставить и перед именем, а не после типа данных:

```
int *intPtr;
```

Можно даже поставить с обеих сторон от звездочки по пробелу (хотя в реальных исходных текстах это делают редко):

```
int * intPtr;
```

Я предпочитаю первую запись, но временами прибегаю и ко второй, и компания Apple также достаточно часто ее использует, так что вы должны четко понимать, что это разные способы записи одной и той же сущности. Не имеет значения, сколько пробелов вы добавите, — имя типа будет int*. Если вы спросите, какому типу данных принадлежит переменная intPtr, ответом будет int* (указатель на int); звездочка в данном случае является частью имени типа этой переменной[3]. Если вам надо выполнить приведение переменной p к этому типу, следует записать выражение наподобие (int*)p. В этом выражении вполне допускаются пробелы перед звездочкой, так что то же выражение может быть записано как (int *)p.

Наиболее общим типом указателя является *обобщенный указатель*, или "*указатель на void*" (void*). Обобщенный указатель может использоваться везде, где используется указатель на конкретный тип данных. По сути, этот указатель опускает все проверки того, что именно находится в указываемом месте. Таким образом, следующий код вполне корректен:

```
int*  p1;   // Считаем, что p1 имеет конкретное значение
void* p2;
p2 = p1;
p1 = p2;
```

Указатели очень важны в языке Objective-C, поскольку в нем почти все является объектами (см. главу 2), а каждая переменная, ссылающаяся на объект, сама по себе является указателем. По сути, язык Objective-C использует преимущества того факта, что указатель в языке C может ссылаться на реальные данные произвольной природы. В данном случае реальные данные представляют собой объект языка Objective-C. Язык Objective-C знает, что это значит, но вас это не должно беспокоить — вы просто работаете с указателем языка C и позволяете Objective-C позаботиться о деталях. Например, я уже упоминал о том, что строковый тип в языке Objective-C называется NSString. Таким образом, способ объявления переменной NSString выглядит как объявление указателя на NSString:

```
NSString* s;
```

Литерал NSString представляет собой значение NSString, так что можно объявить и инициализировать этот объект NSString, записав действительно полезную строку кода Objective-C:

```
NSString* s = @"Hello, world!";
```

[3] Однако если в одном объявлении вы указываете несколько указателей, то звездочка должна находиться перед каждым именем переменной. Так, в объявлении int *ptr1, *ptr2; переменные ptr1 и ptr2 имеют тип данных int*, но в объявлении int* ptr1, ptr2; переменная ptr1 представляет собой указатель на int, а ptr2 — переменную типа int. — *Примеч. пер.*

В языке C, объявив указатель на целое значение под именем `intPtr`, вы позже обращаетесь к этому значению в коде, как в `*intPtr`. Это обозначение вне объявления означает "то, на что указывает указатель `intPtr`". Вы используете запись `*intPtr`, так как хотите получить доступ к целому числу "на дальнем конце" указателя, в том месте, куда он указывает. Такой процесс называется *разыменованием* указателя.

Но в языке Objective-C в общем случае все обстоит *не так*. В вашем коде вы будете рассматривать указатель на объект как объект; вам никогда не придется его разыменовывать. Так, например, объявив `s` как указатель на `NSString`, вы не будете использовать запись `*s`; вы будете просто использовать `s` так, как будто это просто строка. В языке Objective-C для выполнения операций над объектом используется указатель, который на него ссылается, а не сам объект. Язык Objective-C "за кулисами" сам позаботится обо всех необходимых действиях по обращению к блоку памяти, на который указывает переданный указатель, и сделает все, что нужно, в этом блоке памяти. Это чрезвычайно удобно для программиста, но приводит к определенным терминологическим неточностям: как правило, говорят, что "`s` является объектом класса `NSString`", хотя, конечно же, на самом деле это указатель на объект класса `NSString`.

Логика работы указателей как в языке C, так и в языке Objective-C отличается от логики работы простых типов данных. Разница становится особенно очевидной в случае присваивания. Присваивание простых типов данных изменяет значение данных. Присваивание указателя перенаправляет его на другие данные. Предположим, что `ptr1` и `ptr2` — указатели, и мы имеем код

```
ptr1 = ptr2;
```

Теперь `ptr1` и `ptr2` указывают на одно и то же место в памяти. Любые изменения данных, на которые указывает `ptr1`, будут изменять данные, на которые указывает `ptr2`, потому что их значения одинаковы (рис. 1.1). При этом на то, на что раньше, до присваивания, указывал указатель `ptr1`, теперь может не указывать ни один указатель, и это может быть очень плохо. Твердое понимание этих фактов имеет решающее значение при программировании на языке Objective-C, и мы еще вернемся к этой теме в главе 3.

Массивы

Массив C (*K&R* 5.3) состоит из нескольких элементов одного и того же типа данных. Объявление массива указывает тип данных элементов, за которым следует имя массива, а за ним — квадратные скобки, в которых указано количество элементов в массиве:

```
int arr[3]; // arr является массивом из трех 3 int
```

Для обращения к элементу массива используется имя массива, за которым следует номер элемента в квадратных скобках. Первый элемент массива имеет номер 0. Таким образом, можно инициализировать массив, присваивая значения поочередно каждому его элементу:

```
int arr[3];
arr[0] = 123;
arr[1] = 456;
arr[2] = 789;
```

Можно также инициализировать массив во время объявления, присваивая ему список значений в фигурных скобках, так же, как и в случае структуры. В таком случае размер массива в объявлении можно опустить, так как неявно он указывается списком значений инициализации (*K&R* 4.9):

```
int arr[] = {123, 456, 789};
```

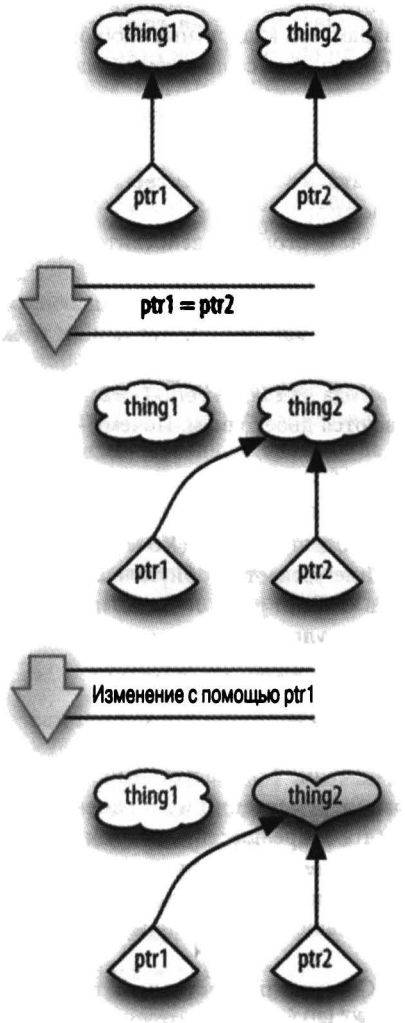


Рис. 1.1. Указатели и присваивание

Интересно, что имя массива является также именем указателя на первый элемент массива. Таким образом, например, имея показанное выше объявление массива, можно использовать `arr` там, где ожидается значение типа `int*` (указатель на `int`). Этот факт является основой некоторых высокоинтеллектуальных идиом языка программирования C, которые вам определенно не требуется знать (именно поэтому я и не рекомендовал вам читать из главы 5 *K&R* что-либо, кроме первых трех разделов).

Вот пример, в котором массив C может оказаться полезным при программировании для iOS. Функция `CGContextStrokeLineSegments` объявлена следующим образом:

```
void CGContextStrokeLineSegments(
        CGContextRef c,
        const CGPoint points[],
        size_t count
);
```

Вторым параметром функции передается С-массив элементов типа CGPoint (об этом говорят следующие за именем параметра квадратные скобки). Таким образом, для вызова функции необходимо знать как минимум, как создать такой массив. Это можно сделать, например, с помощью следующего кода:

```
CGPoint arr[] = {{4,5}, {6,7}, {8,9}, {10,11}};
```

После этого можно передать arr в качестве второго аргумента в вызов функции CGContextStrokeLineSegments.

Кроме того, как я упоминал, С-строка в действительности представляет собой массив. Например, метод NSString с именем stringWithUTF8String: получает (согласно документации) "С-массив байтов в кодировке UTF8 с завершающим нулевым символом", но этот параметр объявлен не как массив, а как char*. Это одно и то же, и оба способа говорят, что данный метод принимает С-строку.

(Двоеточие после имени метода stringWithUTF8String: — не опечатка; имена многих методов Objective-C заканчиваются двоеточием. Почему — я объясню в главе 3.)

Операторы

Арифметические операторы очень просты (*K&R* 2.5), но надо не забывать о следующем правиле: "целочисленное деление усекает дробную часть". Это правило является причиной ошибки множества новичков в С. Если у вас есть два целых числа и вы хотите разделить их так, чтобы получить дробный результат, вы должны представить по крайней мере одно из них как число с плавающей точкой:

```
int i = 3;
float f = i/2; // Осторожно! Не 1.5
```

Для того чтобы получить значение 1.5, следует записать i/2.0 или (float)i/2.

Целочисленные операторы инкрементации и декрементации (K&R 2.8), ++ и --, работают по-разному в зависимости от того, предшествуют ли они переменной или располагаются за ней. Выражение ++i заменяет значение i величиной, на 1 большей ее текущего значения, а затем использует это новое значение. Выражение i++ использует текущее значение i, а затем заменяет его величиной, на 1 большей текущего значения. Это одна из привлекательнейших возможностей С.

Язык программирования С также предоставляет программисту побитовые операторы (K&R 2.9), такие как побитовое "И" (&) и побитовое "ИЛИ" (|); эти операторы работают с отдельными битами, составляющими целые числа. Обычно чаще требуется оператор побитового "ИЛИ", поскольку интерфейс API каркаса Cocoa часто использует биты для одновременного указания нескольких вариантов выбора или настроек. Например, при указании, как именно UIView должен быть анимирован, можно передать аргумент, значение которого взято из перечисления UIViewAnimationOptions, определение которого начинается следующим образом:

```
typedef NS_OPTIONS(NSUInteger, UIViewAnimationOptions) {
    UIViewAnimationOptionLayoutSubviews          = 1 << 0,
    UIViewAnimationOptionAllowUserInteraction    = 1 << 1,
    UIViewAnimationOptionBeginFromCurrentState   = 1 << 2,
    UIViewAnimationOptionRepeat                  = 1 << 3,
    UIViewAnimationOptionAutoreverse             = 1 << 4,
    // ...
};
```

Символы << представляют собой оператор сдвига влево; правый операнд при этом указывает, на какое количество битов должен быть выполнен сдвиг левого операнда. В предположении, что `NSUInteger` состоит из 8 бит (это не так, но для простоты и краткости примем это предположение), это перечисление означает, что определены следующие пары имя-значение (значения показаны в двоичной системе счисления):

```
UIViewAnimationOptionLayoutSubviews          00000001
UIViewAnimationOptionAllowUserInteraction    00000010
UIViewAnimationOptionBeginFromCurrentState   00000100
UIViewAnimationOptionRepeat                  00001000
UIViewAnimationOptionAutoreverse             00010000
```

Причиной для такого представления на основе значений битов является то, что эти значения могут быть объединены в одно значение (битовой маски), которое вы передаете для задания параметров анимации. Каркас Cocoa может понять ваши намерения, выясняя, какие биты в переданном вами значении равны 1. Так, например, значение 00011000 будет означать, что вами выбраны значения `UIViewAnimationOptionRepeat` и `UIViewAnimationOptionAutoreverse` (и что все остальные, естественно, являются не выбранными).

Вопрос теперь в том, как создать эту битовую маску 00011000, чтобы передать ее в качестве аргумента. Это можно сделать чисто математически, так как двоичное значение 00011000 равно десятичному 24, и просто присвоить это значение аргументу. Но это не очень хорошая мысль, так как такой метод делает код запутанным, а при его написании очень легко ошибиться. Лучше вместо этого воспользоваться оператором побитового "ИЛИ" для объединения необходимых значений:

```
(UIViewAnimationOptionRepeat | UIViewAnimationOptionAutoreverse)
```

Эта запись корректно работает, потому что оператор побитового "ИЛИ" объединяет свои операнды, устанавливая в результате те биты, которые установлены хотя бы в одном операнде, так что `00001000 | 00010000` равно `00011000`, именно тому значению, которое мы и хотели получить. (А каким образом во время работы программы выясняется, какие именно биты маски установлены? С помощью оператора побитового "И".)

Простое присваивание (*K&R* 2.10) выполняется с помощью знака равенства. Однако в дополнение к нему имеются операторы составного присваивания, которые объединяют присваивание с некоторой другой операцией. Например,

```
height *= 2; // То же, что и height = height * 2;
```

Тернарный оператор (`?:`) представляет собой способ указать одно из двух значений в зависимости от условия (*K&R* 2.11). Схема применения этого оператора имеет вид (условие) ? выражение1 : выражение2

Если условие истинно (в следующем разделе вы узнаете, что это значит), вычисляется выражение1 и используется полученный результат; в противном случае вычисляется выражение2 и используется полученный результат. Например, можно использовать тернарный оператор при присваивании следующим образом:

```
myVariable = (условие) ? выражение1 : выражение2;
```

Что именно будет присвоено переменной `myVariable`, зависит от истинности условия. С помощью этого оператора нельзя сделать что-то, что невозможно достичь с помощью более многословного управления потоком выполнения, но тернарный оператор обладает исключительной ясностью, и потому я предпочитаю именно его.

Управление потоком выполнения

Управление потоком достаточно простое и обычно включает условие в круглых скобках и блок выполняемого кода в фигурных скобках. Эти фигурные скобки составляют новую область видимости, в которой можно вводить новые переменные, например:

```
if (x == 7) {
    int i = 0;
    i += 1;
}
```

После закрывающей фигурной скобки в четвертой строке переменная `i`, объявленная во второй строке, прекращает свое существование, так как ее область видимости располагается в этих фигурных скобках. Если содержимое фигурных скобок состоит из одной инструкции, то скобки можно опустить, но я не советую новичкам это делать, так как при этом очень легко запутаться. Распространенная ошибка новичков (которую, к счастью, отлавливает компилятор) — отсутствие круглых скобок вокруг условия. Полный набор инструкций управления потоком приводится в главе 3 *K&R*, и я просто подытожу их здесь в примере 1.1.

Пример 1.1. Конструкции управления потоком языка программирования C

```
if (условие) {
    инструкции;
}
if (условие) {
    инструкции;
} else {
    инструкции;
}
if (условие) {
    инструкции;
} else if (условие) {
    инструкции;
} else {
    инструкции;
}
while (условие) {
    инструкции;
}
do {
    инструкции;
} while (условие);
for (перед циклом; условие; после каждой итерации) {
    инструкции;
}
```

Структура `if...else if...else` может иметь любое необходимое количество блоков `else if`, и блок `else` является необязательным. Вместо расширенной структуры `if...else if...else if...else`, в случае, когда условия представляют собой сравнение единственной переменной с разными значениями, можно использовать конструкцию `switch`; однако при ее применении будьте осторожны, так как она достаточно запутанная и в ней легко ошибиться (за подробностями обратитесь к *K&R* 3.4). Главное — не забывать заканчивать каждый блок `case` инструкцией `break`, если только вы не хотите "провалиться" в следующий блок `case` (пример 1.2).

Пример 1.2. Инструкция switch

```
NSString* key;
switch (tag) {
    case 1: { // Т.е. если tag равно 1
        key = @"lesson";
        break;
    }

    case 2: { // Т.е. если tag равно 2
        key = @"lessonSection";
        break;
    }

    case 3: { // Т.е. если tag равно 3
        key = @"lessonSectionPartFirstWord";
        break;
    }
}
```

Цикл `for` в языке программирования C для новичков требует некоторой работы для понимания (см. пример 1.1). Инструкция перед циклом выполняется однократно, когда программа встречается с циклом, и обычно используется для инициализации счетчика. Затем проверяется условие, и если оно истинно, выполняется блок кода; обычно в условии проверяется, не достиг ли счетчик определенного граничного значения. После этого выполняется инструкция после каждой итерации, которая обычно применяется для увеличения или уменьшения счетчика; непосредственно за этим вновь проверяется условие. Таким образом, для выполнения блока, в котором переменная `i` принимает целые значения 1, 2, 3, 4 и 5, `i` используется следующий код:

```
int i;
for (i = 1; i < 6; i++) {
    // ... Инструкции ...
}
```

Потребность в счетчике, существование которого ограничено только рамками цикла, столь распространено, что стандарт C99 допускает объявление счетчика как часть инструкции перед циклом; область видимости объявленной переменной ограничена только кодом в фигурных скобках и инструкциями в круглых скобках заголовка цикла:

```
for (int i = 1; i < 6; i++) {
    // ... Инструкции ...
}
```

Цикл `for` является одной из немногих областей, в которых Objective-C расширяет синтаксис управления потоком языка программирования C. Некоторые объекты Objective-C, такие как `NSArray`, представляют перечислимые коллекции других объектов; "перечислимый" в данном случае означает, что можно выполнить цикл по элементам коллекции, и выполнение такого цикла и есть перечисление элементов коллекции. (Основные типы перечислимых коллекций рассматриваются в главе 10. Для упрощения перечисления Objective-C предоставляет конструкцию `for...in`, которая работает аналогично циклу:

```
SomeType* oneItem;
for (oneItem in myCollection) {
    // ... Инструкции ....
}
```

При каждой итерации цикла переменная `oneItem` (или как вы ее назовете) получает очередное значение из коллекции. Как и в цикле `for` стандарта C99, переменную `oneItem` можно объявить в заголовке цикла `for`, ограничивая тем самым ее область видимости фигурными скобками цикла:

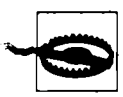
```
for (SomeType* oneItem in myCollection) {
    // ... Инструкции ....
}
```

Для досрочного прекращения работы цикла используется инструкция `break`. Для досрочного прекращения работы текущей итерации в фигурных скобках и перехода к следующей итерации цикла используется инструкция `continue`. В случае циклов `while` и `do` инструкция `continue` означает немедленное выполнение проверки условия цикла; в цикле `for` инструкция `continue` означает немедленное выполнение инструкции после каждой итерации, а затем проверку условия цикла.

В языке программирования C имеется также инструкция `goto`, которая позволяет перейти к именованной (с помощью метки) строке вашего кода (*K&R* 3.8); несмотря на то, что `goto` заведомо "рассматривается как вредная возможность", бывают ситуации, в которых она необходима, в основном из-за примитивности управления потоком в C.

Инструкция языка программирования C может быть составлена из нескольких инструкций, разделенных запятыми и выполняемыми последовательно. Последняя из этих инструкций является значением составной инструкции как единого целого. Такая конструкция, например, позволяет выполнять некоторые дополнительные действия перед каждым тестом условия или более чем одно действие после каждой итерации.

Вернемся теперь к вопросу о том, что собой представляет условие. В языке программирования C нет отдельного логического (булева) типа; вычисление условия либо дает значение 0, рассматриваемое как ложь, либо ненулевое значение, рассматриваемое как истина. Сравнения выполняются с помощью операторов эквивалентности и отношений (*K&R* 2.6); например, оператор `==` выполняет проверку на равенство, а оператор `<` выполняет проверку того, что первый операнд меньше второго. Логические выражения можно комбинировать с помощью операторов логического "И" (`&&`) и логического "ИЛИ"; применяя эти операторы вместе со скобками и оператором логического отрицания (`!`), можно формировать сложные условия. Вычисление выражений логического "И" и логического "ИЛИ" сокращенное, т.е. если левое условие однозначно определяет все выражение (ложное в случае "И" или истинное в случае "ИЛИ"), то правое условие не вычисляется.



Не перепутайте операторы логического "И" (`&&`) и логического "ИЛИ" (`||`) с операторами побитового "И" (`&`) и побитового "ИЛИ" (`|`), которые рассматривались ранее. Если вы напишете `&`, подразумевая `&&` (или наоборот), то можете получить результаты, которые вас немало удивят.

Оператор проверки равенства `==` отличается от обычного одинарного знака равенства; забыть об этом различии — одна из распространенных ошибок начинающих. Основная проблема в том, что код при этом остается корректным: простое присваивание, которое подразумевает одинарный знак равенства, имеет значение, и это значение вполне можно использовать в качестве условия. Рассмотрим следующий (бессмысленный) фрагмент кода:

```
int i = 0;
while (i = 1) {
```

```
    i = 0;
}
```

На первый взгляд начинающий программист может решить, что условие в цикле `while` проверяет, не равно ли значение `i` единице. Затем он может рассуждать так: значение `i` при входе в цикл равно нулю, так что цикл не будет выполнен ни разу. Верно? Нет. Условие в цикле `while` не проверяет, равно ли значение `i` единице, а присваивает `i` это единичное значение. Значение всего выражения присваивания также равно единице, так что условие цикла равно единице, что означает истину. Таким образом, тело цикла будет выполнено, и переменная `i` станет равной нулю. Но это ничего не значит, так как очередное вычисление условия снова присвоит переменной `i` единичное значение, условие будет истинно, и будет выполнена очередная итерация цикла. И так до бесконечности, так что программа просто зависает.

Программисты на C наслаждаются тем, что проверка на равенство нулю и проверка на ложность представляют собой одно и то же, и используют этот факт для создания компактных условных выражений, которые считаются элегантными и идиоматичными. Такие идиомы могут запутывать неискушенного программиста, но одна из них очень часто используется в Objective-C, а именно для проверки ссылки на объект, не является ли она пустой, имеющей значение `nil`. Поскольку значение пустой ссылки нулевое (см. главу 3), проверить, является ли ссылка `s` равной `nil`, можно следующим образом:

```
if (!s) {
    // ...
}
```

В языке Objective-C введен тип `BOOL`, который вы должны использовать, если нужно сохранить значение условия как переменную, и константы `YES` и `NO` (представляющие значения 1 и 0), используемые при установке значения логического типа. Не сравнивайте никакие значения со значениями типа `BOOL`, даже `YES` и `NO`, поскольку значение наподобие 2 представляет собой истину, но при этом не равно ни `YES`, ни `NO`. (Выполнение этих действий — распространенная ошибка начинающего программиста, которая может привести к нежелательным результатам, причину которых будет очень трудно найти.) Используйте переменную типа `BOOL` непосредственно в качестве условия или как часть сложного условия, и все будет хорошо. Например:

```
BOOL isnil = (nil == s);
if (isnil) {
    // не так: if (isnil == YES)
    // ...
}
```

Функции

Язык программирования C является языком на основе функций (*K&R* 4.1). Функция представляет собой блок кода, определяющий, что должно быть сделано. Когда другой код вызывает эту функцию, выполняется ее код. Функция возвращает значение, которое подставляется как результат вызова функции.

Вот определение функции, которая принимает целое число и возвращает его квадрат:

```
int square(int i) {
    return i * i;
}
```

Так выглядит вызов этой функции:

```
int i = square(3);
```

В силу ее определения приведенный код эквивалентен следующему:

```
int i = 9;
```

Это крайне простой пример, но он иллюстрирует многие ключевые аспекты функций. Проанализируем определение функции.

```
int❶ square❷(❸int i) {❹
    return i * i;
}
```

❶ Сначала указывается тип возвращаемого функцией значения; в данном случае это тип `int`.

❷ Затем указывается имя функции; в данном случае это `square`.

❸ После этого идут круглые скобки, в которых помещаются тип данных и имя всех значений, которые функция ожидает получить при вызове. В данном случае функция `square` ожидает получения одного значения типа `int`, которое мы называем `i`. Имя `i` (вместе с ожидаемым типом данных) является *параметром*; когда функция вызывается, его значение передается в функцию в качестве *аргумента*. Если функция ожидает получения более одного значения, несколько параметров в ее определении разделяются запятыми (при вызове функции передаваемые аргументы также разделяются запятыми).

❹ Наконец, в фигурных скобках находятся инструкции, выполняемые при вызове функции.

Эти фигурные скобки образуют область видимости; объявленные в ней переменные являются локальными для данной функции. Имена, используемые для параметров в определении функции, также локальны по отношению к функции. Другими словами, переменная `i` в первой строке определения функции — то же самое, что и `i` во второй строке определения, но она не имеет ничего общего с любой `i` вне определения функции (например, если результат вызова функции присваивается переменной с именем `i`). Значение параметра `i` в определении функции присваивается из соответствующего аргумента, передаваемого при вызове функции; в предыдущем примере это 3, поэтому результатом функции является 9. Таким образом, вызов функции с аргументами является формой присваивания. Предположим, что функция определяется следующим образом:

```
int myfunction(int i, int j) { // ...
```

Предположим также, что мы вызываем ее следующим образом:

```
int result = myfunction(3, 4);
```

Этот вызов функции присваивает значение 3 параметру `i` функции и значение 4 параметру `j`.

Когда встречается оператор `return`, сопутствующее ему значение передается обратно как результат вызова функции, и функция на этом завершается. Функция может не возвращать никакого значения; в таком случае оператор `return` не имеет сопутствующего значения, и тип данных, возвращаемых функцией, указывается в определении функции как `void`. Даже если функция возвращает значение, вызывающий код может его игнорировать. Например, вполне корректна запись

```
square(3);
```

Это выглядит довольно глупо — преодолеть все неприятности вызова функции для вычисления квадрата тройки и просто потерять полученное в результате вычисления значения. Это выглядит так, как будто мы написали код

```
9;
```

Вы можете сказать, что толку от этого мало. Но ведь, с другой стороны, цель функции может быть не столько в возврате некоторого значения, сколько в некоторых других действиях при ее выполнении, так что игнорирование возвращаемого значения вполне может иметь смысл.

Скобки в синтаксисе функции имеют важное значение. Скобки указывают C, что это функция. Скобки после имени функции в ее определении поясняют компилятору, что это определение функции, и они необходимы, даже если функция и не принимает параметры. Скобки после имени в вызове функции указывают C, что это вызов функции, и они необходимы, даже если это вызов функции без аргументов. Использование имени функции без скобок в языке программирования C возможно, потому что просто имя обозначает некоторую сущность (об этом мы поговорим позже), но это не вызов функции.

Вернемся к простому определению функции C и вызову, который уже встречался в примере ранее. Предположим, что я объединяю в одной программе определение функции и ее вызов:

```
int square(int i) {
    return i * i;
}
int i = square(3);
```

Это вполне корректная программа, но только потому, что определение функции `square` предшествует ее вызову. Если мы хотим поместить определение функции `square` в другом месте, например, после ее вызова, то вызову должно предшествовать по крайней мере объявление функции `square` (пример 1.3). Объявление выглядит так же, как первая строка определения, но за которой следует точка с запятой, а не левая фигурная скобка.

Пример 1.3. Объявление, вызов и определение функции

```
int square(int i);
int i = square(3);
int square(int i) {
    return i * i;
}
```

Имена параметров в объявлении не обязаны совпадать с таковыми в определении, но должны совпадать все типы (и, конечно, имя функции). Типы составляют *сигнатуру* функции. Другими словами, первая строка приведенного выше примера вполне может быть записана как

```
int square(int j);
```

Важно то, что и в объявлении, и в определении `square` представляет собой функцию, принимающую один параметр типа `int` и возвращающую значение типа `int`. (В современной программе Objective-C объявление функции обычно не требуется, даже если вызов функции предшествует ее объявлению; см. врезку "Функции и объявления методов в современной версии языка Objective-C").

В Objective-C, когда вы отправляете объекту сообщение (глава 2), вы используете не вызов функции, а вызов метода (глава 3). Тем не менее вы будете также использовать массу вызовов

функций C. Например, ранее мы инициализировали объект типа CGPoint, устанавливая значения ее элементов x и y, но обычно для того, чтобы создать новый объект CGPoint, используется вызов функции CGPointMake, которая объявлена следующим образом:

```
CGPoint CGPointMake(
    CGFloat x,
    CGFloat y
);
```

Несмотря на то что ее объявление располагается на нескольких строках и использует отступы, это настоящее объявление функции C, такое же, как объявление нашей простой функции square. Оно гласит, что CGPointMake представляет собой функцию C, которая принимает два параметра типа CGFloat и возвращает значение типа CGPoint. Так что теперь, надеюсь, вы знаете, что можно записать код с вызовом этой функции наподобие следующего:

```
CGPoint myPoint = CGPointMake(4.3, 7.1);
```

Параметры-указатели и оператор получения адреса

Язык Objective-C переполнен указателями (и звездочками), поскольку именно так он обращается к объектам. Методы Objective-C работают в основном с объектами, так что обычно они ожидают параметры-указатели и возвращают значения-указатели. Но это не усложняет программирование. Указатели — это то, что ожидает Objective-C, но одновременно это и то, что Objective-C дает вам. Так что никаких особых проблем не наблюдается.

Например, можно объединить две строки NSString с помощью вызова метода stringByAppendingString: объекта NSString. Документация говорит нам, что этот метод объявлен следующим образом:

```
- (NSString *)stringByAppendingString:(NSString *)aString
```

Это объявление говорит всем, кто понимает синтаксис Objective-C, что метод ожидает один параметр типа NSString* и возвращает значение типа NSString*. Это звучит неприятно, но все в порядке, поскольку *каждый* объект NSString на самом деле представляет собой NSString*. Так что не может быть ничего проще, чем получить новую строку NSString, состоящую из двух соединенных строк:

```
NSString* s1 = @"Hello, ";
NSString* s2 = @"World!";
NSString* s3 = [s1 stringByAppendingString: s2];
```

Однако иногда функция или метод ожидает в качестве параметра указатель на что-то; при этом у нас есть это "что-то", но нет указателя на него. Решение заключается в применении оператора получения адреса (*K&R* 5.1), который представляет собой амперсанд, размещенный перед именем объекта, адрес которого нам нужен.

Например, имеется метод класса NSString для чтения из файла в объект класса NSString, который объявляется следующим образом:

```
+ (id)stringWithContentsOfFile:(NSString *)path
                      encoding:(NSStringEncoding)enc
                         error:(NSError **)error
```

Не будем пока что беспокоиться о том, что такое id, и о синтаксисе объявления методов в языке Objective-C. Просто рассмотрим типы параметров. Первый из них — NSString*; здесь нет никаких проблем, поскольку каждая ссылка на NSString в действительности является

указателем на объект типа NSString. NSStringEncoding оказывается просто другим именем примитивного типа данных, NSUInteger, так что и здесь нет никаких проблем. Но что такое NSError**?

По логике NSError** должно быть указателем на указатель на NSError. И это в самом деле так. Этот метод требует передачи ему указателя на указатель на NSError. Объявить указатель на NSError легко:

```
NSError* err;
```

Но как получить указатель на этот указатель? С помощью оператора взятия адреса! Так что схематически наш код может выглядеть следующим образом:

```
NSString* path =       // Что-то там
NSStringEncoding enc = // Что-то там
NSError* err = nil;
NSString* result =
    [NSString stringWithContentsOfFile: path
                              encoding: enc
                                 error: &err];
```

Следует отметить важную вещь об амперсанде. Поскольку err является указателем на NSError, &err представляет собой указатель на указатель на NSError, именно то, что нам и надо обеспечить. Таким образом, все получается как надо. В главе 3, мы обсудим, как вы должны использовать возвращаемый результат и значение err после вызова метода.

Оператор взятия адреса можно использовать для создания указателя на любую именованную переменную. Функция С технически является своего рода именованной переменной, так что можно создать даже указатель на функцию! Это пример того, когда имя функции используется без скобок: в этом случае мы не вызываем функцию, а просто говорим о ней. Например, &square представляет собой указатель на функцию square. Кроме того, так же, как неявно имя массива без индекса является указателем на его первый элемент, так и имя функции без скобок неявно является указателем на функцию; оператор взятия адреса при этом является необязательным. В главе 3, описана ситуация, в которой указатель на функцию оказывается очень полезной вещью.

Файлы

По мере развития вашей программы ее можно разделить и организовать в виде нескольких файлов. Такой вид организации может сделать большую программу гораздо более легкой в обслуживании — ее легче читать, легче понимать, легче внести изменения и при этом не внести ошибки. Поэтому большая программа на языке программирования С обычно состоит из двух типов файлов: файлы с кодом, расширение имени файла которых обычно .c, и заголовочные файлы с расширением .h. Система построения автоматически будет "видеть" все файлы и будет знать, что все вместе они составляют одну программу. Однако при этом существует правило, согласно которому код в одном файле не может "видеть" другой файл, если только ему явным образом не указано иное. Таким образом, сам по себе файл представляет область видимости. Это преднамеренная (и весьма ценная) возможность языка С, потому что позволяет поддерживать порядок в вашем коде.

Для того чтобы один файл в С "увидел" другой файл, ему надо явно сказать об этом с помощью директивы #include. Знак # указывает, что данная строка представляет собой директиву препроцессора. В данном случае слово #include, за которым следует имя другого файла, является директивой, которая заставляет препроцессор просто заменить эту директиву полным содержимым указанного в директиве файла.

Стратегия построения большой программы на языке программирования C выглядит примерно следующим образом.

- В каждом `.c`-файле размещается код, о котором должен знать только этот файл; обычно код файла обеспечивает ту или иную функциональность программы.
- В каждый `.h`-файл помещаются объявления функций, которые могут потребоваться в нескольких `.c`-файлах.
- Каждый `.c`-файл включает те `.h`-файлы, в которых содержатся объявления необходимых ему функций.

Так, например, если функция `function1` определена в файле `file1.c`, но в файле `file2.c` может потребоваться вызов функции `function1`, то объявление функции `function1` можно поместить в файл `file1.h`. Теперь файл `file1.c` может включить файл `file1.h`, так что все его функции могут вызывать функцию `function1` независимо от расположения в файле по отношению к определению этой функции. Файл `file2.c` также может включить файл `file1.h`, так что все его функции также смогут вызывать функцию `function1` (рис. 1.2). Короче говоря, заголовочные файлы представляют собой способ обеспечить возможность совместного использования файлами с кодом знаний о другом коде без реального совместного использования самого кода (поскольку в случае совместного использования кода теряется сам смысл разделения кода на отдельные файлы).

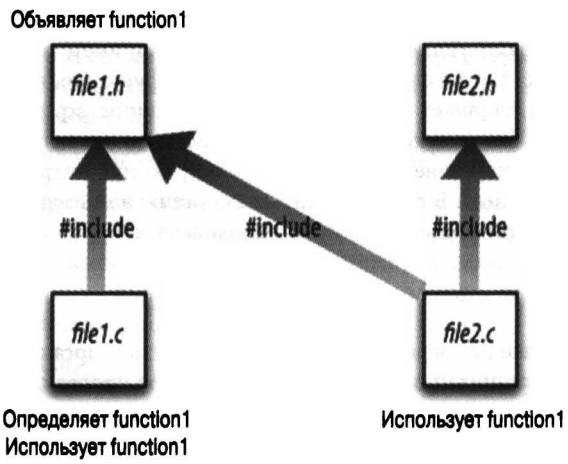


Рис. 1.2. Разделение большой программы на файлы

Но откуда компилятор знает, где среди всего множества `.c`-файлов найти место, с которого надо начать выполнение программы? Каждая реальная программа на языке программирования C содержит в каком-то месте ровно одну функцию с именем `main`, и она всегда является точкой входа для программы в целом: компилятор настраивает все так, что при выполнении программы вызывается функция `main`.

Описанная мною организация больших программ на языке программирования C, по сути, такая же, как и для ваших программ для iOS. Главное отличие будет в том, что вместо `.c`-файлов вы будете использовать `.m`-файлы, потому что по соглашению именно расширение `.m` говорит среде Xcode, что это файл на языке программирования Objective-C, а не на языке C.

Кроме того, если вы взглянете на любой проект Xcode для iOS, вы обнаружите, что он содержит файл с именем `main.m`; и если вы рассмотрите содержимое этого файла, то увидите, что он содержит функцию с именем `main`. Это точка входа в код приложения при его запуске.

Существенное отличие между файлами с кодом Objective-C и кодом C заключается в том, что вместо директивы `#include` в файлах Objective-C используется директива `#import`. Директива препроцессора `#import` не упоминается в *K&R*. Это дополнение Objective-C к языку программирования C. Она основана на директиве `#include`, но используется вместо нее, потому что содержит определенную логику для того, чтобы один и тот же материал не был считан и не вошел в файл с кодом более одного раза. Такое повторное включение представляет собой опасность, когда есть много зависимых один от другого заголовочных файлов; использование директивы `#import` позволяет аккуратно справиться с этой проблемой.

Кроме того, ваша программа для iOS состоит не только из ваших файлов кода и соответствующих им заголовочных файлов, но и файлов кода и соответствующих им заголовочных файлов Apple. Разница в том, что файлы кода Apple (которые составляют каркас Cocoa, см. часть III) уже скомпилированы. Однако ваш код должен по-прежнему использовать директиву `#import` для заголовочных `.h`-файлов Apple, чтобы иметь возможность видеть и использовать объявления Apple. Если вы взглянете на проект Xcode для iOS, то увидите, что любые `.h`-файлы, содержащиеся в нем по умолчанию, а также файл `main.m` содержат строку вида

```
#import <UIKit/UIKit.h>
```

Эта строка по сути представляет собой единую массивную директиву `#import`, которая копирует в вашу программу объявления всего базового API iOS. Более того, каждый из ваших `.m`-файлов импортирует соответствующий `.h`-файл, включая все импортируемое этим `.h`-файлом. Например, если у вас есть файл `AppDelegate.m`, он содержит строку

```
#import "AppDelegate.h"
```

Однако файл `AppDelegate.h` импортирует `<UIKit/UIKit.h>`. Таким образом, все ваши файлы с кодом включают все основные объявления iOS.

Как показывают приведенные примеры, директива `#import`, как и директива `#include` (*K&R* 4.11), может содержать имя файла в угловых скобках и в кавычках. Разные ограничители имени по-разному влияют на работу компилятора.

Кавычки. Компилятор ищет указанный файл в той же папке, где находится и включающий файл (`.m`-файл, в котором расположена строка `#import`).

Угловые скобки. Компилятор ищет указанный файл среди различных путей для поиска файлов, указанных в настройках компилятора. (Эти пути устанавливаются автоматически, и обычно вам не требуется их изменять.)

В общем случае угловые скобки используются для заголовочных файлов, владельцем которых является интерфейс API каркаса Cocoa, а кавычки — для заголовочных файлов, написанных вами. Если вам интересно, что именно импортирует директива `#import`, щелкните, удерживая нажатой клавишу <Command>, на имени импортируемого файла, чтобы просмотреть непосредственно включаемый файл.

В системе iOS 7 и среде Xcode 5 имеется дополнительный механизм импорта — модули, которые используются неявно, "за сценой", как часть процесса построения любого нового проекта iOS (и могут быть включены в старых проектах через настройки построения). Но вам никогда не придется непосредственно соприкасаться с ними. Основное назначение модулей — ускорение компиляции ваших проектов.

Без модулей процесс компиляции должен начинаться с буквального включения заголовочных файлов для UIKit и iOS API в каждом из ваших файлов. Например, ранее я говорил, что `CGPoint` определен как

```
struct CGPoint {
    CGFloat x;
    CGFloat y;
};
typedef struct CGPoint CGPoint;
```

После обработки ваших файлов препроцессором `.m`-файлы содержат определение `CGPoint` буквально — именно поэтому ваш код может использовать этот тип данных. Определение `CGPoint` и всех прочих импортируемых материалов добавляет к каждому из ваших файлов более 30 000 строк кода, с которым приходится иметь дело компилятору. Модули позволяют снизить накладные расходы. Импортируемый материал компилируется один раз, автоматически (обычно при создании или открытии вашего проекта), и кешируется в отдельном месте; этот кешированный код состоит из модулей. В процессе построения препроцессор вставляет в ваш код только несколько строк, таких как

```
@import UIKit;
@import Foundation;
```

(Для подтверждения можете выбрать Product⇨Perform Action⇨Preprocess [Имя файла].) Директива компилятора `@import` с символом `@` вместо `#` является новинкой в среде Xcode 5 и обращается к модулю по его имени. Поскольку модуль уже скомпилирован, никакой дополнительной работы компилятору делать не приходится. Более подробно о директиве `@import` и модулях речь пойдет в главе 6.

Объявления функций и методов в современном Objective-C

Начиная с компилятора LLVM версии 3.1, который был представлен в среде Xcode 4.3, Objective-C больше не требует, чтобы объявление функции предшествовало ее использованию, если определение этой функции располагается позже в этом же файле. Это относится как к функциям C, так и к методам Objective-C. Другими словами, код внутри класса Objective-C может вызывать функции C или методы Objective-C, даже если этот вызов предшествует определению данной функции или метода, и при этом не имеется отдельного объявления указанной функции или метода. Таким образом, в современном Objective-C порядок функций и методов в `.m`-файле не имеет значения, и даже не требуется объявлений функций и методов в `.m`-файле вовсе! Единственное место, где вы должны объявить функцию или метод, — `.h`-файл, так что `.m`-файл, отличный от того, в котором эта функция или метод определены, для возможности ее вызова должен всего лишь импортировать этот заголовочный файл.

Это соглашение является возможностью Objective-C, но не C. Я говорю только о `.m`-файлах, но не о `.c`-файлах. Более подробно о том, где в `.m`-файлах применимо это соглашение (а именно в разделе реализации класса), я расскажу в главе 4.

Стандартная библиотека

В вашем распоряжении имеется большая коллекция встроенных библиотечных файлов C. Библиотечный файл представляет собой централизованный набор функций C, вместе с сопутствующим `.h`-файлом, который можно включать в свой код, чтобы сделать эти библиотечные функции доступными для вашего кода.

Предположим, например, что мы хотим округлить число с плавающей точкой до ближайшего большего целого числа. Для этого используется библиотечная функция `ceil` ("потолок"). Об этой функции можно прочесть в руководстве, набрав в терминале команду `man ceil`. Документация говорит о том, какие директивы `#include` должны быть использованы для включения нужного заголовочного файла, а также показывает объявление функции и описывает выполняемые ею действия. Небольшая программа на языке программирования C, использующая эту функцию, может выглядеть следующим образом:

```
#include <math.h>
float f = 4.5;
int i = ceilf(f); // Теперь i равно 5
```

В программу для iOS файл `math.h` включается как часть UIKit, так что его не надо включать отдельно. Однако некоторые библиотечные функции могут потребовать явной директивы `#import`.

Стандартная библиотека рассматривается в приложении Б книги *K&R*. Однако со времени написания *K&R* библиотека несколько эволюционировала; и в настоящее время представляет собой надмножество библиотеки, описанной в *K&R*. Например, в приложении B в *K&R* описана функция `ceil`, но функции `ceilf` там нет. Аналогично, если вы хотите сгенерировать случайное число (например, в игровой программе, в которой желательно некоторое непредсказуемое поведение персонажа), вы, вероятно, не прибегнете к функции `rand`, описанной в *K&R*; вы можете использовать некоторые заменяющие ее функции, такие как `random` или даже `arc4random_uniform`.

Начинающие программисты очень часто забывают о том, что язык Objective-C по сути представляет собой язык C и что в нем есть множество готовых функций из стандартной библиотеки C.

Другие директивы препроцессора

Из других доступных программисту директив препроцессора чаще всего используется директива `#define`. За ней следует имя и значение; во время обработки исходного текста препроцессором имя в файле с кодом заменяется указанным значением. Как хорошо пояснено в книге *K&R* (раздел 1.4), это отличный способ избежать жесткого кодирования "магических чисел" в вашей программе, которое затрудняет ее понимание и поддержку.

Например, в приложении iOS, в котором некоторые текстовые поля расположены одно над другим, я хочу добиться, чтобы промежутки между ними были одинакового размера. Пусть этот промежуток имеет размер, скажем, равный 3.0. Я не должен использовать в своем коде 3.0 непосредственно. Вместо этого я пишу

```
#define MIDSPACE 3.0
```

Теперь вместо "магического числа" 3.0 мой код использует значащее имя `MIDSPACE`; встречая в исходном тексте `MIDSPACE`, препроцессор заменяет его текстом 3.0. В качестве премии за использование имени, если я позже решу, что это значение должно быть иным, мне не придется менять каждое встреченное 3.0 в файле на другое значение — достаточно будет изменить только одну строку с соответствующей директивой `#define`.

Директива `#define` выполняет простую замену текста, так что в качестве значения может быть использовано любое выражение. Часто требуется, чтобы это выражение было литералом `NSString`. Дело вот в чем. В каркасе Cocoa литералы `NSString` могут использоваться как ключ в словаре или имя уведомления. (Пока пусть вас не волнует, что такое словарь или уведомление.) Эта ситуация — прямое приглашение сделать ошибку. Если у вас в словаре есть

ключ @"mykey" и вы по ошибке где-то в коде введете его как @"myKey" или @"mikey", компилятор не будет жаловаться на это, но как положено программа работать не будет. Элегантным решением является определение имени для этого строкового литерала:

```
#define MYKEY @"mykey"
```

Теперь везде в своем коде вы используете MYKEY вместо @"mykey", и если ошибетесь (например, введете MYKKEY), то препроцессор не выполнит замену и компилятор *сообщит* об обнаруженной ошибке.

Директива #define может также использоваться для создания макроса (*K&R* 4.11.2), более сложного вида замены текста. При программировании для iOS вы встретитесь с некоторыми макросами Cocoa, но они будут казаться неотличимыми от функций; пусть их секрет пока остается нераскрытым, так как обычно макросы не касаются вас непосредственно.

Директива #warning заставляет среду Xcode вывести предупреждение при компиляции; эта возможность может использоваться для напоминания самому себе о некоторой недоделанной задаче или требовании:

```
#warning Не забыть исправить этот участок кода
```

Имеется еще такая полезная в среде Xcode директива, как #pragma; о ней я расскажу, когда мы будем рассматривать среду программирования Xcode в главе 9.

Квалификаторы типов данных

Тип данных переменной может быть объявлен с квалификатором перед именем типа. Это добавление влияет на то, как эта переменная может использоваться в программе. Например, объявлению может предшествовать слово const, которое означает (*K&R* 2.4), что изменять значение этой переменной нельзя; такая переменная должна быть инициализирована в той же строке объявления, и это единственное значение, которое она может иметь в программе.

Переменную, объявленную как const, можно использовать как альтернативный способ (вместо #define) предотвратить использование "магических чисел" и аналогичных выражений. Например:

```
NSString* const MYKEY = @"Howdy";
```

Интерфейс API каркаса Cocoa сам часто использует эту возможность. Например, в некоторых случаях каркас Cocoa передает вашему коду словарь с информацией. Документация сообщает, какие именно ключи содержит словарь. Но вместо того, чтобы передать вам ключи в виде строковых литералов, документация передает ключ как имя переменной типа const NSString:

```
UIKIT_EXTERN NSString *const
    UIApplicationStatusBarOrientationUserInfoKey;
```

(Сейчас неважно, что означает UIKIT_EXTERN.) Это объявление говорит вам, что UIApplicationStatusBarOrientationUserInfoKey является именем строки NSString и что вы можете довериться ее значению. Вы должны использовать это имя, когда вам нужен этот конкретный ключ. Вы можете быть уверены в том, что будет использовано нужное вам фактическое значение строки, несмотря на то, что вы не знаете, что это за значение. Если же вы допустите ошибку при вводе имени переменной, компилятор сообщит вам об ошибке, потому что вы будете пытаться использовать имя не определенной переменной.

Еще одним распространенным квалификатором является static. К сожалению, это ключевое слово используется в языке программирования C двумя способами; способ, которым

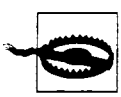
обычно пользуюсь я, применим в функции или методе. Здесь `static` указывает, что память, выделенная для переменной, не освобождается после возврата из функции или метода; переменная, таким образом, хранит свое значение до следующего вызова функции или метода. Статическая переменная полезна, например, когда вы хотите вызывать функцию многократно без накладных расходов на вычисление результата всякий раз заново (после первого вызова). Сначала проверим, вычислено ли уже значение статической переменной: если нет, то это должен быть первый вызов функции, и следует вычислить значение переменной; если же она уже вычислена, надо просто ее вернуть. Вот схема такого применения статической переменной:

```
int myfunction() {
    static int result = 0; // 0 означает, что вычислений
                           // еще не было
    if (result == 0) {
        // Вычисляем и сохраняем результат
    }
    return result;
}
```

Очень распространено использование статической переменной в Objective-C для реализации синглтона — единственного экземпляра класса, возвращаемого методом фабрики класса. Звучит пугающе, но не бойтесь — это не страшно. Ниже приведен пример из моего собственного кода, который вы можете понять, несмотря на то, что мы еще не обсуждали Objective-C.

```
+ (CardPainter*) sharedPainter {
    static CardPainter* sp = nil;
    if (nil == sp)
        sp = [CardPainter new];
    return sp;
}
```

Этот код гласит: если экземпляр `sp` типа `CardPainter` не был создан, его следует создать, и в любом случае его следует вернуть. Таким образом, независимо от того, сколько раз был вызван метод, экземпляр создается только один раз, и при каждом вызове возвращается одно и то же значение.



Статические переменные являются возможностью языка C, но не языка Objective-C. Таким образом, статическая переменная не знает ничего о классах и экземплярах; даже если она находится внутри функции или метода, она определена на уровне файла, что по сути означает — на уровне вашей программы в целом. Все в порядке, когда вы используете ее в методе фабрики класса, потому что класс является уникальным для вашей программы в целом. Но никогда не используйте статическую переменную в методе экземпляра Objective-C, потому что ваша программа может иметь несколько таких экземпляров и значение этой одной статической переменной будет использоваться во всех них. Другими словами, не используйте статическую переменную C как заменитель переменной экземпляра Objective-C (глава 2). Я однажды сделал эту ошибку, и, поверьте, этого урока мне вполне хватило.

ГЛАВА 2

Объектно-ориентированное программирование

Моя цель — очищение.
У.Ш. Гильберт (W.S. Gilbert), "Микадо"

Objective-C, "родной" язык для программирования интерфейса API каркаса Cocoa, является объектно-ориентированным языком, и чтобы его использовать, программист должен понимать природу объектов и объектно-ориентированного программирования. Мало смысла в изучении синтаксиса отправки сообщения Objective-C или создания экземпляра без четкого понимания того, что такое сообщение или экземпляр. Именно этому и посвящена данная глава.

Объекты

Объект в программировании основан на концепции объекта реального мира. Это независимая самодостаточная вещь. Такие объекты, в отличие от чисто инертных объектов реального мира, имеют определенные возможности. Объект в программировании, если можно так выразиться, больше напоминает часовой механизм, чем камень. Он не просто где-то находится, но еще и активно что-то делает. Возможно, лучше сравнивать объект в программировании с одушевленными объектами реального мира, в отличие от неодушевленных предметов, за исключением того, что в отличие от реальных одушевленных объектов объект программирования является предсказуемым: в частности, он делает то, что вы ему приказываете. В реальном мире вы можете приказать собаке сидеть, но при этом может произойти что угодно; в мире программирования вы говорите собаке сидеть, и она будет сидеть. (Вот почему так много программистов предпочитают иметь дело с программами и компьютерами, а не с реальным миром.)

В объектно-ориентированном программировании программа организована в виде множества дискретных объектов. Такая организация может сделать жизнь намного проще для программиста. Каждый объект обладает некоторыми способностями, которые являются специфичными для данного объекта. Вы можете представить это как работу автомобильного сборочного конвейера. Каждый работник или автомат вдоль конвейера делает только одну операцию (прикручивает бампер, красит дверь или делает что-то еще), и делает ее хорошо. Сразу же оказывается очевидно, как такая организация помогает программисту. Если автомобиль сходит с конвейера с плохо окрашенной дверью, весьма вероятно, что вина лежит на объекте, который занимается окраской двери, так что мы знаем, где искать ошибку в коде. Или, если мы решим изменить цвет двери, надо внести небольшие изменения только в объект,

окрашивающий дверь. Тем временем все другие объекты просто продолжают делать то, что они делают. Они ничего не знают и не должны знать ни об объекте, окрашивающем дверь, ни о том, как он работает. Объекты связаны с другими объектами только в той мере, в какой они взаимодействуют с ними. Эти взаимодействия имеют вид сообщений.

Сообщения и методы

В компьютерной программе ничто не происходит без указания о том, что оно должно произойти. В программе на языке программирования C весь код принадлежит функции и не выполняется, если функция не вызывается. В объектно-ориентированной программе весь код принадлежит объектам и не выполняется, пока объекту не прикажут его выполнить. Все действия в объектно-ориентированной программе происходят потому, что объект получил приказ действовать. Но что значит — отдать приказ объекту?

Объект в объектно-ориентированном программировании имеет точно определенный набор способностей — вещей, о которых он знает, как их делать. Например, вообразите объект, который представляет собаку. Мы можем разработать весьма упрощенную схему собаки, которая умеет выполнять только крайне ограниченный набор действий: есть, гулять, лаять, сидеть, лежать, спать. Цель этих умений объекта заключается в выполнении при необходимости соответствующих действий, когда вы говорите об этом объекту. Так что мы можем представить себе нашу схематичную собаку скорее как детскую игрушку-робот, способную реагировать на простые команды: Сидеть! Лежать! Голос!

В объектно-ориентированном программировании команда, отправленная объекту, называется *сообщением*. Для того чтобы заставить объект "собака" есть, мы отправляем сообщение `eat` (есть) объекту `dog` (собака). Этот механизм отправки сообщений является основой всей деятельности в программе. Программа полностью состоит из объектов, так что вся ее деятельность полностью состоит из отправки сообщений одним объектом другому. Сообщения имеют столь важное значение для деятельности объектно-ориентированной программы, что возникает соблазн предложить (несколько зацикленное) определение понятия *объекта* в терминах сообщений: *объект — это сущность, которой можно отправить сообщение.*

Минуту назад я сказал, что в программе на языке программирования C весь код принадлежит функции. Объектно-ориентированный аналог функции называется *методом*. Так, например, объект `dog` может иметь метод `eat`. Когда объекту `dog` отправляется сообщение `eat`, он отвечает на него вызовом метода `eat`.

Может показаться, что я не провожу четкого различия между сообщением и методом, но разница есть. Сообщение представляет собой то, что один объект говорит другому. Метод же представляет собой некоторый код, который вызывается. Связь между ними не идеально непосредственная. Вы можете отправить объекту сообщение, которое не соответствует методу этого объекта. Например, можно приказать собаке прочесть монолог Гамлета. Я не знаю, что произойдет, если вы это сделаете, — детали зависят от реализации. (Собака может просто сидеть молча или разозлится и укусит вас. А может, она возьмет томик Шекспира, выучит монолог и прочтет его.) Но эта зависимость от реализации и есть основным различием между сообщением и методом.

Тем не менее в целом различия между отправкой сообщения и вызовом метода обычно не самое важное в реальной жизни. Большую часть времени использования Objective-C причиной для отправки сообщения объекту является то, что этот объект реализует соответствующий метод, и вам нужен вызов этого метода. Так что отправка сообщения объекту и вызов метода объекта оказывается одним и тем же действием.

Язык программирования Objective-C является объектно-ориентированным; язык программирования C таковым не является. Поэтому мы говорим о функциях C, но о методах Objective-C.

Классы и экземпляры

Теперь мы переходим к очень характерным возможностям объектно-ориентированного программирования. Как и в реальном мире, каждый объект в мире объектно-ориентированного программирования имеет некоторый тип. Этот тип, именуемый *классом*, является объектно-ориентированным аналогом типа данных в C. Так же как простая переменная в C может иметь тип данных `int` или `float`, так и объект в мире объектно-ориентированного программирования может быть `dog` или `NSString`. В мире объектно-ориентированного программирования идея этого механизма заключается в том, сколько индивидуальных объектов будут действовать одинаковым образом.

Например, может быть несколько собак. Возможно, у вас есть собака Тузик, у меня — собака Шарик. Но обе наши собаки знают, как сидеть, лежать и лаять. В объектно-ориентированном программировании они знают это, потому что оба принадлежат к одному классу `Dog`. Перечисленные выше знания являются частью класса `Dog`. Ваш Тузик и мой Шарик обладают этим знанием только потому, что они являются объектами класса `Dog`.

С точки зрения программиста смысл сказанного прост: весь код, который мы создаем, размещается в классе. Все методы, которые мы пишем, будут частью того или иного класса. Вы не программируете отдельную собаку как отдельный объект — вы программируете класс `Dog`.

Я только что говорил, что объектно-ориентированная программа работает путем отправки сообщений отдельным объектам. Так что, даже несмотря на то, что программист не пишет код для отдельных "собачьих объектов", все равно в программе должны *существовать* отдельные собаки, чтобы было кому отправлять сообщения. Класс `Dog` знает, как лают, но только конкретная собака может залаять в ответ на команду. Соответственно возникает вопрос: если весь код собаки находится в классе `Dog`, то откуда берутся отдельные собаки?

Ответ заключается в том, что они должны создаваться в ходе работы программы. Когда программа запускается, она содержит код класса `Dog`, но не объекты отдельных собак. Чтобы получить лай какой-то собаки, программе необходимо сначала создать эту собаку — объект класса `Dog`. Этот объект будет принадлежать классу `Dog`, поэтому ему может быть отправлено сообщение "голос". Отдельный объект, принадлежащий классу `Dog` (или любому иному классу), является *экземпляром* этого класса. Изготовление отдельного объекта, являющегося экземпляром класса, называется *созданием экземпляра* этого класса.

Классы существуют как отражение того факта, что на первом месте находится программа: именно в них находится выполняемый код программы. Экземпляры же класса создаются преднамеренно и индивидуально в процессе работы программы. Каждый экземпляр создается из класса, является экземпляром этого класса и имеет методы в силу того, что эти методы имеет класс. Экземпляру могут быть отправлены сообщения; что он будет делать в ответ на получение сообщения, зависит от того, какой код класс содержит в своих методах. Экземпляр является отдельным объектом, которому можно отправлять сообщения; класс же с его методами является средоточием возможности реагировать на них (рис. 2.1).

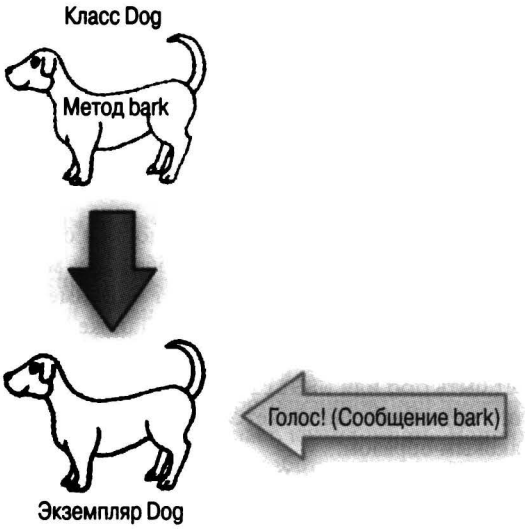


Рис. 2.1. Класс и экземпляр

Поскольку каждый отдельный объект является экземпляром класса, чтобы знать, какие сообщения официально можно отправлять этому объекту, необходимо по меньшей мере знать, какими методами наделил объект его класс. Такая открытая информация является API этого класса. (Класс может также иметь методы, которые вы не должны вызывать вне объекта; они не являются открытыми, и другие объекты не имеют права отправлять такие сообщения экземпляру данного класса.) Именно по этой причине документация каркаса Cacao от Apple состоит в основном из перечисления и описания методов, предоставляемых разными классами. Например, чтобы узнать, какие сообщения вы можете отправлять объекту (экземпляру) `NSString`, следует начать с изучения документации класса `NSString`. Это просто большой список методов, который говорит о том, что может делать объект `NSString`. Это хотя и не вся информация о классе `NSString`, но по крайней мере большая ее часть.

Методы класса

Я уже говорил, что в программе ничего не происходит до тех пор, пока объекту не отправляется сообщение. Но я также сказал, что экземпляров нет до тех пор, пока программа их не создаст. С самого начала в программе имеются только классы. Так как же создаются эти экземпляры? Для создания экземпляров во время работы программы должна иметься возможность отправить сообщение чему-то, иначе ничего не произойдет. Но если нет экземпляров, то куда же отправлять сообщение "сделай экземпляр"?

Ответ заключается в том, что классы сами по себе являются объектами и им могут быть отправлены сообщения. И действительно, одно из самых важных заданий, которое вы можете попросить сделать класс, отправив ему сообщение, — это создать свой экземпляр. Таким образом, есть шаг, отсутствующий на приведенном ранее рис. 2.1. На этом рисунке показан класс и экземпляр, с кодом в классе и сообщением, посланным экземпляру, но не показано, как был создан экземпляр класса. Более полная картина будет выглядеть так, как показано на рис. 2.2.

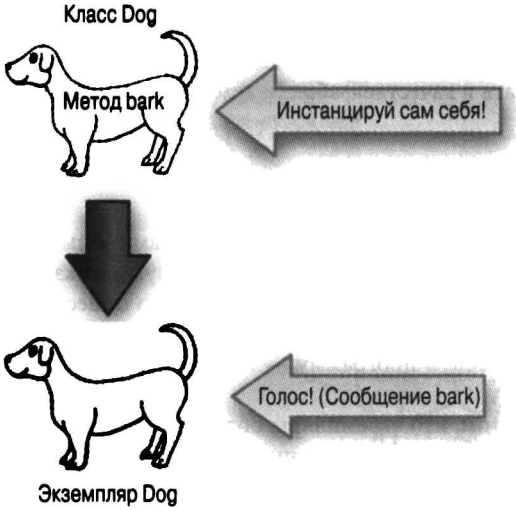


Рис. 2.2. Создание экземпляра класса

Таким образом, все начинает выглядеть так, как будто имеются два вида сообщений: сообщения, которые вы можете отправить классу (например, потребовать от класса `Dog` создания экземпляра собаки), и сообщения, которые вы можете отправить экземпляру (например, потребовать от конкретной собаки залаять). Это на самом деле так. Говоря более строго, весь код находится в методах класса, но сами методы бывают двух видов: методы класса и методы экземпляра. Если метод является методом класса, сообщение можно отправить классу. Если метод является методом экземпляра, сообщение отправляется экземпляру класса.

В синтаксисе Objective-C методы класса и методы экземпляра отличаются использованием знака "плюс" или "минус". Например, в документации класса `NSString` список его методов начинается следующим образом:

```
+ string
- init
```

Метод `string` является методом класса, а метод `init` — методом экземпляра.

В общем случае, хотя и не всегда, методами класса являются фабричные методы, то есть методы для создания экземпляров. Это имеет смысл, потому что скорее всего первое, что вы потребуете от класса, — это создать экземпляр самого себя. Вы можете решить, что классу в действительности нужен только один метод класса для создания своего экземпляра (и, строго говоря, это так и есть), но на самом деле классы, как правило, предоставляют несколько фабричных методов исключительно для удобства программиста. Например, вот три метода класса `NSString`:

```
+ string
+ stringWithFormat:
+ stringWithContentsOfFile:encoding:error:
```

Все они создают экземпляры. Первый метод класса, `string`, создает пустой экземпляр `NSString` (строка без текста). Второй метод класса, `stringWithFormat:`, создает экземпляр `NSString` на основе предоставленного вами текста, который может быть преобразован в текст других значений. Например, вы можете использовать его, чтобы из целого

числа 9 создать экземпляр NSString @"9". Третий метод класса считывает содержимое файла и генерирует экземпляр NSString из этого содержимого. Когда вы приходите к написанию своих собственных классов, вы также вполне можете создать несколько методов класса, которые будут действовать в качестве фабрики экземпляров для большего удобства программирования.

Переменные экземпляра

Теперь, когда я показал, что классы являются объектами и им могут быть отправлены сообщения, вы можете удивиться — зачем тогда вообще нужны экземпляры? Почему для объектно-ориентированного программирования недостаточно существования классов как объектов? Зачем вообще заниматься созданием экземпляров классов? Почему бы не писать вообще весь код как методы классов, а программу — как отправку сообщений от одного класса к другому?

Ответ заключается в том, что экземпляры обладают одной особенностью, которой нет у классов: переменными экземпляра. Переменная экземпляра представляет собой именно то, что следует из ее названия: это переменная, принадлежащая экземпляру. Подобно методам экземпляров, переменные экземпляров определяются как часть класса. Однако *значение* переменной экземпляра устанавливается во время работы программы и принадлежит только одному экземпляру. Иными словами, разные экземпляры могут иметь разные значения одной и той же переменной экземпляра.

Предположим, например, что у нас есть класс Dog и мы решили, что каждая собака должна иметь имя. Имя реального пса можно узнать по надписи на его ошейнике. Вот так же мы хотим иметь возможность назначать имя каждому экземпляру класса Dog, а впоследствии иметь возможность получать это имя. Поэтому при проектировании класса Dog мы объявляем, что этот класс имеет переменную экземпляра с именем name, значение которого представляет собой строку (вероятно, типа NSString, так как мы используем Objective-C). Теперь при работе нашей программы мы можем создать собаку (экземпляр класса Dog) и дать этой собаке имя (то есть присвоить значение переменной экземпляра name). Мы можем создать экземпляр класса Dog вновь и дать другое имя *этому* новому экземпляру класса. Скажем, это два разных имени: одно — @"Тузик", а второе — @"Шарик". В результате у нас есть два экземпляра класса Dog, которые отличаются значениями их переменных экземпляра name (рис. 2.3).

Таким образом, экземпляр является отражением методов экземпляра своего класса, но это не все; он также является совокупностью переменных экземпляра. Класс отвечает за то, какие переменные имеются у экземпляра, но не за значения этих переменных. Значения могут изменяться в процессе работы программы, и применимы они только к конкретному экземпляру. Экземпляр по сути представляет собой кластер значений переменных конкретного экземпляра.

Говоря короче, экземпляр объединяет код и данные. Код, который он получает от своего класса, он в некотором смысле разделяет со всеми другими экземплярами этого класса, но данные безраздельно принадлежат экземпляру. Данные могут сохраняться до тех пор, пока сохраняется сам экземпляр. В любой момент времени экземпляр имеет состояние — полную коллекцию значений своих собственных переменных экземпляра. Можно сказать, что экземпляр представляет собой устройство для поддержания состояния — эдакий ящик для хранения данных.

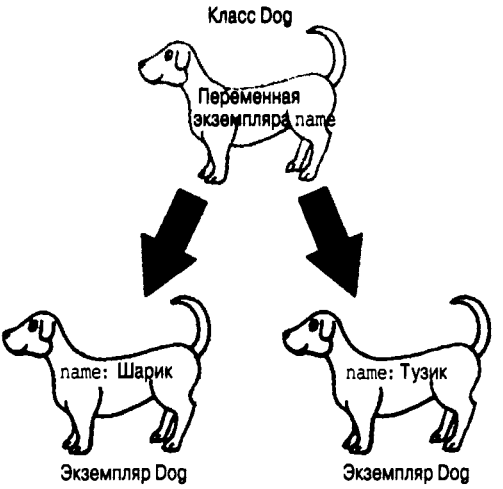


Рис. 2.3. Переменные экземпляров

Объектно-ориентированная философия

Подытожить природу объектов можно двумя фразами: инкапсуляция функциональности и поддержка состояния. (Много лет назад я уже воспользовался этими фразами в своей книге *REALbasic: The Definitive Guide.*)

Инкапсуляция функциональности

Каждый объект выполняет свою собственную работу и для внешнего мира — для других объектов, а в известном смысле, для программиста — представляет собой непрозрачный черный ящик, входами в который являются методы, которые он обещает выполнить при получении соответствующих сообщений. Вся информация о том, как именно реализованы эти методы и как именно выполняются действия, скрыта внутри объекта; никакие другие объекты об этом не знают и не должны знать.

Поддержка состояния

Каждый отдельный экземпляр представляет собой набор поддерживаемых им данных. Обычно эти данные являются закрытыми, что означает их инкапсуляцию; никакой другой объект не знает, что собой представляют эти данные и в каком виде они хранятся. Единственный способ ознакомиться с поддерживаемыми данными объекта извне — воспользоваться соответствующим методом, если таковой имеется.

В качестве примера представьте себе объект, работа которого заключается в реализации стека, — это может быть экземпляр класса `Stack`. *Стек* — это структура данных, которая поддерживает набор данных в порядке LIFO (last in, first out; последним вошел — первым вышел). Она реагирует на два сообщения: `push` и `pop`. Сообщение `push` означает добавление в набор данных нового элемента. Сообщение `pop` означает удаление из стека элемента, который был помещен в него последним. Стек можно представить как стопку тарелок: тарелки помещают в стопку сверху и забирают из стопки также сверху, так что тарелка, помещенная

в стопку первой, не может быть забрана из стопки, пока из нее не будут удалены все тарелки, помещенные туда после первой (рис. 2.4).

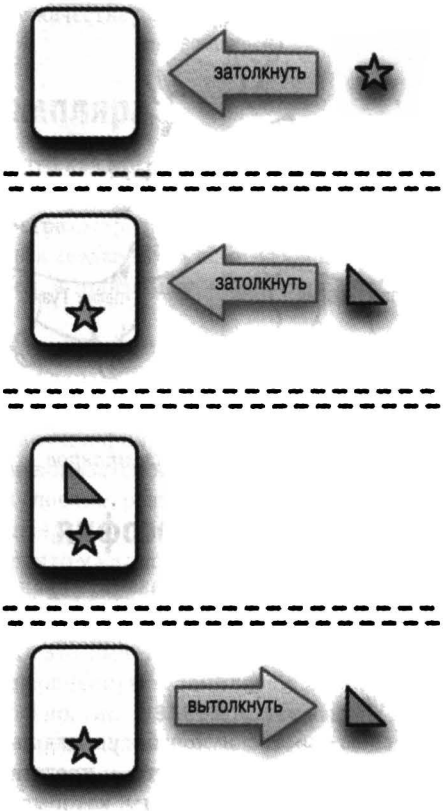


Рис. 2.4. Стек

Объект стека иллюстрирует инкапсуляцию функциональности, потому что внешний мир не знает ничего о том, как фактически реализован стек. Это может быть массив, это может быть связанный список, это может быть любая из множества других реализаций. Но объект-клиент, то есть тот, который фактически отправляет сообщения push и pop объекту стека, ничего об этом не знает, да и вообще не должен об этом беспокоиться: пока объект стека придерживается своего контракта и ведет себя как стек, все в порядке. Это хорошо и для программиста, который может по мере развития программы безопасно заменять одну реализацию стека другой без ущерба механизму программы в целом. И наоборот, объект стека ничего не знает и не заботится о том, кто именно отправляет сообщение push или pop и зачем. Он просто добросовестно выполняет то, о чем его просят, не интересуясь никакими внешними событиями и процессами.

Объект стека иллюстрирует поддержку состояния, потому что объект не просто способ доступа к данным стека — это сами данные стека. Другие объекты могут получить доступ к этим данным только в силу доступа к самому объекту стека и только таким образом, каким это позволяет делать объект стека. Данные стека полностью скрыты внутри объекта стека; никто другой не может их видеть. Все, что может сделать другой объект, — это послать объекту стека сообщения push и pop. Если на вершине стека имеется некоторый элемент, то любой

объект, который отправит стеку сообщение pop, получит в ответ этот элемент. Если никакой объект не отправляет сообщение pop объекту стека, то этот элемент на вершине стека будет просто ожидать, когда он кому-нибудь потребуется.

За вторым примером философии и природы объектно-ориентированного программирования я хочу обратиться к воображаемому сценарию, использованному мною в книге о REALbasic. Представьте, что мы пишем игру, в которой пользователь "стреляет" в движущиеся цели, и при каждом попадании в цель растет счет. Думаю, у вас сразу же получается представление о том, как можно организовать код игры с использованием объектно-ориентированного программирования.

- В игре имеется класс целевого объекта `Target`. Каждый целевой объект является экземпляром этого класса. Такое решение имеет смысл, потому что мы хотим, чтобы все цели вели себя одним и тем же образом. Цель должна знать, как изобразить себя на экране, и это знание является частью целевого класса — что имеет смысл, поскольку все цели будут рисовать себя одинаково. Таким образом, здесь мы имеем взаимоотношения между классом и экземпляром.
- Цели могут рисовать себя одним и тем же способом, но при этом они могут отличаться внешним видом. Возможно, некоторые из целей синие, другие красные, и так далее. Такое различие между отдельными целями может быть выражено с помощью переменной экземпляра; назовем ее `color`. Каждый раз при создании экземпляра мы будем назначать цвет объект. Код класса `Target` для рисования отдельных объектов будет получать значение переменной экземпляра `color` объекта и использовать его при рисовании. Очевидно, что можно не ограничиваться цветом: цели могут иметь различные размеры, формы и так далее, и все эти различия можно описывать с помощью переменных экземпляра. Таким образом, у нас есть как инкапсуляция функциональных возможностей, так и поддержка состояния. Объект цели имеет состояние (параметры, которые описывают, как цель должна выглядеть на экране), а также возможность изображать самого себя, т.е. визуально выражать это состояние.
- Когда пользователь попадает в цель, она взрывается. Вероятно, класс `Target` будет иметь метод экземпляра `explode`, и, таким образом, каждый объект цели знает, как надо взрываться. При взрыве цели должно выполниться еще одно действие — увеличение счета пользователя. Соответственно, представим еще один объект счета — экземпляр класса `Score`. Когда цель взрывается, одно из действий, выполняемых методом `explode`, — отправка сообщения `increase` (увеличить) объекту счета. Таким образом, у нас есть как инкапсуляции функциональных возможностей, так и поддержка состояния. Объект счета одинаково реагирует на любой объект, который отправляет ему сообщение `increase`; ему не нужно знать, почему он получил сообщение. Ему не надо знать ни о существовании объекта цели, ни о том, что сам он является частью игры — он просто поддерживает счет и при получении сообщения `increase` увеличивает его.

Когда мы представляем себе создание объектно-ориентированной программы для выполнения некоторой задачи, мы учитываем природу объектов, в общих чертах изложенную в этой главе. Есть классы и экземпляры. Класс является набором функциональных возможностей (методов), описывающих то, что могут делать все экземпляры этого класса (инкапсуляции функциональных возможностей). Экземпляры одного и того же класса отличаются только значениями своих переменных экземпляра (поддержка состояния). Соответственно этой модели выполняется и проектирование программы. Объекты представляют собой инструмент организации программы, набор черных ящиков, инкапсулирующих выполняющий

определенную задачу код. Они также являются концептуальным инструментом. Программист, вынужденный думать в терминах дискретных объектов, с необходимостью разделяет цели и поведение программы на дискретные подзадачи, каждая из которых решается соответствующим объектом.

Но объект не является островом. Объекты могут сотрудничать, общаясь один с другим, т.е. путем отправки сообщений друг другу. Соответствующие линии связи могут быть организованы бесчисленными способами. Конвейерная аналогия, которую я использовал в начале этой главы, иллюстрирует один такой способ — сначала с конечным продуктом взаимодействует объект 1; затем он передает его объекту 2, который выполняет над ним свои действия, и так далее. Но такой способ не подходит для большинства задач. Поиск надлежащего способа организации взаимоотношений между объектами — *архитектуры* приложения — является одним из самых сложных аспектов объектно-ориентированного программирования.

Эффективное применение объектно-ориентированного программирования для того, чтобы программа делала то, что вы от нее хотите, и при этом оставалась понятной и легко поддерживаемой — скорее искусство; по мере приобретения опыта ваши способности также будут улучшаться. Возможно, вы захотите почитать дополнительную литературу по эффективному планированию и построению архитектуры объектно-ориентированных программ. В таком случае позвольте порекомендовать вам две классические книги. Это *Refactoring* Мартина Фаулера (Martin Fowler) (Addison-Wesley, 1999), в которой описано, как почувствовать, как именно следует распределить методы по классам (и как победить свой страх перед этой работой). Другой книгой является *Design Patterns* Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides) (известных также как "Банда четырех") (Addison-Wesley, 1994), которая фактически стала Библией в области проектирования объектно-ориентированных программ.

ГЛАВА 3

Объекты и сообщения Objective-C

Одним из первых объектно-ориентированных языков программирования, получивших широкое распространение, был Smalltalk. Он был разработан в 1970-х годах в Xerox PARC под руководством Алана Кея (Alan Kay) и стал широко известен в начале 1980-х годов. Цель языка Objective-C, созданного Брэдом Коксом (Brad Cox) и Томом Лавом (Tom Love) в 1986 году, заключалась в создании синтаксиса, аналогичного языку Smalltalk, и расширении языка программирования C. Язык программирования Objective-C был лицензирован компанией NeXT в 1988 году и стал основой для интерфейса API каркаса приложений NeXTStep. В конце концов компании NeXT и Apple объединились, каркас приложений NeXT эволюционировал в Cocoa, каркас приложений OS X, по-прежнему основанный на языке Objective-C. Эта история поясняет, почему Objective-C является базовым языком программирования для iOS. (Это также объясняет, почему имена классов Cocoa часто начинаются с "NS" — это всего лишь означает "NeXTStep".)

Усвоив основы C (глава 1) и природу объектно-ориентированного программирования (глава 2), вы готовы к встрече с Objective-C. В этой главе описаны структурные основы Objective-C; следующие две главы предоставляют более подробные сведения о том, как работают классы и экземпляры Objective-C. (Несколько дополнительных возможностей языка обсуждаются в главе 10.) Как и в случае языка программирования C, мое намерение состоит не в том, чтобы полностью описать язык Objective-C, а в том, чтобы, основываясь на собственном опыте и знаниях, обеспечить фундамент для его практического применения, в первую очередь тех аспектов языка, которые необходимы как основа программирования для iOS.

Ссылка на объект является указателем

Ссылка представляет собой именно то, что вы себе представляете, — способ выбора некоторой определенной отдельной сущности. Особенно хороший вид ссылки — имя. Если мы хотим обратиться к Сократу, то утомительно каждый раз описывать его как "толстый лысый парень, который вечно задает дурацкие вопросы". Куда проще указать его по имени — "Сократ". Переменная языка программирования C представляет собой эквивалент имени. Присваивание некоторого значения переменной приводит к тому, что данная переменная (то есть имя) становится ссылкой на это значение.

В языке программирования C каждая переменная должна быть объявлена как имеющая некоторый тип данных. В языке C очень немного фундаментальных типов данных. Эти типы определенно не готовы к роли объектных типов. Для того чтобы добавить к C объекты и тем самым превратить C в объектно-ориентированный язык C, Objective-C использует гибкость указателей C (см. главу 1). Указатель представляет собой тип данных C, но при этом он может

указывать на все, что угодно. Таким образом, в Objective-C каждая ссылка на объект является указателем (и Objective-C сам заботится о выяснении, на что этот указатель указывает).

Тот факт, что ссылки на объекты в Objective-C являются указателями, особенно очевидно в случае ссылки на экземпляр (см. главу 2). В объектно-ориентированном языке, таком как Objective-C, типом экземпляра является его класс. Таким образом, хотелось бы использовать в Objective-C имя класса так, как в C мы используем имя любого типа данных. И позволяют это сделать именно указатели. С одной стороны, указатели удовлетворяют требованию C о том, что ссылка должна быть некоторым определенным типом данных C, а с другой стороны — требованию Objective-C о том, что мы должны иметь возможность указать любой тип класса из огромного их множества. Если в Objective-C переменная явно ссылается на экземпляр класса `MyClass`, то она имеет тип `MyClass*`, т.е. является указателем на `MyClass`. В общем случае в Objective-C ссылка на экземпляр класса является указателем, а имя типа данных, на который указывает этот указатель, представляет собой имя класса этого экземпляра.

Обратите внимание на соглашение об использовании прописных букв. Обычно имена переменных начинаются со строчной буквы, а имена классов — с прописной.

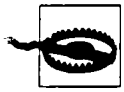
Как упоминалось в главе 1, тот факт, что в Objective-C ссылка на экземпляр является указателем, обычно не вызывает каких-либо трудностей, так как указатели последовательно используются везде в языке. Например, сообщение, направленное экземпляру, отправляется указателю, так что нет необходимости в разыменовании последнего. Действительно, установив, что переменная, представляющая экземпляр, является указателем, можно забыть об этом факте и просто непосредственно работать с этой переменной:

```
NSString* s = @"Hello, world!";
NSString* s2 = [s uppercaseString];
```

Один раз установив, что переменная `s` имеет тип `NSString*`, вы больше никогда не будете разыменовывать `s` (т.е. никогда не будет говорить о `*s`) для доступа к "реальным" строкам `NSString`. Так что все выглядит так, как будто указатель является реальным типом `NSString`. Таким образом, в предыдущем примере, после того как переменная `s` была объявлена как указатель на объект класса `NSString`, сообщение `uppercaseString` отправляется непосредственно этой переменной. (Сообщение `uppercaseString` запрашивает у `NSString` создание и возвращение версии хранимой объектом строки в верхнем регистре; таким образом, после выполнения приведенного кода `s2` представляет собой строку `@"HELLO, WORLD!"`.)

Связь между указателем, экземпляром и классом этого экземпляра столь тесная, что вполне естественно говорить о выражении наподобие `MyClass*` как об означающем "экземпляр `MyClass`", а о значении `MyClass*` как о "`MyClass`". Программист на языке Objective-C, говоря о приведенном выше примере, просто сказал бы, что `s` представляет собой `NSString`, что `uppercaseString` возвращает `NSString` и так далее. Это прекрасно, что можно так говорить, и я постоянно делаю это сам — надо только помнить, что это просто в определенном смысле сокращение. Такое выражение на самом деле означает "экземпляр `NSString`", что, в свою очередь, означает, что поскольку экземпляр представлен в виде указателя C, то он является не чем иным, как `NSString*`, т.е. указателем на `NSString`.

Хотя тот факт, что ссылки на экземпляры в Objective-C являются указателями, не вызывает каких-либо особых трудностей, вы все равно должны осознавать, что такое указатели и как они работают. Как я подчеркивал в главе 1, когда вы работаете с указателями, ваши действия имеют особый смысл. Так что вот некоторые основные факты об указателях, которые вам следует иметь в виду при работе со ссылками на экземпляры в Objective-C.



Распространенная ошибка начинающего — забыть о звездочке в объявлении экземпляра, на что компилятор позже отреагирует загадочными сообщениями типа "Interface type cannot be statically allocated".

Ссылки на экземпляры, инициализация и nil

Простое объявление типа ссылки на экземпляр не влечет за собой существование соответствующего экземпляра. Например:

```
NSString* s; // Только объявление; экземпляра, на который
             // указывает s, не существует
```

После этого объявления `s` имеет тип указателя на `NSString`, но на самом деле эта переменная не указывает на `NSString`. Вы создали указатель, но не предоставили объект `NSString`, на который он должен указывать. Он пока что ждет, пока вы укажете, на что он должен указывать (обычно путем присваивания, как мы это делали с помощью строки `@"Hello, world!"` выше). Такое присваивание инициализирует переменную, придавая ей первоначальное имеющее смысл значение надлежащего типа.

Можно объявить переменную как ссылку на экземпляр в одной строке кода, а инициализировать ее позже, как в приведенном примере:

```
NSString* s;
// ... Прошло время ...
s = @"Hello, world!";
```

Однако обычно так не делается. Гораздо более распространена практика объявления и инициализации переменной в одной строке кода (если это возможно):

```
NSString* s = @"Hello, world!";
```

Объявление без инициализации до появления механизма ARC (автоматического подсчета ссылок, см. главу 12) приводило к достаточно опасным ситуациям.

```
NSString* s;
```

Без механизма ARC переменная `s` после объявления может иметь *любое* значение. Беда в том, что эта переменная *претендует* на то, чтобы являться указателем на `NSString`. Обманувшись, вы можете *рассматривать* мусор в памяти, на который указывает переменная `s`, как строку `NSString`. Это может привести к серьезным неприятностям, если вы попытаетесь использовать мусор в качестве экземпляра. Отправка сообщения мусору или использование его иным образом может привести к сбою программы. Еще хуже — это может *не* привести к сбою программы, а всего лишь заставить ее работать неверно, — и при этом будет очень, очень сложно выяснить, в чем же причина такого поведения программы.

Назначение ссылке значения `nil` приводит к тому, что, хотя она и не указывает ни на какой реальный экземпляр, она не указывает и на мусор. До появления механизма ARC распространенной защитой от "мусорных указателей" была инициализация каждого указателя в момент его объявления значением `nil`, если инициализация реальным объектом по каким-то причинам оказывается в этот момент невозможной:

```
NSString* s = nil;
```

Небольшим, но очаровательным дополнением к возможностям механизма ARC является то, что такое присваивание выполняется автоматически, неявно и незаметно, в случае объявления переменной без ее инициализации:

```
NSString* s; // ARC присваивает s значение nil
```

Что такое nil? Это разновидность нуля — нуль, который подходит для присваивания ссылке на экземпляр. Значение nil просто означает: "эта ссылка на экземпляр не указывает ни на один реальный экземпляр". Действительно, вы можете проверить, указывает ли ссылка на реальный экземпляр, сравнив ее с nil. Это очень распространенная практика:

```
if (nil == s) // ...
```

Как я упоминал в главе 1, явное сравнение с nil не является строго необходимым; поскольку nil является разновидностью нуля, а также, так как нуль в условии означает ложь, тот же тест можно выполнить проще:

```
if (!s) // ...
```

Я буду использовать именно эту, вторую разновидность проверки, но некоторые программисты будут считать, что у меня плохой стиль программирования. Первая разновидность имеет то преимущество, что ее реальный смысл сразу становится очевиден всем и не приходится полагаться на некоторую неявную функциональность C. В первой разновидности nil также преднамеренно занимает место слева от оператора ==, так что если программист случайно пропустит один знак равенства, выполняя вместо сравнения присваивание, компилятор обнаружит эту ошибку (потому что присваивать что-то nil нельзя).

Многие методы Cocoa используют возвращаемое значение nil вместо ожидаемого экземпляра для указания, что что-то *выполнено* не так, как надо. Чтобы убедиться, что вызов метода прошел успешно, вы должны проверить возвращаемое значение на равенство nil. Например, документация метода stringWithContentsOfFile:encoding:error: класса NSString гласит, что он "возвращает строку, полученную путем считывания данных из файла с именем path, используя кодировку enc. Если файл не может быть открыт или имеется ошибка кодирования, возвращается значение nil". Так что, как я писал в главе 1, этот метод должен вызываться следующим образом:

```
NSString* path =          // Какое-то имя
NSStringEncoding enc =    // Какое-то значение
NSError* err = nil;
NSString* result =
    [NSString stringWithContentsOfFile: path
     encoding: enc error: &err];
```

Почему stringWithContentsOfFile:encoding:error: имеет такой вид? По сути этот метод может возвращать *два* результата. Он возвращает строку, которую мы сохраняем, присваивая ее указателю на NSString, и которую мы называем результатом. Но если произошла ошибка, то метод задает значение другого объекта, а именно объекта NSError. Идея заключается в том, что вы можете впоследствии изучить этот объект NSError и выяснить, что именно пошло не так. (Возможно, не было файла с указанным именем, или он имел неверную кодировку.) Передавая указатель на указатель на NSError, вы позволяете методу выполнить описанное действие. Перед вызовом stringWithContentsOfFile:encoding:error: переменная err была инициализирована значением nil; во время вызова stringWithContentsOfFile:encoding:error:, если произошла ошибка, указатель перенаправляется, тем самым присваивая err значение NSError, описывающее ошибку. (Перенаправление указателя таким образом иногда называют *косвенным* обращением.)

Таким образом, следующий шаг после вызова этого метода и сохранения результата — проверить равенство результата nil, просто чтобы убедиться, что действительно получен экземпляр строки. Если результат не nil, все хорошо; это и есть та строка, которую вы хотели получить. Но если результат *равен* nil, то надо исследовать NSError, чтобы узнать, что пошло неправильно.

```
NSString* path =        // Какое-то имя
NSStringEncoding enc = // Какое-то значение
NSError* err = nil;
NSString* result =
    [NSString stringWithContentsOfFile: path
     encoding: enc error: &err];
if (nil == result) { // Ой! Что-то пошло не так...
    // Из err можно узнать, что именно случилось
}
```

Этот шаблон часто используется в каркасе Cocoa. *Не поймите его неправильно!* Очень распространенная ошибка начинающих программистов — вызов метода `stringWithContentsOfFile:encoding:error:` с немедленной проверкой значения переменной ошибки (в данном случае — `err`). Не делайте этого! Если ошибки не было, значение переменной `err` никак не гарантируется. Вместо этого начните с проверки результата (в данном случае — переменной `result`). И только если результат показывает, что произошла ошибка, то вот тогда и только тогда вы должны исследовать значение `NSError`, которое было установлено путем косвенного обращения.

В исходном тексте на чистом C иногда приходится встречаться с "указателем на ничто", выраженным как `NULL`. Функционально `NULL` и `nil` эквивалентны, так что не стесняйтесь везде использовать только `nil`.

Ссылки на экземпляры и присваивание

Как я уже говорил в главе 1, присвоение указателю не приводит к изменению значения, которое находится "на дальнем конце указателя", на которое он указывает; это просто перенацеливание указателя. Кроме того, присваивание значения одного указателя другому перенацеливает указатель таким образом, что после этого оба указателя указывают на один и тот же объект. Если вы забудете эти простые факты, результаты будут в диапазоне от удивительных до катастрофических.

Предположим, что мы реализовали класс `Stack`, описанный в главе 2, и рассмотрим следующий код:

```
Stack* myStack1 = // ... Создание экземпляра Stack
                  // и инициализация myStack1 ...
Stack* myStack2 = myStack1;
```

Распространенное заблуждение — что присваивание `myStack2 = myStack1` создает новый, отдельный экземпляр, который дублирует `myStack1`. Это вовсе не так. Присваивание не создает новый экземпляр; оно просто приводит к тому, что `myStack2` указывает на тот же экземпляр, что и `myStack1`. Да, можно сделать новый экземпляр, который дублирует данный, но такая возможность не является данностью, а это действие не выполняется путем простого присваивания. (Как создаются экземпляры-дубликаты, описано в главе 10; см. описание протокола `NSCopying` и метода `copy`.)

Кроме того, в общем случае экземпляры изменяемые: они, как правило, имеют переменные экземпляров, которые можно изменить. Если две ссылки указывают на один и тот же экземпляр, то когда экземпляр изменяется с помощью одной ссылки, это изменение становится видимым через другие ссылки.

```
Stack* myStack1 = // ... Создание экземпляра Stack
                  // и инициализация myStack1 ...
Stack* myStack2 = myStack1;
[myStack1 push: @"Hello"];
```

```
[myStack1 push: @"World"];
NSString* s = [myStack2 pop];
```

Когда мы снимем элемент с вершины стека myStack2, значением переменной s окажется @"World", хотя мы ничего не помещали в стек myStack2; стек же myStack1 после этого содержит только строку @"Hello", хотя мы ничего не удаляли из стека myStack1. Это связано с тем, что мы внесли две строки в myStack1 и удалили одну строку из myStack2, а myStack1 является myStack2 — в том смысле, что это два указателя на один и тот же экземпляр стека. Это прекрасно до тех пор, пока вы понимаете и правильно используете это поведение.

Впрочем, иногда такое поведение является нежелательным. Если ваша программа имеет более чем одну ссылку на один и тот же экземпляр, можно получить удивительные результаты только потому, что (как и в предыдущем примере) этот экземпляр можно изменить с помощью одной ссылки в то время, когда владелец другой ссылки ничего подобного не ожидает. В реальной жизни проблемы такого рода могут возникнуть, в частности, потому, что экземпляры могут иметь переменные экземпляров, которые указывают на другие объекты, и эти указатели могут сохраняться до тех пор, пока существуют сами экземпляры. Предположим, у нас есть объект myObject и мы передаем ему ссылку на объект нашего стека.

```
Stack* myStack = // ... Создание экземпляра Stack
                 // и инициализация myStack ...
[myObject doSomethingWithThis: myStack]; // передача myStack
                                         // в myObject
```

После этого кода myObject имеет указатель на тот же экземпляр, на который мы уже указываем с помощью myStack. Поэтому мы должны быть очень осторожны и внимательны. Объект myObject теперь может изменять наш myStack прямо у нас под носом! Более того, этот объект myObject может *хранить* свою ссылку на экземпляр стека (например, в переменной экземпляра) и изменить его *позже* — возможно, намного позже, и способом, который нас удивит. Такая ситуация может быть создана преднамеренно, но и в этом случае следует быть внимательным и точно понимать, что вы делаете и зачем (рис. 3.1).

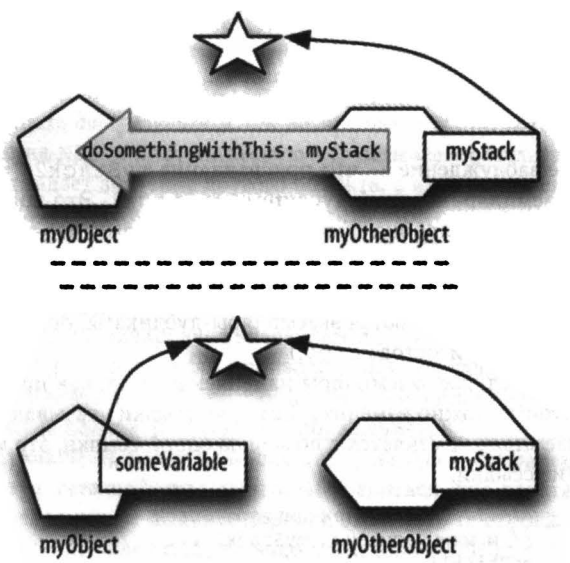


Рис. 3.1. Два экземпляра с указателями на один и тот же третий экземпляр

Ссылки на экземпляры и управление памятью

"Указательная" природа ссылок на экземпляры в Objective-C имеет определенные последствия для системы управления памятью. Правила области видимости и, в частности, времени жизни переменных в чистом C обычно довольно просты: если вы создаете переменную путем ее объявления в некоторой области видимости, то когда эта область перестает существовать, перестает существовать и переменная. Такого рода переменные называются автоматическими (K&R 1.10). Например:

```
void myFunction() {
    int i; // Выделено место для хранения int
    i = 7; // В этом месте размещено значение 7
} // Область видимости закончилась, выделенное
  // место и его содержимое уничтожаются
```

Но в случае указателя есть два вида памяти, о которых следует беспокоиться: сам указатель, который представляет собой целое число, представляющее адрес в памяти, и все, что находится по этому адресу. В языке программирования C при автоматическом уничтожении указателя ничто не ведет к автоматическому уничтожению того, на что он указывает:

```
void myFunction() {
    NSString* s = @"Hello, world!";      // Указатель и NSString
    NSString* s2 = [s uppercaseString];  // Указатель и NSString
}  // Два указателя прекращают существование...
   // ... но что происходит со строками, на которые они указывали?
```

Некоторые объектно-ориентированные языки программирования, в которых ссылка на экземпляр является указателем, автоматически управляют памятью, на которую указывают ссылки на экземпляры (примерами являются REALbasic и Ruby). Но Objective-C не является одним из таких языков. Поскольку язык C ничего не говорит об автоматическом уничтожении того, на что указывают ссылки на экземпляры, Objective-C реализует явный механизм управления памятью. Позже (в главе 12) я расскажу об этом механизме и о том, какие обязанности ложатся на программиста при его использовании. К счастью, при наличии механизма ARC эти обязанности меньшие, чем были ранее; тем не менее памятью по-прежнему необходимо управлять, и вы по-прежнему должны отчетливо понимать, как работает управление памятью экземпляров.

Методы и сообщения

Метод в Objective-C определен как часть класса. Он имеет три аспекта.

Он может быть методом класса или методом экземпляра

Если это метод класса, вы вызываете его с помощью сообщения самому классу. Если это метод экземпляра, вы вызываете его с помощью сообщения экземпляру класса.

Он имеет параметры и возвращаемое значение

Как и функции C (глава 1), методы Objective-C принимают некоторое количество параметров; каждый параметр имеет некоторый конкретный тип. И, как и функции C, методы могут иметь возвращаемое значение, также определенного типа. Если метод не возвращает ничего, его возвращаемый тип объявляется как `void`.

Он имеет имя

Имя метода Objective-C должно иметь столько двоеточий, сколько параметров он принимает; и, если метод принимает параметры, его имя должно завершаться двоеточием.

Вызов метода

Чтобы отправить сообщение объекту, используется выражение сообщения, которое также называют для простоты и по аналогии с функциями C вызовом метода. Как вы уже и сами догадались, синтаксис отправки сообщения объекту включает использование квадратных скобок. Первым в квадратных скобках указывается объект, которому отправляется сообщение; этот объект является получателем сообщения. Затем следует само сообщение:

```
NSString* s2 = [s uppercaseString]; // Отправка сообщения
             // "uppercaseString" объекту s и присваивание
             // результата переменной s2
```

Если сообщение представляет собой метод, который получает параметры, значение каждого соответствующего аргумента указывается после двоеточия:

```
[myStack1 push: @"Hello"]; // Отправка объекту myStack1
    // сообщения "push:" с одним аргументом типа
    // NSString и значением @"Hello"
```

Чтобы отправить сообщение классу (вызов метода класса), класс может быть представлен его именем:

```
NSString* s = [NSString string]; // Сообщение "string"
                                 // для класса NSString
```

Для отправки сообщения экземпляру (вызов метода экземпляра) требуется ссылка на этот экземпляр, которая (как вы знаете) представляет собой указатель:

```
NSString* s                  // s инициализируется как
    = @"Hello, world!";      // экземпляр класса NSString
NSString* s2                 // Отправка сообщения
    = [s uppercaseString];   // "uppercaseString" объекту s
```

Можно отправлять метод класса классу, а метод экземпляра экземпляру, независимо от того, как получен и представлен этот класс или экземпляр. Например, строка `@"Hello, world!"` сама по себе является экземпляром `NSString`, так что вполне можно написать

```
NSString* s2 = [@"Hello, world!" uppercaseString];
```

Если метод не принимает никаких параметров, то его имя не содержит двоеточия, как, например, метод экземпляра `uppercaseString` класса `NSString`. Если метод принимает один параметр, то его имя содержит одно двоеточие, которое представляет собой последний символ имени метода, наподобие метода экземпляра `push:` нашего гипотетического стека. Если метод принимает два и более параметра, его имя содержит соответствующее количество двоеточий и последнее двоеточие является последним символом имени метода. В минимальном случае имя метода заканчивается всеми его двоеточиями. Например, метод, принимающий три параметра, может называться `hereAreThreeStrings:::`. Для того чтобы его вызвать, мы должны разбить имя и поместить после каждого двоеточия аргумент, т.е. получить выражение наподобие следующего:

```
[someObject hereAreThreeStrings: @"string1" : @"string2" : @"string3"];
```

Это вполне корректный вызов метода, хотя и не очень распространенный, главным образом потому, что он не слишком информативный. Обычно применение имени более информативно, так что, как правило, перед каждым двоеточием идет часть имени, описывающая значение, следующее за двоеточием.

Например, вот метод класса `UIColor`, генерирующий экземпляр `UIColor` из четырех чисел `CGFloat`, представляющих красный, зеленый, синий компоненты и прозрачность, и он называется `colorWithRed:green:blue:alpha:`. Обратите внимание на ясность конструкции этого имени. Часть `colorWith` говорит о предназначении метода: он генерирует цвет на основе некоторой информации. Остальная часть имени, `Red:green:blue:alpha:`, описывает смысл каждого параметра. Вы можете вызвать этот метод таким способом:

```
UIColor* c = [UIColor colorWithRed: 0.0 green: 0.5
              blue: 0.25 alpha: 1.0];
```

В этой главе я уже ссылался несколько раз на метод класса `NSString` с именем `stringWithContentsOfFile:encoding:error:`. Это имя весьма информативно. Даже до изучения типов данных параметров мы можем предположить, что первый параметр представляет собой ссылку на файл, второй описывает кодировку, а третий используется для возврата информации об ошибке путем косвенного обращения (и что этот метод возвращает строку).

Правила именования методов Objective-C, вместе с соглашениями, регулирующими применение таких имен (наподобие сделать имя информирующим о предназначении метода и его параметров), приводят к довольно длинным и громоздким именам методов, например, таким: `getBytes:maxLength:usedLength:encoding:options:range:remainingRange:`. Такая многословность является характерной для Objective-C. Вызовы методов и даже объявления методов часто разделены на несколько строк, как для ясности, так и для предотвращения насильственного переноса редактором строки кода.

Объявление метода

Объявление метода представляет собой определяющую открытую инструкцию, указывающую имя метода, тип данных возвращаемого значения и типы данных каждого из его параметров. Например, документация Apple о классе состоит главным образом из списка объявлений его методов. Таким образом, важно уметь читать объявления методов.

Объявление метода состоит из трех частей.

- Знак + или –, означающий, что метод представляет собой метод класса или мл экземпляра соответственно.
- Тип данных возвращаемого значения в скобках.
- Имя метода, разбитое после каждого двоеточия. За каждым двоеточием следует соответствующий параметр, выраженный как тип данных параметра в скобках, за которым следует имя заполнителя для параметра.

Так, например, документация Apple гласит, что объявление метода `colorWithRed:green:blue:alpha:` класса `UIColor` имеет вид

```
+ (UIColor*) colorWithRed: (CGFloat) red  green: (CGFloat) green
                     blue: (CGFloat) blue alpha: (CGFloat) alpha
```

(Обратите внимание, что я разделил объявление на две строки, для удобочитаемости и чтобы поместить его на листе книги. В документации это объявление расположено на одной строке.)

Полезное сокращение

Не редкость, когда за пределами кода, ссылаясь на метод, перед его именем указывают знак + или –, чтобы было ясно, является ли этот метод методом класса или методом экземпляра. Полуофициальное сокращение представляет собой знак + или –, после чего идут квадратные скобки, содержащие имя класса и имя метода. Например, в примечаниях к выпуску iOS 7 сказано, что в iOS 7 добавлен следующий метод, отсутствовавший в iOS 6:

```
-[NSScanner scanUnsignedLongLong:]
```

Это не вызов метода и не объявление. Это вообще не код Objective-C. Это просто метод сокращенного описания метода, указывающий, что это метод экземпляра класса `NSScanner` с именем `scanUnsignedLongLong:`.

Убедитесь, что вы можете прочитать и понять это объявление! Вы должны уметь посмотреть на него и мгновенно сказать себе: "имя метода — `colorWithRed:green:blue:alpha:`. Это метод класса, который принимает четыре параметра типа `CGFloat` и возвращает `UIColor`".

Пробел после каждого двоеточия в объявлении или вызове метода является необязательным. Пробел перед двоеточием также является корректным, хотя на практике редко применяется. Пробел до или после любой из круглых скобок в объявлении метода также является необязательным.

Вложенные вызовы методов

Там, где в вызове метода должен находиться объект определенного типа, можно поместить еще один вызов метода, который возвращает этот тип. Таким образом, вызовы метода могут быть вложенными. Вызов метода может быть приемником в вызове метода:

```
// Глупо, но допустимо:
NSString* s = [[NSString string] uppercaseString];
```

Приведенный код корректен, поскольку результатом метода `string` класса `NSString` является экземпляр `NSString` (формально, значение `NSString*`), так что мы можем отправить метод экземпляра `NSString` этому результату. Аналогично вызов метода может находиться в другом вызове в качестве аргумента:

```
[myStack push: [NSString string]]; // Все в порядке, если метод
                                  // push: ожидает параметр типа NSString*
```

Тем не менее я должен предостеречь вас от злоупотребления этой возможностью. Код с большим количеством вложенных квадратных скобок очень трудно читать (и писать). Кроме того, если один из вложенных вызовов возвращает непредвиденное значение, нет способа легко обнаружить этот факт. Зачастую куда лучше писать более подробный код и объявлять временные переменные для хранения результатов каждого вызова метода. Вот пример из моего собственного кода. Вместо чтобы написать

```
NSArray* arr = [[MPMediaQuery albumsQuery] collections];
```

я воспользовался временной переменной:

```
MPMediaQuery* query = [MPMediaQuery albumsQuery];
NSArray* arr = [query collections];
```

Несмотря на то что первая версия короче и достаточно ясна и что во втором варианте переменная `query` никогда не будет использоваться еще раз (она существует исключительно для того, чтобы быть приемником сообщения `collection` во второй строке), ее все же стоило создать. Как минимум, это упрощает пошаговую отладку кода в отладчике, когда я хочу сделать паузу после вызова `albumsQuery` и посмотреть, правильный ли результат возвращается этим вызовом (см. главу 9).

Неверное количество или несоответствие вложенных квадратных скобок может привести к появлению несколько запутанных сообщений от компилятора. Например, слишком большое количество пар квадратных скобок (`[[query collections]]`) или несбалансированное количество левых скобок (`[[query collections]`) влекут за собой сообщение "Expected identifier" ("Ожидается идентификатор").

Отсутствие перегрузки

Тип данных, возвращаемых методом вместе с типами данных каждого из его параметров в порядке передачи этих параметров методу, составляют *сигнатуру* этого метода. Существование двух методов одного и того же типа (метода класса или метода экземпляра) в одном и том же классе с одним и тем же именем недопустимо, даже если они имеют разные сигнатуры.

Так, например, вы не может иметь два метода экземпляра класса `MyClass` с одним именем `myMethod`, один из которых возвращает `void`, а второй — `NSString`. Аналогично нельзя иметь два метода экземпляра класса `MyClass` с именем `myMethod:`, оба возвращающие `void`, но один из которых принимает параметр `CGFloat`, а второй — `NSString`. Попытки нарушить это правило будут заблокированы компилятором, который объявит об ошибке "duplicate declaration" ("повторное объявление"). Причиной появления этого правила является то, что, если бы два таких метода могли существовать, не имелось бы способа определить при получении сообщения, какой именно из этих методов должен быть вызван.

Вы можете подумать, что этот вопрос можно решить на основе типов параметров, участвующих в вызове. Если один `myMethod:` принимает параметр `CGFloat`, а другой — `NSString`, то, когда вызывается `myMethod:`, Objective-C мог бы взглянуть на фактический аргумент и понять, какой из методов должен быть вызван. Но Objective-C таким образом не работает. Да, есть языки программирования, которые обеспечивают такую возможность, которая называется перегрузкой, но Objective-C не является одним из них.

Списки параметров

В языке Objective-C не такая уж редкость методы, требующие неизвестное количество параметров. Хорошим примером является метод `arrayWithObjects:` класса `NSArray`, который выглядит, как будто он принимает один параметр, но на самом деле он принимает любое количество параметров, разделенных запятыми. Параметры являются объектами, которые представляют собой элементы `NSArray`. Хитрость здесь заключается в том, что список должен заканчиваться элементом `nil`. Это значение не является ни одним из объектов, помещаемых в `NSArray` (`nil` не является объектом, поэтому `NSArray` не может его содержать), и предназначено только для того, чтобы указать, где заканчивается список.

Таким образом, вполне корректен следующий вызов этого метода:

```
NSArray* pep = [NSArray arrayWithObjects:@"Manny",
                                @"Moe", @"Jack", nil];
```

В объявлении метода `arrayWithObjects:` используется многоточие, указывающее на то, что здесь может находиться список параметров, разделенных запятыми:

```
+ (id)arrayWithObjects:(id)firstObj, ... ;
```

(Позже в этой главе я объясню смысл встречающегося в объявлении `id`.) Без ограничителя `nil` программа будет не в состоянии определить, где завершается список, и при выполнении программы могут случиться разные неприятности — ведь она заберется в поисках новых параметров туда, где их нет, и запросто заполнит `NSArray` всяческим мусором.

Распространенная ошибка начинающего программиста — забыть об ограничителе `nil`, но современный компилятор Objective-C достаточно умен, чтобы выдать предупреждение об отсутствии ограничителя — "missing sentinel in method dispatch". Несмотря на то что это всего лишь предупреждение, не запускайте такой код на выполнение! Еще один способ не забыть об ограничителе `nil` в этом конкретном примере — избежать вызова `arrayWithObjects:` вообще, что возможно начиная с версии компилятора LLVM 4.0 (Xcode 4.4 или более поздний), где вы можете образовать литеральный объект `NSArray` непосредственно, с помощью синтаксиса `@[...]`:

```
NSArray* pep = @[@"Manny", @"Moe", @"Jack"];
```

Это просто упрощенная запись, которая "за кулисами" разворачивается в вызов `arrayWithObjects:`, но этот вызов корректно оформлен, с ограничителем `nil`, так что шансов не допустить ошибку у программиста существенно больше, чем при явном вызове `arrayWithObjects:` (да и количество вводимых символов при этом куда меньше).

Тем не менее вы будете встречаться со многими другими методами Objective-C, которые принимают параметр, являющийся списком переменной длины с завершающим `nil`. Например, у протокола `UIAppearance` имеется метод класса `appearanceWhenContainedIn:` или метод `initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:` в `UIAlertView`. Жаль, что Apple не изменит каким-то образом Objective-C (или эти методы), чтобы избежать использования ограничителя `nil`; но до тех пор, пока это не сделано, следует знать, как правильно вызываются такие методы.

Язык C явным образом обеспечивает функционирование списков аргументов неопределенной длины, чем методы Objective-C наподобие `arrayWithObjects:` пользуются "за сценой". Я не собираюсь пояснять работу механизма языка программирования C, поскольку не думаю, что вам придется писать метод или функцию, для которых он потребуется. Но если вам нужны соответствующие подробности — обратитесь к K&R 7.3.

Когда отправка сообщений не работает

Механизм отправки сообщений имеет фундаментальное значение для Objective-C. По этой же причине это основной источник проблем. Слишком легко может случиться, что, когда сообщение отправляется объекту, все пойдет не так, как следует. Но как что-то может пойти не так в таком простом и понятном механизме? Что ж, рассмотрим следующее, казалось бы, тривиальное, выражение:

```
[s uppercaseString]
```

В этом выражении s представляет собой ссылку, а uppercaseString — сообщение. Сообщение uppercaseString отправляется ссылке s. Что может пойти не так?

Ссылка может указывать на неверный объект

Ссылка на объект является указателем. Предполагается, что она указывает на реальный объект. Ссылка — это всего лишь имя. Как из выражения узнать, на что именно указывает s? Вы не видели, каким значением эта ссылка инициализирована. Она может быть равна nil. Это может не быть строка. Это может быть строка, но не та, которую вы ожидаете увидеть. Ссылка может быть ссылкой вовсе не на то, на что вы думаете.

Сообщение может быть неверным

Эта возможность идет рука об руку с предыдущей. Если ссылка может быть неверной, то объект, на который она указывает, может не принимать отправленное сообщение. Или объект может принять сообщение, но, будучи не тем объектом, что надо, он может дать неверный результат, отличающийся от того, который вы ожидаете.

Это не просто придирки; это распространенная повсеместная реальность. Подавляющее количество вопросов, с которыми я сталкиваюсь в Интернете, о том, почему некоторый код неверно работает, и подавляющее количество ответов на эти вопросы — из-за указанных проблем. Программирование обманчиво; вы можете легко обмануться в своих предположениях. Вы думаете, что знаете, что собой представляют некоторые ссылки, но на самом деле это вовсе не так... В результате выясняется, что все ведет себя не так, как надо, или программа попросту аварийно завершает работу. Понимание, как работает отправка сообщений и что при этом может пойти не так — а я заверяю вас, что это часто так и будет, — это залог успешного написания программы.

В оставшейся части этого раздела мы сосредоточимся на двух особенно коварных причинах, по которым отправка сообщения может пойти не так, как надо: сообщение направляется по ссылке nil и отправляется объекту, которому оно не нравится.

Сообщения для nil

Ссылке на экземпляр очень легко оказаться указывающей на неэкземпляр — то есть на nil. Я уже говорил, что простое объявление экземпляра без инициализации устанавливает ссылку равной nil (при наличии механизма ARC) и что многие методы каркаса Cocoa преднамеренно возвращают nil как способ указать, что что-то пошло не так. Кроме того, экземпляр переменной, объявленной как ссылка на объект, начинает жизнь как nil, и если эта ссылка не будет установлена впоследствии указывающей на реальный объект, как вы ожидаете, то она может так и остаться nil. (В главе 7", мы рассмотрим все наиболее распространенные способы, которыми это может случиться.)

Рассмотрим последствия отправки сообщения неэкземпляру, т.е. по указателю nil. Вы можете отправить сообщение экземпляру NSString следующим образом:

```
NSString * s2 = [s uppercaseString];
```

Этот код посылает сообщение uppercaseString объекту s. Предположительно s представляет собой экземпляр NSString. Но что если s равен nil? В некоторых объектно-ориентированных языках программирования отправка сообщения ссылке nil вызывает ошибку времени выполнения и приводит к преждевременному завершению работы программы (например, в REALbasic и Ruby). Но Objective-C работает иначе. В Objective-C отправка сообщения nil допустима и не прерывает выполнение программы. Более того, если вы сохраните

результат вызова метода, он будет разновидностью нуля — что означает, что если вы присвоите этот результат ссылке на экземпляр, то он получит значение nil:

```
NSString* s = nil;                  // Сейчас s равно nil
NSString* s2 = [s uppercaseString]; // Теперь s2 тоже равно nil
```

Вопрос о том, является ли это поведение Objective-C хорошим решением или нет, вызывает целые религиозные войны и является предметом бурных споров среди программистов. С одной стороны, это полезно, с другой — при этом очень легко обмануться. Обычный сценарий — когда вы случайно отправляете сообщение ссылке nil, не осознавая этого, и позже ваша программа работает не так, как ожидалось. А поскольку точка, в которой обнаруживается некорректное поведение, находится позже (может быть, даже существенно позже) момента первого появления нулевого указателя, это появление может быть очень трудно отследить (на самом деле программисты часто не считают нулевой указатель первоочередной причиной своих неприятностей).

Имеется не так много вариантов ваших действий — по сути, вы можете только наполнить ваш код сплошными проверками на равенство nil используемых ссылок на экземпляры. Описанное поведение в случае сообщения nil встроено в язык, и его никто не собирается менять. Поэтому вы просто обязаны знать его и постоянно о нем помнить! Считать, что у вас есть ссылка на реальный экземпляр, в то время как на самом деле у вас ссылка "в никуда" — это очень, очень распространенная ошибка начинающих, отягощенная тем, что при использовании ссылки nil никаких жалоб от среды выполнения не поступает, а ссылки nil в результате размножаются едва ли не в геометрической прогрессии. При этом ничего страшного не происходит, программа как-то работает, и некому разуверить вас в этом заблуждении (за исключением того, что программа ведет себя совершенно не так, как ожидалось). Если все тихо и таинственно идет не так, в первую очередь следует заподозрить наличие нулевой ссылки и использовать для расследования методы отладки (глава 9).

Словом, если вы заранее знаете, что вызов некоторого метода может вернуть nil, не думайте, что все пойдет хорошо и что он не вернет этот nil. Наоборот, если что-то может пойти не так — оно пойдет не так с большой вероятностью! Например, опустить проверку на равенство nil после вызова stringWithContentsOfFile:encoding:error: — это просто тупость. Меня не интересует, что вы точно знаете, что нужный файл имеется и что он именно в той кодировке, которую вы указали, — вы просто **обязаны** проверить результат на равенство nil!

Нераспознанные селекторы

Возможна ситуация с отправкой сообщения объекту, у которого нет соответствующего метода. Если такое случится, последствия могут быть фатальными: ваше приложение может просто аварийно завершиться. Единственный защитник от таких непредвиденных обстоятельств — компилятор; но это не слишком сильный защитник. В некоторых случаях компилятор предупредит вас, что, возможно, вы делаете что-то нежелательное; но в других случаях компилятор с радостью позволит вам наступить на эти грабли.

Вот ситуация, в которой компилятор действительно спасет вас от самого себя:

```
NSString* s = @"Hello, world!";
[s rockTheCasbah];
```

Класс NSString не имеет метода rockTheCasbah. До появления механизма ARC компилятор разрешил бы компиляцию этого кода, но предупредил бы о том, что вы сами ищете себе проблемы. Однако при наличии механизма ARC компилятор с негодованием отвергает вашу попытку с сообщением о фатальной ошибке "No visible @interface for 'NSString'

declares the selector 'rockTheCasbah'". (Компилятор с механизмом ARC гораздо строже, так как механизм ARC должен работать с управлением памятью, а при разрешении отправлять бессмысленные сообщения это не удается делать эффективно.)

Немного замутив воду, мы можем проскользнуть мимо строгого механизма ARC. Предположим, что у нас есть класс MyClass, в котором объявлен метод rockTheCasbah. Тогда мы можем записать

```
MyClass* m = @"Hello, world!";
[m rockTheCasbah];
```

Мы говорим, что m — экземпляр класса MyClass, но на деле присваиваем этой ссылке экземпляр NSString. Это странное, но не строго запрещенное действие. Компилятор предупредит нас о неполной корректности этого присваивания ("incompatible pointer types"), но программу скомпилирует. Остается ехидно засмеяться и запустить программу на выполнение. И что получится?

Когда программа заработает и класс NSString получит сообщение rockTheCasbah, программа аварийно завершится с сообщением (в журнале консоли) о случившейся неприятности в виде

```
-[__NSCFConstantString rockTheCasbah]: unrecognized
    selector sent to instance 0x8650.
```

Это сообщение детально описывает происшедшее.

- Фраза о *нераспознанном селекторе* (unrecognized selector) является самой сутью происшедшего. Термин "селектор" грубо эквивалентен термину "сообщение", так что это способ сказать, что определенному объекту послано сообщение, с которым он не в состоянии справиться.
- -[__NSCFConstantString rockTheCasbah] описывает сообщение и его получателя, используя сокращение, о котором я говорил ранее: экземпляру NSCFConstantString послано сообщение экземпляра rockTheCasbah. (Описание NSString как NSCFConstantString представляет собой внутренние подробности языка.)
- 0x8650 — значение указателя на экземпляр; это адрес в памяти, на который в действительности указывает указатель m.

Ситуация нераспознанного селектора приводит к генерации *исключения*, внутреннего сообщения о том, что при работе программы произошло что-то плохое. Код на Objective-C может "поймать" исключение, и тогда аварийное завершение не произойдет. Технически причина завершения работы программы не в том, что объекту отправлено сообщение, с которым тот не смог справиться, а в том, что не было перехвачено сгенерированное при этом исключение. Вот почему журнал сбоев может также гласить "Terminating app due to uncaught exception" (завершение приложения из-за неперехваченного исключения).

В этом примере мы сознательно создали неприятности сами себе, чтобы продемонстрировать, что происходит, когда возникает ситуация нераспознанного селектора. Пример, конечно, достаточно надуманный, но последствия таковыми не являются. Я заверяю, что такая вещь рано или поздно *случится* и с вами, хотя, конечно, не преднамеренно. Это произойдет потому, что, как я сказал в начале этого раздела, ваша ссылка окажется не совсем тем, что вы о ней думаете. Существует множество путей, которыми может возникнуть такая ситуация;

например, как я покажу в следующем разделе, вы можете случайно соврать компилятору о классе объекта, или компилятор может не иметь никакой информации о классе объекта, так что он даже не сможет предупреждать вас о сообщении, которое объект не в состоянии обработать. Следите за фразой "unrecognized selector", когда ваша программа аварийно завершает работу, — теперь вы знаете, что это значит и как интерпретировать такое сообщение! Детали сообщения помогут вам понять, что именно не так и как исправить ситуацию.

Приведение типа и тип `id`

Иногда компилятор, пытаясь спасти вас от ситуации нераспознанного селектора, выдает предупреждение или сообщение об ошибке, в то время как вы очень хорошо знаете, что то, что вы пытаетесь сделать, — безопасно и правильно. Проблема в данном случае заключается в том, что вы знаете больше компилятора о том, что в действительности происходит. Чтобы продолжить работу, вам нужно поделиться своими знаниями с компилятором. Как правило, это делается с помощью *приведения типа* ссылки на объект (см. главу 1). Такое приведение служит в качестве объявления класса, которому объект будет принадлежать при выполнении программы. Компилятор верит в то, что вы говорите ему в выражении приведения; таким образом, вы можете развеять сомнения компилятора в корректности выполняемых вами действий.

Эта ситуация часто возникает в связи с наследованием классов. Пока что мы не будем обсуждать наследование (см. главу 4), но пример я все равно приведу.

Возьмем встроенный класс Cocoa `UINavigationController`. Его метод `topViewController` объявлен как возвращающий экземпляр `UIViewController`. В реальной же жизни он скорее вернет экземпляр некоторого созданного вами подкласса `UIViewController`. Для того чтобы вызов метода созданного вами класса для экземпляра, возвращенного методом `topViewController`, не свел с ума компилятор, последнему следует пояснить, что этот экземпляр на самом деле будет экземпляром созданного вами класса.

Вот пример из одного из моих собственных приложений:

```
[[navigationController topViewController] setAlbums: arr];
```

Эта строка кода не компилируется; компилятор выдает ту же ошибку "no visible `@interface`", о которой я говорил в предыдущем разделе. Встроенный метод `topViewController` возвращает `UIViewController`, а `UIViewController` не имеет метода `setAlbums`.

Однако я знаю, что в данном случае контроллер верхней панели контроллера навигации является экземпляром моего собственного класса `RootViewController`. А мой класс `RootViewController` *имеет* метод `setAlbums`; и именно его я и пытаюсь вызвать. То, что я делаю — разумно, законно и желательно. Мне надо действовать именно таким образом! Чтобы компилятор не мешал мне работать, я должен убедить его, что знаю, что делаю, сообщив, что объект, возвращаемый вызовом метода `topViewController`, на самом деле представляет собой `RootViewController`. Я делаю это с помощью приведения типов:

```
[(RootViewController*)[navigationController topViewController]
        setAlbums: arr];
```

Этого достаточно. Не будет никаких предупреждений и сообщений об ошибках. Приведение типов заставляет компилятор замолчать, когда я собираюсь отправить этому экземпляру сообщение `setAlbums:`, поскольку мой класс `RootViewController` имеет метод экземпляра `setAlbums:`, и компилятор знает об этом. А программа при запуске не завершается аварийно, поскольку я не лгу компилятору: вызов метода `topViewController` действительно возвращает экземпляр `RootViewController`.

Но чем больше власть — тем больше и ответственность. Не лгите компилятору! Вспомните пример из предыдущего раздела:

```
MyClass* m = @"Hello, world!";
[m rockTheCasbah];
```

Первая строка заставляет компилятор предупредить нас о "incompatible pointer types" (несовместимых типах указателей); и в данном случае компилятор совершенно прав: ведь в дальнейшем мы получим аварийное завершение ("unrecognized selector") при попытке отправить сообщение `rockTheCasbah` объекту `NSString`. Мы можем заставить компилятор промолчать с помощью приведения типов:

```
MyClass* m = (MyClass*)@"Hello, world!";
[m rockTheCasbah];
```

Да, предупреждения не будет. Но проблема останется: ведь мы солгали компилятору и будем отправлять сообщение `rockTheCasbah`, как и ранее, экземпляру `NSString`. Мораль проста: поскольку компилятор верит всему, что вы говорите ему с помощью приведения типов, врать ему категорически нельзя.

Приведение типов не меняет чудесным образом реальные типы объектов. Это просто метод подсказки компилятору об информации о типе. На сам приводимый объект это никак не влияет. Выражение приведения `(MyClass*)@"Hello, world!"` не превращает экземпляр `NSString`, который представляет собой `@"Hello, world!"`, в экземпляр `MyClass`! На удивление распространенная ошибка среди новичков — считать, что при этом действительно выполняются какие-то преобразования.

Язык Objective-C предоставляет также специальный тип, предназначенный для того, чтобы все заботы компилятора о типах данных выполнялись в тишине. Это тип `id`. Он представляет собой указатель, так что вы не должны использовать выражение `id*`. Этот тип определен как "указатель на просто объект", без каких бы то ни было дальнейших уточнений. Таким образом, каждая ссылка на экземпляр также представляет собой `id`.

Использование типа `id` заставляет компилятор перестать беспокоиться о взаимосвязи между типами объектов и сообщениями. Компилятор не может ничего знать о том, какого типа в действительности будет тот или иной объект, поэтому он просто поднимает (или опускает — как кому нравится) руки и не предупреждает больше ни о чем. Кроме того, объекту типа `id` может быть присвоено любое объектное значение, и к типу `id` может быть приведен любой другой тип. Значение типа `id` может быть использовано в любом присваивании там, где ожидается значение некоторого конкретного объектного типа. Понятие назначения включает в себя передачу параметров; таким образом, значение типа `id` можно передать в качестве аргумента везде, где ожидается параметр некоторого конкретного объектного типа. (Мне нравится думать об `id` как об аналоге групп крови AB и O: это универсальный реципиент и универсальный донор.) Вот пример применения типа `id`:

```
NSString* s = @"Hello, world!";
id unknown = s;
[unknown rockTheCasbah];
```

Вторая строка корректна, поскольку любое объектное значение может быть присвоено переменной типа `id`. Третья строка не генерирует предупреждения компилятора, поскольку типу `id` может быть послано любое сообщение. (Очевидно, что программа при запуске все равно завершится аварийно, когда выяснится, что `unknown` имеет тип `NSString` — который, конечно же, не может получать сообщения `rockTheCasbah`!)

На самом деле это чрезмерное упрощение. При использовании механизма ARC этот код не может быть скомпилирован. Вместо скомпилированного кода вы получите сообщение об ошибке: "No known instance method for selector 'rockTheCasbah'". Оно означает, что компилятор ничего не знает о методе `rockTheCasbah` ни в каком классе. Однако если `rockTheCasbah` объявляется в заголовочном файле любого класса, импортированном в данный, или если `rockTheCasbah` реализован в текущем классе, то код будет компилироваться без предупреждения.

Если способность типа `id` получать любые сообщения напоминает вам `nil`, то так и должно быть. Я уже говорил, что `nil` представляет собой разновидность нуля; теперь я могу сказать, какой разновидностью нуля является `nil`. Это нуль, приведенный к типу `id`. Конечно, во время выполнения имеется разница, равен ли `id` значению `nil` или некоторому иному; отправка сообщения `nil` не вызовет аварийного завершения программы, но отправка неизвестного сообщения реальному объекту, вероятно, приведет к таковому.

Таким образом, результат применения `id` выражается в полном отключении проверки типов компилятором. Выяснения, каким является тип объекта, откладываются до тех пор, пока программа не будет запущена на выполнение. Но я не рекомендую широко применять `id` таким образом. Компилятор — ваш друг; вы должны позволить ему разведывать и перехватывать возможные ошибки в коде. Поэтому я почти никогда не объявляю переменную или параметр как имеющие тип `id`. Я хочу, чтобы типы моих объектов были максимально конкретными, чтобы компилятор мог максимально проверить мой код. С другой стороны, тип `id` часто используется в интерфейсе API каркаса Cocoa — и это именно то, что, как я предупреждал вас в предыдущем разделе, может приводить к аварийному завершению из-за нераспознанного селектора.

Рассмотрим, например, класс `NSArray`, который представляет собой объектно-ориентированную версию массива. В чистом C вы должны объявить, какого типа объекты находятся в массиве; например, у вас может быть "массив целых чисел `int`". В Objective-C, используя `NSArray`, вы не можете этого сделать. Каждый `NSArray` представляет собой массив элементов типа `id`, что означает, что каждый элемент массива может быть любого объектного типа. Вы можете поместить в массив `NSArray` объект некоторого типа, поскольку любой тип объекта может быть присвоен ссылке на `id` (`id` является универсальным реципиентом). Вы можете получить обратно из `NSArray` любой конкретный тип объекта, так как `id` может быть присвоен объекту любого типа (`id` является универсальным донором).

Метод `lastObject` класса `NSArray`, таким образом, определен как возвращающий тип `id`. Для данного массива `arr` типа `NSArray` последний элемент можно получить следующим образом:

```
id unknown = [arr lastObject];
```

Теперь мы находимся в потенциально опасной ситуации. После этого кода объекту `unknown` можно послать любое сообщение; компилятор не будет этому препятствовать. Следовательно, если мне известен тип элементов массива, я всегда могу выполнить присваивание или приведение типа к типу, который я получаю из массива. Пусть, например, я знаю, что массив `arr` не содержит ничего, кроме экземпляров `NSString` (поскольку я перед этим поместил их туда). Тогда я могу написать

```
NSString* s = [arr lastObject];
```

Компилятор не будет жаловаться на этот код, поскольку тип `id` может быть присвоен любому конкретному типу объекта (`id` является универсальным донором). Кроме того, далее компилятор рассматривает `s` как `NSString` и использует свои возможности проверки типов для того, чтобы убедиться, что я не посылаю `s` никаких сообщений, кроме тех, что можно

посылать NSString. И я не лгу компилятору; во время выполнения s действительно представляет собой объект NSString, так что все в порядке.

Однако есть еще одна опасность: эта ситуация — открытое приглашение случайно солгать компилятору. Предположим, что последний элемент в этом массиве NSArray не является NSString. Компилятор этого не знает; он вообще ничего не знает о том, что NSArray на самом деле будет содержать во время выполнения, а lastObject возвращает id, который компилятор с радостью позволит присвоить ссылке, объявленной как NSString. И тут мы просто напрашиваемся на неприятности. Предположим, например, что в следующей строке я отправлю сообщение uppercaseString объекту s. Компилятор молчит: в конце концов, я объявил эту ссылку как NSString, а uppercaseString является методом NSString. Но если s *не* является NSString, то, вероятно, мы попадем в ситуацию нераспознанного селектора и получим аварийное завершение программы.

Еще одной связанной с id ловушкой являются конфликты имен методов. Ранее я говорил, что один и тот же класс не может определять методы одного и того же типа (методы класса или методы экземпляра) с одинаковыми именами, но различными сигнатурами. Но я не говорил, что происходит, когда два *разных* класса объявляют методы с одинаковыми именами, но различными сигнатурами. Если в коде указан тип объекта-получателя сообщения, никаких проблем нет, потому что нет никаких сомнений в том, какой метод вызывается: он один в классе этого объекта. Но если объектом-получателем сообщения является id, то с использованием механизма ARC вы получите сообщение об ошибке: "Multiple methods named 'rockTheCasbah' found with mismatched result, parameter type or attributes" (найдено несколько методов с именем rockTheCasbah с несоответствующими типами результата, параметров или атрибутами). Это еще одна причина, по которой имена методов так многословны: для того, чтобы сделать каждое имя метода уникальным, предотвращая объявление в разных классах конфликтующих сигнатур для одного и того же имени метода.

Сообщения как тип данных

Предыдущие разделы вращалась вокруг того факта, что Objective-C до времени выполнения не знает, какому объекту посылается сообщение. Но справедливо большее: Objective-C до времени выполнения не должен знать, *какое сообщение* отправляется объекту. Некоторые важные методы в действительности принимают в качестве параметров и сообщение, и его получателя; они не собраны и использованы для формирования выражения фактическое сообщение до времени выполнения.

Рассмотрим, например, такое объявление метода из класса NSNotificationCenter Cocoa:

```
- (void)addObserver:(id)notificationObserver
           selector:(SEL)notificationSelector
               name:(NSString *)notificationName
             object:(id)notificationSender
```

Позже (в главе 11) я поясню, что делает этот метод. Сейчас для понимания важно то, что он составляет команду для отправки некоторого сообщения определенному объекту позже, в подходящее время. Например, наша цель при вызове этого метода может заключаться в том, чтобы иметь сообщение tickleMeElmo: для отправки позже объекту myObject. Для этого нам надо передать соответствующие данные в качестве двух первых аргументов.

Давайте рассмотрим, какие аргументы следует передать. Первый параметр (observer:) представляет собой объект, которому будет отправлено сообщение и который имеет тип id, что позволяет указать в качестве получателя объект любого типа. Так что в нашем случае в

качестве аргумента `observer:` мы передаем myObject. Само сообщение представляет собой параметр `selector:`, который имеет специальный тип данных `SEL`. Теперь вопрос заключается в том, как выразить имя сообщения `tickleMeElmo:`.

Вы не можете просто набрать `tickleMeElmo:` как обычный текст: это не сработает синтаксически. Вы можете подумать, что сообщение можно выразить как строку `NSString`, т.е. как `@"tickleMeElmo:"`, но это тоже не сработает. Оказывается, корректно следует поступить следующим образом:

```
@selector(tickleMeElmo:)
```

Текст `@selector()` представляет собой директиву компилятору, которая говорит о том, что то, что находится в круглых скобках, представляет собой имя сообщения. Обратите внимание, что находящееся в скобках не является `NSString`; это просто имя сообщения. А поскольку это имя, оно не должно включать пробелы и должно включать в качестве своей части двоеточия.

Так что правило исключительно простое: там, где ожидается `SEL`, вы обычно передаете выражение `@selector`. Ошибка в этом синтаксисе достаточно распространена среди начинающих программистов.

К сожалению, этот синтаксис — также прямое приглашение наделать ошибок при вводе, возможно, с очень неприятными результатами. Если `myObject` реализует метод `tickleMeElmo:`, а я случайно наберу текст, скажем, `@selector(tickleMeElmo)`, забыв о двоеточии, то даже если сообщение `tickleMeElmo` без двоеточия будет отправлено объекту `myObject`, приложение, вероятно, все равно завершится аварийно из-за исключения нераспознанного селектора.

В Xcode 5, впервые в истории Objective-C, введено предупреждение компилятора в случае, когда селектор не соответствует известному методу ("Undeclared selector 'tickleMeElmo'"). Таким образом, вы можете быть предупреждены о возможности будущего аварийного завершения — или нет. Компилятор не может заглянуть в класс аргумента `observer:` и выяснить, реализован ли в нем данный метод. Вместо этого компилятор предупреждает только в ситуации, если класса с данным методом вообще нет. Справедливо и обратное: если компилятор знает о наличии метода `tickleMeElmo` в каком-то классе, то никакого предупреждения вы не получите, даже если объект, которому в действительности во время работы программы будет послано сообщение `tickleMeElmo`, такого метода не имеет.

Функции C

Хотя ваш код, несомненно, будет вызывать массу методов Objective-C, вероятно, он также будет вызывать и некоторые функции C. Например, я уже упоминал в главе 1, что обычный способ описания `CGPoint`, основанный на его значениях `x` и `y`, заключается в вызове функции `CGPointMake`, которая объявлена следующим образом:

```
CGPoint CGPointMake (
    CGFloat x,
    CGFloat y
);
```

Убедитесь, что вы с первого взгляда видите, что это функция C, а не метод Objective-C и что вы понимаете разницу в синтаксисе вызова. Для вызова метода Objective-C вы отправляете в квадратных скобках сообщение объекту, с аргументами после двоеточий в имени метода; для вызова функции C используется имя функции, за которым следуют круглые скобки, содержащие аргументы.

У вас даже могут иметься причины для написания вместо методов собственных функций C как части класса. Функции C имеют более низкие накладные расходы, чем полноценные методы; так что, даже несмотря на отсутствие объектно-ориентированных возможностей методов, иногда полезно писать такие функции, в частности, когда некоторые вспомогательные вычисления должны выполняться быстро и часто.

Кроме того, вы можете встретиться с методами или функциями Cocoa, которые требуют передачи им функции C как "функции обратного вызова". Примером является метод `sortedArrayUsingFunction:context:` класса `NSArray`. Его первый параметр имеет тип

```
NSInteger (*) (id, id, void *)
```

Это выражение с помощью довольно сложного синтаксиса C описывает указатель на функцию, которая принимает три параметра и возвращает `NSInteger`. Три параметра функции имеют типы `id`, `id` и указатель на `void` (что означает любой указатель C). В языке программирования C как указатель функции может использоваться ее имя (см. главу 1). Таким образом, чтобы вызвать `sortedArrayUsingFunction:context:`, вам нужно написать функцию C, которая соответствует приведенному описанию, и использовать ее имя в качестве первого аргумента метода.

Для иллюстрации я напишу функцию обратного вызова, которая поможет мне отсортировать массив `NSArray` строк `NSString` по последнему символу каждой строки. (Да, это странное решение, но это ведь всего лишь иллюстрация!) Значение `NSInteger`, возвращаемое функцией, имеет особое значение: оно указывает, меньше ли первый параметр второго, равен ему или больше. Это значение я буду получать с помощью метода `compare:` класса `NSString`, который возвращает `NSInteger` с тем же смыслом. В примере 3.1 определена функция и показано, как должен вызываться метод `sortedArrayUsingFunction:context:` с нашей функцией как функцией обратного вызова.

Пример 3.1. Конструкции управления потоком языка программирования C

```
NSInteger sortByLastCharacter(id string1, id string2,
                              void* context) {
    NSString* s1 = string1;
    NSString* s2 = string2;
    NSString* string1end =
            [s1 substringFromIndex:[s1 length] - 1];
    NSString* string2end =
            [s2 substringFromIndex:[s2 length] - 1];
    return [string1end compare:string2end];
}
// А вот как используется эта функция (предполагается,
// что arr представляет собой массив NSArray строк NSString)
NSArray* arr2 = [arr
        sortedArrayUsingFunction:sortByLastCharacter
        context:nil];
```

CFTypeRef

Многие объектные типы Objective-C имеют низкоуровневые аналоги C, вместе с функциями C для работы с ними.

Например, помимо `NSString`, в Objective-C имеется нечто, именуемое `CFString`; "CF" означает "Core Foundation" и представляет собой низкоуровневый интерфейс API на основе C. `CFString` является "непрозрачной" структурой C; "непрозрачная" означает, что элементы, составляющие эту структуру, держатся в секрете и что вы должны работать с `CFString`

только с помощью соответствующих функций C. Как и в случае NSString или любого другого объекта, в коде вы обычно обращаетесь к CFString через указатель C; указатель на CFString имеет имя типа CFStringRef.

При случае можно решить работать с таким типом даже тогда, когда существует соответствующий объектный тип. Например, может выясниться, что NSString, при всей его мощи, не обеспечивает некоторую необходимую часть функциональности, которая доступна для CFString. К счастью, NSString (значение, типизированное как NSString*) и CFString (значение CFStringRef) являются взаимозаменяемыми: вы можете использовать один из типов там, где ожидается другой, хотя, чтобы успокоить компилятор, может потребоваться приведение типов. Эта взаимозаменяемость описана и в документации.

Для иллюстрации я использую CFString для преобразования строки NSString, представляющей целое значение, в целое число (для этого не так уж необходимо применять CFString, и я делаю это только в иллюстративных целях; тип NSString имеет решающий эту задачу метод intValue):

```
NSString* answer = @"42";
int ans = CFStringGetIntValue((CFStringRef)answer);
```

Объектные типы C, представляющие собой указатели на структуры, обычно имеют имена, заканчивающиеся на "Ref", и которые можно в целом именовать как CFTypeRef, в действительности представляют собой обобщенный указатель на void. Таким образом, можно использовать упомянутую выше взаимозаменяемость типов как приведение типов между объектными и обобщенными указателями, т.е. от id к void* или от void* к id. Даже там, где взаимозаменяемости между *конкретными* типами (как между типами NSString и CFString) нет, всегда можно обеспечить взаимозаменяемость с верхним уровнем иерархии, т.е. между id или NSObject (базовый класс объекта, см. главу 4) и CFTypeRef.

Блоки

Блок представляет собой расширение языка C, введенное в OS X 10.6 и доступное начиная с iOS 4.0. Это способ связывания некоторого кода в единую сущность и передачи его как аргумента функции C или методу Objective-C. Это похоже на то, что мы делали в примере 3.1, передавая в качестве аргумента указатель на функцию; однако теперь мы будем передавать *сам код*. Этот способ имеет некоторые важные преимущества перед передачей указателя, о чем мы поговорим чуть позже.

Для иллюстрации я перепишу пример 3.1 с использованием блока вместо указателя на функцию. Вместо вызова метода sortedArrayUsingFunction:context: я вызываю метод sortedArrayUsingComparator:, который получает блок в качестве параметра. Этот блок типизирован как

```
NSComparisonResult (^)(id obj1, id obj2)
```

Это выражение аналогично синтаксису определения типа указателя на функцию, но с символом ^ вместо *. Здесь описывается блок, который получает два параметра типа id и возвращает NSComparisonResult (который представляет собой просто NSInteger, с тем же смыслом, что и в примере 3.1). Мы можем определить блок прямо в качестве аргумента в вызове sortedArrayUsingComparator:, как в примере 3.2.

Пример 3.2. Использование встроенного блока вместо функции

```
NSArray* arr2 = [arr sortedArrayUsingComparator: ^(id obj1, id obj2) {
    NSString* s1 = obj1;
```

```
    NSString* s2 = obj2;
    NSString* string1end = [s1 substringFromIndex:[s1 length] - 1];
    NSString* string2end = [s2 substringFromIndex:[s2 length] - 1];
    return [string1end compare:string2end];
}];
```

Синтаксис определения встроенного блока имеет вид

```
^❶(id obj1, id obj2)❷ {❸
    //...
}
```

❶ Символ ^

❷ Скобки с параметрами, аналогично параметрам в определении функции C.

❸ Наконец, содержимое блока в фигурных скобках. Фигурные скобки образуют область видимости.

Возвращаемый тип в определении встраиваемого блока обычно опускается. Если он включен, то находится перед символом ^, а не в круглых скобках. Если же он опущен, то вы можете использовать выражение приведения типа в строке `return`, чтобы возвращаемый тип соответствовал ожидаемому.

Встроенный блок, как показанный в примере 3.2, не может использоваться повторно; если бы у нас было два вызова `sortedArrayUsingComparator:`, то мы должны были писать этот блок в полном объеме дважды. Чтобы избежать такого повторения, или просто для ясности, блок может быть присвоен переменной, которая затем передается методу как аргумент (пример 3.3).

Пример 3.3. Присваивание блока переменной

```
NSComparisonResult (^sortByLastCharacter)(id, id) = ^(id obj1, id obj2) {
    NSString* s1 = obj1;
    NSString* s2 = obj2;
    NSString* string1end = [s1 substringFromIndex:[s1 length] - 1];
    NSString* string2end = [s2 substringFromIndex:[s2 length] - 1];
    return [string1end compare:string2end];
};
NSArray* arr2 = [arr sortedArrayUsingComparator: sortByLastCharacter];
NSArray* arr4 = [arr3 sortedArrayUsingComparator: sortByLastCharacter];
```

Возможно, наиболее примечательной возможностью блоков является следующая: переменные в области видимости в точке, где определен блок, сохраняют свои значения в блоке на этот момент, даже несмотря на то, что блок может быть выполнен в некоторый более поздний момент. (Технически мы говорим, что блок является замыканием и что значения переменных вне блока могут захватываться блоком.) Этот аспект блоков делает их полезными для определения функциональности, которая будет выполнена в более позднее время или даже в некотором другом потоке.

Вот пример из реальной практики:

```
CGPoint p = [v center];
CGPoint pOrig = p;
p.x += 100;
void (^anim) (void) = ^{
    [v setCenter: p];
};
void (^after) (BOOL) = ^(BOOL f) {
```

```
    [v setCenter: pOrig];
};
NSUInteger opts = UIViewAnimationOptionAutoreverse;
[UIView animateWithDuration:1 delay:0 options:opts
                animations:anim completion:after];
```

Этот код делает нечто достаточно удивительное. Ìàòîä `animateWithDuration:delay:options:animations:completion:` настраивает анимацию с помощью блоков. Но сам вывод осуществляется позже; анимация, а следовательно, и блоки, будет выполняться в неопределенный момент в будущем, после завершения вызова метода, когда ваш код будет заниматься совсем другими вещами. Сейчас в области видимости имеется объект `v` типа `UIView`, наряду с `CGPoint p` и еще одним `CGPoint pOrig`. Переменные `p` и `pOrig` — локальные автоматические; они выходят из области видимости и прекращают существование до начала анимации и выполнения блоков. Тем не менее значения этих переменных используются внутри блоков в качестве параметров сообщений, отправляемых `v`.

Поскольку блок может выполняться позже, не совсем корректно выполнять присваивание значений локальным автоматическим переменным, определенным вне блока; поэтому компилятор постарается остановить вас сообщением "variable is not assignable" (переменная не допускает присваивание):

```
CGPoint p;
void (^aBlock) (void) = ^{
    p = CGPointMake(1,2); // Ошибка
};
```

Локальная автоматическая переменная может быть сделана подчиняющейся специальным правилам сохранения путем объявления переменной с помощью квалификатора `__block`. Этот квалификатор обеспечивает существование переменной вместе с блоком, который ее использует, и имеет два основных применения. Вот первое из них: если блок будет выполняться немедленно, квалификатор `__block` позволит ему установить переменную вне блока равной значению, которое будет необходимо после завершения блока.

Например, метод `enumerateObjectsUsingBlock:` класса `NSArray` принимает блок и немедленно вызывает его для каждого элемента массива. Это основанный на блоке эквивалент цикла `for...in`, который циклически проходит по элементам перечислимой коллекции (глава 1). Здесь мы предлагаем циклический проход по циклу до тех пор, пока не будет найдено искомое значение; когда мы найдем его, мы устанавливаем переменную (`dir`) равной этому значению. Однако эта переменная должна быть объявлена вне блока, поскольку она должна использоваться *после* выполнения блока — нам надо, чтобы ее область видимости выходила за пределы фигурных скобок блока. Поэтому мы объявляем ее с квалификатором `__block`, так что можем присваивать ей значение и внутри блока:

```
CGFloat h = newHeading.magneticHeading;
__block NSString* dir = @"N";
NSArray* cards = @[@"N", @"NE", @"E", @"SE",
                   @"S", @"SW", @"W", @"NW"];
[cards enumerateObjectsUsingBlock:^(id obj,
                            NSUInteger idx, BOOL *stop) {
    if (h < 45.0/2.0 + 45*idx) {
        dir = obj;
        *stop = YES;
    }
}]; // Теперь мы можем использовать dir
```

(Обратите внимание, что присваивание выполняется разыменованному указателю на BOOL. Это способ преждевременного завершения цикла; если мы уже нашли интересующее нас значение, нет смысла продолжать работу цикла. Мы не можем использовать инструкцию break, поскольку на самом деле это не цикл for. Поэтому метод enumerateObjectsUsingBlock: передает блоку параметр, который представляет собой указатель на BOOL и которому блок может косвенно присвоить значение YES как сигнал методу о том, что можно остановить выполнение. Это одна из немногих ситуаций в программировании для iOS, где требуется разыменование указателя.)

Второе из двух применений квалификатора __block обратно первому. Оно реализуется тогда, когда блок будет выполняться через некоторое время после его определения, и мы хотим, чтобы блок использовал значение, которое переменная имеет в момент выполнения, а не захватывал ее значение в момент определения блока. Обычно это связано с тем, что тот же вызов метода, который получает блок (для последующего выполнения) одновременно и устанавливает значение этой переменной (сейчас).

Например, метод beginBackgroundTaskWithExpirationHandler: получает блок, который будет выполнен в некоторый будущий момент времени. Он также генерирует и возвращает UIBackgroundTaskIdentifier, который в действительности представляет собой просто целое число, — и мы хотим использовать это целое число в блоке, если и когда этот блок будет выполнен. Так что мы пытаемся действовать следующим образом: блок передается методу как аргумент, метод вызывается, метод возвращает значение, блок использует это значение. Квалификатор __block делает такую последовательность возможной:

```
__block UIBackgroundTaskIdentifier bti =
    [[UIApplication sharedApplication]
        beginBackgroundTaskWithExpirationHandler: ^{
            [[UIApplication sharedApplication]
                endBackgroundTask:bti];
        }];
```

Тогда же, когда в Objective-C были введены блоки, Apple предоставила системную библиотеку функций C под названием Grand Central Dispatch (GCD), которая интенсивно их использует. Главной целью GCD является управление потоками, но она также оказывается удобной для аккуратного и компактного выражения определенных понятий о том, когда должен быть выполнен код. К примеру, GCD может помочь задержать выполнение нашего кода (отложенное выполнение). В приведенном далее коде блок используется для того, чтобы сказать "измени границы UIView v1, но не прямо сейчас, а через две секунды":

```
dispatch_time_t popTime =
    dispatch_time(DISPATCH_TIME_NOW, 2 * NSEC_PER_SEC);
dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
    CGRect r = [v1 bounds];
    r.size.width += 40;
    r.size.height -= 50;
    [v1 setBounds: r];
});
```

Последний пример блока в действии представляет собой переписанный код из конца главы 1, где метод класса предоставляет объект-синглтон. Функция GCD dispatch_once — очень быстрая и (в отличие от примера из главы 1) безопасная в смысле потоков, — обеспечивает выполнение ее блока (который в данном случае создает синглтон) только один раз в течение всей работы программы. Таким образом, она гарантирует, что синглтон действительно является синглтоном:

```
+ (CardPainter*) sharedPainter {
    static CardPainter* sp = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sp = [CardPainter new];
    });
    return sp;
}
```

Блок в состоянии выполнить присваивание локальной переменной `sp` без квалификатора `__block`, потому что `sp` имеет квалификатор `static`, который дает тот же результат, но даже еще более сильный: так же, как `__block` продляет время жизни переменной до времени жизни блока, `static` продляет время жизни переменной до времени жизни самой программы. Таким образом, его побочный эффект — обеспечение возможности присваивания значений переменной `sp` внутри блока, так же как если бы она была объявлена с квалификатором `__block`.

Если вы хотите детальнее познакомиться с блоками, обратитесь к документации Apple по адресу `http://developer.apple.com/library/ios/#documentation/cocoa/Conceptual/Blocks/`, или к разделу "Blocks Programming Topics" в окне справки Xcode. Полная техническая спецификация синтаксиса блоков имеется по адресу `http://clang.llvm.org/docs/BlockLanguageSpec.html`.

ГЛАВА 4

Классы Objective-C

В этой главе описаны некоторые лингвистические и структурные особенности Objective-C, связанные с классами; в следующей главе мы рассмотрим то же самое по отношению к экземплярам.

Подкласс и суперкласс

В языке Objective-C, как и во многих других объектно-ориентированных языках, предоставляется механизм для указания отношения между двумя классами: они могут быть *подклассом* и *суперклассом* по отношению один к другому. Например, у нас может быть класс `Quadruped` (четвероногих) и класс `Dog` (собак), и мы делаем `Quadruped` суперклассом для `Dog`. Класс может иметь много подклассов, но один непосредственный суперкласс. Я говорю "непосредственный", потому что суперкласс может иметь свой суперкласс, и так далее в растущей цепи, до тех пор, пока мы не достигнем конечного суперкласса, именуемого *базовым*, или *корневым, классом*.

Поскольку класс может иметь много подклассов, но только один суперкласс, существует иерархическое дерево подклассов, получающееся путем ветвления из суперкласса, где каждый подкласс может стать источником дерева своих подклассов. Полное дерево начинается с одного базового класса в верхней части. В действительности каркас Cocoa состоит из одного (огромного!) дерева иерархически организованных классов еще до того, как вы напишете хоть одну строку кода или создадите собственный класс. Диаграмму этого дерева можно представить как схему, на вершине которой находится один глобальный суперкласс, все его непосредственные подклассы — уровнем ниже, а каждый из их непосредственных подклассов — еще одним уровнем ниже и т.д. Среда Xcode готова показать вам такую схему (рис. 4.1): для этого в окне проекта iOS выберите View⇨Navigators⇨Show Symbol Navigator и щелкните на Hierarchical при выбранных (синего цвета) первой и третьей пиктограммах на панели фильтров.

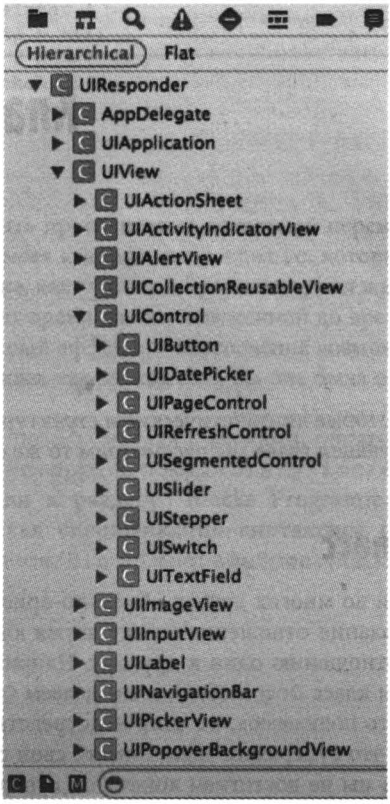


Рис. 4.1. Просмотр иерархии встроенных классов в среде Xcode

Причина существования отношения класс–подкласс заключается в том, чтобы разрешить классам совместно использовать имеющуюся функциональность. Предположим, например, у нас есть класс `Dog` и класс `Cat`, и мы рассматриваем определение метода ходьбы для обоих из них. Мы могли бы рассуждать, что собаки и кошки ходят в значительной степени одинаково, так как оба являются четвероногими. Поэтому может иметь смысл определить `walk` (ходьбу) как метод класса `Quadruped` и сделать `Dog` и `Cat` подклассами класса `Quadruped`. В результате получится, что и для `Dog`, и для `Cat` могут быть отправлены сообщения `walk`, даже если ни один из них не имеет такого метода, потому что каждый из них имеет суперкласс, в котором *есть* метод `walk`. Мы говорим, что подкласс *наследует* методы своего суперкласса.

Целью создания подклассов является не просто то, что класс может наследовать методы другого класса. Подкласс может также определять свои собственные методы. Как правило, подкласс состоит из методов, наследуемых от своего суперкласса, и еще *некоторых собственных* методов. Если класс `Dog` не имеет собственных методов, то трудно понять, почему он должен существовать отдельно от класса `Quadruped`. Но если `Dog` знает, как сделать что-то, что умеет делать не каждый из представителей класса `Quadruped` — скажем, лаять, — то появляется смысл в его существовании как отдельного класса. Если мы определим метод `bark` в классе `Dog` и метод `walk` в классе `Quadruped` и сделаем `Dog` подклассом `Quadruped`, то `Dog` наследует способность ходить от класса `Quadruped`, но при этом будет уметь лаять.

Подкласс может также переопределять методы, унаследованные от его суперкласса. Например, возможно, некоторые собаки лают не так, как другие. У нас может быть класс,

например NoisyDog, который является подклассом Dog. Класс Dog определяет метод bark, но и NoisyDog также определяет метод bark, причем определяет его не так, как класс Dog. Это называется *замещением* (overriding). Совершенно естественное правило гласит, что если подкласс переопределяет метод, унаследованный от его суперкласса, то когда соответствующее сообщение отправляется экземпляру этого подкласса, вызывается версия метода, определенная в подклассе.

Наряду с методами подкласс также наследует переменные экземпляра суперкласса. Естественно, подкласс может также определять свои собственные переменные экземпляра.

Интерфейс и реализация

Как вы знаете из главы 2, весь ваш код располагается в том или ином классе. Поэтому первое, что мы должны сделать, — это указать, что понимается под помещением кода "в класс" в языке Objective-C. Как Objective-C решает, лингвистически и структурно, что "данный код предназначен для такого-то класса"?

Для того чтобы написать код для класса, необходимо предоставить два куска, или два раздела кода, именуемые *интерфейсом* и *реализацией*. Вот как выглядит полный минимальный код, необходимый для определения класса с именем MyClass. Этот класс настолько минимален, что даже не имеет никаких методов:

```
@interface MyClass
@end
@implementation MyClass
@end
```

Директивы компилятора @interface и @implementation указывают компилятору начала разделов интерфейса и реализации определяемого класса, MyClass; соответствующие строки @end показывают, где каждый из этих разделов заканчивается.

В реальной жизни методы MyClass определяются в разделе реализации. Сейчас перед вами класс, который действительно что-то делает:

```
@interface MyClass
@end
@implementation MyClass
- (NSString*) sayGoodnightGracie {
    return @"Good night, Gracie!";
}
@end
```

Обратите внимание, как определен метод. Первая строка представляет собой просто объявление метода, указывающее его тип (метод класса или экземпляра), тип возвращаемого значения, и имя метода, а также типы параметров и их локальные имена (глава 3). Затем в фигурных скобках идет код, выполняемый при вызове метода, так же, как и в случае функции C (глава 1).

Наш минимальный класс по-прежнему бесполезен, потому что он не может быть инстанцирован. В каркасе Cocoa знания о том, как создать экземпляр класса, а также как сделать ряд других вещей, которые должен уметь делать любой класс, находятся в базовом классе, которым является класс NSObject. Таким образом, все классы каркаса Cocoa в конечном итоге должны основываться на классе NSObject, объявляя в качестве суперкласса данного класса NSObject или другой класс, унаследованный от NSObject. (В действительности без явного указания суперкласса объявление нашего класса в Xcode 5 даже не будет компилироваться, а мы получим сообщение "Class 'MyClass' defined without specifying a base class" (класс MyClass определен без указания базового класса)).

Синтаксис объявления суперкласса класса представляет собой двоеточие, за которым следует имя суперкласса в строке `@interface`:

```
@interface MyClass : NSObject
@end
@implementation MyClass
- (NSString*) sayGoodnightGracie {
    return @"Good night, Gracie!";
}
@end
```

`NSObject` — не единственный базовый класс в каркасе Cocoa. Раньше это было так, но теперь есть и другой базовый класс, `NSProxy`, который используется только в особых обстоятельствах. Если у вас нет причины наследовать свой класс от некоторого другого класса — наследуйте его от `NSObject`.

В своей наиболее полной форме раздел интерфейса может содержать гораздо больший материал. В частности, если мы хотим объявить наши методы так, чтобы другие классы могли узнать о них и вызывать, объявления этих методов должны находиться в разделе интерфейса. Объявление метода соответствует имени и сигнатуре определения метода и заканчивается (обязательной) точкой с запятой:

```
@interface MyClass : NSObject
- (NSString*) sayGoodnightGracie;
@end
@implementation MyClass
- (NSString*) sayGoodnightGracie {
    return @"Good night, Gracie!";
}
@end
```

(В действительности определение метода также может иметь точку с запятой перед фигурными скобками. Но такая запись очень редка, и я ее никогда не использую.)

Следует также упомянуть переменные экземпляра. Если наш класс должен иметь какие-либо переменные экземпляра (за исключением тех, что унаследованы от суперкласса), они должны быть объявлены. В современной версии языка Objective-C вы, вероятно, будете объявлять большинство ваших переменных экземпляра неявно, с помощью методики, которая будет описана в главах 5, и 12. Но и сейчас иногда можно объявить переменную экземпляра явно; в любом случае вам нужно знать, как это делается.

В свое время явное объявление переменных экземпляра должно было находиться в фигурных скобках в начале раздела интерфейса:

```
@interface MyClass : NSObject {
    // Здесь располагаются объявления переменных экземпляра
}
- (NSString*) sayGoodnightGracie;
@end
@implementation MyClass
- (NSString*) sayGoodnightGracie {
    return @"Good night, Gracie!";
}
@end
```

Однако начиная с версии компилятора LLVM 3.0 (в среде Xcode 4.2 и более поздних версиях) разрешается размещать объявления переменных экземпляра в фигурных скобках в начале раздела реализации. Это более логичное место для объявления переменных, поскольку, как

я объясню в следующем разделе, раздел интерфейса может быть видимым другим классам, в то время как обычно нет никаких причин, по которым другим классам должны быть видимы переменные экземпляра, которые в общем случае являются закрытыми. Поэтому я предпочитаю новый стиль:

```
@interface MyClass : NSObject
- (NSString*) sayGoodnightGracie;
@end
@implementation MyClass {
    // Здесь располагаются объявления переменных экземпляра
}
- (NSString*) sayGoodnightGracie {
    return @"Good night, Gracie!";
}
@end
```

(Однако, если подкласс должен наследовать переменную экземпляра суперкласса, это может послужить причиной для суперкласса объявить переменную экземпляра в разделе интерфейса, так, чтобы подкласс мог увидеть его и получить доступ к переменной экземпляра.)

Более подробно о переменных экземпляра будет рассказано в главе 5.

Заголовочный файл и файл реализации

Интерфейс и реализация класса вполне могут находиться в одном и том же файле, как и несколько классов могут быть определены в одном файле, но обычно это не так. Распространенным стилем программирования является соглашение "один класс — два файла": в одном файле содержится раздел интерфейса, а во втором — раздел реализации. Файл реализации импортирует заголовочный файл (см. директиву `#import` в главе 1), что эффективно объединяет воедино определение класса, несмотря на то, что оно разделено между двумя файлами.

Предположим, например, что мы определили класс `MyClass`. В таком случае у нас обычно имеются два файла, `MyClass.h` и `MyClass.m`. (Именование файла именем класса не является необходимым — это просто достаточно удобно.) Раздел интерфейса находится в файле `MyClass.h`, который называется *заголовочным файлом*. Раздел реализации размещается в файле `MyClass.m`, который носит название *файл реализации*. Файлы реализации импортируют заголовочные файлы. Разделение на два файла ничуть не мешает, так как Xcode, ожидая, что вы будете следовать описанному соглашению, упрощает переход от редактируемого `.h`-файла к соответствующему `.m`-файлу, и наоборот (Navigate⇨Jump to Next Counterpart).

При таком размещении легко настраивать дальнейший импорт. Заголовочный файл импортирует базовый заголовочный файл для всего каркаса Cocoa; в случае программы для iOS это файл `UIKit.h` (см. главу 1). Файлам реализации нет необходимости импортировать `UIKit.h`, поскольку его импортирует заголовочный файл, который, в свою очередь, импортируется файлом реализации. Если классу надо знать о другом классе, который не импортируется таким способом, он должен импортировать заголовочный файл этого класса.

В примере 4.1 приведен пример описанной схемы; в нем я предполагаю наличие другого класса, `MyOtherClass`, к которому должен иметь возможность обратиться код `MyClass`.

Пример 4.1. Типичная схема определения класса

```
// MyClass.h:

#import <UIKit/UIKit.h>

@interface MyClass : NSObject
```

```
- (NSString*) sayGoodnightGracie;
@end


// MyClass.m:

#import "MyClass.h"
#import "MyOtherClass.h"

@implementation MyClass {
    // Здесь располагаются объявления переменных экземпляра
}
- (NSString*) sayGoodnightGracie {
    return @"Good night, Gracie!";
}
@end
```

Результатом такого размещения является то, что все оказывается корректно видимым. Никакой файл не импортирует файл реализации; таким образом, все, находящееся в этом файле реализации, оказывается закрытым от посторонних и доступно только данному классу. Если один класс должен общаться с другим классом, он импортирует его заголовочный файл. В примере 4.1:

- Заголовочный файл класса `MyClass` импортирует файл `UIKit.h`, поскольку в противном случае класс `MyClass` не может быть объявлен как унаследованный от класса `NSObject`.
- Файл реализации класса `MyClass` импортирует файл `MyClass.h`, поскольку в противном случае объявление класса будет неполным. В качестве полезного побочного эффекта класс `MyClass` может таким образом общаться с классом `NSString` и другими классами, встроенными в каркас `UIKit`, не импортируя при этом файл `UIKit.h`.
- Файл реализацииклассa `MyClass` импортирует файл `MyOtherClass.h`, поскольку в противном случае класс `MyClass` не может работать с классом `MyOtherClass` (что, как мы полагаем, он должен делать).

Небольшая проблема возникает тогда, когда заголовочный файл должен упоминать класс, который он при этом не должен импортировать. Предположим, например, что класс `MyClass` имеет открытый метод, который принимает или возвращает экземпляр класса `MyOtherClass`, так что в файле `MyClass.h` необходимо упомянуть об указателе `MyOtherClass*`; но мы предпочли бы, чтобы файл `MyClass.h` не импортировал файл `MyOtherClass.h`. Но если файл `MyClass.h` ничего не знает о файле `MyOtherClass`, то при использовании этого имени компилятор сообщит об ошибке. Для того чтобы заставить компилятор промолчать, можно воспользоваться директивой `@class`. За словом `@class` следует список разделенных запятыми имен классов, заканчивающийся точкой с запятой. Так что `MyClass.h` может начинаться так:

```
#import <UIKit/UIKit.h>
@class MyOtherClass;
```

Затем, как и ранее, следует раздел интерфейса. Директива `@class` просто говорит компилятору: "Не беспокойся, встретив слово `MyOtherClass`, — это имя реально существующего класса". Это все, что достаточно знать компилятору, чтобы разрешить упоминание типа `MyOtherClass*` в заголовочном файле.

Заголовочный файл также вполне подходящее место для определения констант. В главе 1, например, я говорил о проблеме опечаток при вводе словарного ключа, который представляет собой литерал `NSString`, и о том, как легко решить эту проблему с помощью определения имени для такой строки:

```
#define MYKEY @"mykey"
```

Возникает вопрос: где следует разместить это определение? Если о нем должен знать только один класс, определение может находиться в начале файла реализации (ему не нужно находиться в самом разделе реализации). Но если об этом имени должны знать несколько классов, то наиболее подходящим местом является заголовочный файл; каждый файл реализации, который импортирует этот заголовочный файл, будет видеть указанное макроопределение, и вы сможете использовать имя `MYKEY` в этом файле реализации.

(Если знать об определении должно много файлов, его может быть более удобно разместить в `.pch`-файле проекта, единого "предкомпилированного заголовка", который неявно импортирует все `.h`-файлы Более подробно о `.pch`-файлах будет рассказано в главе 6.)

Заголовочные файлы каркаса Cocoa

Классы каркаса Cocoa также следуют соглашению, описанному в примере 4.1: каждый класс разделен на заголовочный файл (содержащий интерфейс) и файл реализации. Однако файлы реализации классов каркаса Cocoa нам не видны. Это одно из основных ограничений каркаса Cocoa; в отличие от многих других каркасов программирования, вы не можете увидеть исходный код каркаса Cocoa — он является секретом для программиста. Для того чтобы понять, как работает каркас Cocoa, вам придется полностью положиться на документацию (и эксперименты). Однако заголовочные файлы каркаса Cocoa открыты, и вам стоит их просмотреть, так как они могут оказаться полезным дополнением к документации (см. главу 8).

Методы классов

Методы классов в общем случае имеют два основных применения.

Фабричные методы

- Метод фабрики — это метод, который создает экземпляр данного класса. Например, класс `UIFont` имеет метод класса `fontWithName:size:`. Вы указываете имя и размер, и класс `UIFont` возвращает вам экземпляр `UIFont`, соответствующий шрифту с этим именем и размером. Метод класса, который возвращает экземпляр-синглтон (описанный в конце главы 1), также является фабричным методом.

Глобальные вспомогательные методы

- Классы являются глобальными (т.е. видимыми из всего кода; см. главу 13), поэтому класс является хорошим местом для размещения вспомогательного метода, который может вызвать кто угодно и для вызова которого не требуются накладные расходы по созданию экземпляра. Например, класс `UIFont` имеет метод класса `familyNames`.

Он возвращает массив строк (т.е. NSArray экземпляров NSString), состоящий из имен семейств шрифтов, установленных в данном устройстве. Поскольку этот метод работает с шрифтами, класс UIFont является логичным местом для его размещения.

Большинство методов, которые будут написаны вами, будут методами экземпляров, но вы можете создавать и методы классов. При их создании вы, скорее всего, будете преследовать цели, аналогичные описанным примерам.

Секретная жизнь классов

Метод класса может вызываться с помощью отправки сообщения непосредственно имени класса. Например, метод класса familyNames из класса UIFont, о котором я говорил выше, может быть вызван следующим образом:

```
NSArray* fams = [UIFont familyNames];
```

Понятно, что это возможно потому, что класс является объектом (глава 2), и здесь имя класса представляет этот объект.

Вы не должны делать ничего для создания объекта класса. Один объект класса для каждого класса, определенного в вашей программе, создается автоматически при запуске программы. (Сюда входят и классы, импортируемые вашей программой, так что существует объект класса MyClass, поскольку вами определен MyClass, и объект класса NSString, поскольку вы импортируете UIKit.h и весь каркас Cocoa.) Именно этому объекту и посылаются сообщения, когда вы отправляете их имени класса.

Способность послать сообщение непосредственно имени класса представляет собой своего рода синтаксическую стенографию. "Голое" имя класса можно использовать только двумя способами (и мы уже знаем их оба).

Для отправки сообщения

В выражении [UIFont familyNames] сообщение familyNames отправляется имени UIFont.

Для указания типа экземпляра

В выражении NSString* имя NSString, за которым следует звездочка, определяет указатель на экземпляр этого класса.

В противном случае, чтобы говорить об объекте класса, необходимо формально получить этот объект. Один из способов сделать это заключается в том, чтобы отправить сообщение class классу или экземпляру. Например, [MyClass class] возвращает действительный объект класса. Некоторые встроенные методы каркаса Cocoa требуют в качестве параметра объект класса, тип которого описывается как Class. Для того чтобы передать этот объект класса в качестве аргумента, необходимо формально получить его.

Рассмотрим, например, анализ объектом класса, к которому он принадлежит. Метод экземпляра isKindOfClass: объявлен следующим образом:

```
- (BOOL)isKindOfClass:(Class)aClass
```

Это означает, что его можно вызвать с помощью следующего кода:

```
if ([someObject isKindOfClass: [MyClass class]]) // ...
```

Объект класса не является экземпляром, но он определенно представляет собой полноправный объект. Следовательно, объект класса можно использовать везде, где может использоваться объект. Например, его можно присвоить переменной типа id:

```
id classObject = [MyClass class];
```

Затем вы можете вызвать метод класса, отправляя сообщение этому объекту, поскольку он является объектом класса:

```
id classObject = [MyClass class];
[classObject someClassMethod];
```

Все объекты класса являются также членами класса Class, так что вы можете написать следующий код:

```
Class classObject = [MyClass class];
[classObject someClassMethod];
```

Эти примеры будут компилироваться до тех пор, пока someClassMethod представляет собой известный метод класса *любого* класса.

Ссылка на объект является указателем (глава 3). Это верно и для ссылки на объект класса так же, как и для ссылки на экземпляр. Подобно id, термин Class типизирован как указатель, так что вам не надо использовать с ним звездочку.

Глобальное пространство имен

Определяя классы, для предотвращения конфликтов имен разумно выбирайте имена классов. В языке Objective-C нет пространства имен; есть одно огромное пространство имен, содержащее все имена. Нежелательно, чтобы имя вашего собственного класса (или любое имя константы верхнего уровня) совпадало с именем, определенным в каркасе Cocoa. Вместо пространства имен имеется соглашение: каждый каркас Cocoa добавляет к именам в качестве префиксов определенную группу из прописных букв (NSString и NSArray, CGFloat и CGRect и т.д.). Компания Apple предлагает вам также использовать свой собственный префикс; в действительности, когда вы создаете новый проект в среде Xcode, вам будет предложена возможность задать префикс, который будет отображаться перед автоматически создаваемыми именами классов. Не используйте никакой из префиксов компании Apple! Ничто не ограничивает ваш префикс двумя буквами, как и не требует, чтобы буквы были прописными. Фактически, поскольку Apple использует в префиксах только прописные буквы, "My" в качестве префикса оказывается достаточно безопасным.

ГЛАВА 5

Экземпляры Objective-C

Экземпляры являются сердцем активности программы на языке Objective-C. Получение и управление экземплярами будет иметь решающее значение для всего, что вы делаете. Почти каждая строка вашего кода будет связана с одним или несколькими из следующих видов деятельности.

- Ссылка на уже существующий экземпляр.
- Создание нового, ранее не существовавшего экземпляра.
- Присваивание экземпляра переменной.
- Отправка сообщения экземпляру.
- Передача экземпляра в вызов метода в качестве параметра.

Создание экземпляров

Объекты ваших классов создаются автоматически во время запуска программы, но экземпляры должны создаваться индивидуально в процессе работы программы. В конечном счете каждый экземпляр начинает существование единственным способом: когда некто просит класс создать свой экземпляр (см. главу 4). Однако это может быть сделано тремя путями: с помощью готовых экземпляров, созданием экземпляра с нуля и созданием экземпляра на основе nib.

Готовые экземпляры

Один из способов создания экземпляров — косвенный, путем вызова кода, который создает экземпляр за вас. Экземпляр, полученный таким путем, можно рассматривать как "готовый экземпляр". (Это придуманное мною название, а не официальный термин.) Рассмотрим следующий простой код:

```
NSString* s2 = [s uppercaseString];
```

Документация по методу экземпляра `uppercaseString` класса `NSString` гласит, что он возвращает "строку, в которой каждый символ получателя заменен соответствующим значением в верхнем регистре". Другими словами, вы отправляете сообщение `uppercaseString` экземпляру класса `NSString` и получаете обратно другой, *отличный* от исходного, вновь созданный экземпляр класса `NSString`. После выполнения приведенной строки кода `s2` указывает на экземпляр класса `NSString`, который до этого не существовал.

Экземпляр класса NSString, созданный методом uppercaseString, передается вам готовым. Ваш код ничего не говорит об создании его экземпляра, он просто отправляет сообщение uppercaseString. Однако понятно, что *кто-то* отправляет сообщение о создании экземпляра, поскольку таковое имеет место; вы получаете новоиспеченный экземпляр класса NSString. Этот "кто-то", по всей видимости, — код в классе NSString. Но мы не должны беспокоиться о деталях. Нам гарантируется получение нового, готового к немедленному употреблению экземпляра класса NSString, и это все, о чем нам надо знать.

Аналогично любой фабричный метод создает экземпляр класса и возвращает получившийся экземпляр как готовый. Так, например, метод stringWithContentsOfFile:encoding:error: класса NSString читает файл и создает его экземпляр, представляющий содержимое файла. Вся работа по созданию экземпляра выполнена за вас. Вы просто получаете результирующую строку и можете делать с ней, что вам нужно.

Создание экземпляра класса с нуля

Альтернативой получению готового экземпляра является явное требование к классу создать экземпляр. Для этого классу отправляется сообщение alloc. Метод класса alloc реализован классом NSObject, корневым классом, из которого наследуются все иные классы. Он приводит к тому, что для экземпляра выделяется необходимая память, так что указатель на экземпляр может на нее указывать. (Управление этой памятью — отдельный вопрос, который рассматривается в главе 12.)

Вы никогда, *никогда*, ***никогда*** не должны вызывать метод alloc сам по себе. Вы должны *немедленно* вызвать другой метод, метод экземпляра, который *инициализирует* вновь созданный экземпляр, вводит его в известное корректное состояние, так что ему могут отправляться другие сообщения. Такой метод именуется *инициализатором*. Кроме того, инициализатор возвращает экземпляр — обычно тот же самый, но уже инициализированный. Следовательно, вы можете (и всегда должны) вызывать alloc и инициализатор в одной строке кода. Минимальным инициализатором является init. Такой базовый шаблон, неформально известный как "alloc—init", показан в примере 5.1.

Пример 5.1. Базовый шаблон создания с нуля

```
SomeClass* aVariable = [[SomeClass alloc] init];
```

Вы не можете создать экземпляр с нуля, если не знаете, как выполнить инициализацию, так что мы тотчас же перейдем к этому важному вопросу.

Инициализация

Каждый класс определяет или наследует как минимум один инициализатор. Это метод экземпляра; соответствующий экземпляр был только что создан путем вызова метода класса alloc, и именно этому новоиспеченному экземпляру должно быть отправлено сообщение инициализации. Сообщение инициализации должно быть отправлено экземпляру немедленно после создания этого экземпляра сообщением alloc и не должно отправляться экземпляру в любое другое время.

Основной шаблон инициализации, как показано в примере 5.1, представляет собой вызов alloc, вложенный в вызов инициализатора, результат вызова которого (не alloc!) присваивается переменной. Одной из причин для такой вложенной структуры является то, что если что-то идет не так и экземпляр не может быть создан или инициализирован, то инициализатор вернет значение nil; следовательно, очень важно сохранить результат инициализации и проверить его значение (не результат alloc!) на равенство nil.

Для того чтобы помочь определить инициализаторы, все они именуются в соответствии с определенным соглашением, согласно которому все инициализаторы, и только инициализаторы имеют имена, начинающиеся с `init`. Простейший инициализатор называется `init` и не принимает ни одного параметра. Прочие инициализаторы принимают параметры, и их имена обычно начинаются с фразы `initWith`, за которой следует описание их параметров. Например, документация класса `NSArray` перчисляет следующие методы инициализации:

```
- initWithArray:
- initWithArray:copyItems:
- initWithContentsOfFile:
- initWithContentsOfURL:
- initWithObjects:
- initWithObjects:count:
```

Рассмотрим реальный пример. В главе 3, мы создавали массив `NSArray` из трех строк, составляющих его элементы, с помощью фабричного метода `arrayWithObjects:`, принимающего список объектов, завершающийся `nil`, и возвращающий готовый экземпляр:

```
NSArray* pep =
    [NSArray arrayWithObjects:@"Manny", @"Moe", @"Jack", nil];
```

Оказывается, однако, что имеется также инициализатор `NSArray` с именем `initWithObjects:`, который работает точно таким же образом, как и `arrayWithObjects:`. Отличие заключается в том, что последний является фабричным методом класса, который возвращает готовый объект, в то время как первый является инициализатором, методом экземпляра, который может использоваться только в одном ряду с `alloc`. Таким образом, мы можем выполнить в точности те же действия, что и в главе 3, с тем отличием, что в этот раз мы сами создаем экземпляры с нуля:

```
NSArray* pep =
    [[NSArray alloc] initWithObjects:@"Manny", @"Moe",
                                     @"Jack", nil];
```

В современной версии языка Objective-C, как я упоминал в главе 3, вы вряд ли будете вызывать `arrayWithObjects:` или `initWithObjects:`, поскольку теперь имеется удобный литеральный синтаксис массивов, который генерирует массив на основе списка его содержимого:

```
NSArray* pep = @[@"Manny", @"Moe", @"Jack"];
```

Поэтому я приведу другой пример. Предположим, что, так или иначе, но у вас есть массив `pep`, содержащий три строки, `@"Manny"`, `@"Moe"` и `@"Jack"`, и вы хотите создать на основе первого массива второй, который содержит те же три строки. Заметим, что для этого недостаточно просто присвоить другой переменной `NSArray` значение переменной `pep`:

```
NSArray* pep2 = pep; // Нет, этим другой массив не создать
```

Ссылки на объекты являются указателями, и присваивание указателей просто приводит к тому, что две ссылки указывают на одну и ту же сущность (глава 3). Так что `pep2` в этом коде не является вторым массивом; это тот же самый массив, что, определенно, не то, что нам хотелось получить. Чтобы получить второй экземпляр массива на основе первого, мы можем вызвать метод класса `arrayWithArray:` следующим образом:

```
NSArray* pep2 = [NSArray arrayWithArray: pep];
```

Теперь `pep2` представляет собой новоиспеченный экземпляр, отличный от `pep`. Это готовый экземпляр, возвращаемый фабричным методом класса. И вновь оказывается, что

существует соответствующий инициализатор, `initWithArray:`. Таким образом, чтобы создать экземпляр с нуля, мы можем использовать этот инициализатор:

```
NSArray* pep2 = [[NSArray alloc] initWithArray: pep];
```

Часто бывает так, что встроенный класс каркаса Cocoa предлагает фабричный метод и инициализатор, которые, начиная с одного и того же типа данных, производят один и тот же результат. В конечном счете нет никакой разницы, что именно вы используете; при одних и тех же аргументах оба подхода дают экземпляры, неотличимые друг от друга. (Эти два подхода имеют различные последствия для управления памятью, как я поясню в главе 12, но при использовании ARC эти различия, вероятно, не будут играть для вас никакой роли.)

Просматривая документацию об инициализаторах, не забывайте пройтись вверх по иерархии классов. Например, в документации класса `UIWebView` инициализаторы не перечислены, но класс `UIWebView` наследуется от `UIView`, в документации класса `UIView` вы можете обнаружить метод `initWithFrame:`. Кроме того, метод `init` определен как метод экземпляра класса `NSObject`, так что каждый класс наследует его, и сообщение `init` может быть отправлено всем вновь созданным экземплярам. Таким образом, если класс не определяет своего собственного инициализатора, можно инициализировать его экземпляр с помощью метода `init`. Например, в документации класса `UIResponder` нет ни инициализаторов, ни фабричных методов. Значит, чтобы создать экземпляр `UIResponder` с нуля, вы должны вызвать методы `alloc` и `init`.

Если вы планируете вызывать инициализатор `init`, последовательные вызовы методов `alloc` и `init` можно объединить в вызов метода класса `new`. Другими словами, `[MyClass new]` представляет собой синоним для `[[MyClass alloc] init]`. Это удобное сокращение, но оно применимо только тогда, когда инициализатором является метод `init`; для использования любого другого инициализатора необходимо вызывать метод `alloc` и требуемый инициализатор явно.

Назначенный инициализатор

Если класс определяет инициализаторы, один из них может быть описан в документации как назначенный инициализатор. (В имени метода нет ничего, что говорило бы о том, что это назначенный инициализатор; для того чтобы это определить, следует обратиться к документации.) Например, в документации класса `UIView` метод `initWithFrame:` описан как назначенный инициализатор. Класс, который не определяет назначенный инициализатор, наследует назначенный инициализатор своего суперкласса. Последним назначенным инициализатором, наследуемым всеми классами без иных назначенных инициализаторов, является `init`. Таким образом, каждый класс имеет ровно один назначенный инициализатор.

Назначенный инициализатор класса должен вызываться в процессе создания экземпляра класса. Если у класса имеется несколько инициализаторов, неважно, унаследованных или определенных в классе, назначенным инициализатором является инициализатор, от которого зависят другие инициализаторы: в конечном итоге они должны его вызывать.

Прочие инициализаторы могут находиться с назначенным в различных взаимоотношениях. Назначенный инициализатор может иметь больше параметров, обеспечивая явную установку большего количества переменных экземпляра (в то время как другие инициализаторы для некоторых переменных экземпляра используют значения по умолчанию). Он может быть также базовым способом инициализации. Вот несколько примеров из реальной практики.

- Документация класса NSDate гласит, что назначенным инициализатором является initWithTimeIntervalSinceReferenceDate: и что другие инициализаторы (такие, как initWithTimeIntervalSinceNow:) вызывают его.
- Документация класса UIView гласит, что назначенным инициализатором является initWithFrame:. Класс UIView не имеет других инициализаторов, но их имеют некоторые из его подклассов. Класс UIWebView, подкласс UIView, не имеет инициализатора, так что initWithFrame: является его назначенным инициализатором (посредством наследования). Класс UIImageView, подкласс UIView, имеет инициализаторы, такие как initWithImage:, но ни один из них не является назначенным; таким образом, initWithFrame: является назначенным инициализатором для этого класса, и initWithImage: должен вызывать initWithFrame:.

Класс, который реализует свой собственный назначенный инициализатор, должен перекрывать назначенный инициализатор, унаследованный от суперкласса, так что последний должен вызывать первый.

- Класс NSDate выполняет замещение унаследованного метода init для вызова собственного назначенного инициализатора, initWithTimeIntervalSinceReferenceDate:, со значением, которое генерирует дату, определяемую текущим моментом времени.
- Класс UIView замещает унаследованный метод init для вызова собственного назначенного инициализатора, initWithFrame:, со значением рамки CGRectZero.

Создание экземпляра класса на основе nib

Третий способ создания экземпляра — с помощью nib-файла. Nib-файл представляет собой файл в созданном приложении, сгенерированный из .storyboard-файла или .xib-файла, в котором вы "рисуете" части пользовательского интерфейса. Большинство проектов Xcode включают как минимум один .storyboard- или .xib-файл, и, таким образом, большинство приложений содержит как минимум один nib-файл в пакете приложения. Для использования nib-файла последний должен быть явно загружен с использованием некоторого механизма в процессе выполнения приложения. Nib-файл состоит из имен классов вместе с инструкциями для создания и инициализации их экземпляров. Когда в процессе работы приложения nib-файл загружается, эти инструкции выполняются — создаются и инициализируются экземпляры этих классов. Таком образом, работающее приложение генерирует экземпляры на основании того, что вы изначально нарисовали в .storyboard- или .xib-файле.

Предположим, например, что вы хотите, чтобы пользователю было показано представление, в котором имеется кнопка с надписью "Howdy!". Среда Xcode позволяет создать и разместить эту кнопку графически, редактируя .storyboard- или .xib-файл. Для этого вы должны перетащить кнопку из библиотеки объектов, как показано на рис. 5.1. Затем следует поместить ее на свое место и настроить ее заголовок "Howdy!", как показано на рис. 5.2. По сути, тем самым вы создаете чертеж того, как должно выглядеть представление и что в нем должно быть.

При работе приложения выполняется загрузка nib-файла, и чертеж превращается в реальность. Для этого чертеж рассматривается как набор инструкций по созданию объектов. Кнопка, которую вы перетаскиваете на представление, рассматривается как экземпляр класса UIButton. Класс UIButton получает команду создать свой экземпляр. Созданный экземпляр затем инициализируется, располагаясь в позиции, выбранной вами для него во время

создания чертежа (переменная `frame` экземпляра), получая заголовок, который был указан в чертеже (переменная `title` экземпляра), и выводится на представлении. По сути, загрузка вашего nib-файла эквивалентна приведенному ниже коду (в предположении, что `self.view` представляет собой ссылку на объект представления).

```
UIButton* b =                          // Создание экземпляра
        [UIButton buttonWithType:UIButtonTypeSystem];
[b setTitle:@"Howdy!"
        forState:UIControlStateNormal];  // Настройка заголовка
[b setFrame: CGRectMake(100,100,52,30)]; // Настройка рамки
[self.view addSubview:b];              // Размещение в представлении
```

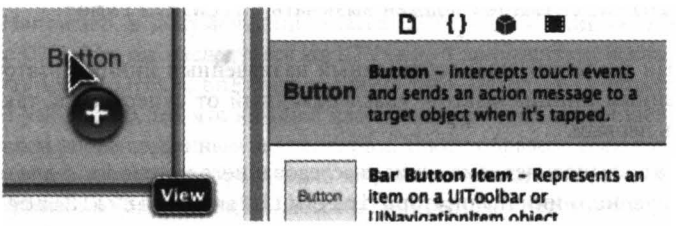


Рис. 5.1. Перетягивание кнопки в представление

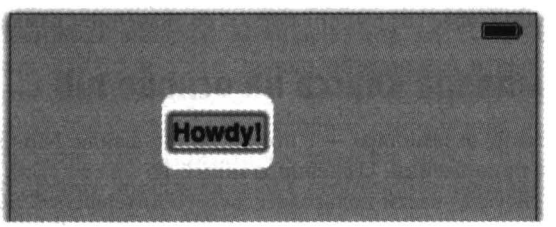


Рис. 5.2. Графическая настройка кнопки

В приведенном исходном тексте экземпляр класса `UIButton` создается как готовый, с помощью вызова фабричного метода, и затем присваивается переменной `b`. Однако в случае создания экземпляра на основе nib-файла получение экземпляра из nib-файла для присваивания переменной оказывается достаточно сложным делом, предусматривающим значительные подготовительные усилия и использование устройства под названием *выход* (outlet). Тот факт, что nib-файлы являются источником экземпляров и что эти экземпляры пробуждаются к жизни при загрузке nib-файла, вместе с проблемой присваивания этих экземпляров переменным, является для начинающих программистов неиссякающим источником путаницы. Детально этот вопрос будет рассматриваться в главе 7.

Полиморфизм

Для данных подкласса и суперкласса можно свободно подставлять экземпляр подкласса там, где требуется экземпляр суперкласса. Например, класс `UIButton` является подклассом `UIControl`, который является подклассом `UIView`. Поэтому вполне корректным является следующий исходный текст:

```
UIButton* b = [UIButton buttonWithType:UIButtonTypeSystem];
UIView* v = b;
```

Переменная b объявлена как класс UIButton, но я присваиваю ее переменной, объявленной как класс UIView. Это вполне законно и приемлемо, поскольку класс UIView является предком (находится выше по цепочке суперклассов) класса UIButton. Говоря иначе, я веду себя так, как будто класс UIButton является классом UIView, и компилятор с этим согласен — именно потому, что класс UIButton действительно является классом UIView.

Однако при работе приложения важен не *объявленный* класс переменной, а *фактический класс объекта*, на который ссылается эта переменная. После того, как я присвоил экземпляр класса UIButton переменной v, объект, на который указывает переменная v, представляет собой класс UIButton. Так что, несмотря на то, что переменная v имеет тип данных, отличный от класса UIButton, не будет никаких проблем в отправке ей сообщений, соответствующих классу UIButton, поскольку фактически она представляет экземпляр класса UIButton. Например:

```
UIButton* b = [UIButton buttonWithType:UIButtonTypeSystem];
UIView* v = b;
[v setTitle:@"Howdy!" forState:UIControlStateNormal];
```

Класс UIView не может принимать сообщения setTitle:forState:. Тем не менее приведенный код вполне осмыслен и безопасен; переменная v может быть типизирована как простой класс UIView, но в действительности, при выполнении кода, она будет указывать на экземпляр класса UIButton, который может принимать сообщение setTitle:forState:. Однако компилятор ничего не знает о том, на что в действительности будет указывать переменная v во время выполнения приложения, так что (при наличии механизма ARC) он сообщает об ошибке компиляции. Заставить компилятор замолчать можно с помощью приведения типа получателя сообщения:

```
UIButton* b = [UIButton buttonWithType:UIButtonTypeSystem];
UIView* v = b;
[(UIButton*)v setTitle:@"Howdy!" forState:UIControlStateNormal];
```

Приведение успокаивает компилятор и он благополучно компилирует код, который отлично работает. Он работает не потому, что я выполнил приведение переменной v к типу UIButton (приведение типов не выполняет никаких реальных преобразований; это просто намек компилятору), а потому, что объект v действительно *является* экземпляром класса UIButton. Я выполнил приведение типа получателя сообщения, которое компилятор всегда принимает к сведению как истину в последней инстанции; но, кроме того, мое приведение говорит правду, поэтому, когда сообщение setTitle:forState: достигает объекта, на который указывает v, все отлично работает. С другой стороны, если бы v был на самом деле экземпляром класса UIView, но *не* класса UIButton, то программа аварийно завершилась бы.

Теперь давайте развернем ситуацию на 180 градусов. Мы назвали объект класс UIButton объектом класса UIView и отправили ему сообщение для объекта класса UIButton. Оставим теперь объект класса UIButton как объект класса UIButton, но отправим ему сообщение для объекта класса UIView.

То, чем в действительности является объект, зависит не только от его класса, но и от наследования этого класса. Сообщение является приемлемым, даже если класс объекта сам не реализует соответствующий метод, но при этом метод реализован где-то выше по цепочке суперклассов. Например:

```
UIButton* b = [UIButton buttonWithType:UIButtonTypeSystem];
[b setFrame: CGRectMake(100,100,52,30)];
```

Этот код нормально работает. Но вы не найдете метод setFrame: в документации класса UIButton. Это потому, что вы ищете в неверном месте. Класс UIButton является подклассом

класса UIControl, а класс UIControl является подклассом класса UIView. Чтобы узнать о методе setFrame:, обратитесь к документации класса UIView. (Ну, если честно, все еще сложнее, и вы не найдете setFrame: и там. Но вы найдете термин frame, который является "свойством", а это по сути то же самое, как я поясню позже в этой главе.) Так что сообщение setFrame: отправляется классом UIButton, но оно соответствует методу, определенному в классе UIView. Тем не менее все нормально работает, потому что объект класса UIButton *является* объектом класса UIView.

Еще одна распространенная ошибка начинающего программиста — обратиться к документации, но не пройтись при этом по цепочке суперклассов. Если вы хотите узнать, какие сообщения можно посылать UIButton, не ограничивайтесь документацией только класса UIButton: просмотрите также документацию класса UIControl, класса UIView и т.д.

Мы рассматривали объект UIButton как UIView, но (с помощью приведения типов) были в состоянии отправить ему сообщение UIButton. Мы рассматриваем объект UIButton как UIButton, но (по наследству) мы в состоянии отправить ему сообщение UIView.

Объект отвечает на отправленное ему сообщение не потому, что что-то в коде говорит, чем он является; он отвечает исходя из того, чем на самом деле он является при работе программы, и того, что сообщение в действительности отправляется именно этому объекту. Когда сообщение отправляется объекту, важно не то, как объявлена или приведена переменная, указывающая на этот объект, а то, объектом какого класса он в действительности является. То, чем в действительности является объект, зависит от его класса, а также цепочки наследования этого класса от суперклассов. Эти факты внутренне присущи объекту и не зависят от того, как ваш код характеризует переменную, указывающую на этот объект. Такое независимое поддержание целостности типа объекта является основой того, что называется *полиморфизмом*.

Но это не все, что касается полиморфизма. Чтобы понять полиморфизм полностью, мы должны пойти дальше и разобраться с динамикой отправки сообщения.

Ключевое слово self

Распространена ситуация, когда в коде метода экземпляра, определенного в классе, необходимо вызывать другой метод экземпляра, определенный в том же классе. Мы еще не обсуждали, как это сделать. Метод вызывается путем отправки сообщения объекту; но чем в этой ситуации является объект? Ответ заключается в ключевом слове self. Вот простой пример:

```
@implementation MyClass
- (NSString*) greeting {
    return @"Goodnight, Gracie!";
}
- (NSString*) sayGoodnightGracie {
    return [self greeting];
}
@end
```

Когда сообщение sayGoodnightGracie отправляется экземпляру класса MyClass, выполняется метод экземпляра sayGoodnightGracie. Он отправляет сообщение greeting для экземпляра self. В результате вызывается метод экземпляра greeting; он возвращает строку @"Goodnight, Gracie!", и эта строка затем возвращается методом sayGoodnightGracie.

Пример выглядит достаточно простым, и это действительно так. В реальной практике ваш код класса часто будет состоять из нескольких открытых методов экземпляра вместе с множеством других методов экземпляра, которые используют открытые методы. В классе одни методы экземпляров будут постоянно вызывать другие. Они будут делать это с помощью отправки сообщений для экземпляра `self`.

Тем не менее за этим простым примером стоит тонкий и важный механизм, который должен выяснить реальное значение ключевого слова `self`. На самом деле ключевое слово `self` не означает "в том же классе". В конце концов, экземпляр — не класс. Но какой именно экземпляр? Это тот же экземпляр, которому было отправлено первоначальное сообщение, в результате обработки которого впервые появилось ключевое слово `self`.

Давайте более подробно рассмотрим, что происходит, когда мы создаем экземпляр класса `MyClass` и отправляем сообщение `sayGoodnightGracie` созданному экземпляру:

```
MyClass* thing = [MyClass new];
NSString* s = [thing sayGoodnightGracie];
```

Мы создаем экземпляр класса `MyClass` и присваиваем созданный экземпляр переменной `thing`. Затем мы отправляем сообщение `sayGoodnightGracie` переменной `thing`, т.е. экземпляру, который только что создали. Сообщение достигает получателя, и оказывается, что это — экземпляр класса `MyClass`. Конечно же, класс `MyClass` реализует метод экземпляра `sayGoodnightGracie`, и этот метод вызывается. Когда он начинает работу, в коде встречается ключевое слово `self`. Оно означает "экземпляр, которому первоначально было отправлено исходное сообщение". Это экземпляр, на который указывает переменная `thing`. Так что теперь сообщение `greeting` отправляется указанному экземпляру (рис. 5.3).

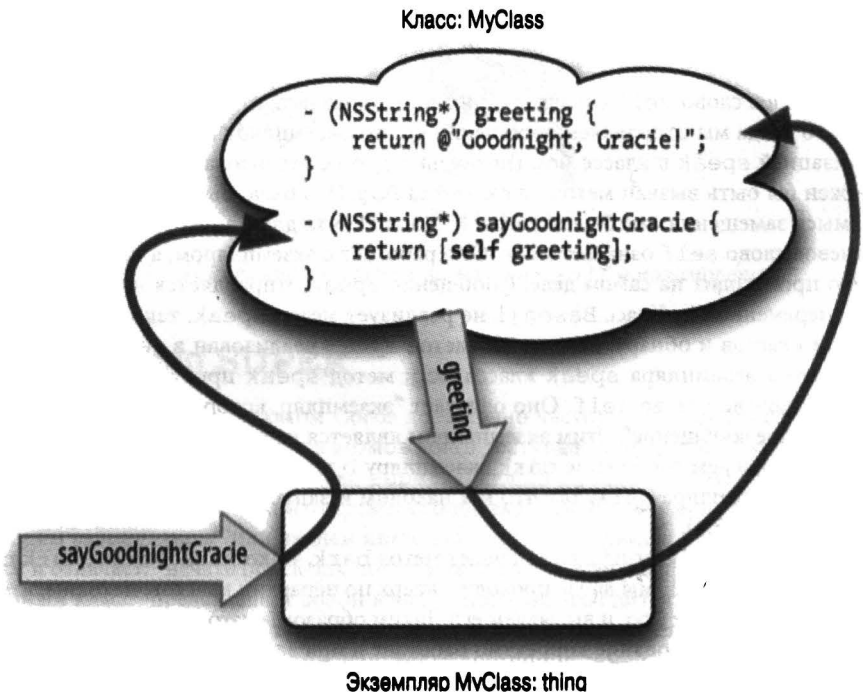


Рис. 5.3. Значение ключевого слова `self`

Этот механизм может показаться довольно сложным, учитывая, что получается интуитивно ожидаемый результат. Однако этот механизм должен быть сложным, чтобы всегда давать верный результат. Это в особенности очевидно в случае, когда в игру включаются суперклассы и класс перекрывает метод суперкласса.

Для иллюстрации предположим, что у нас есть класс Dog с методом экземпляра bark. Предположим также, что класс Dog имеет метод экземпляра speak, который просто вызывает метод bark. Теперь предположим, что у нас имеется подкласс Basenji класса Dog , который замещает метод bark. Что произойдет, когда мы пошлем сообщение speak экземпляру Basenji, как в примере 5.2?

Пример 5.2. Полиморфизм в действии

```
@implementation Dog
- (NSString*) bark {
    return @"Woof!";
}
- (NSString*) speak {
    return [self bark];
}
@end

@implementation Basenji : Dog
- (NSString*) bark {
    return @""; // Пустая строка, Basenjis не лает
}
@end

// В некотором другом классе:
Basenji* b = [Basenji new];
NSString* s = [b speak];
```

Если ключевое слово self означает просто "тот же класс, где встречается данное ключевое слово", то когда мы отправляем сообщение speak экземпляру Basenji, дело заканчивается реализацией speak в классе Dog (поскольку speak реализован именно в нем), и после этого должен бы быть вызван метод bark класса Dog. Это было бы ужасно, так как убивало бы весь смысл замещения; мы бы получили @"Woof!", что для Basenji неверно. Но, к счастью, ключевое слово self означает иное. Оно работает с экземпляром, а не классом.

Вот что происходит на самом деле. Сообщение speak отправляется нашему экземпляру Basenji, переменной b. Класс Basenji не реализует метод speak, так что мы идем вверх по иерархии классов и обнаруживаем, что метод speak реализован в суперклассе Dog. Мы вызываем метод экземпляра speak класса Dog; метод speak приступает к работе, и тут встречается ключевое слово self. Оно означает "экземпляр, которому первоначально было послано исходное сообщение". Этим экземпляром является наш экземпляр b класса Basenji. Так что мы отправляем сообщение bark экземпляру b класса Basenji. Класс Basenji реализует метод экземпляра bark, так что мы находим и запускаем этот метод, и возвращаем пустую строку (рис. 5.4).

Конечно, если класс Basenji *не* замещает метод bark, то когда сообщение bark посылается экземпляру Basenji, мы *вновь* проходим вверх по иерархии классов, находим, что метод bark реализован в классе Dog, и вызываем его. Таким образом, благодаря способу работы ключевого слова self наследование корректно работает и при замещении, и при его отсутствии.

Если вы смогли понять этот пример, то понимаете, что такое полиморфизм. Механизм, который я только что описал, имеет решающее значение для полиморфизма и является основой объектно-ориентированного программирования. (Заметим, что сейчас я говорю об объектно-

ориентированном, а не объектно-основанном программировании, как в главе 2. Это потому, что, на мой взгляд, именно добавление полиморфизма делает объектно-основанное программирование объектно-ориентированным.)

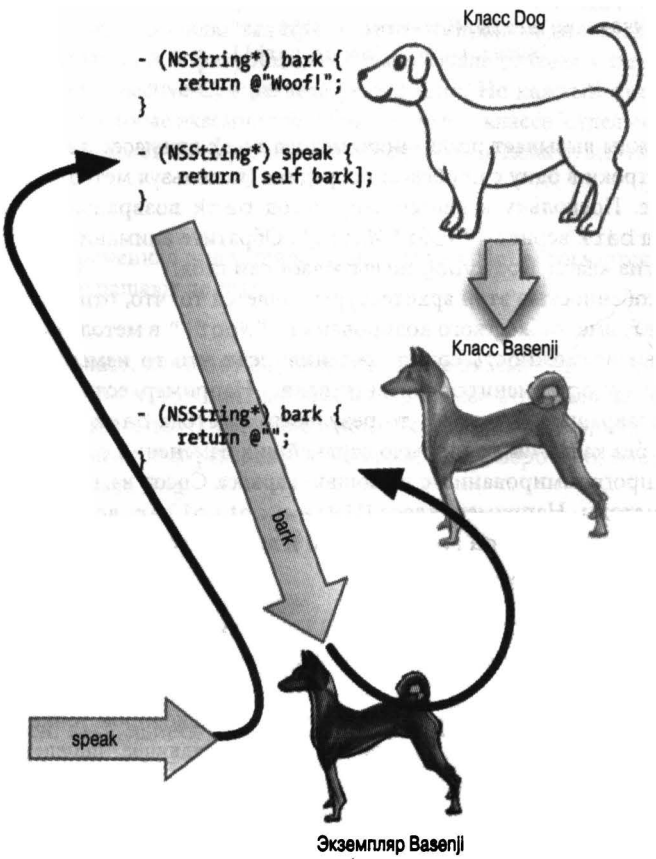


Рис. 5.4. Наследование классов, замещение, `self` и полиморфизм

Ключевое слово SUPER

Иногда (а при работе с каркасом Cocoa достаточно часто) требуется заместить унаследованный метод, но при этом оставить возможность доступа к замещаемой функциональности. Для этого используется ключевое слово `super`. Подобно `self`, ключевое слово `super` представляет собой нечто, чему можно отправить сообщение. Но его значение не имеет ничего общего с "этим экземпляром" или любым иным экземпляром. Ключевое слово `super` основано на классе и означает: "начать поиск для полученного сообщения в суперклассе данного класса" (где "данный класс" представляет собой класс, в котором находится ключевое слово `super`).

Вы можете делать с `super` все, что хотите, но его основная цель, как я только что сказал, — доступ к перекрытой функциональности (обычно в самой перекрывающей функции, чтобы получить как перекрытую функциональность, так и некоторые дополнительные возможности).

Предположим, например, что мы определили класс NoisyDog, являющийся подклассом Dog. Когда передается сообщение bark, он лает дважды:

```
@implementation NoisyDog : Dog
- (NSString*) bark {
    return [NSString stringWithFormat: @"%@ %@",
                    [super bark], [super bark]];
}
@end
```

Этот код дважды вызывает реализацию метода bark из класса super; он объединяет две получающиеся строки в одну с пробелом посредине (используя метод stringWithFormat:) и возвращает ее. Поскольку в классе Dog метод bark возвращает @"Woof!", в классе NoisyDog метод bark вернет @"Woof! Woof!". Обратите внимание, что рекурсии при этом нет: метод bark из класса NoisyDog не вызывает сам себя.

Приятной особенностью этой архитектуры является то, что, отправив сообщение ключевому слову super, вместо жесткого кодирования @"Woof!" в методе bark класса NoisyDog мы обеспечиваем возможность сопровождения: если что-то изменится, результат метода bark класса NoisyDog изменится соответственно. Например, если позже метод bark класса Dog станет возвращать @"Arf!", то результатом метода bark класса NoisyDog станет @"Arf! Arf!", без каких бы то ни было дальнейших изменений с нашей стороны.

В реальном программировании с помощью каркаса Cocoa вам придется очень часто перекрывать его методы. Например, класс UIViewController, встроенный в каркас Cocoa, реализует метод viewDidAppear:, описанный в документации следующим образом:

```
- (void)viewDidAppear:(BOOL)animated
```

Документация гласит, что UIViewController является классом, для которого с большой вероятностью потребуется определить подкласс. Документация предполагает, что ваш подкласс класса UIViewController будет перекрывать упомянутый метод, и предупреждает, что если вы это делаете, то "должны вызвать super в некоторой точке вашей реализации". Фраза "вызвать super" — своего рода сокращение, означающее "передать super тот же вызов и те же аргументы, которые были отправлены вам".

Скажем, вы перекрываете метод viewDidAppear: в вашем подклассе UIViewController, который называется MyViewController. Ваша реализация может иметь следующий вид:

```
- (void) viewDidAppear: (BOOL) animated {
    [super viewDidAppear: animated];
    // ... Выполняете свои действия ...
}
```

В результате при вызове viewDidAppear: в экземпляре класса MyViewController мы выполняем как стандартные действия, которые выполняет суперкласс UIViewController в ответ на сообщение viewDidAppear:, так и собственные действия, относящиеся к нашему классу MyViewController. В этом конкретном примере мы даже не знаем, что именно делает класс UIViewController, да нас это и не интересует. Когда документация требует вызывать super при перекрытии — вызывайте super при перекрытии! Пренебрежение этим вызовом, когда его требует документация, может привести к некорректному поведению приложения и является распространенной ошибкой начинающих программистов.

Переменные экземпляра и методы доступа

В главе 3, я пояснял, что одной из главных причин существования экземпляров, а не просто классов, является то, что экземпляры могут иметь переменные экземпляра. Как вы помните, переменные экземпляра объявляются при определении класса, и в главе 4, я говорил, что эти объявления располагаются в фигурных скобках в начале раздела интерфейса, или, в современной версии языка Objective-C, в разделе реализации. Но каждый отдельный экземпляр класса хранит свои переменные экземпляра, объявленные в классе, отдельно; значение любой переменной экземпляра может быть установлено и получено, пока существует сам экземпляр класса.

Термин "переменная экземпляра" (instance variable) встречается так часто, что его часто сокращают до *ivar*.

Давайте напишем класс, использующий переменные экземпляров. Предположим, у нас есть класс `Dog` и мы хотим, чтобы каждый экземпляр класса `Dog` имел номер, который представляет собой число типа `int`. (Этот номер может соответствовать, например, номеру лицензии собаки.) В современной версии языка Objective-C мы, вероятно, объявили бы `number` в разделе реализации класса Dog:

```
@implementation Dog {
    int number;
}
// Здесь находятся реализации методов
@end
```

(Вы можете спросить, а почему я, например, не использую кличку собаки? Причина в том, что кличка была бы экземпляром `NSString`, который представляет собой объект. Переменные экземпляров, являющиеся указателями на объекты, вызывают дополнительные проблемы, которые я не хочу обсуждать в данный момент. Переменные экземпляров, представляющие собой простые типы данных C, таких проблем не вызывают. Мы вернемся к этому вопросу в главе 12.)

В классе его собственные переменные экземпляра являются глобальными по отношению ко всем методам экземпляра. Любой метод экземпляра класса `Dog` может просто использовать эту переменную `number`, как любую другую. Но код, с помощью которого это делается, может вас смутить: вы видите переменную по имени `number` и не понимаете, что это такое, потому что поблизости нет никакого объявления этой переменной. Поэтому я часто использую различные обозначения наподобие такого: `self->ivarName`. Оператор "стрелка", образованный знаками "минус" и "больше", называется оператором указателя на структуру в связи с его исходным применением в C (*K&R* 6.2).

Еще одно соглашение об именовании переменных экземпляра — их имена начинаются с символа подчеркивания: `_ivarName`. Таким образом, даже если вы не напишете `self->_ivarName`, одно имя `_ivarName` уже будет содержать подсказку о том, где может быть объявлена эта переменная.

Тот факт, что переменные экземпляров глобальны для всех методов экземпляров класса и что они сохраняются до тех пор, пока существует сам экземпляр, является вполне достаточной причиной для наличия переменных экземпляров. Рассматривайте переменные экземпляров как удобный и мощный способ совместного использования значений различными методами экземпляров одного и того же класса. Вам часто придется в одном методе экземпляра

присваивать значение переменной экземпляра с тем, чтобы позже другой метод мог им воспользоваться. Эта возможность становится особенно важной в мире каркаса Cocoa, управляемом событиями; см. главу 11.

Однако переменные экземпляров могут также служить средством общения экземпляра с другим экземпляром вне данного. При создании кода вашего класса и размышлениях о месте последнего в вашей программе вы можете захотеть позволить экземплярам других классов получать (или даже устанавливать) значение переменных у экземпляров данного класса. Однако по умолчанию переменные экземпляров являются *защищенными*; это значит, что другие классы, за исключением подклассов данного класса, не могут видеть переменные экземпляров данного класса. Если же переменные экземпляров объявлены в разделе реализации класса, то никакие другие классы, даже подклассы данного, не могут их видеть. Так что если где-то в другом месте я создаю экземпляр класса `Dog`, я не смогу получить доступ к переменной `number` этого экземпляра класса `Dog`. Это преднамеренно спроектированная особенность языка Objective-C; при желании ее можно обойти, но в общем случае вы не должны этого делать. Вместо этого следует предоставить *методы доступа* к переменным экземпляров.

Для именования этих методов имеется свое соглашение: они должны иметь имена `setXxx:` и `xxx`, где "`xxx`" совпадает у обоих имен (и может совпадать с именем переменной экземпляра). Например, наши методы доступа могут именоваться `setNumber:` и `number`.

Давайте запишем в разделе реализации класса `Dog` метод доступа `setNumber:`, который позволяет устанавливать значение переменной экземпляра `number`:

```
- (void) setNumber: (int) n {
    self->number = n;
}
```

Конечно, для того чтобы сделать метод `setNumber:` открытым для любого другого класса, который импортирует файл `Dog.h` интерфейса класса `Dog`, мы должны также объявить его в разделе интерфейса `Dog`:

```
@interface Dog : NSObject
- (void) setNumber: (int) n;
@end
```

Теперь можно в любом импортирующем файл `Dog.h` классе создать экземпляр `Dog` и присвоить этому экземпляру номер:

```
Dog* fido = [Dog new];
[fido setNumber: 42];
```

Теперь мы можем установить значение переменной экземпляра `number` класса `Dog`, но не можем получить его (извне этого экземпляра класса `Dog`). Для того чтобы исправить эту оплошность, напишем второй метод доступа, `number`, который позволяет получать значение переменной экземпляра `number`:

```
- (int) number {
    return self->number;
}
```

Мы вновь объявляем этот метод в разделе интерфейса `Dog`. Теперь в любом классе, который импортирует файл `Dog.h`, мы можем как устанавливать, так и получать значение переменной экземпляра `number` класса `Dog`:

```
Dog* fido = [Dog new];
[fido setNumber: 42];
int n = [fido number]; // Конечно же, n равно 42!
```

К счастью, современная версия языка Objective-C предоставляет механизм для автоматической генерации методов доступа (обсуждаемый в главе 12), так что вам не надо долго и утомительно создавать их вручную всякий раз, когда вы хотите сделать некоторую переменную экземпляра общедоступной. (Хотя, чтобы быть честным, должен сказать, что не вижу особых причин, по которым вы не должны пройти через эту скуку; до появления версии Objective-C 2.0 мы все делали это, так почему вы должны избежать этого? Пока что вы еще дети в программировании и должны пройти суровую школу и на своей шкуре почувствовать, что это такое — реальное программирование.)

В моем коде класс `Dog` имеет одновременно и метод `number`, и переменную экземпляра `number`. Этот факт не должен вас смущать. И он совершенно не смущает компилятор, потому что имя метода и имя переменной экземпляра используются в коде совершенно различными способами. Если компилятор может увидеть разницу, то вы и подавно. Тем не менее применение упомянутого мною выше соглашения об именовании переменных экземпляров устраняет любую возможную путаницу. Вспомните — мы начинаем имена наших переменных экземпляров с символа подчеркивания: `_number`, а не `number`. Следуя соглашению, нам надо также переписать (но не переименовать) методы доступа к переименованной переменной экземпляра:

```
@implementation Dog {
    int _number;
}
- (void) setNumber: (int) n {
    self->_number = n;
}
- (int) number {
    return self->_number;
}
@end
```

Все переменные экземпляра при создании экземпляра получают значение, представляющее собой ту или иную разновидность нуля (как часть вызова метода класса `alloc`). Это важно, потому что означает, что, в отличие от локальной автоматической переменной, простое объявление переменной экземпляра без указания начального значения не опасно — что хорошо, хотя бы потому, что такое объявление невозможно. Мы не можем сказать:

```
@implementation Dog {
    int _number = 42; // Нет, простите, это не Objective-C!
}
```

Однако по крайней мере мы знаем, что переменная `_number` начнет свое существование, имея значение, равное 0, а не некоторому опасно неопределенному значению. Точно так же экземпляр переменной типа `BOOL` начинает существование как `NO`, разновидность нулевого значения типа `BOOL`, а переменная экземпляра, которая представляет собой объект, получит значение `nil`. Если вы хотите получить ненулевое значение переменной экземпляра при создании экземпляра, напишите код, который присваивает ей это ненулевое значение. И не забывайте выполнять проверки на равенство переменных экземпляров нулю (или `nil`).

Кодирование ключ–значение

Язык Objective-C предоставляет средство для перевода строки в вызов метода доступа, которое называется *кодированием ключ–значение* (key–value coding). Такой перевод полезен, например, когда имя требуемого метода доступа не известно заранее, до времени выполнения.

Строка является ключом. Эквивалентом вызова метода доступа для получения значения переменной экземпляра в кодировании ключ–значение является вызов метода `valueForKey:`; эквивалентом вызова метода доступа для установки значения переменной экземпляра является вызов `setValue:forKey:`.

Таким образом, предположим, например, что мы хотим вызвать метод `number` экземпляра `fido`. Это можно сделать с помощью отправки сообщения `valueForKey:` экземпляру `fido` с ключом `@"number"`. Однако, несмотря на то, что метод `number` возвращает значение типа `int`, значение, возвращаемое методом `valueForKey:`, является объектом — в данном случае это объект класса `NSNumber`, эквивалентный числу (см. главу 10). Если нам нужен реальный экземпляр типа `int`, то можно воспользоваться методом экземпляра `intValue` класса `NSNumber`, который позволяет получить это число:

```
NSNumber* num = [fido valueForKey: @"number"];
int n = [num intValue];
```

Аналогично, чтобы использовать кодирование ключ–значение для вызова метода `setNumber:` экземпляра `fido`, мы должны воспользоваться кодом

```
NSNumber* num = [NSNumber numberWithInt:42];
[fido setValue: num forKey: @"number"];
```

Перед передачей числа 42 в качестве значения аргумента `setValue:forKey:` мы должны "завернуть" его в объект, в данном случае — в объект `NSNumber`. Начиная с версии компилятора LLVM 4.0 (Xcode 4.4) для этого имеется синтаксическое сокращение; так же как мы можем создавать `NSString`, помещая текст в директиву компилятора `@"..."`, мы можем создать и `NSNumber`, помещая числовое выражение в директиву компилятора `@(...)`; или, если числовое выражение является просто литералом, перед ним можно поставить символ `@`. Так что предыдущий пример может быть переписан следующим образом:

```
NSNumber* num = @42;
[fido setValue: num forKey: @"number"];
```

В реальном программировании вы, вероятно, предпочтете опустить промежуточную переменную `num` и запишете весь код как одно выражение:

```
[fido setValue: @42 forKey: @"number"];
```

В этих примерах нет никакого преимущества применения кодирования ключ–значение перед вызовом методов доступа. Предположим, что мы получили значение `@"number"` в переменной (скажем, как результат вызова метода). Пусть эта переменная имеет имя `something`. Тогда мы можем написать

```
id result = [fido valueForKey: something];
```

Таким образом, мы можем обратиться в разных ситуациях к разным методам доступа. Такая гибкость возможна благодаря тому, что Objective-C настолько динамический язык программирования, что отправляемое объекту сообщение оказывается не сформировано до реального выполнения программы.

Когда вы вызываете `valueForKey:` или `setValue:forKey:`, вызывается корректный метод доступа, если таковой имеется. Таким образом, когда в качестве ключа мы используем `@"number"`, вызываются методы `number` и `setNumber:` (если таковые существуют). Это одна из причин, по которой ваши методы доступа должны иметь корректные имена. С другой стороны, если метода доступа не существует, но есть переменная экземпляра с тем же именем, что и ключ, будет *выполнено непосредственное обращение к этой переменной* (даже если ее

имя начинается с подчеркивания)! Такое непосредственное обращение нарушает закрытость переменных экземпляров, так что имеется способ отключения этой возможности для определенных классов, если вы не хотите предоставления такого доступа. (Более подробно на эту тему мы поговорим в главе 12.)

Свойства

Свойство представляет собой синтаксическую возможность современного языка программирования Objective-C, разработанную в качестве альтернативы стандартному синтаксису методов доступа. В качестве синтаксического приема, облегчающего программирование, вы можете просто добавить имя свойства к ссылке на экземпляр с помощью записи с точкой. Получающееся в результате выражение можно использовать как слева от знака равенства (для вызова соответствующего метода доступа для установки значения), так и в любых других местах (для вызова соответствующего метода доступа для получения значения). Имя свойства по умолчанию использует соглашения по именованию методов доступа.

Я воспользуюсь в качестве примера классом `Dog`. Если класс `Dog` имеет открытый метод доступа для получения значения переменной экземпляра с именем `number` и открытый метод доступа для установки `setNumber:`, то класс `Dog` имеет также свойство `number`, которое может быть использовано слева от знака равенства или в любом ином месте. Это означает, что вместо

```
[fido setNumber: 42];
int n = [fido number];
```

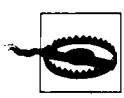
можно записать

```
fido.number = 42;
int n = fido.number;
```

Применение свойств совершенно не обязательно. Наличие свойства эквивалентно наличию соответствующих методов доступа к данным; использование свойства эквивалентно вызову метода доступа. Вызывать метод доступа можно с помощью любого синтаксиса. В случае класса Dog вы можете как вызывать методы доступа для установки и получения значения переменной экземпляра (`number` и `setNumber:`), так и воспользоваться свойством (`number`).

Для того чтобы использовать свойство в классе, которое им обладает, следует явно указывать ключевое слово `self`, например

```
self.number = 42;
```



Не путайте свойство с переменной экземпляра. Выражение наподобие `self->number = n`, или даже просто `number = n`, устанавливает значение переменной экземпляра непосредственно (и это возможно только в самом классе, поскольку переменные экземпляров по умолчанию являются защищенными).Выражение наподобие `fido.number` или `self.number` включает свойство и эквивалентно вызову метода доступа. Такой метод доступа может обращаться к переменной экземпляра, и эта переменная экземпляра может даже иметь то же имя, что и свойство, но это не означает, что они представляют собой одну и ту же сущность.

К свойствам мы вернемся в главе 12, где вы узнаете, что на самом деле это более мощная и интересная возможность, чем я рассказал здесь. Сейчас я говорю о свойствах просто по-

тому, что они широко применяются в каркасе Cocoa и поскольку вам очень часто придется сталкиваться с ними в документации.

Например, ранее в этой главе я вызывал метод экземпляра `setFrame:` класса `UIView` с экземпляром класса `UIButton`:

```
[b setFrame: CGRectMake(100,100,52,30)];
```

Я упоминал, что `setFrame:` не документирован в `UIButton`; вы должны обратиться к документации по `UIView`. Но в действительности такой метод не описан и там. Вот что говорит документация `UIView` по этому поводу.

`frame`

- Прямоугольник рамки, описывающий положение и размер представления в координатной системе родительского представления.
- **`@property(nonatomic) CGRect frame`**

В документации говорится о том, что у класса `UIView` есть свойство с именем `frame`. Приведенная выше строка является объявлением свойства. С точки зрения клиента класса `UIView` — в данном случае с моей точки зрения — объявление свойства является просто сокращенной записью, говорящей, что такое свойство существует. (Пока что не будем говорить о смысле ключевого слова `nonatomic`.) Но это то же самое, что сказать о наличии в классе `UIView` методов экземпляра `frame` и `setFrame:`; я могу использовать эти методы либо посредством свойства `frame` и записи с точкой, либо явно их вызывая. Таким образом я и узнал о существовании метода доступа `setFrame:`. В моем коде я вызвал `setFrame:` явно; но сейчас вы уже знаете, что я мог бы воспользоваться свойством `frame`:

```
b.frame = CGRectMake(100,100,52,30);
```

Язык Objective-C для свойств использует запись с точкой, а язык C использует такую запись для структур; эти записи могут быть объединены в цепочки. Так, например, свойство `frame` класса `UIView` является свойством, значение которого представляет собой структуру (`CGRect`); таком образом, можно записать `myView.frame.size.height`, где `frame` представляет собой свойство, возвращающее структуру, `size` является элементом этой структуры, а `height`, в свою очередь, элементом структуры `size`. У этого синтаксиса имеются ограничения; вы не можете, например, непосредственно установить значение `height` с помощью цепочки, начинающейся с `UIView`:

```
myView.frame.size.height = 36.0; // Ошибка "Expression is not assignable"
```

Вместо этого, если вы хотите изменить компонент свойства, являющегося структурой, вы должны получить значение свойства в переменную структуры, изменить ее значение и установить значение всего свойства, используя эту переменную:

```
CGRect f = myView.frame;
f.size.height = 0;
myView.frame = f;
```

Как написать инициализатор

Теперь, когда вы знаете о ключевых словах `self` и `super` и переменных экземпляра, мы можем вернуться к теме, которую я беспечно пропустил ранее. Я описал, как инициализировать новоиспеченные экземпляры путем вызова инициализаторов, и подчеркнул, что вы всегда должны делать это, но при этом ничего не сказал о том, как написать инициализатор

Войны из-за точки

Естественно, по поводу синтаксиса свойств и записи с точкой разгораются целые религиозные войны и крестовые походы. Одна сторона утверждает, что синтаксис свойств удобный и компактный, и делает язык Objective-C больше похожим на другие языки с подобной записью. Другая сторона возражает, что этот синтаксис слишком ограничен.

Например, противники синтаксиса свойств говорят, что класс `UIScrollView` имеет свойство `contentView`, но, устанавливая его, вы скорее всего захотите одновременно анимировать прокрутку, что можно сделать с помощью вызова `setContentView:animated:`. Это своего рода метод доступа, но он принимает два параметра. Выразить такое с помощью синтаксиса свойств не получится, так что вам придется вернуться к явному вызову метода. В результате синтаксис свойств не дает никакого выигрыша и скорее в состоянии ввести в заблуждение (вы можете просто забыть добавить анимацию).

Другим возражением против синтаксиса свойств является то, что компилятор ограничивает его использование. Например, можно использовать формальный вызов метода для отправки сообщения `number` экземпляру `Dog`, типизированному как `id`, но добавить свойство `number` с записью с помощью точки в такой экземпляр невозможно.

Еще одной проблемой является то, что возможно ошибочно использовать синтаксис свойств для вызова метода, который, строго говоря, методом получения значения переменной экземпляра не является. Например, метод `lastObject` класса `NSArray` — это просто метод; его нет в списке свойств. Тем не менее программисты часто пишут `myArray.lastObject` только потому, что написать это проще и быстрее, чем `[myArray lastObject]`, и еще потому, что этот способ работает. Но опять же так нельзя поступить с методами, которые не возвращают значение или принимают какие-либо параметры. Соответствующий метод должен выглядеть как метод доступа для получения значения, даже если в действительности он им не является. Такое неправомерное использование синтаксиса свойств выглядит отвратительно, но одновременно столь заманчиво, что даже встречается в примерах исходных текстов Apple.

своего собственного класса. Создавая свой класс, вы, вероятно, захотите обеспечить удобный инициализатор, функциональность которого выходит за рамки функциональности наследуемых инициализаторов. Чаще всего ваша цель будет заключаться в том, чтобы принять некоторые параметры и использовать их для задания начальных значений некоторых переменных экземпляра.

Пусть, например, в случае класса `Dog` с переменной экземпляра `number` мы не хотим создания экземпляров класса `Dog` без конкретного номера; каждый `Dog` должен иметь таковой. Так что первоочередной задачей при создании экземпляра класса `Dog` является присваивание значения его переменной экземпляра `number`. Инициализатор озвучивает это правило и обеспечивает его выполнение — особенно если это назначенный инициализатор класса. Так что давайте считать, что этот инициализатор — назначенный.

Кроме того, будем считать, что номер собаки не может изменяться. Когда экземпляр класса Dog начинает свое существование, он получает значение номера, который остается неизменным до конца жизненного цикла экземпляра.

Поэтому удалим метод setNumber: и его объявление, тем самым уничтожив возможность другим классам вмешаться и изменить значение переменной экземпляра number после инициализации. Вместо этого значение переменной экземпляра number устанавливается при инициализации с помощью метода, объявленного как

```
- (id) initWithNumber: (int) n;
```

Возвращаемое значение типизировано как id, а не как указатель на объект класса Dog, несмотря на тот факт, что в действительности возвращается объект класса Dog. Это соглашение, которому мы должны подчиняться. Имя метода также соответствует соглашению; как вы знаете, начало метода init говорит о том, что он является инициализатором.

Теперь я хочу показать вам реальный код инициализатора (пример 5.3). Большая часть этого кода обычна — так сказать, обязательная программа инициализатора. Вы не должны спрашивать, зачем — просто сделайте это. Я опишу смысл кода, но не собираюсь оправдывать все части используемого соглашения.

Пример 5.3. Схема инициализатора

```
- (id) initWithNumber: (int) n {
    self = [super init];❶ ❷
    if (self) {
        self->_number = n;❸
    }
    return self;❹
}
```

Частями соглашения являются следующие.

❶ Мы отправляем сообщение об инициализации, вызывая назначенный инициализатор. Если метод, который мы создаем, является назначенным инициализатором нашего класса, это сообщение отправляется объекту super и вызывает назначенный инициализатор суперкласса. В противном случае оно отправляется объекту self и вызывает назначенный инициализатор данного класса либо другой инициализатор класса, который вызывает назначенный инициализатор. В данном случае мы пишем назначенный инициализатор, а назначенным инициализатором суперкласса является init.

❷ Мы сохраняем результат отправки сообщения инициализации в экземпляре self. Для начинающих программистов (и не только для них) оказывается сюрпризом, что можно что-то присвоить self и что это действие имеет смысл. Однако присваивание экземпляра self вполне допустимо (в силу того, как "за кулисами" работает система сообщений языка Objective-C), и это присваивание имеет смысл, поскольку в некоторых случаях экземпляр, возвращаемый отправленным нами сообщением инициализации, может не совпадать с тем экземпляром self, с которым мы начинаем работу.

❸ Если значение self не равно nil, мы инициализируем все переменные экземпляра, о которых нам надо позаботиться. Это та часть кода, которую вы будете писать по своему усмотрению; остальное будет соответствовать шаблону. Заметьте, что я не использую никакие методы доступа для установки значений переменных экземпляров (как не использую и свойства); в инициализации переменные экземпляра не наследуются от суперкласса, вы должны выполнять присваивание непосредственно переменной экземпляра.

(Ранее я упоминал, что переменные экземпляров начинают свое существование с нулевыми значениями, и если это вас не устраивает, то вы можете присвоить им необходимые значения. Ясно, что одним из способов сделать это является написание назначенного инициализатора. Если же нулевые значения по умолчанию вас устраивают, вы можете не беспокоиться о них в назначенном инициализаторе.)

❶ Мы возвращаем экземпляр `self`.

Но это не конец. Вспомните, что класс, который определяет назначенный инициализатор, должен также перекрыть унаследованный назначенный инициализатор (в данном случае `init`). И вы можете увидеть, почему: если этого не сделать, то кто-то сможет сказать `[[Dog alloc] init]` (или `[Dog new]`) и создать собаку без номера — сделать именно то, от чего призван защищать наш инициализатор. Просто в качестве примера я заставлю перекрытый `init` присваивать отрицательное значение переменной экземпляра `number` в качестве сигнала о проблемах. Обратите внимание, что мы все еще подчиняемся правилам: этот инициализатор не является назначенным, так что он вызывает назначенный инициализатор данного класса:

```
- (id) init {
    return [self initWithNumber: -9999];
}
```

Просто чтобы довести дело до конца, вот код, демонстрирующий, как мы теперь можем создать экземпляр класса `Dog`:

```
Dog* fido = [[Dog alloc] initWithNumber:42];
int n = fido.number; // n равно 42; наша инициализация работает!
```

Несмотря на то что инициализатор возвращает `id`, а `is` является универсальным донором, компилятор предупреждает нас в случае присваивания результата некорректно типизированнной переменной:

```
NSString* s = [[Dog alloc] initWithNumber:42];
// Предупреждение компилятора
```

Эта магия выполняется современным компилятором LLVM, который из того, что имя этого метода начинается с `init`, выводит, что это инициализатор, и эффективно заменяет `id` ключевым словом `instancetype`, указывающим, что значение, возвращаемое этим методом, должно быть того же типа, что и его получатель (в данном случае — `Dog`).

Ссылки на экземпляры

Эта глава в основном сконцентрирована на механике создания экземпляра. Однако зачастую ваш код будет связан не с созданием нового экземпляра, а со ссылкой на экземпляр, который уже существует. На экземпляр можно ссылаться либо с использованием имени ссылки на объект, которой он был ранее присвоен (это может быть переменная экземпляра), либо получая его как результат вызова метода — и никак иначе. Если у вас еще нет именованной ссылки на некоторый экземпляр (в виде переменной или переменной экземпляра), то нужный экземпляр может вернуть вызов метода. Например, вот как вы запрашиваете последний элемент массива (`NSArray`):

```
id myThing = [myArray lastObject];
```

Массив myArray типа NSArray не *создает* объект, который передает вам. Этот объект уже существует; myArray просто содержал указатель на него. Теперь он поделился этим объектом с вами, только и всего.

Аналогично многие классы могут обходиться одним создаваемым объектом. Например, ваше приложение имеет ровно один экземпляр класса UIApplication (мы называем такой экземпляр синглтоном); для доступа к нему надо отправить сообщение sharedApplication классу UIApplication:

```
UIApplication* theApp = [UIApplication sharedApplication];
```

Экземпляр этого синглтона существует еще до обращения к нему; фактически он существует еще до начала выполнения вашего кода. Вам не надо заботиться о его создании; все, что вам надо знать, — что вы можете сохранить его, когда захотите. Несколько подробнее о глобально доступных синглтонах речь пойдет в главе 13.

В обоих примерах код решает две задачи: найти способ указать уже существующий экземпляр и присвоить этот существующий экземпляр переменной, тем самым создавая (не новый экземпляр, но) новое имя, с помощью которого можно обращаться к уже существующему экземпляру. Это две стороны одной медали — ссылки на экземпляр.

Задача ссылки на экземпляр — это то, с чем вы постоянно имеете дело при программировании для iOS. Этот вопрос может показаться тривиальным, но на самом деле это не так. В действительности задача ссылки на уже существующий экземпляр является настолько важной, что я посвящаю ей большую часть главы 13. Экземпляр не имеет смысла, если к нему нельзя обратиться. Если вы не можете ссылаться на экземпляр, ему нельзя отправить сообщение. Если вы не можете ссылаться на экземпляр, его нельзя использовать в качестве аргумента в вызове метода. Если вы не можете ссылаться на экземпляр, нельзя ни получить информацию о нем, ни сделать с ним что-нибудь. Довольно часто бывают ситуации, когда вы знаете, что экземпляр существует, но, чтобы обратиться к нему, приходится попотеть. Возможно, вам даже придется переписать или заново спроектировать всю вашу программу, меняя порядок событий в программе, чтобы иметь возможность получить ссылку на конкретный экземпляр в тот момент, когда она вам понадобится.

Кроме того, после того как вы *получите* ссылку на желаемый экземпляр, может понадобиться ее *сохранить*. В случае синглтона, такого, как [UIApplication sharedApplication], это не является проблемой. Существует только один разделяемый экземпляр приложения, и класс UIApplication будет рад предоставить его вам снова и снова, всякий раз, когда вы попросите его об этом. Таким образом, его можно хранить, просто используя в соответствующих местах вашей программы выражение [UIApplication sharedApplication], и вы, вероятно, так и будете поступать. Присвоение разделяемого экземпляра приложения своей переменной, как, например, переменной theApp в предыдущем примере, — не более чем вопрос удобства. Это короткое простое имя для того же самого экземпляра.

Но все совсем не так в случае экземпляра [myArray lastObject]. Здесь мы уже имеем ссылку на массив (myArray) и теперь озабочены получением ссылки на один из его элементов, а именно на последний элемент. Экземпляр, который в настоящее время функционирует в качестве последнего элемента массива myArray, может не всегда быть последним элементом myArray. Он даже не всегда может быть элементом myArray. На самом деле массив myArray может сам прекратить существование. Если конкретный экземпляр, на который в настоящее время мы можем сослаться как [myArray lastObject], будет иметь важное значение для нас в будущем, нам может потребоваться принять меры для поддержания некоторых других ссылок на него, которые не зависят от статуса или состояния некоторого массива.

Это основная причина для присваивания экземпляра, возвращаемого вызовом `[myArray lastObject]`, переменной `myThing`. Теперь, независимо от того, что происходит в массиве, имя `myThing` будет продолжать ссылаться на указанный экземпляр.

В предыдущем коде `myThing` — просто локальная автоматическая переменная; эта переменная выйдет из области видимости, когда та закончится (например, когда завершится выполнение текущего метода), и возможность сослаться на этот экземпляр будет потеряна. Что мы можем поделать с этим? Все это происходит в пределах кода некоторого класса, выполняемого в рамках некоторого экземпляра; скажем, это экземпляр класса `Dog`. Если нам требуется экземпляр `myThing`, продолжительность жизни которого соответствует времени жизни экземпляра `Dog`, мы можем присвоить это значение переменной экземпляра. Такой поступок имеет дополнительное преимущество: после этого к нему смогут обращаться другие методы экземпляра данного экземпляра `Dog`! Более того, если имеется метод доступа, другие объекты, имеющие ссылку на наш экземпляр `Dog`, будут иметь ссылку и на экземпляр `myThing` также! Так мы можем управлять временем жизни экземпляра, обеспечивая при этом возможность получить ссылку на него в нужный нам момент в будущем. Это настолько важная возможность, что зачастую именно она оказывается главной причиной для наличия в классе переменных экземпляров.

ЧАСТЬ II

Интегрированная среда разработки

К данному моменту вы, несомненно, страстно желаете вскочить и начать писать приложение. Чтобы сделать это, вам необходимо хорошо освоить инструменты, которые вы будете использовать. Тело и душа этих инструментов — среда Xcode. В этой части мы исследуем интегрированную среду разработки Xcode, в которой вы будете программировать приложения для системы iOS. Xcode — большая программа и разработка приложения подразумевают координирование большого количества ее компонентов; эта часть книги поможет вам изучить Xcode. По ходу мы создадим простое рабочее приложение и дадим практические рекомендации.

- Глава 6 содержит обзор среды Xcode. В ней описана архитектура проекта, т.е. коллекции файлов, из которых генерируется приложение.
- Глава 7 посвящена nib-файлам. Nib-файл — это файл, содержащий внешнее представление вашего интерфейса. Понимать смысл nib-файлов — т.е. знать, как они работают и как связаны с вашим кодом, — крайне важно для правильного использования среды Xcode и разработки любого приложения.
- В главе 8 мы обсудим документацию среды Xcode и другие источники информации об интерфейсе прикладного программирования.
- В главе 9 объясняется, как редактировать, тестировать и отлаживать код, и показаны этапы вплоть до размещения вашего приложения в интернет-магазине App Store. Вероятно, сначала вам захочется поскорее пролистать эту главу, чтобы впоследствии, после разработки реального приложения, вернуться к ней за рекомендациями.

ГЛАВА 6

Анатомия проекта Xcode

Xcode — это приложение, которое используется для разработки приложений в системе iOS. Проект Xcode — это исходный код приложения; он представляет собой набор файлов и параметров настройки для создания приложения. Чтобы создать, разработать и поддерживать приложение, вы должны знать, как манипулировать проектом Xcode и перемещаться по нему. Таким образом, вы должны знать основные факты о среде Xcode, о природе и структуре проектов Xcode и о том, как Xcode показывает их вам. Именно это является предметом настоящей главы.

Термин "Xcode" имеет два значения. С одной стороны, это название приложения, в котором вы редактируете и собираете свое приложение. С другой стороны, это название набора утилит, которые это приложение сопровождают. Например, во втором смысле утилиты Instruments и Simulator являются частью среды Xcode. Эта многозначность обычно не должна вызывать затруднений.

Xcode — мощная, сложная и чрезвычайно большая программа. Мой подход к описанию среды Xcode состоит в том, чтобы предложить вам преднамеренно сузить свой кругозор: если вы не понимаете что-то, не беспокойтесь об этом — даже не смотрите на это и не касайтесь этого, потому что вы можете изменить что-то важное. Наш обзор среды Xcode позволит проложить безопасный, узкий и прямой путь к цели, сосредоточившись на тех аспектах среды Xcode, которые важно понять сразу, решительно игнорируя все остальное.

Полную информацию можно получить из документации компании Apple (выберите команду Help⇨Xcode User Guide); сначала она может показаться ошеломляющей, но то, что вы хотите узнать, вероятно, находится где-то в этом справочнике. Описанию и объяснению среды Xcode посвящены целые книги.

Структура процесса инсталляции программы Xcode изменилась с версии Xcode 4.3. Папка `Developer` в Xcode 4.2 и более ранних версиях была папкой инсталляции верхнего уровня, в которой содержалась программа Xcode и много других файлов и папок. В Xcode 4.3 и более поздних версиях папка `Developer` находится в комплекте `Xcode.app/Contents/Developer`.

Новый проект

Еще до того, как вы напишете первую строчку кода, проект Xcode уже имеет довольно богатое содержание. Для того чтобы увидеть это, создадим новый, совсем "пустой" проект; вы убедитесь, что на самом деле он совсем не пустой.

1. Запустите программу Xcode и выберите команду File⇨New⇨Project.
2. На экране появится диалоговое окно Choose a template. Шаблон — это изначальный набор файлов и настроек. Выбирая шаблон, вы по существу выбираете существующую папку, заполненную файлами; как правило. это одна из папок, вложенных в папку `Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/Library/Xcode/Templates/Project Templates/Application`. Для того чтобы ваш проект был создан, эта папка шаблона будет скопирована, а некоторые значения будут заранее заданы.

   В данном случае слева на панели IOS (не OS X!) выберите команду Application. Затем на правой панели выберите пиктограмму Single View Application и щелкните на кнопке Next.
3. Программа попросит вас ввести имя своего проекта (Product Name). Назовем наш проект `Empty Window`.

   Создавая реальный проект, следует хорошо продумать его название, потому что вам придется с ним тесно работать. Поскольку программа Xcode копирует папку шаблона, она будет использовать имя проекта в нескольких местах, в частности, в именах файлов и других настройках, например в имени приложения. Таким образом, имя, которое вы введете на этом этапе, вы будете встречать во всем вашем проекте. Впрочем, это имя не задается раз и навсегда. Существует отдельная настройка, позволяющая изменить имя приложения в любой момент. Процедуру изменения имени я опишу в конце главы.

   В имени проекта удобно использовать пробелы. Пробелы допускаются в именах папок, проектов, приложений и разнообразных файлов, которые программа Xcode будет генерировать автоматически; впрочем, в некоторых местах пробелы могут вызвать проблемы (например, в идентификаторах комплектов, которые обсуждаются в следующем разделе), поэтому пробелы в имени, которое вы введете в поле Product Name, будут преобразованы в дефисы.
4. Обратите внимание на поле Company Identifier. Когда вы впервые создаете проект, это поле является пустым, и вы должны его заполнить. Это делается для того, чтобы создать уникальную строку, идентифицирующую вас или вашу компанию. По соглашению идентификатор компании должен начинаться символами `com.`, за которыми следует строка (возможно, с несколькими точками), которая вряд ли будет использоваться кем-то другим. Например, я использую строку `com.neuburg.matt`. Каждое приложение, инсталлированное на устройстве или посланное в интернет-магазин App Store, должно иметь уникальный идентификатор комплекта. Идентификатор комплекта вашего приложения, который выводится серым цветом ниже идентификатора компании, будет состоять из идентификатора компании и имени проекта; если вы зададите уникальное имя своего проекта, то идентификатор комплекта однозначно идентифицирует и его, и приложение (впоследствии вы можете изменить идентификатор комплекта вручную).
5. Выберите в меню Devices команду iPhone. (Пока пропустите поле Class Prefix; оно должно оставаться пустым, а его значение по умолчанию "`XYZ`" должно подсвечиваться серым цветом. Его предназначение описано в разделе "Глобальное пространство имен".) Щелкните на кнопке Next.

6. Теперь укажите программе Xcode, как конструировать ваш проект. По существу, она будет копировать папку `Single View Application.xctemplate` из папки `Project Templates/Application`, о которой я уже говорил. Однако вы должны указать, куда именно следует копировать папку шаблона. По этой причине программа Xcode откроет диалоговое окно Save. Вы должны указать место создаваемой папки — в ней будет храниться проект.

   Папка проекта может находиться в любом месте, причем после ее создания ее можно переместить. Обычно я создаю новые проекты на рабочем столе.

7. Программа Xcode предлагает также гит-репозиторий для вашего проекта. В реальной жизни это может быть очень удобным (см. главу 9), но пока этот флажок следует сбросить. Щелкните на кнопке Create.

8. На диске будет создана папка проекта Empty Window (в нашем случае она будет создана на рабочем столе), и в среде Xcode будет открыто окно проекта Empty Window.

Созданный нами проект является рабочим; он действительно конструирует приложение `Empty Window` для системы iOS. Для того чтобы убедиться в этом, убедитесь, что схема и предназначение в инструментальной панели окна проекта перечислены как Empty Window⇨iPhone Retina (3.5-inch). (Схема и предназначение на самом деле задаются в раскрывающемся списке, поэтому для их изменения можно щелкнуть на них позднее.) Выберите команду Product⇨Run. После некоторой задержки приложение iOS Simulator откроет и выведет на экран ваше приложение — пустой белый экран.

Скомпоновать проект — значит скомпилировать его код и собрать скомпилированный код вместе с различными ресурсами в реальное приложение. Обычно, если вы хотите узнать, правильно ли компилируется ваш код, вы компонуете проект (Product⇨Build). В версии Xcode 5 можно компилировать отдельный файл (Product⇨Perform Action⇨Compile [Filename]). Запустить проект — значит запустить скомпонованное приложение с помощью утилиты Simulator или на присоединенном устройстве. Если вы хотите проверить, правильно ли работает ваш код, необходимо выполнить проект (Product⇨Run), который автоматически компонуется при необходимости.

Окно проекта

Проект Xcode содержит множество информации о том, какие файлы образуют проект и как они используются при создании приложения.

- Исходные файлы (ваш код), которые должны компилироваться.
- Все файлы `.storyboard` или `.xibs`, графически выражающие объекты интерфейса, которые должны инициализироваться при запуске приложения.
- Все ресурсы, такие как пиктограммы, изображения или звуковые файлы, являющиеся частью приложения.
- Все настройки (инструкции компилятору, редактору связей и т.д.), которые должны учитываться при компоновке приложения.
- Все каркасы, необходимые для выполнения кода.

Вся эта информация содержится в отдельном окне проекта Xcode, которое открывает доступ к ней и дает возможность редактировать текст программы и перемещаться по нему, а также выводит информацию о выполнении таких процедур, как сборка или отладка приложения. Это окно содержит много информации и обладает массой возможностей! Окно проекта — мощный и сложный инструмент; для того чтобы научиться перемещаться по нему и понимать все процедуры, требуется время. Предлагаю остановиться на время и исследовать это окно, чтобы посмотреть, как оно сконструировано.

Окно проекта состоит из четырех частей (рис. 6.1):

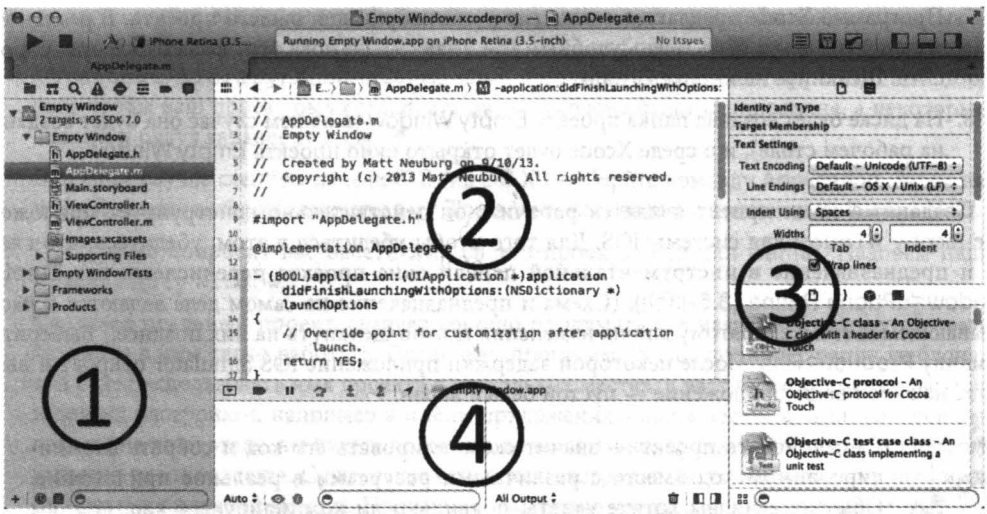


Рис. 6.1. Окно проекта

1. Слева расположена панель навигатора. Для того чтобы показать и скрыть эту панель, выполните команды View⇨Navigators⇨Show/Hide Navigator (или нажмите клавиши <Command+0>) или щелкните на первой кнопке View, считая с правого конца инструментальной панели.
2. Посередине расположена панель редактора (или просто "редактор"). Это основная область окна проекта. Окно проекта почти всегда содержит хотя бы одну панель редактора и может отображать несколько панелей редактора одновременно.
3. Справа расположена панель утилит. Для того чтобы показать и скрыть эту панель, выполните команды View⇨Utilities⇨Show/Hide Utilities (или нажмите клавиши <Command+Option+0>) или щелкните на третьей кнопке View, считая с правого конца инструментальной панели.
4. Внизу расположена панель отладки. Для того чтобы показать или скрыть эту панель, выполните команды View⇨Show/Hide Debug Area (или нажмите клавиши <Command+Shift+Y>) или щелкните на второй кнопке View, считая с правого конца инструментальной панели.

Все комбинации клавиш в среде Xcode можно настроить самостоятельно; см. панель Key Bindings в окне Preferences. Комбинации клавиш, используемые в книге, заданы по умолчанию.

Панель навигатора

Панель навигатора представляет собой столбец информации, расположенный в левой части окна проекта. Помимо прочего, она является основным механизмом контроля того, что вы видите в главной области окна проекта (поле редактора). Важный шаблон использования среды Xcode выглядит следующим образом: вы что-нибудь выбираете на панели навигатора, и это отображается в поле редактора.

Видимостью панели навигатора можно управлять (с помощью команд View⇨Navigators⇨Hide/Show Navigator или комбинации клавиш <Command+0>); например, если вы хотите использовать панель навигатора для того, чтобы увидеть элемент или поработать с ним в поле редактора, временно скройте панель навигатора, чтобы сделать размеры экрана максимальными (особенно на маленьком дисплее). Ширину панели навигатора можно изменить, перетаскивая вертикальную линию на ее правом крае.

Панель навигатора может отображать восемь разных наборов информации; таким образом, на самом деле существует восемь навигаторов. Они представлены восемью пиктограммами, расположенными вдоль ее верхнего края; для переключения этих наборов можно использовать пиктограммы или комбинации клавиш (<Command+1>, <Command+2> и т.д.). Вы скоро привыкнете переключать навигатор по своему желанию, а нажатие соответствующих комбинаций клавиш станет рефлексом. Если панель навигатора скрыта, то нажатие комбинации клавиш одновременно открывает панель навигатора и выполняет переключение в этот навигатор.

В зависимости от настроек предпочтений, выбранных на панели поведения, навигатор может открываться автоматически, когда вы выполняете определенное действие. Например, по умолчанию, когда во время сборки генерируются предупреждения или сообщения об ошибках, появляется навигатор проблем. Это автоматическое поведение не вызывает проблем, потому что оно является естественным, а если вы так не думаете, то можете изменить его; кроме того, в любом момент вы можете легко переключиться на другой навигатор.

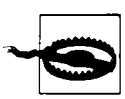
Приступим к экспериментам с разными навигаторами.

Навигатор проекта (<Command+1>)

Щелкните здесь, чтобы перемещаться по файлам, образующим ваш проект. Например, щелкните на файле `AppDelegate.m` в папке `Empty Window` (эти образования, напоминающие папку, называются группами), чтобы открыть его исходный код в поле редактирования (рис. 6.1).

На верхнем уровне навигатора проекта рядом с голубой пиктограммой Xcode располагается сам проект Empty Window; щелкните на ней, чтобы увидеть настройки, связанные с вашим проектом и целевыми устройствами. Не изменяйте никаких настроек, смысла которых не знаете! Позднее я все вам объясню.

Панель фильтров, расположенная внизу навигатора проекта, позволяет ограничить виды отображаемых файлов; если файлов слишком много, очень удобно быстро находить файл с известным именем. Например, попытайтесь набрать строку "`delegate`" в поле поиска на панели фильтров. Не забудьте удалить ваши фильтры, когда закончите экспериментировать.



Если вы установили фильтр навигатора, он будет действовать, пока не будет удален, даже если вы закроете проект! Пользователи часто делают распространенную ошибку: устанавливают фильтр навигатора, забывают об этом, не

обращают внимания на панель фильтров (потому что они чаще всего смотрят на сам навигатор, а не на панель фильтров, расположенную внизу) и спрашивают: "Эй, а куда делись все мои файлы?"

Рис. 6.2. Навигатор проекта

Навигатор символов (<Command+2>)

Символ — это имя, обычно имя класса или метода. В зависимости от того, какая из трех пиктограмм выделена на панели фильтров внизу панели, вы можете увидеть стандартные символы Cocoa или символы, определенные в вашем проекте. Первая возможность может быть полезной формой документирования; вторая возможность облегчает навигацию по коду. Например, выделив первые две пиктограммы на панели фильтров (т.е. первые две пиктограммы подсвечены голубым цветом, а третья остается темной), вы увидите, насколько быстро можно найти определение метода `AppDelegate's applicationDidBecomeActive:`.

Попробуйте выделить пиктограммы на панели фильтров разными способами, чтобы увидеть, как изменяется содержимое навигатора символов. Наберите строку в поле поиска на панели фильтров, чтобы ограничить символы, отображаемые в навигаторе символов; Например, попробуйте набрать строку "`active`" в поле поиска и посмотрите, что получится.

Навигатор поиска (<Command+3>)

Это мощное средство для глобального поиска текста в вашем проекте и даже в заголовочных файлах каркасов Cocoa. Его можно вызвать с помощью команды Find⇨Find in Project (<Command+Shift+F>). Слова, которые выводятся над окном поиска, описывают опции, которые доступны в данный момент; Они образуют раскрывающийся список, так что опции можно изменять, щелкая на них. Попытайтесь найти слово "`delegate`" (рис. 6.3). Для того чтобы перейти в место кода, содержащее данное слово, щелкните на результате поиска.

Искомые слова можно вводить в другом поисковом окне, расположенном на панели фильтров в нижней части экрана. (Пока я прекращаю упоминать панель фильтров, но каждый навигатор имеет его в том или ином виде.)

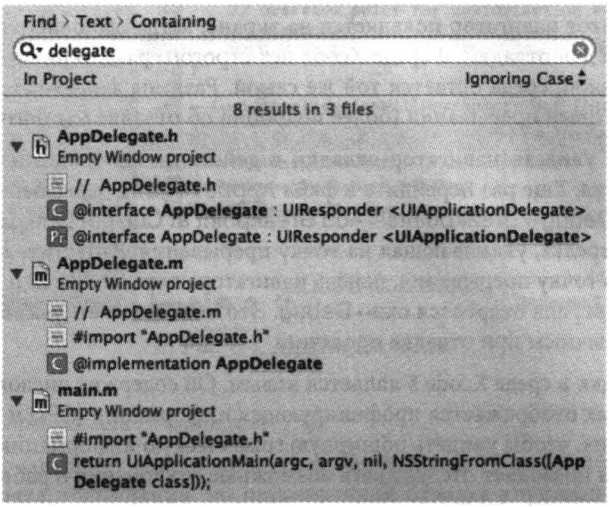


Рис. 6.3. Навигатор поиска

Навигатор проблем (<Command+4>)

Этот навигатор нужен в основном тогда, когда ваша программа имеет проблемы. Это не психологические проблемы. Этим термином в среде Xcode называют предупреждения и сообщения об ошибках, возникших во время создания вашего проекта.

Для того чтобы увидеть навигатор проблем в действии, внесите в вашу программу преднамеренную ошибку. Например, перейдите (как вы уже знаете, это можно сделать как минимум тремя способами) в файл `AppDelegate.m` и в пустой строке после последнего комментария в верхней части файла, над строкой `#import`, наберите слово `howdy`. Соберите проект (нажав клавиши <Command+B>). Навигатор проблем выведет на экран несколько сообщений об ошибках, показывая, что компилятор не может распознать это недопустимое слово, находящееся в недопустимом месте. Щелкните на сообщении, чтобы увидеть проблему в файле. В вашем коде проблема может выделяться с помощью "шариков", расположенных справа от строк, содержащих проблему; если навигатор поиска вам мешает, переключите его режим видимости с помощью команды Editor⇨Issues⇨Hide/Show All Issues (клавиши <Command+Control+M>).

Теперь выделите слово `howdy` и удалите его; соберите проект снова, и ваша проблема исчезнет. Если бы в реальной жизни все было так же просто!

Навигатор тестирования (<Command+5>)

Этот навигатор является новшеством в среде Xcode 5. В нем перечисляются тестовые файлы и отдельные тестовые методы. Он позволяет выполнять тесты и проверять их результаты. Тест — это код, который не является частью вашего приложения; он выполняет часть кода вашего приложения, чтобы оценить, работает ли она так, как ожидалось.

По умолчанию проект в новой среде Xcode 5 содержит один тестовый файл, содержащий тестовый метод. Более подробно мы поговорим об этом в главе 9.

Навигатор отладки (<Command+6>)

По умолчанию этот навигатор появляется на экране, когда выполнение вашего кода приостанавливается для отладки. В среде Xcode нет строгого разделения между выполнением кода и его отладкой; среда остается той же самой. Разница заключается лишь в наличии или отсутствии точек прерывания (более подробно об отладке мы поговорим в главе 9).

Для того чтобы увидеть навигатор отладки в действии, необходимо установить в коде точки прерывания. Еще раз перейдите в файл `AppDelegate.m`, выберите строку `return YES` и команду Debug⇨Breakpoints⇨Add Breakpoint at Current Line. На этой строке появится голубая стрелка, указывающая на точку прерывания. Запустите проект. По умолчанию, обнаружив точку прерывания, панель навигатора переключится на панель отладки, и в нижней части экрана откроется окно Debug. Это общий макет (рис. 6.4), который скоро станет вам привычным при отладке проектов.

Навигатор отладки в среде Xcode 5 является новым. Он содержит числовую и графическую панели, в которых отображается профилирующая информация (CPU и Memory); щелкните на одной из них, чтобы увидеть обширную графическую информацию в окне редактора. Эта информация позволяет отслеживать возможные отклонения в работе приложения, не вызывая утилиту Instruments (см. главу 9). Для того чтобы изменить режим видимости профилирующей информации в верхней части навигатора отладки, щелкните на пиктограмме с изображением градусника (справа от имени процесса).

Навигатор отладки также выводит на экран стек вызовов, содержащий имена вложенных методов, в которых возникла пауза; как и следовало ожидать, для перехода в метод достаточно щелкнуть на его имени. Список вызовов можно сократить или удлинить, перемещая ползунок в нижней части панели навигатора. Вторая пиктограмма справа от имени процесса позволяет переключаться между выводом потоков и выводом очереди.

Панель Debug можно делать видимой или невидимой (View⇨Debug Area⇨Hide/Show Debug Area или <Command-Shift-Y>). Она состоит из двух вложенных панелей — списка переменных и консоли. Их можно скрыть или сделать видимыми, щелкая на кнопках, расположенных в нижнем правом углу панели. Консоль также можно открыть, выбрав команду View⇨Debug Area⇨Activate Console.

Список переменных (слева)

Этот список содержит переменные, находящиеся в области видимости выбранного метода в стеке вызовов.

Консоль (справа)

Здесь отладчик выводит текстовые сообщения, читая которые вы можете узнать об исключениях, возникших в ходе выполнения вашего приложения. Кроме того, вы можете заставить свой код выводить на консоль сообщения, описывающие ход выполнения и поведение приложения. Такие сообщения важны, поэтому следует внимательно следить за консолью во время выполнения приложения. Консоль можно также использовать для ввода команд для отладчика. Часто этот способ изучения значений оказывается удобнее, чем список переменных.

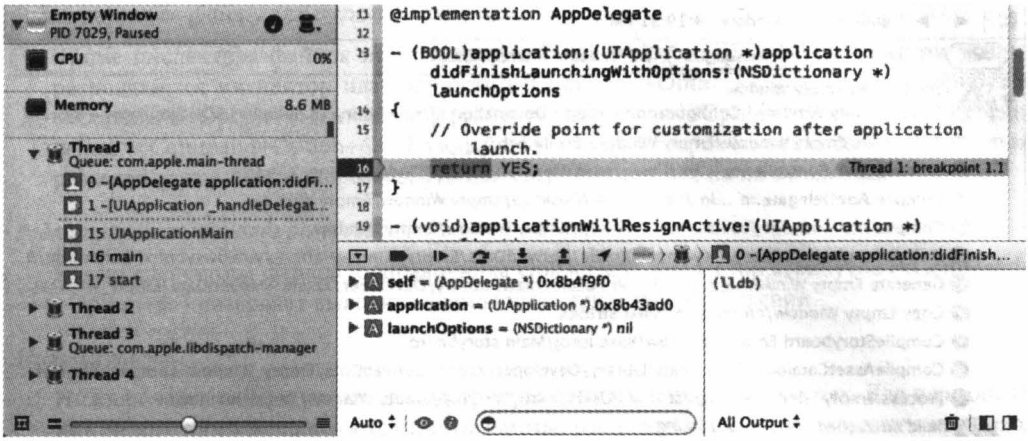


Рис. 6.4. Макет отладки

Навигатор точек прерывания (<Command+7>)

Этот навигатор содержит список всех точек прерывания. В данный момент установлена только одна точка прерывания, но при активной отладке крупного проекта с многими точками прерывания этот навигатор оказывается незаменимым помощником. Кроме того, здесь можно создавать специальные точки прерывания (например, символические точки прерывания). В целом этот навигатор является центром управления существующими точками прерывания. Мы вернемся к нему в главе 9.

Навигатор журналов (<Command+8>)

Этот навигатор содержит список недавно выполненных действий, например команды сборки или запуска (отладки) приложения. Щелкните на листинге, чтобы увидеть (в окне редактора) файл журнала, сгенерированный при выполнении вами данного действия. Файл журнала может содержать информацию, которую невозможно получить никаким другим способом. Кроме того, он позволяет разбираться в консольных сообщениях, которые выводились ранее ("Какое исключение возникло во время последней отладки?").

Например, щелкнув на листинге успешной сборки и выбрав команды вывода всех сообщений All и All Messages с помощью фильтров, расположенных в верхней части журнала, вы увидите этапы выполнения сборки (рис. 6.5). Для вывода всей информации об этапе щелкните на нем, а затем на кнопке Expand Transcript, которая появляется справа (см. пункты меню Editor).

Перемещаясь по окнам с помощью панели навигатора, можно модифицировать щелчки, чтобы уточнить, где именно происходит навигация. По умолчанию щелчок при нажатой клавише <Option> выполняет навигацию в панели помощника (см. далее), двойной щелчок открывает новое окно, а щелчок при нажатой комбинации клавиш <Option+Shift> открывает окно навигации, которое представляет собой маленькую панель, в которой можно указать, где следует выполнять навигацию (в новом окне, в новой вкладке или новом окне помощника). Настройки, управляющие этими модификациями щелчков, находятся на панели навигации среди настроек среды Xcode.

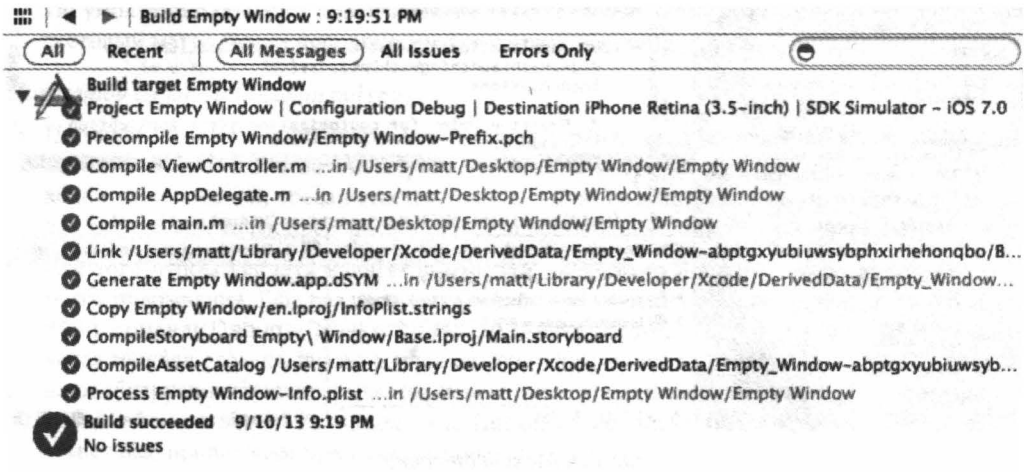


Рис. 6.5. Просмотр журнала

Панель утилит

Панель утилит — это столбец в правой части окна проекта. Она содержит инспекторы, предоставляющие информацию о текущем выборе или настройках; если настройки допускают изменения, то это можно сделать именно здесь. Эта панель также содержит библиотеки, играющие роль источников объектов, необходимых при редактировании проекта. Важность панели утилит проявляется в основном при редактировании файлов `.storyboard` или `.xib` (глава 7). В то же время она может быть полезной при редактировании кода, потому что, помимо прочего, в ней выводится справочник Quick Help, представляющий собой разновидность документации (см. главу 8). Кроме того, панель утилит служит источником сниппетов кода (см. главу 9). Для переключения режима видимости панели утилит выберите команду View⇨Utilities⇨Hide/Show Utilities (или нажмите клавиши <Command+Option+0>). Чтобы изменить ширину панели утилит, перетаскивайте вертикальную линию на ее левом крае.

Панель утилит состоит из многочисленных палитр, объединенных в многочисленные множества, которые сами подразделяются на две основные группы: верхнюю половину панели и нижнюю. Относительную высоту каждой из этих двух половинок можно изменить, перетаскивая горизонтальную линию, которая их разделяет.

Верхняя половина

Содержимое верхней половины зависит от текущего выбора в окне редактора. Возможны три варианта.

Редактируется исходный файл

Верхняя половина панели утилит содержит либо инспектор файлов, либо справочник Quick Help. Переключаться между ними можно с помощью пиктограмм, расположенных на верхнем крае этой половины панели утилит, или с помощью комбинаций клавиш <Command+Option+1> и <Command+Option+2>. Инспектор файлов используется редко, а справочник Quick Help может оказаться полезным. Инспектор файлов содержит несколько разделов, каждый из которых можно развернуть или свернуть, щелкая на его заголовке.

Редактируется файл `.storyboard` *или* `.xib`

Кроме инспектора файлов и справочника Quick Help, в верхней части панели утилит располагается инспектор идентичности (<Command+Option+3>), инспектор атрибутов (<Command+Option+4>), инспектор размеров (<Command+Option+5>) и инспектор соединений (<Command+Option+6>). Каждый из этих инспекторов может содержать несколько разделов, которые можно развернуть или свернуть, щелкая на их заголовках.

Редактируется каталог ресурсов

Кроме инспектора файлов и справочника Quick Help, инспектор атрибутов (<Command+ +Option+4>) позволяет выяснить возможные варианты изображений.

Нижняя часть

Нижняя часть панели утилит демонстрирует одну их четырех библиотек. Для переключения между ними используются пиктограммы, расположенные вдоль верхнего края этой половины или соответствующие комбинации клавиш. Это библиотеки файловых шаблонов (<Command+Option+Control+1), сниппетов кода (Command+Option+Control+2), объектов (<Command+Option+Control+3) и мультимедиа (Command+Option+Control+4). Самой важной является библиотека объектов; мы будем интенсивно использовать ее при редактировании файлов `.storyboard` или `.xib`.

Для того чтобы увидеть раскрывающийся список, описывающий пункт, выбранный в библиотеке, нажмите клавишу <Spacebar>.

Редактор

В центре окна проекта находится редактор. Именно здесь выполняется реальная работа, чтение и запись кода (см. главу 9), а также разработка интерфейса в файлах `.storyboard` или `.xib` (глава 7). Редактор — это сердцевина окна проекта. Панель навигатора, утилит или отладки можно скрыть, но окна проекта без редактора не существует (хотя редактор можно полностью закрыть панелью отладки).

Редактор предоставляет свою форму навигации — панель быстрых переходов вдоль верхнего края. Панель быстрых переходов не только демонстрирует место редактируемого файла в файловой иерархии, но и позволяет переключиться на другой файл. В частности, каждый компонент пути на панели быстрых переходов сам представляет раскрывающийся список. Эти списки можно активизировать, щелкая на компоненте пути или нажимая комбинации клавиш (показанные во втором разделе подменю View⇨Standard Editor). Например, комбинация клавиш <Control+4> открывает иерархический раскрывающийся список, по которому можно передвигаться с помощью клавиш и выбирать разные файлы проекта для редактирования. Более того, каждый раскрывающийся список на панели быстрых переходов имеет поле фильтра; для того чтобы увидеть его, активизируйте всплывающее меню и начните ввод символов. Таким образом, можно передвигаться по проекту, даже не используя навигатор проекта.

Символ в левом конце панели быстрых переходов (<Control+1>) открывает иерархическое меню Related Files, позволяющее перемещаться по файлам, связанным с текущим файлом. То, что появляется при этом на экране, зависит не только от текущего редактируемого файла, но и от текущего выбора в этом файле. Это чрезвычайно мощное и удобное меню, заслуживающее внимательного изучения. Если редактируемый файл является одним из пары файлов классов (`.m` или `.h`), можно переключиться на другой член этой пары (команда Counterparts). Можно также перемещаться по связанным и заголовочным файлам (команды Superclasses,

Subclasses и Siblings; одноуровневые классы (siblings) — это классы, имеющие один и тот же суперкласс). Существует возможность переходить в файл, который содержится в заданном файле, и в файл, который содержит заданный файл; можно увидеть методы, которые вызываются выбранным методом, а также метод, который вызвал текущий выбранный метод.

Редактор запоминает историю отображаемых сущностей. Благодаря этому можно вернуться к ранее просмотренному содержимому, щелкнув на кнопке Back панели быстрых переходов, которая открывает также раскрывающийся список. В качестве альтернативы можно выбрать команду Navigate⇨Go Back (Command+Control+Left).

Скорее всего, при разработке проекта вы будете одновременно редактировать несколько файлов или откроете несколько представлений одного и того же файла для редактирования в двух областях одновременно. Это можно сделать тремя способами: с помощью помощников, вкладок и вспомогательных окон.

Помощники

Окно редактора можно разделить на несколько окон, активизировав панель помощника. Для этого щелкните на второй кнопке Editor инструментальной панели или выберите команду View⇨Assistant Editor⇨Show Assistant Editor (или нажмите комбинацию клавиш <Command+Option+Return>). Кроме того, по умолчанию панель помощника открывается после нажатия клавиши <Option> при навигации; например, комбинация клавиши <Option> и щелчка на панели навигатора или клавиши <Option> и выбора на панели быстрых переходов позволяет осуществлять навигацию в открывающейся (или существующей) панели помощника. Для того чтобы удалить панель помощника, сначала щелкните на кнопке Editor инструментальной панели, или выберите команду View⇨Standard Editor⇨Show Standard Editor (или нажмите клавиши <Command+Return>), или щелкните на кнопке X в верхнем правом углу панели помощника.

Вкладки

Все окно проекта можно реализовать в виде вкладки. Для этого выберите команду File⇨New⇨Tab (или нажмите клавиши <Command+T>), которая выводит на экран панель вкладок (непосредственно под инструментальной панелью), если она еще не была активизирована. Используйте интерфейс вкладок, который должен быть вам известен по другим приложениям, например по браузеру Safari. Переходить с одной вкладки на другую можно, щелкая на вкладках или нажимая комбинацию клавиш <Command+Shift+}>. Сначала новая вкладка будет очень похожей на оригинальное окно, из которого она была порождена. Но теперь у вас есть возможность вносить в нее изменения — например, указывать, какие панели должны демонстрироваться и какие файлы должны редактироваться, — не влияя на другие вкладки. Таким образом, можно получить несколько представлений своего проекта. Каждую вкладку можно назвать информативным именем: для того чтобы редактировать имя вкладки, дважды щелкните на нем.

Вспомогательные окна

Вспомогательное окно проекта похоже на вкладку, но появляется как отдельное окно, а не как вкладка в том же окне. Для его создания выберите команду File⇨New⇨Window (или нажмите комбинацию клавиш <Command+Shift+T>). В качестве альтернативы можно превратить вкладку в окно, перетащив его за пределы текущего окна.

Пользователь может самостоятельно определить внешний вид панели помощника. Для этого выберите команду в подменю View⇨Assistant Editor. Обычно я предпочитаю опцию All

Editors Stacked Vertically, но это дело вкуса. Открыв панель помощника, вы можете разделить ее на дополнительные панели помощника. Для этого щелкните на кнопке Plus в правом верхнем углу панели помощника. Для перемещения содержимого текущей панели помощника в главный редактор выберите команду Navigate⇨Open in Primary Editor. Для того чтобы закрыть панель помощника, щелкните на кнопке X в его правом верхнем углу.

Панель помощника становится действительно помощником, а не разновидностью редактирования на нескольких панелях, благодаря тому, что у него есть особое отношение с главным окном редактирования. Содержимое главного окна редактирования по умолчанию зависит от того, на каком пункте вы щелкнете на панели навигатора; в то же время панель помощника может реагировать на то, какой файл редактируется в главном окне редактирования, изменяя файл, который редактируется на панели помощника. Этот процесс называется слежением (tracking).

Для того чтобы увидеть слежение в действии, откройте отдельное окно помощника и активизируйте его первый компонент на панели быстрых переходов (<Control+4>). Это меню называется Tracking. Оно похоже на меню Related Files, о котором говорилось выше, но вы можете выбрать категорию для определения автоматического слежения. Например, выберите команду Counterparts (рис. 6.6). Теперь в навигаторе проекта выберите файл `AppDelegate.m`; этот файл появится в главном окне редактирования, а панель помощника отобразит файл `AppDelegate.h`. Затем используйте навигатор проекта для выбора файла `AppDelegate.h`; главное окно редактирования отобразит этот файл, а на панели помощника автоматически появится файл `AppDelegate.m`. Если категория содержит несколько файлов, помощник сначала перейдет на первый файл, затем на правом конце панели быстрых переходов появится пара кнопок со стрелками, с помощью которых можно перемещаться по другим файлам (или используйте второй компонент панели быстрых переходов, нажав клавиши <Control+5>). Эта панель обеспечивает большое удобство и мощь, поэтому заслуживает внимательного изучения. Слежение можно отключить, установив настройку первого компонента панели быстрых переходов равной `Manual`.

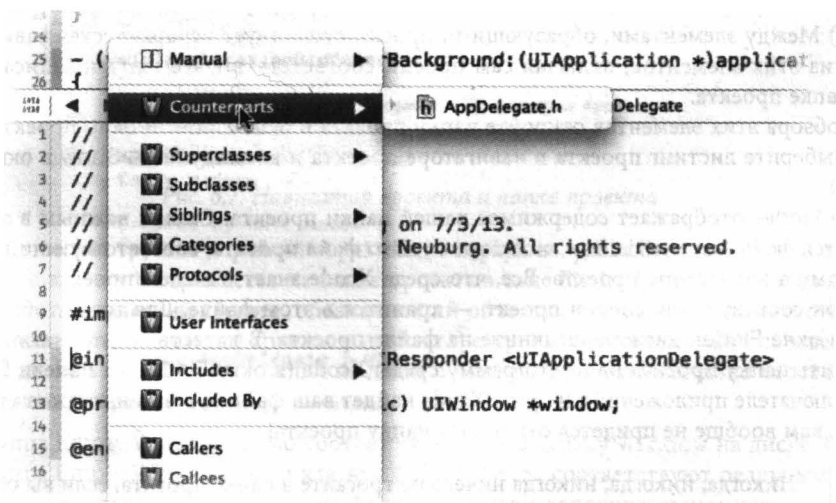


Рис. 6.6. Приказ панели ассистента отслеживать сопряженные файлы

Между вкладкой и вспомогательным окном нет принципиальной разницы, поэтому выбор одного из этих механизмов — дело вкуса. Я предпочитаю использовать вспомогательное окно,

потому что его можно открыть одновременно с главным окном и в то же время сделать маленьким. Таким образом, если я работаю с файлом, на который приходится часто ссылаться, я часто открываю этот файл во вспомогательном окне, делаю это окно достаточно маленьким и не открываю никаких других панелей, кроме редактора.

Поведение вкладок и окон для удобства можно настраивать. Например, как я указывал выше, важно иметь возможность видеть консоль в процессе отладки; я предпочитаю раскрывать ее на все окно проекта и в то же время хотел бы иметь возможность переключаться на просмотр исходного кода. Для этого я создаю поведение (щелкая на кнопке Plus на нижнем крае панели Behaviors в окне Preferences), которое предусматривает два действия: Show tab named Console in active window (Открыть вкладку Console в активном окне) и Show debugger with Console View (Открыть отладчик в представлении консоли). Более того, я назначаю для этого поведения комбинацию клавиш. Таким образом, каждый раз, когда я нажимаю эту комбинацию клавиш, я переключаюсь на вкладку Console (создавая ее, если она не была создана ранее), в которой не выводится ничего, кроме содержимого консоли. Это обычная вкладка, поэтому я могут переходить от нее к своему коду, нажимая клавиши <Command+Shift+}>.

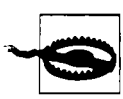
Существует много способов изменить содержимое, которое отображается в окне редактирования, и навигаторы не поддерживают автоматическую синхронизацию этих изменений. Для того чтобы выбрать в навигаторе проекта файл, отображаемый в окне редактирования, выберите команду Navigate⇨Reveal in Project Navigator. Кроме того, существуют пункты Reveal in Symbol Navigator и Reveal in Debug Navigator в меню Navigate.

Файл проекта и его зависимости

Первый элемент в навигаторе проекта (<Command+1>) представляет собой сам проект. (В проекте `Empty Window`, созданном в этой главе ранее, этот пункт назывался Empty Window.) Между элементами, образующими проект, существует иерархическая зависимость. Многие из этих элементов, включая сам проект, соответствуют элементам, записанным на диск в папке проекта.

Для обзора этих элементов откройте папку проекта в окне Finder и окно проекта в среде Xcode. Выберите листинг проекта в навигаторе проекта и команду File⇨Show в окне Finder (рис. 6.7).

Окно Finder отображает содержимое вашей папки проекта. Самым важным в этой папке является файл `Empty Window.xcodeproj`. Это файл проекта, соответствующий проекту, указанному в навигаторе проекта. Все, что среда Xcode знает о вашем проекте, — из каких файлов он состоит и как собран проект, — хранится в этом файле. Для того чтобы открыть проект в окне Finder, дважды щелкните на файле проекта. В качестве альтернативы можно перетащить папку проекта на пиктограмму среды Xcode (в окне Finder, на панели Dock или на переключателе приложений), и среда Xcode найдет ваш файл проекта и откроет его; таким образом, вам вообще не придется открывать папку проекта!



Никогда, никогда, никогда ничего не трогайте в папке проекта, если вы открыли ее в окне Finder. Можете только дважды щелкнуть на файле проекта, чтобы открыть его. Ничего не записывайте в эту папку непосредственно. Ничего не удаляйте из этой папки непосредственно. Ничего не переименовывайте в папке проекта. Ни к чему не прикасайтесь в папке проекта! Все ваше взаимодействие

с проектом должно происходить с помощью окна проекта в среде Xcode. (Когда вы станете опытным пользователем среды Xcode, то узнаете, в каких ситуациях можно нарушать это правило. А пока слепо выполняйте его!)

Дело в том, что изменения в папке проекта должны происходить определенным образом. Если вы вносите изменения в проект непосредственно в окне Finder, так сказать, за спиной проекта, то этот порядок нарушается и проект разрушается. Когда вы работаете в окне проекта, среда Xcode сама внесет необходимые изменения в папку проекта, и все будет в порядке.

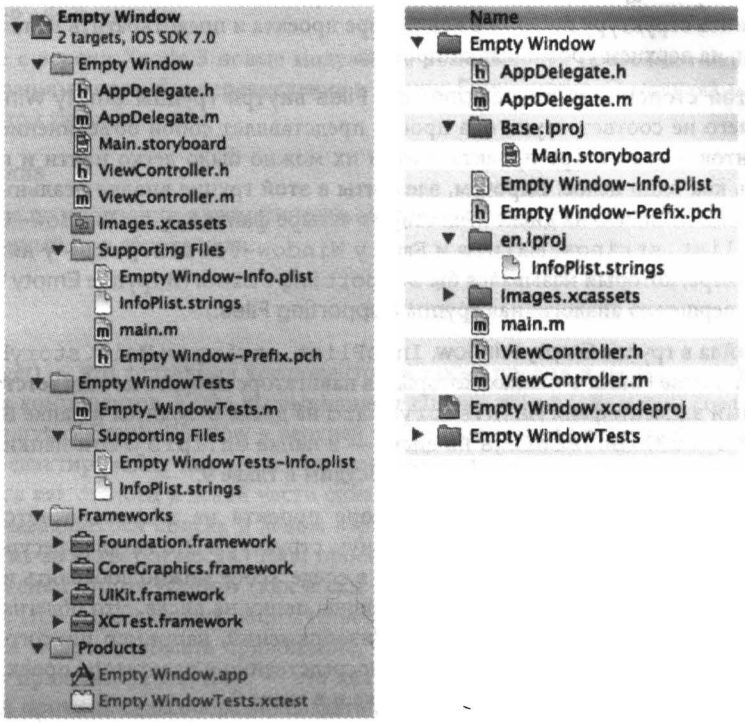


Рис. 6.7. Навигатор проекта и папка проекта

Посмотрим, как группы и файлы, образующие иерархию, удаляются из проекта в навигаторе проекта и как это отражается на папке проекта в окне Finder (рис. 6.7). (Напомним, что группа — это технический термин, обозначающий объекты, похожие на папки и отображающиеся в навигаторе проекта.) Группы в навигаторе проекта не обязательно соответствуют папкам на диске в окне Finder, а папки на диске в окне Finder не обязательно соответствуют группам в навигаторе проекта.

- Группа Empty Window прямо соответствует папке `Empty Window` на диске. Файлы в группе Empty Window, такие как `AppDelegate.m`, соответствуют реальным файлам на диске в папке `Empty Window`. Если вы создали дополнительные исходные файлы (которые в реальной жизни вы практически обязательно создадите в ходе разработки своего проекта), то скорее всего включите их в группу Empty Window в навигаторе проекта, и они окажутся в папке `Empty Window` на диске. (Однако это не является

обязательным требованием; ваши файлы могут находиться где угодно, и ваш проект при этом будет прекрасно работать.)

Аналогично группа Empty Window Tests соответствует папке `Empty Window Tests` на диске, а файл `Empty_WindowTests.m` в группе Empty Window Tests находится в папке `Empty Window Tests`.

Эти две пары группа–папка соответствуют двум целям вашего проекта. Что такое цель, я объясню в следующем разделе. Нет правила, утверждающего, что каждая цель должна иметь соответствующую группу в навигаторе проекта и соответствующую папку в папке проекта, но для удобства шаблон проекта делает именно так: это позволяет прояснить структуру проекта в навигаторе проекта и предотвратить появление многих файлов на верхнем уровне папки проекта.

- С другой стороны, группе Supporting Files внутри группы Empty Window на диске ничего не соответствует; она просто представляет собой объединение нескольких элементов в навигаторе проекта, чтобы их можно было легко найти и показать или скрыть как одно целое. Впрочем, элементы в этой группе вполне реальны; вы можете сами убедиться, что на диске существуют четыре файла `Empty Window-Info.plist`, `InfoPlist.strings`, `main.m` и `Empty Window-Prefix.pch` — у них просто нет контейнера, который назывался бы `Supporting Files`. (В группе Empty WindowTests есть совершенно аналогичная группа Supporting Files.)
- Два файла в группе Empty Window, `InfoPlist.strings` и `Main.storyboard`, появляются в окне Finder в папках, которым в навигаторе проекта не соответствует ни один видимый элемент: файл `Main.storyboard` на диске находится в папке `Base.lproj`, а файл `InfoPlist.strings` на диске — в папке `en.lproj`. Эти папки относятся к механизму локализации, который мы обсудим в главе 9.
- Элементу Images.xcassets в навигаторе проекта на диске соответствует папка `Images.xcassets`, имеющая специальную структуру. Это каталог ресурсов (новшество в среде Xcode 5); в каталог ресурсов в среде Xcode можно добавлять изображения, которые будут записаны в соответствующую папку на диске. Это облегчает работу со множеством связанных друг с другом изображений, например пиктограмм разного размера, не просматривая их список непосредственно в навигаторе проекта. О каталоге ресурсов мы поговорим немного позже и в главе 9.

Возможно, все это покажется вам слишком запутанным. Совсем нет! Помните, что я говорил вам не вмешиваться в содержание вашей папки проекта на диске в окне Finder. Сосредоточьте свое внимание на навигаторе проекта, вносите свои модификации только здесь, и все будет хорошо.

Не стесняйтесь создавать новые группы в ходе разработки проекта и добавлять в них файлы. Цель групп — помочь навигатору проекта хорошо выполнять ваши требования. Они не влияют на сборку приложения и по умолчанию не соответствуют ни одной папке на диске; это просто удобный способ организации файлов в навигаторе проекта. Для создания новой группы выберите команду File⇨New⇨Group. Для того чтобы переименовать группу, выберите ее в навигаторе проекта и нажмите клавишу <Return>, чтобы отредактировать ее имя. Например, если какие-то ваши файлы связаны с экраном приветствия вашего приложения, вы можете собрать их в группу Login. Если ваше приложение содержит звуковые файлы, вы можете собрать их в группу Sounds. И так далее.

Элементы групп Frameworks и Products не имеют аналогов в папке проекта, но им соответствуют реальные сущности, необходимые для сборки и запуска проекта.

Группа Frameworks

В этой группе по принятому соглашению перечисляются каркасы (код Cocoa), от которых зависит ваш код. Каркасы существуют на диске, но они не встраиваются в ваше приложение во время его сборки; это не нужно, потому что каркасы находятся также на целевом устройстве (iPhone, iPod touch или iPad). Вместо этого каркасы связываются с приложением, т.е. приложение знает о них и ожидает найти их на целевом устройстве во время выполнения. Таким образом, весь код каркасов в приложении отсутствует, экономя значительный объем памяти.

(Начиная с версии Xcode 5 новые модульные возможности позволяют каркасам связываться с вашим кодом без перечисления в группе Frameworks. О модулях мы поговорим в конце этой главы.)

Группа Products

Эта группа по умолчанию автоматически содержит ссылку на выполняемый комплект, сгенерированный при сборке цели.

Цель

Цель (target) — это коллекция компонентов, а также правил и настроек для сборки продукта из этих компонентов. Когда вы выполняете сборку, вы на самом деле создаете цель.

Выберите пункт Empty Window в верхней части навигатора проекта. Вы увидите в левой части окна редактирования два элемента: сам проект и список его целей. (Этот список может появиться как столбец в левой части окна редактирования, как показано на рис. 6.8, или в виде раскрывающегося списка в левом верхнем углу окна редактирования, если столбец был свернут из-за нехватки места.) Наш проект Empty Window имеет две цели: целевое приложение с именем Empty Window (как и сам проект) и цель тестирования с именем Empty WindowTests. При определенных обстоятельствах вы можете добавлять в проект новые цели. Например, вы можете написать приложение, которое можно собрать и как приложение для iPhone, и как приложение для iPad, — это два разных приложения, у которых много общего кода. По этой причине целесообразно создать один проект, который будет выполнять сборку обоих приложений с разными целями.

Если вы выберете проект в левом столбце или во всплывающем меню редактора, то будете редактировать проект. Если вы выберете цель в левом столбце или во всплывающем меню редактора, то будете редактировать цель. Я буду часто использовать эти выражения в дальнейших инструкциях.

Сосредоточьтесь на целевом приложении Empty Window. Это цель, которую мы будем использовать для сборки и выполнения вашего приложения. Ее настройки сообщают среде Xcode, как собрать ваше приложение; ее результат — это само приложение. (Цель тестирования Empty WindowTests создает специальный выполняемый код, цель которого — протестировать код вашего приложения. Более подробно тестирование описывается в главе 9.)

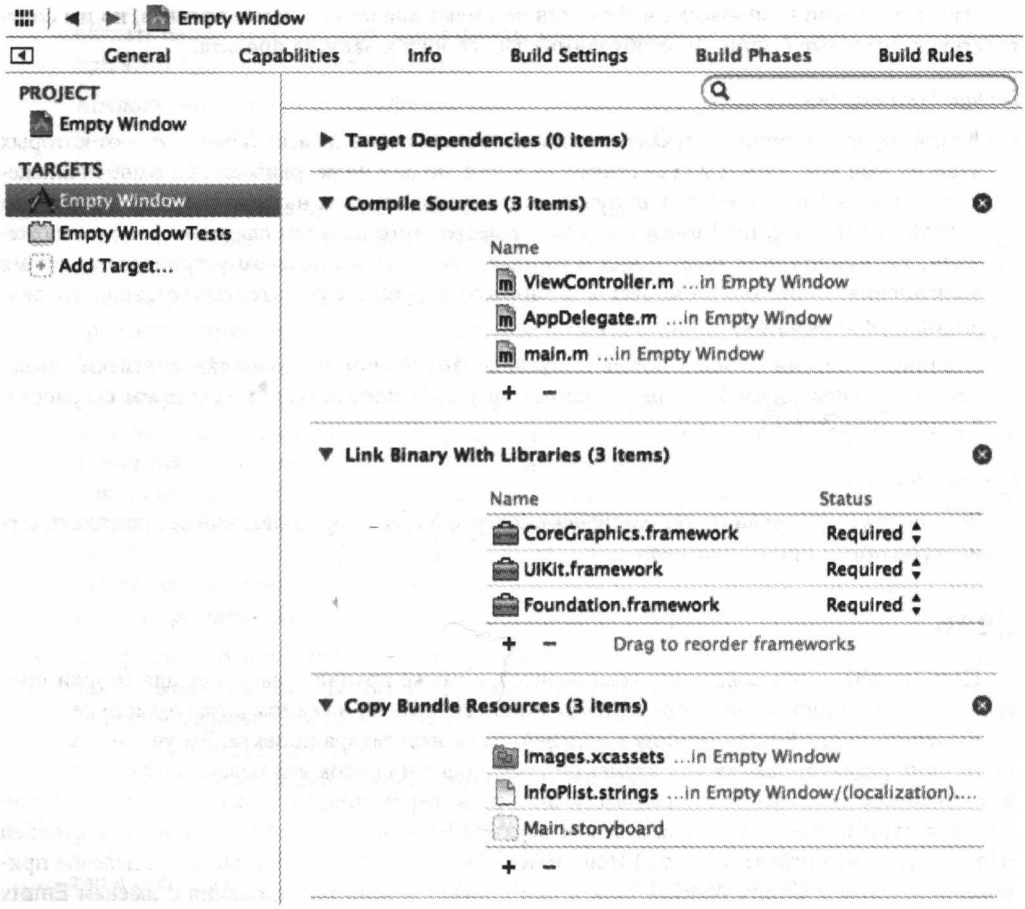


Рис. 6.8. Редактирование целей приложения для демонстрации его этапов

Фазы сборки

Отредактируйте цель Empty Window и щелкните на пункте Build Phases в верхней части редактора (рис. 6.8). Вы увидите этапы, из которых состоит сборка приложения. По умолчанию самые необходимые из них имеют содержание — Compile Sources, Link Binary With Libraries и Copy Bundle Resources, — в то время как остальные являются необязательными. Фазы сборки подразумевают создание отчета о сборке цели и набор инструкций, необходимых среде Xcode для сборки цели. Если изменить фазы сборки, изменится сам процесс сборки. Щелкните на каждой фазе сборки, чтобы увидеть список файлов в вашей цели, к которым будет применяться процесс сборки.

Смысл каждой из фаз сборки довольно очевиден.

Фаза Compile Sources

Компилируются некоторые файлы (ваш код), а скомпилированный код копируется в приложение.

Эта фаза сборки обычно применяется ко всем целевым файлам `.m`; это файлы исходного кода, образующие цель. В настоящее время цель содержит файлы `ViewController.m`, `AppDelegate.m` и `main.m`. Если вы добавите в свой проект новый класс, то должны будете указать, что он является частью целевого приложения, а файл `.m` будет автоматически добавлен в фазу Compile Sources.

Фаза Link Binary With Libraries

Некоторые библиотеки (обычно каркасы) связываются со скомпилированным кодом (теперь он является бинарным), так что приложение теперь будет ожидать, что во время его выполнения эти библиотеки будут находиться на устройстве.

Эта фаза сборки в настоящее время содержит три каркаса. Механизм связывания бинарного кода с дополнительными каркасами будет обсуждаться позднее.

Фаза Copy Bundle Resources

Некоторые файлы копируются в приложение, чтобы ваш код или система могли найти их во время выполнения приложения. Например, если приложение имеет пиктограмму, ее нужно скопировать в приложение, чтобы устройство могло его найти и отобразить на экране.

Эта фаза сборки в настоящее время применяется к каталогу ресурсов; в каталог изображения можно добавить любые изображения, которые будут скопированы в ваше приложение. В настоящий момент каталог содержит файлы `InfoPlist.strings` и `.storyboard`.

Копирование не обязательно означает создание идентичной копии. Файлы некоторых типов при копировании в комплект приложения автоматически обрабатываются специальным образом. Например, копирование каталога ресурсов означает, что пиктограммы и заставки в каталоге записываются на верхний уровень комплекта приложения; копирование файла `.storyboard` означает, что он будет преобразован в файл `.storyboardc`, который в свою очередь представляет собой комплект, состоящий из nib-файлов.

Этот список можно изменять вручную. Например, если звуковой файл не был включен в фазу Copy Bundle Resources и вы хотите скопировать его в приложение в ходе сборки, перетащите его из навигатора проекта в список Copy Bundle Resources или (что легче) щелкните на кнопке Plus под списком Copy Bundle Resources, чтобы открыть полезное диалоговое окно, в котором перечисляются все компоненты вашего проекта. И наоборот, если какой-то компонент вашего проекта был включен в фазу Copy Bundle Resources и вы не хотите копировать его в приложение, удалите его из списка; это не значит, что вы удаляете его из проекта, навигатора проекта или окна Finder, — вы просто удаляете его из списка элементов, которые должны копироваться в ваше приложение.

Существует удобный трюк — добавить фазу сборки Run Script, запускающую оболочку специального сценария на поздних этапах сборки. Для этого выберите команду Editor⇨Add Build Phase⇨Add Run Script Build Phase. Откройте вновь добавленную фазу сборки Run Script и отредактируйте сценарий. Минимальный сценарий может содержать лишь команду

```
echo "Running the Run Script build phase"
```

Флаг Show environment variables in build log регулирует перечисление переменных окружения сборки и их значения в журнале сборки в ходе выполнения фазы Run Script. Это сама по себе достаточная причина для добавления фазы сборки Run Script; вы можете изучить множество деталей процесса сборки, проанализировав переменные окружения.

Настройки сборки

Фазы сборки — это лишь один аспект механизма, позволяющего цели выяснять способ сборки приложения. Вторым аспектом являются настройки. Для того чтобы их увидеть, откройте цель и щелкните на пункте Build Settings на верхнем крае редактора (рис. 6.9). Здесь вы найдете длинный список настроек, большинство из которых вы никогда не будете изменять. В то же время среда Xcode проверяет этот список, чтобы узнать, что именно делать на разных стадиях процесса сборки. Эти настройки определяют способ компиляции и сборки проекта.

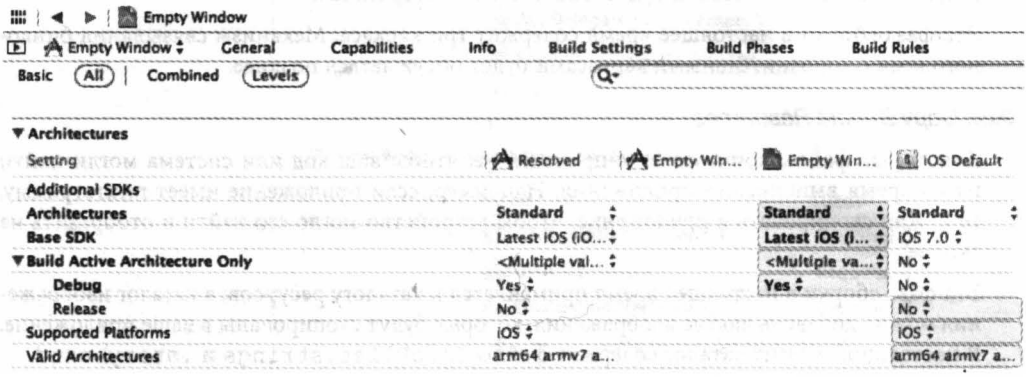


Рис. 6.9. Настройки сборки

Вы можете самостоятельно определять, какие настройки выводить на экран, щелкая на кнопке Basic или All. Настройки подразделяются на две категории. Для того чтобы сэкономить место на экране, эти категории можно открывать или закрывать. Если вам уже известно что-то о настройках, которые вы хотите увидеть, например их имена, используйте поле поиска в правом верхнем углу, чтобы фильтровать отображаемые настройки.

Для того чтобы определить способ отображения настроек на экране, можно щелкать на кнопке Combined или Levels; на рис. 6.9 продемонстрирована ситуация, сложившаяся после того, как я щелкнул на кнопке Levels, чтобы обсудить ее смысл. Она показывает, что не только цель содержит значения настроек сборки, но и проект; более того, среда Xcode имеет несколько встроенных настроек, имеющих значения, заданные по умолчанию. Кнопка Levels показывает все эти уровни сразу, и пользователь может понять происхождение реальных значений, используемых для каждой настройки сборки.

Для того чтобы понять этот рисунок, прочитайте его справа налево. Например, рассматривая раздел iOS default, мы видим, что параметр Debug настройки Build Active Architecture Only по умолчанию равен `No`. Однако затем проект (второй столбец справа) устанавливает его равным `Yes`. Цель (третий столбец справа) не изменяет эту настройку, так что результат (четвертый столбец) в разделе Resolved остается равным `Yes`.

Вам редко придется непосредственно манипулировать настройками сборки, поскольку параметры, заданные по умолчанию, как правило. вполне приемлемы. Тем не менее такая возможность существует, и мы показали вам, как это можно сделать. Вы можете изменить значение на уровне проекта или цели. Можете выбрать настройку сборки и открыть справочник Quick Help на панели утилит, чтобы получить информацию об этой настройке. Более подробные сведения обо всех настройках сборки можно узнать в документации компании Apple, особенно в справочнике Xcode Build Setting Reference.

Конфигурации

Значения настроек сборки образуют несколько списков, хотя в конкретной сборке применяется только один список. Такой список называется конфигурацией. Несколько конфигураций необходимы для того, чтобы сборку можно было выполнять разными способами в разные моменты времени для разных целей, поэтому удобно иметь определенные настройки сборок, принимающие разные значения в разных ситуациях.

По умолчанию существуют две конфигурации.

Конфигурация Debug

Используется на всем протяжении процесса разработки, когда вы пишете и запускаете свое приложение.

Конфигурация Release

Используется на поздних этапах тестирования, когда необходимо проверить производительность устройства.

Конфигурации фиксируют те параметры, которые задал проект. Для того чтобы увидеть, где именно проект задает параметры, откройте его в окне редактирования и щелкните на кнопке Info в верхней части редактора (рис. 6.10). Обратите внимание на то, что конфигурации — это просто имена. Вы можете создавать дополнительные конфигурации, а потом добавлять их в список имен. Важность конфигурации проявляется только тогда, когда эти имена объединяются со значениями настроек сборки. Конфигурации могут влиять на значения настроек сборок как на уровне проекта, так и на уровне цели.

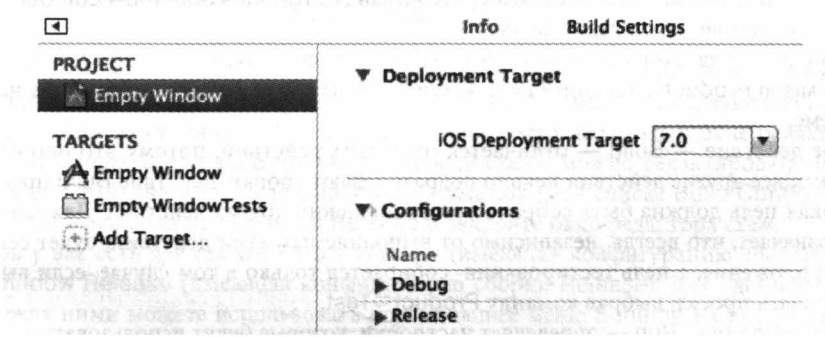


Рис. 6.10. Конфигурации

Например, вернитесь к настройкам цели (см. рис. 6.9) и наберите строку "`Optim`" в поисковом поле. Теперь вы увидите настройку сборки Optimization Level (рис. 6.11). Значение параметра Optimization Level в конфигурации Debug равно `None`: разрабатывая свое приложение, вы собираете его с помощью конфигурации Debug, поэтому ваш код просто компилируется строка за строкой. Значение параметра Optimization Level в конфигурации Release равно `Fastest`, `Smallest`. Когда ваше приложение будет готово для поставки, вы соберете его с использованием конфигурации Release, и полученный бинарный код будет быстрее и меньше, что очень важно для ваших пользователей, устанавливающих и запускающих ваше приложение на своих устройствах, но мешает работе на этапе разработки приложения, потому что нарушает механизмы точек прерывания и пошаговой отладки.

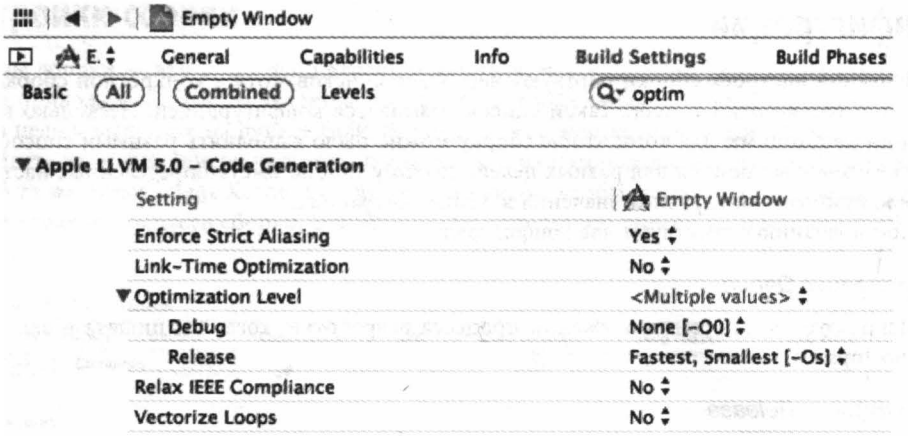


Рис. 6.11. Влияние конфигураций на настройки сборки

Схемы и предназначения

До сих пор я ничего не говорил о том, как среда Xcode узнает, какую конфигурацию использовать для конкретной сборки. Это определяется схемой.

Схема объединяет цель (или несколько целей) и конфигурацию сборки с учетом предназначения сборки. Новый проект открывается по умолчанию с одной схемой, которая называется по имени проекта. Таким образом, схема проекта Empty Window называется Empty Window. Для того чтобы увидеть ее, выберите команду Product⇨Scheme⇨Edit Scheme. Она открывает диалоговое окно редактора схем.

В левой части редактора схем перечислены разные действия, которые можно выполнить с помощью меню Product. Щелкните на действии, чтобы увидеть соответствующие настройки в этой схеме.

Первое действие — Build — отличается от других действий, потому что оно общее для всех, так как все другие действия неявно подразумевают сборку. Действие Build просто определяет, какая цель должна быть собрана при выполнении других действий. Для нашего проекта это означает, что всегда, независимо от выполняемых вами действий, будет собираться целевое приложение, а цель тестирования собирается только в том случае, если вы решили протестировать проект, выбрав команду Product⇨Test.

Второе действие — Run — определяет настройки, которые будут использоваться при сборке и выполнении приложения (рис. 6.12). Раскрывающийся список Build Configuration настроен для конфигурации Debug. Это объясняет, как определяется текущая конфигурация сборки: в данный момент, если мы соберем и запустим приложение (выбрав команду Product⇨Run или щелкнув на кнопке Run инструментальной панели), будет использоваться конфигурация сборки Debug и соответствующие ей настройки, потому что вы выбрали именно такую схему. Так схема определяет, что делать, когда вы собираете и выполняете приложение.

Вы можете редактировать эту схему или создавать дополнительные схемы. Например, допустим, что вы хотите собрать и выполнить приложение, применяя конфигурацию сборки Release (чтобы протестировать приложение как можно ближе к условиям, в которых его будет использовать пользователь). Для этого можно отредактировать конфигурацию сборки для действия Run. Среда Xcode позволяет сделать это просто и удобно: нажмите клавишу Option, одновременно выбрав команду Product⇨Run (или щелкнув на кнопке Run). В результате

откроется окно редактора схемы, содержащее кнопку Run. Теперь вы можете внести изменения в схему, а затем перейти к непосредственной сборке и выполнению приложения, щелкнув на кнопке Run.

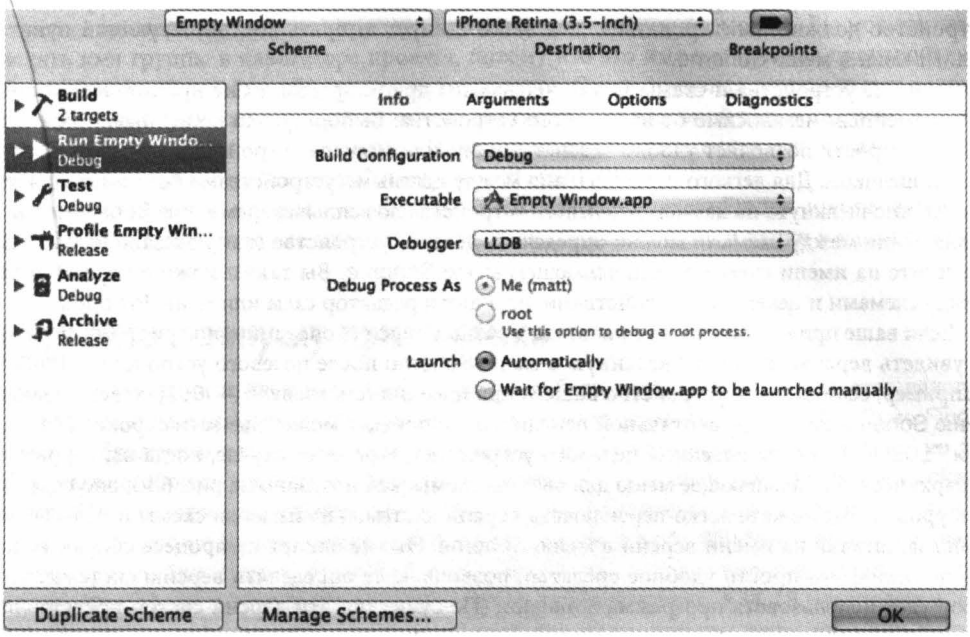


Рис. 6.12. Редактор схемы

Если вы часто хотите переключаться между конфигурациями сборки и выполнения Debug и Release, то можете создать отдельную, дополнительную схему, использующую конфигурацию Release для действия Run. Сделать это очень просто: находясь в окне редактора схем, щелкните на кнопке Duplicate Scheme. Имя новой схемы можно редактировать; назовем ее Empty Window Release. Измените выбор в раскрывающемся списке Build Configuration для действия Run в вашей новой схеме на Release и закройте окно редактора схем.

Теперь у вас есть две схемы: Empty Window (имеющая конфигурацию сборки Debug) и Empty Window Release (имеющая конфигурацию сборки Release). Для удобного переключения между ними можете использовать всплывающее меню Scheme на инструментальной панели окна проекта (рис. 6.13) до сборки и выполнения приложения.

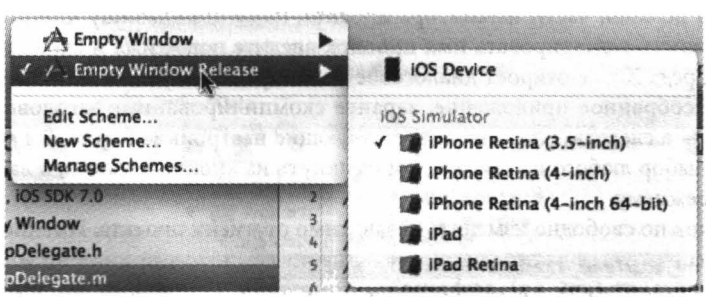


Рис. 6.13. Всплывающее меню Scheme

Во всплывающем меню Scheme перечислены все схемы вместе с их целевым устройством, на котором будет выполняться ваше приложение. В конце концов, ваше приложение будет выполняться либо на реальном физическом устройстве, либо программой Simulator. Если оно будет выполняться программой Simulator, вы можете указать, какое конкретно физическое устройство должно имитироваться. Для этого следует выбрать соответствующий пункт во всплывающем меню Scheme.

Целевые устройства и схемы никак не связаны друг с другом; ваше приложение останется неизменным независимо от выбранного устройства. Выбор пункта во всплывающем меню Scheme просто позволяет удобно задавать схему, или целевое устройство, или и то и другое одним щелчком. Для легкого переключения между целевыми устройствами без изменения схем достаточно щелкнуть на названии целевого устройства во всплывающем меню Scheme. Для переключения между схемами можно определить целевое устройство (как показано на рис. 6.13): щелкните на имени схемы во всплывающем меню Scheme. Вы также можете переключаться между схемами и целевыми устройствами, используя редактор схем или меню Product.

Если ваше приложение может работать в разных версиях операционной системы, вы можете увидеть версию системы, указанную в меню Scheme после целевого устройства Simulator. Например, если целевое устройство вашего приложения (см. главу 9) — `6.1`, то всплывающее меню Scheme на инструментальной панели в окне проекта может вывести строки "`iOS 7.0`" или "`iOS 6.1`" после названия целевого устройства. В данном случае, когда вы открываете иерархическое всплывающее меню для выбора схемы, как показано на рис. 6.13, оно содержит три уровня. Вы можете легко переключать версии системы, не изменяя схемы и целевые устройства, щелкая на имени версии в меню Scheme. Это не влияет на процесс сборки вашего приложения; это просто удобное средство, позволяющее определять версию системы, которую будет использовать программа Simulator. (По существу, эти версии систем представляют собой комплекты SDK; они будут рассмотрены в конце главы. Инсталляция дополнительных комплектов SDK для программы Simulator SDK описана в разделе "Дополнительные комплекты SDK для программы Simulator".)

Дальнейшее управление схемами осуществляется с помощью диалогового окна Manage Schemes (выберите команду Product⇨Scheme⇨Manage Schemes или соответствующую команду в меню Scheme). Например, если вы создали схему Empty Window Release и она вам больше не нужна, в этом окне вы можете ее удалить.

Переименование частей проекта

Имя, присвоенное вашему проекту в процессе его создания, используется во многих местах. Это создает у новичков ложное мнение, что они никогда не смогут переименовать проект, не испортив его. Это совсем не так! Для того чтобы переименовать проект, выберите листинг проекта в верхней части навигатора проекта, нажмите клавишу <Return>, чтобы появилась возможность редактировать имя проекта, введите новое имя и снова нажмите клавишу <Return>. Среда Xcode откроет диалоговое окно, предлагающее изменить и другие имена, включая цель, собранное приложение, заранее скомпилированные заголовочные файлы и файл Info.plist — а следовательно, и соответствующие настройки сборки. Вы можете выбрать или отменить выбор любого имени, а затем щелкнуть на кнопке Rename, и ваш проект будет работать по-прежнему.

Имя цели можно свободно изменять независимо от имени проекта. Именно имя цели, а не имя проекта используется для создания имени продукта, а значит, и имени комплекта, имени дисплея комплекта, а также идентификатора комплекта. Таким образом, выбирая реальное имя приложения, достаточно изменить имя цели.

Изменение имени проекта (или цели) не приводит автоматически к изменению имени соответствующей схемы. Это не обязательно, но возможно; выберите команду Product⇨Manage Schemes и щелкните на имени схемы, чтобы появилась возможность его редактировать.

Изменение имени проекта (или имени цели) не приводит автоматически к изменению имени соответствующей главной группы. Это необязательно, но возможно. Вы можете легко изменять имя группы в навигаторе проекта, потому что это имя выбирается произвольно и не влияет ни на настройки сборки, ни на ее процесс. Однако главная группа имеет особый характер, потому что (как я уже говорил) она соответствует реальной папке на диске, в корне которой хранится файл вашего проекта. Новички не должны изменять имя папки на диске, потому что оно влияет на "зашитые" настройки сборки.

Вы можете в любой момент изменить имя папки проекта в окне Finder или перенести ее в другое место, потому что все ссылки в настройках сборки на файл и папки в папке проекта являются относительными.

От проекта к запуску приложения

Файл приложения представляет собой папку особого вида, которая называется пакетом (package), а специальная разновидность пакета называется комплектом (bundle). Окно Finder обычно отображает пакет в виде файла, не раскрывая пользователю его содержимое, но эту защиту можно обойти и изучить комплект приложения с помощью команды Show Package Contents. Это позволяет исследовать внутреннюю структуру собранного комплекта приложения.

Мы будем использовать минимальное приложение Empty Window, собранное ранее для примера. Найдите его в окне Finder; по умолчанию оно должно находиться в папке `Library/Developer/Xcode/DerivedData`, как показано на рис. 6.14. (Я предполагаю, что вам известно, как найти каталог `Library`.) Теоретически вы должны уметь находить приложение в разделе Products в навигаторе проекта и выбирать команду File⇨Show в окне Finder, но, кажется, эта команда имеет долговременный изъян.

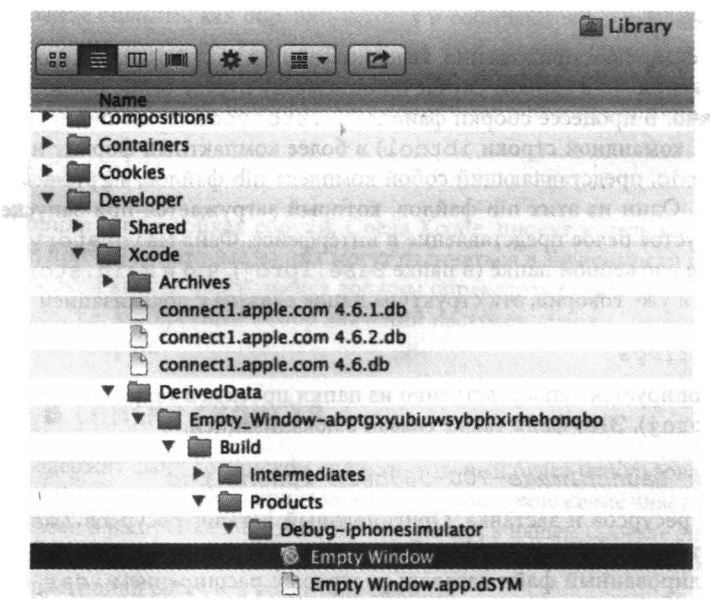


Рис. 6.14. Собранное приложение в окне Finder

Находясь в окне Finder, нажмите клавишу <Control> и щелкните на приложении Empty Window, а затем выберите в контекстом меню команду Show Package Contents. Здесь вы увидите результаты процесса сборки (рис. 6.15).

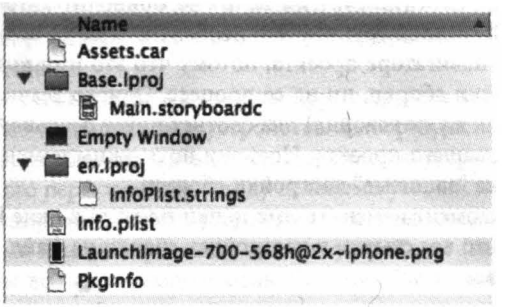


Рис. 6.15. Содержимое комплекта приложения

Можете считать комплект приложения трансформированной папкой проекта.

`Empty Window`

Это скомпилированный код нашего приложения. В процессе сборки компилируются файлы `ViewController.m`, `AppDelegate.m` и `main.m`. Перед этим выполняется предварительная компиляция в соответствии с директивами `#import` (а также импортирование результатов предварительной компиляции файла `Empty Window-Prefix.pch`). В результате возникает отдельный файл, представляющий собой бинарный код нашего приложения. Это сердцевина приложения, реально выполняемый код. При запуске приложения бинарный код связывается с разными каркасами, и код начинает выполняться с функции `main`.

`Main.storyboard`

Это файл раскадровки приложения. Именно файл `Main.storyboard` определяет интерфейс приложения — в данном случае он описывает пустое белое представление, занимающее все окно. В процессе сборки файл `Main.storyboard` компилируется (с помощью инструмента командной строки `ibtool`) в более компактный формат и возникает файл `.storyboardc`, представляющий собой комплект nib-файлов, загружаемых при запуске приложения. Один из этих nib-файлов, который загружается при запуске приложения, описывает пустое белое представление в интерфейсе. Файл `Main.storyboardc` хранится в такой же вложенной папке (в папке `Base.lproj`), что и `Main.storyboard` в папке проекта; как я уже говорил, эта структура папок связана с локализацией (см. главу 9).

`InfoPlist.strings`

Этот файл копируется непосредственно из папки проекта в такую же вложенную папку (в папке `en.lproj`). Этот файл также связан с локализацией.

`Assets.car` и `LaunchImage-700-568h@2x~iphone.png`

Это каталог ресурсов и заставка. Оригинальный каталог ресурсов `Images.xcassets` был обработан с помощью инструмента командной строки `actool`. В результате возник скомпилированный файл каталога ресурсов с расширением `.car`, содержащий все

изображения, добавленные в каталог, и заставку, которая по умолчанию является черным экраном (она моментально выводится во время анимации при запуске приложения). Если бы наше приложение было реальным, то, вероятно, вы увидели бы дополнительную пиктограмму приложения и файлы запуска, скопированные из каталога ресурсов на верхний уровень комплекта приложения.

Info.plist

Это файл конфигурации в текстовом формате (файл, содержащий список свойств). Он выводится из файла проекта `Empty Window-Info.plist` и содержит инструкции, сообщающие системе, как обрабатывать и запускать приложение. Например, если наше приложение имеет пиктограмму, то файл `Info.plist` сообщит системе ее имя, так что система сможет найти ее в комплекте приложения и отобразить на экране. Он также сообщает системе такую информацию, как имя бинарного файла, чтобы система могла найти его и запустить приложение.

PkgInfo

Это маленький текстовый файл, содержащий строку `APPL????`, означающую тип и идентификатор автора приложения. Файл `PkgInfo` — это в некотором роде динозавр; на самом деле для функционирования приложения в системе iOS он не нужен и генерируется автоматически. Редактировать его практически никогда не требуется.

В реальном мире комплект приложения может содержать больше файлов, но разница заключается только в их количестве, а не в разнообразии. Например, наш проект может иметь дополнительные файлы `.storyboard` или `.xib`, файлы с пиктограммами, изображения или звуковые файлы. Все это упаковывается в комплект приложения. Кроме того, комплект приложения, собранное для запуска на устройстве, будет содержать несколько файлов, связанных с безопасностью.

Теперь вы можете оценить, как обрабатываются и собираются в приложение компоненты проекта и что должны делать вы как программист, чтобы ваше приложение было собрано корректно. Оставшаяся часть главы описывает процесс сборки приложения из проекта, а также составные части приложения, используемые для его запуска и приложения.

Настройки сборки

Мы уже говорили о настройках сборки. Среда Xcode, проект и цель выясняют значения настроек сборки, причем некоторые из них могут отличаться в зависимости от конфигурации сборки. Перед сборкой вы как программист должны определить схему; схема определяет конфигурацию сборки, т.е. конкретный набор значений настроек сборки, которые будут использоваться в процессе сборки.

Настройки в списке свойств

Ваш проект содержит файл со списком свойств, который будет использоваться для генерации файла `Info.plist` собранного приложения. Целевое приложение знает, что это за файл, потому что он указан в настройке Info.plist File. Например, в нашем проекте значение целевого приложения Info.plist File установлено равным `Empty Window/Empty Window-Info.plist`. (Посмотрите на настройки сборки и убедитесь сами!!)

Поскольку имя файла в вашем проекте, из которого будет генерироваться файл `Info.plist`, может изменяться в зависимости от имени вашего проекта, я буду называть его просто `Info.plist`.

Файл со списком свойств представляет собой коллекцию пар ключ–значение. Вы можете редактировать его и часто изменять. Существуют три способа редактирования файла `Info.plist`.

- Выберите файл `Info.plist` в навигаторе проекта и отредактируйте его в окне редактора. По умолчанию имена ключей (и некоторые значения) выводятся в описательном стиле, в терминах их функциональности; например, вместо фактического ключа выводится строка "`Bundle name`", а не `CFBundleName`. Однако вы можете увидеть фактические ключи: щелкните на окне редактирования и выберите команду Editor⇨Show Raw Keys & Values или используйте контекстное меню. Кроме того, файл `Info.plist` можно увидеть в его естественной форме XML: нажмите клавишу <Control> на файле `Info.plist` в навигаторе проекта и выберите команду Open As⇨Source Code в контекстном меню.
- Откройте цель и щелкните на кнопке Info в верхней части редактора. Раздел Custom iOS Target Properties содержит практически ту же информацию, что и файл `Info.plist` в окне редактора.
- Откройте цель и щелкните на кнопке General в верхней части редактора. Некоторые из настроек в этом разделе по существу редактируют файл `Info.plist`. Например, когда вы устанавливаете флаг Device Orientation, вы изменяете значение ключа "`Supported interface orientations`" в файле `Info.plist`. (Другие настройки являются способом редактирования настроек сборки. Например, когда вы изменяете значение Deployment Target, вы изменяете значение настройки сборки iOS Deployment Target.)

Некоторые значения в файле `Info.plist` требуют обработки перед записью окончательных значений в файл `Info.plist` собранного приложения. Этот шаг выполняется на поздних этапах сборки. Например, значение ключа "`Executable file`" в файле `Info.plist` проекта равно `${EXECUTABLE_NAME}`; для этого необходимо заменить значение переменной окружения `EXECUTABLE_NAME` (ее можно обнаружить на фазе сборки Run Script). Кроме того, некоторые пары ключ–значение вставляются в файл `Info.plist` на этапе обработки.

Полный список возможных ключей и их значений можно найти в справочнике Information Property List Key Reference компании Apple. Настройки в файле `Info.plist`, которые приходится редактировать особенно часто, рассматриваются в главе 9.

Nib-файлы

Nib-файл — это описание фрагмента пользовательского интерфейса в скомпилированном формате в файле с расширением `.nib`. Каждое приложение, которые вы пишите, практически наверняка содержит хотя бы один nib-файл. Nib-файл генерируется во время процесса сборки путем компиляции (с помощью инструмента командной строки `ibtool`), которая превращает файл `.xib` в nib-файл, а файл `.storyboard` — в комплект `.storyboardc`, содержащий несколько nib-файлов. Эта компиляция выполняется, если файлы `.storyboard` или `.xib` перечислены в фазе сборки Copy Bundle Resources целевого приложения.

Файлы .xib или .storyboard графически редактируются в среде Xcode; по существу, вы графически описываете объекты, которые будут созданы во время запуска приложения и загрузки nib-файла (см. главу 5). Благодаря такой архитектуре nib-файл загружается только при необходимости; это сокращает период запуска приложения, во время которого загружаются только те nib-файлы, которые необходимы для генерации начального интерфейса приложения, а также экономит память во время выполнения программы, потому что объекты, описанные в nib-файле, не создаются, пока не будут необходимы и могут быть удалены, когда они больше не нужны.

Наш проект Empty Window, сгенерированный из шаблона Single View Application, содержит один файл .storyboard с именем Main.storyboard. Этот файл является предметом особого внимания, потому что это главная раскадровка приложения, не столько из-за имени, сколько потому, что он задается в файле Info.plist как значение Main (без расширения .storyboard) ключа "Main storyboard file base name" (UIMainStoryboardFile). Откройте файл Info.plist в окне редактирования и убедитесь сами! В результате при запуске приложения первый nib-файл, сгенерированный из файла .storyboard, загружается автоматически, помогая создать начальный интерфейс приложения.

Если вы решите использовать шаблон Single View Application для создания универсального приложения, т.е. приложения, работающего и на iPad, и на iPhone, то получите два файла .storyboard: один для iPad (Main_iPad.storyboard), а второй для iPhone (Main_iPhone.storyboard). Таким образом, это приложение сможет использовать два разных интерфейса на двух разных типах устройств. В зависимости от типа устройства в момент запуска одна из этих раскадровок будет интерпретироваться как главный файл раскадровок, предназначенный для создания начального интерфейса приложения. Для этой цели используется второй ключ Info.plist: ключ "Main storyboard file base name" (UIMainStoryboardFile) указывает на "Main_iPhone", а ключ "Main storyboard file base name (iPad)" (UIMainStoryboardFile~ipad) указывает на "Main_iPad".

Более подробно процесс запуска приложения и главную раскадровку мы рассмотрим позднее в этой главе. Редактирование файлов .xib и .storyboard, а также процесс создания экземпляров в ходе выполнения приложения описываются в главе 7.

Дополнительные ресурсы

Ресурсы — это вспомогательные файлы, встроенные в комплект вашего приложения, для того чтобы извлечь их при выполнении приложения. К ним относятся, например, изображения, которые вы хотите вывести на экран, или звуки, которые хотите воспроизвести. В реальном приложении бывает много таких дополнительных ресурсов. Для того чтобы добавить эти ресурсы в проект, необходимо, чтобы они были указаны в фазе сборки Copy Bundle Resources. Доступ к этим ресурсам во время выполнения приложения обычно должен обеспечивать ваш код (или код, подразумеваемый при загрузке nib-файла): в принципе среда выполнения приложения просто открывает ваш комплект приложения и извлекает из него требуемый ресурс. Фактически ваш комплект приложения рассматривается как папка, заполненная дополнительными файлами.

Для того чтобы добавить ресурс в свой проект, откройте навигатор проекта и выберите команду File⇨Add Files to [Project] или перетащите ресурс из окна Finder в навигатор проекта. При этом откроется диалоговое окно (рис. 6.16), в котором необходимо установить следующие настройки.

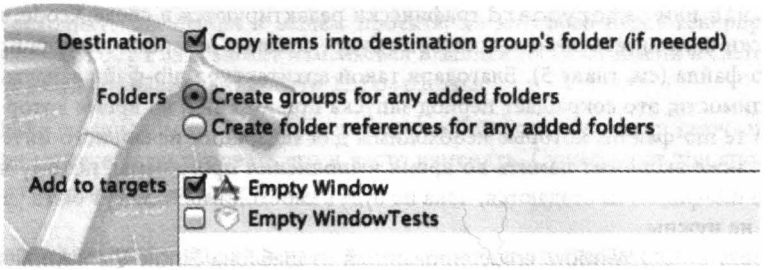


Рис. 6.16. Опции при добавлении ресурса в проект

Copy items into destination group's folder (if needed)

Этот флаг следует устанавливать практически во всех случаях. Он означает копирование ресурса в папку проекта. Если оставить этот флаг сброшенным, ваш проект будет искать файл, расположенный за пределами папки проекта, где он может быть непреднамеренно удален или изменен. Хранение всех ресурсов в папке проекта намного безопаснее.

Folders

Переключатели из этого раздела устанавливаются только тогда, когда вы добавляете в проект папку; разница между ними заключается только в том, как проект ссылается на содержимое этой папки.

Create groups for any added folders

Имя папки становится именем обычной группы в навигаторе проекта; содержимое папки появляется в этой группе, но при этом его элементы перечисляются в фазе сборки Copy Bundle Resources индивидуально, так что по умолчанию они будут копироваться по отдельности на верхний уровень комплекта приложения.

Create folder references for any added folders

Папка изображается синим цветом в навигаторе проекта (этим цветом выделяются ссылки на папки); более того, она перечисляется в фазе сборки Copy Bundle Resources как папка, т.е. в процессе сборки она будет скопирована в комплект приложения как целая папка вместе с ее содержимым. Это значит, что ресурсы внутри папки должны находиться не на верхнем уровне комплекта, а во вложенной папке. Такая организация целесообразна, если у вас много ресурсов и вы хотите разделить их на категории (а не сваливать их в одну кучу на верхнем уровне комплекта приложения) или если иерархия файлов играет важную роль в вашем приложении. Недостатком такой организации является то, что код, написанный вами для доступа к ресурсу, должен быть точно настроен на вложенную папку, содержащую данный ресурс.

Add to Targets

Установка этого флага для цели означает, что ресурс будет добавлен в фазу сборки Copy Bundle Resources данной цели. Таким образом, вы, скорее всего, хотите проверить его в своем целевом приложении. А зачем же еще добавлять ресурс в свой проект? Если же этот флаг будет случайно сброшен и вы позднее поймете, что ресурс, указанный в навигаторе проекта, должен быть добавлен в фазу сборки Copy Bundle Resources для конкретной цели, то сможете сделать это вручную, как было описано ранее.

Изображения для программ iOS обычно образуют пары: одно изображение для экрана с обычным разрешением, а второе — для экрана с двойным разрешением. Для того чтобы обеспечить правильную работу методов загрузки изображений, такие пары получают специальные имена: например `listen_normal.png` и `listen_normal@2x.png`, где суффикс `@2x` во втором имени файла означает, что он является вариантом первого файла, предназначенным для экрана с двойным разрешением. Это может вызвать путаницу среди графических файлов в навигаторе проекта. Именно этот недостаток призваны устранить новые средства Xcode 5 — каталоги ресурсов.

Вместо того чтобы добавить файл `listen_normal.png` в свой проект так, как описано выше, можно использовать каталог ресурсов. Я использовал каталог ресурсов, заданный по умолчанию, — `Images.xcassets`. Для того чтобы отредактировать каталог, заданный по умолчанию, я щелкнул на кнопке **Plus** в нижней части первого столбца и выбрал команду **New Image Set**. В результате возник шаблон `Image`, предоставляющий место для изображения с одинарным разрешением и место для изображения с двойным разрешением. Затем я перетащил туда два изображения, `listen_normal.png` и `listen_normal@2x.png`, и они автоматически заняли свои места. Более того, в этом случае диалоговое окно не открывается (как на рис. 6.16); изображения автоматически копируются в папку проекта (в папку с каталогом ресурсов), цель для изображений указывать не обязательно, потому что они являются частью каталога ресурсов, который уже имеет связь с целью. В заключение я могу дать шаблону более осмысленное имя, чем `Image`, и более простое, чем `listen_normal`; назовем его `Listen` (рис. 6.17).

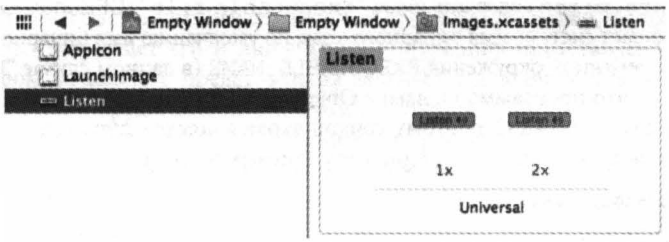


Рис. 6.17. Пара изображений, добавленная в каталог активов

В результате моя программа теперь может загружать правильное изображение, соответствующее текущему разрешению экрана, используя его псевдоним `@"Listen"`, не упоминая оригинальное имя (или расширение) графического файла. Более того, эти изображения не обязаны храниться порознь в комплекте приложения; для приложений iOS 7 они могут оставаться в скомпилированном каталоге ресурсов.

Другое преимущество каталогов ресурсов состоит в том, что они не обязаны подчиняться соглашениям о названиях файлов. Допустим, у меня есть два изображения, `little.png` и `big.png`, причем второе изображение предназначено для экрана с двойным разрешением. Мы не используем суффикс `@2x`, поэтому каталог ресурсов не может распознать. что эти два файла связаны друг с другом. Но я могу сообщить ему об этом, перетащив файл `little.png` на место изображения с одинарным разрешением, и файл `big.png` на место для изображения с двойным разрешением.

Элементы каталога ресурсов можно проверять, выбирая изображения и используя инспектор атрибутов (<Command+Option+4>). В результате на экране будут показаны имя и размер изображения. Например, в описанном выше случае, проверяя вариант изображения с одинарным разрешением, я узнаю, что оно называется `little.png` и в два раза меньше, чем его аналог с двойным разрешением `big.png`.

Кодирование и запуск приложения

В процессе сборки исходные файлы, подлежащие компиляции для создания бинарного приложения, известны, потому что они перечисляются в фазе сборки целевого приложения Compile Sources. (В этом случае наш проект Empty Window содержит файлы `ViewController.m`, `AppDelegate.m` и `main.m`.) Кроме того, в ходе компиляции используется предварительно скомпилированный заголовок; фактически он обрабатывается до всех остальных исходных файлов. На фазе сборки Compile Sources этот файл не указывается; цель знает о нем, потому что он указан в настройках сборки Prefix Header. (В проекте Empty Window этот файл называется `Empty Window-Prefix.pch`.)

Предварительно скомпилированный заголовок — это средство ускорения компиляции. Это заголовочный файл; он компилируется только один раз (или, по крайней мере, очень редко), а результаты компиляции кешируются (в папке `DerivedData`) и неявно импортируются во все исходные файлы. Таким образом, предварительно скомпилированные заголовки, которые никогда не изменяются (например, встроенные заголовки каркаса Cocoa), должны состоять в основном из директив `#import`; это удобное место для включения директивы `#defines`, которая никогда не изменяется и должна использоваться всеми исходными файлами, как указано в главе 4. По умолчанию предварительно скомпилированный заголовочный файл импортирует файлы `Foundation.h` (заголовочный файл каркаса Core Foundation) и `UIKit.h` (заголовочный файл каркаса Cocoa). Они будут рассмотрены в следующем разделе.

При запуске приложения система знает, где найти бинарный код в комплекте приложения, потому что файл `Info.plist` содержит ключ "`Executable file`" (CFBundleExecutable), значение которого сообщает системе имя бинарного кода; по умолчанию имя бинарного кода определяется значением переменной окружения `EXECUTABLE_NAME` (в данном случае "`Empty Window`").

Приложение — это программа на языке Objective-C. В свою очередь, язык Objective-C является подмножеством языка C, поэтому точкой входа является функция `main`. Эта функция определена в файле проекта `main.m`. Приведем пример функции `main`:

```
int main(int argc, char *argv[])
{
    @autoreleasepool { return UIApplicationMain(argc, argv, nil,
                                    NSStringFromClass([AppDelegate class]));
    }
}
```

Функция `main` делает две вещи.

- Настраивает окружение системы управления памятью с помощью директивы `@autoreleasepool` и фигурных скобок, которые ее сопровождают. Их смысл разъясняется в главе 12.
- Вызывает функцию `UIApplicationMain`, которая разворачивает ваше приложение и запускает его.

Функция `UIApplicationMain` решает несколько сложных проблем. Где взять первичные экземпляры для вашего приложения? Какие методы экземпляра следует вызвать из этих экземпляров? Где ваше приложение должно искать начальный интерфейс? На эти вопросы отвечает функция `UIApplicationMain`.

1. Функция `UIApplicationMain` создает первый экземпляр — общий экземпляр приложения (shared application instance), к которому можно обращаться из кода с помощью вызова `[UIApplication sharedApplication]`. Третий аргумент функции

UIApplicationMain, представляющий собой строку, задает имя класса, к которому должен относиться общий экземпляр вашего приложения. Если этот параметр равен nil, что является довольно распространенным случаем, по умолчанию используется класс UIApplication. Если по каким-то причинам вам нужен подкласс класса UIApplication, вы должны указать этот подкласс здесь, выполнив замену (она зависит от того, как называется подкласс) третьего аргумента в вызове функции UIApplicationMain.

```
NSStringFromClass([MyUIApplicationSubclass class])
```

2. Функция UIApplicationMain создает второй экземпляр — делегата экземпляра приложения. Делегирование — важный шаблон каркаса Cocoa, подробно описанный в главе 11. Очень важно, чтобы каждое приложение, созданное вами, имело экземпляр делегата приложения. Четвертый аргумент в вызове функции UIApplicationMain задает в виде строки имя класса, к которому должен относиться экземпляр делегата приложения. В нашем файле эта спецификация выглядит следующим образом:

```
NSStringFromClass([AppDelegate class])
```

   Она приказывает функции UIApplicationMain создать экземпляр класса AppDelegate и связать его с общим экземпляром приложения в качестве его делегата. Исходные файлы этого класса, AppDelegate.h и AppDelegate.m, создаются шаблоном; разумеется, вы можете (и, вероятно, будете) редактировать этот код.

3. Если в файле Info.plist указан главный файл раскадровки, функция UIApplicationMain загружает его и ищет в нем контроллер представления, назначенный контроллером начального представления данной раскадровки; она создает экземпляр этого контроллера представления, тем самым создавая третий экземпляр приложения. В случае проекта Empty Window, созданного по шаблону Single View Application, этим контроллером представления является экземпляр класса ViewController; исходные файлы, определяющие этот класс, ViewController.h и ViewController.m, также создаются шаблоном.

4. Если главный файл раскадровки существует, то функция UIApplicationMain создает окно приложения, создавая экземпляр класса UIWindow, — это уже четвертый экземпляр приложения. Функция назначает этот экземпляр окна свойству делегата window; она также назначает начальный экземпляр контроллера представления корневым контроллером представления этого экземпляра окна (свойство rootViewController).

   (Я упрощаю. На самом деле функция UIApplicationMain сначала дает экземпляру делегата приложения возможность создать экземпляр окна, вызвав свой метод window. Именно пользовательский код создает экземпляр подкласса класса UIWindow.)

5. Затем функция UIApplicationMain переключается на экземпляр делегата приложения и начинает вызывать его методы — в частности, метод application:didFinishLaunchingWithOptions:. Это дает возможность выполнить ваш код! Таким образом, метод application:didFinishLaunchingWithOptions: — хорошее место для размещения вашего кода, инициализирующего значения и выполняющего запуск, но при этом этот код должен выполняться быстро, потому что ваш интерфейс еще не появился на экране.

   (Я снова упрощаю. Начиная с версии iOS 6, последовательность вызовов кода делегата приложения на само деле начинается с метода application:willFinishLaunchingWithOptions:, если он существует.)

6. Если главная раскадровка существует, то функция `UIApplicationMain` выводит на экран интерфейс приложения. Для этого она вызывает метод экземпляра `makeKeyAndVisible` класса `UIWindow`.
7. Теперь окно готово появиться на экране. Это, в свою очередь, заставляет окно обратиться к своему корневому контроллеру представления и запросить у него главное представление, которое будет занимать окно. Если контроллер представления получает свое главное представление из файлов `.storyboard` или `.xib`, то загружается nib-файл; его объекты создаются, инициализируются и становятся объектами начального интерфейса.

Теперь приложение запускается и становится видимым пользователю. Оно имеет несколько экземпляров — как минимум, общий экземпляр приложения, окно, контроллер начального представления и представление контроллера начального представления и все, что содержат объекты интерфейса. Часть вашего кода (метод делегата `application:didFinishLaunchingWithOptions:`) уже выполнена, а метод `UIApplicationMain` продолжает выполняться (метод `UIApplicationMain` никогда ничего не возвращает), отслеживая действия пользователя и поддерживая цикл событий, возникающих вследствие действий пользователя.

Описывая процесс запуска приложения, я несколько раз использовал выражение "если главная раскадровка существует." В большинстве шаблонов Xcode 5, таких как Single View Application, который мы использовали для создания проекта Empty Window, главная раскадровка существует. Впрочем, шаблон может не содержать главную раскадровку. В этом случае такие действия, как создание экземпляра окна, назначение корневого контроллера представления, назначение свойства `window` делегата приложения и вызов метода `makeKeyAndVisible` из окна для демонстрации интерфейса, должен выполнять ваш код.

Для того чтобы понять, что я имею в виду, создайте новый проект для iPhone на основе шаблона Empty Application; назовем его Truly Empty. Он имеет класс `App Delegate`, но не имеет ни раскадровки, ни контроллера начального представления. Метод делегата приложения `application:didFinishLaunchingWithOptions:` в файле `AppDelegate.m` содержит код для создания окна, присваивает это окно свойству делегата приложения `window`.

```
self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
// Точка замещения на настройки после запуска приложения.
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
```

После сборки и запуска это приложение выводит пустое белое окно; но среда выполнения приложений выводит на консоль предупреждение "`Application windows are expected to have a root view controller at the end of application launch`" ("В конце процесса запуска окна приложения должны иметь корневой контроллер представления"). Это предупреждение можно предотвратить, создав контроллер представления и присвоив его свойству окна `rootViewController`.

```
self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
// Точка замещения на настройки после запуска приложения.
self.window.rootViewController = [UIViewController new]; // предотвращаем предупреждение
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
```

Все работает: при сборке и запуске приложения предупреждение больше не появляется. Однако у нас пока нет разумного способа настройки поведения приложения, так как нет

подкласса класса UIViewController. Более того, у нас нет возможности для графического создания интерфейса; нет ни файла .storyboard, ни файла .xib. Обе эти проблемы можно решить, создав подкласс класса UIViewController вместе с файлом .xib.

1. Выберите команду File⇨New⇨File. В диалоговом окне Choose a template в разделе iOS щелкните на пункте Cocoa Touch слева, выберите пункт Objective-C Class и щелкните на кнопке Next.
2. Присвойте классу имя ViewController и укажите, что он является подклассом класса UIViewController. Установите флаг With XIB for user interface. Щелкните на кнопке Next.
3. Откроется диалоговое окно Save. Убедитесь, что вы сохраняете проект в папке Truly Empty, что вплывающее меню Group также настроено на приложение Truly Empty и что выбрана цель Truly Empty, — мы хотим, чтобы эти файлы были частью целевого приложения. Щелкните на кнопке Create.

   Среда Xcode создала для нас три файла: ViewController.h и ViewController.m, определив ViewController как подкласс класса UIViewController, а также файл ViewController.xib, из которого экземпляр класса ViewController автоматически получит свое представление по умолчанию.
4. Теперь вернемся к методу делегата приложения application:didFinishLaunchingWithOptions: в файле AppDelegate.m и изменим класс контроллера корневого представления ViewController; кроме того, в начало файла поместим директиву #import "ViewController.h".

```
self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
// Точка замещения на настройки после запуска приложения.
self.window.rootViewController = [ViewController new]; // наш подкласс
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
```

Таким образом, мы создали превосходный минимальный проект без раскадровки. Наш код выполняет работу, которую при наличии главной раскадровки класс UIApplicationMain выполняет автоматически: мы создаем экземпляр класса UIWindow, задаем экземпляр окна как свойство делегата window, создаем экземпляр контроллера начального представления, задаем экземпляр контроллера представления как свойство окна rootViewController и открываем окно на экране. Более того, появление окна автоматически заставляет экземпляр класса ViewController извлечь свое представление из nib-файла ViewController, который был скомпилирован из файла ViewController.xib; таким образом, мы можем использовать файл ViewController.xib для настройки начального интерфейса приложения.

Каркасы и пакеты SDK

Каркас — это библиотека скомпилированного кода, который используется вашей программой. Большинство каркасов, использующихся при разработке приложения для системы iOS, являются встроенными каркасами компании Apple. Во время выполнения программы эти каркасы уже являются частью системы, установленной на устройстве; они находятся в папке /System/Library/Frameworks, расположенной на устройстве, но увидеть их на устройствах iPhone или iPad невозможно, потому что в обычном режиме увидеть непосредственно файловую иерархию нельзя.

Ваш скомпилированный код также должен быть связан с этими каркасами во время сборки проекта на вашем компьютере. Для того чтобы это было возможно, папка `System/Library/Frameworks` с устройства iOS дублируется на компьютере в самой среде Xcode. Это продублированное подмножество системы, установленной на устройстве, называется пакетом SDK (software development kit — пакет инструментальных средств для разработки программного обеспечения). То, как используется пакет SDK, зависит от того, для какого устройства вы разрабатываете приложение.

Связывание — это процесс соединения вашего скомпилированного кода с необходимыми каркасами, несмотря на то, что во время сборки каркасы находятся в одном месте, а во время выполнения приложения — в другом.

Когда вы собираете ваш код для запуска на устройстве

Используется копия необходимых каркасов. Эта копия находится в папке `System/Library/Frameworks/` в пакете iPhone SDK, который хранится в папке `Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS7.0.sdk`.

Когда ваш код выполняется на устройстве

После запуска код ищет необходимые каркасы на устройстве в папке `/System/Library/Frameworks/`.

Таким образом, каркасы являются частью искусного механизма Apple для инкорпорации кода в выполняемое приложение. Каркасы содержат все, что необходимо для работы приложений; совокупность этих элементов называется Cocoa. Этих элементов очень много. Ваше приложение может использовать мощь каркасов благодаря связи с ними. Ваш код работает так, будто каркас инкорпорирован в него. Даже если ваше приложение относительно маленькое, его каркасы могут быть огромными.

Связывание обеспечивает соединение скомпилированного кода с необходимыми каркасами, но для успеха этого недостаточно. Каркасы заполнены классами (например, `NSString`) и методами (например, `uppercaseString`), которые ваш код будет вызывать. Для того чтобы удовлетворить требования компилятора, каркасы делают то, что делает любая программа на языке C или Objective-C: они публикуют свой интерфейс в заголовочных файлах (`.h`), которые импортируются вашим кодом. Таким образом, например, ваш код может обратиться к классу `NSString` и вызвать функцию `uppercaseString`, потому что он импортирует заголовочный файл `NSString.h`. На самом деле ваш код импортирует заголовочный файл `UIKit.h`, который в свою очередь импортирует файл `Foundation.h`, который в свою очередь импортирует файл `NSString.h`. Это можно увидеть в первой строке вашего кода:

```
#import <UIKit/UIKit.h>
```

(Импорт файлов классов и его предназначение описан в примере 4.1.)

Таким образом, каркас представляет собой двухэтапный процесс.

Импорт заголовка каркаса

Эта информация необходима вашему коду для успешной компиляции. Ваш код импортирует заголовок каркаса с помощью директивы `#import`, либо непосредственно указывая заголовок каркаса, либо импортируя заголовок, который в свою очередь импортирует заголовок каркаса.

Связывание с каркасом

Скомпилированный исполняемый бинарный код должен быть связан с каркасами, которые будут использоваться во время выполнения приложения, на самом деле инкорпорируя скомпилированный код, образующий эти каркасы. После сборки код связывается со всеми необходимыми каркасами в соответствии со списком каркасов, заданном на фазе сборки цели Link Binary With Libraries.

По умолчанию шаблон уже связал три каркаса с вашим целевым приложением. Он сделал это, перечислив их на фазе сборки Link Binary With Libraries (затем каркасы появляются также в окне навигатора проекта в группе Frameworks, и с помощью этого списка можно просматривать и проверять заголовочные файлы).

Каркас `UIKit`

Классы Cocoa, специализированные для системы iOS, имена которых начинаются с букв "UI," являются частью каркаса `UIKit`. Каркас `UIKit` импортируется (`<UIKit/UIKit.h>`) в предварительно скомпилированный заголовочный файл и в заголовочные файлы классов, образующих шаблон приложения, такие как `AppDelegate.h`, а также в файлы классов, которые пишете вы.

Каркас `Foundation`

Многие классы каркаса Cocoa, такие как `NSString` и `NSArray`, а также другие файлы, имена которых начинаются с букв "NS", являются частью каркаса Foundation framework. Этот каркас импортируется в предварительно скомпилированный заголовочный файл, но делать это не обязательно, потому что многие заголовки, импортируемые файлом `UIKit.h`, импортируют каркас Foundation (`<Foundation/Foundation.h>`). В свою очередь, файл `Foundation.h` включает заголовки каркаса Core Foundation (`<CoreFoundation/CoreFoundation.h>`) и загружает каркас Core Foundation как вложенный каркас; таким образом, нет необходимости явно импортировать Core Foundation (наполненный функциями и типами указателей, имена которых начинаются с букв "CF", например `CFStringRef`) или связывать с ним свой код.

Каркас `Core Graphics`

Каркас Core Graphics определяет множество структур и функций. связанных с рисованием, имена которых начинаются с букв "CG". Он импортируется многими заголовками каркаса `UIKit`, поэтому необязательно импортировать его явно.

Система iOS имеет около пятидесяти каркасов, но шаблоны не импортируют их в ваш код и не связывают с вашей целью. Это объясняется тем, что дополнительные каркасы означают дополнительную работу как для компиляции, так и для запуска приложения. Следовательно, для того чтобы использовать эти дополнительные каркасы, вы должны выполнить оба этапа: импортировать заголовки каркасов там, где это необходимо, и связать ваш код с самими каркасами. Этот двухэтапный процесс довольно неудобен, но если вы пропустите один из этапов, то ничего не получится.

Например, допустим, что нас интересует каркас Address Book и мы редко используем его в нашем приложении. Тогда в нашем коде мы создадим объект

```
ABNewPersonViewController:ABNewPersonViewController* ab =
    [ABNewPersonViewController new];
```

В следующий раз, кода мы будем собирать приложение, компилятор выдаст сообщение о том, что ABNewPersonViewController — необъявленный идентификатор. Тогда мы поймем, что нужно импортировать заголовок каркаса. Пара букв, с которых начинается имя ABNewPersonViewController, может служить подсказкой для поиска дополнительного каркаса; оно начинается не с букв "NS", "UI" или "CG". В документации к классу ABNewPersonViewController сказано, что он принадлежит каркасу AddressBookUI. Вы можете догадаться (и правильно), что в начале файла реализации следует поместить директиву импортирования каркаса:

```
#import <AddressBookUI/AddressBookUI.h>
```

Это позволяет предотвратить предупреждения компилятора, но код собран не будет. На этот раз мы получим сообщение об ошибке на этапе редактирования связей "Symbol(s) not found", связанное с именем _OBJC_CLASS_$_ABNewPersonViewController. Это загадочное сообщение просто означает, что мы забыли о втором этапе: мы должны связать нашу цель с каркасом AddressBookUI.framework.

Для того чтобы связать цель с каркасом, необходимо отредактировать цель, щелкнуть на пункте General в верхней части редактора и прокрутить раздел Linked Frameworks and Libraries. (Эту же информацию можно получить, щелкнув на пункте Build Phases в верхней части редактора и открыв фазу сборки Link Binary With Libraries.) Щелкните на кнопке Plus, расположенной слева непосредственно под списком каркасов. Появляется диалоговое окно, в котором перечисляются существующие каркасы, являющиеся частью активного пакета SDK. Выберите каркас AddressBookUI.framework и щелкните на кнопке Add. Каркас AddressBookUI добавляется в фазу сборки Link Binary With Libraries. (Она появляется в навигаторе проекта.) Теперь можно собирать (и выполнять) приложение.

Начиная с версий Xcode 5 и LLVM 5.0, можно упростить подключение дополнительных каркасом с помощью модулей. Использование модулей регулируется настройкой сборки Enable Modules (C and Objective-C). По умолчанию этот параметр равен Yes для новых проектов, созданных из шаблонов приложения.

Модули — это кешированная информация, хранящаяся на компьютере в папке Library/Developer/Xcode/DerivedData/ModuleCache. Когда вы собираете приложение, импортирующее заголовок каркаса, а информация об этом каркасе не кеширована в модуле, она моментально кешируется в модуле. Перечислим преимущества использования модулей.

После предварительной компиляции получается код меньшего размера

После выполнения предварительной компиляции все, что было импортировано, буквально копируется в ваш код. Заголовки каркаса UIKit вместе с заголовками, которые они импортируют, состоят из более чем 30 000 строк кода. Это значит, что для компиляции любого файла .m компилятор должен работать с файлом, который на 30 000 строк длиннее, чем ваша программа. Однако, используя модули, мы оставляем информацию об импортируемом заголовке внутри модуля, и длина кода в результате предварительной компиляции увеличивается незначительно. Это также может ускорить процесс компиляции.

Более простые и короткие директивы импорта

Импортирование заголовка может оказаться неудобным. Имя заголовка должно быть в угловых скобках, причем нужно указать не только имя каркаса (которое фактически является именем папки), но и имя заголовочного файла.

```
#import <AddressBookUI/AddressBookUI.h>
```

С помощью модулей вместо директивы `#import` можно использовать новую директиву `@import` (перед новой директивой `import` стоит символ `@`, а не `#`), в которой достаточно указать лишь имя каркаса и точку с запятой.

```
@import AddressBookUI;
```

Более того, директивы `#import` автоматически конвертируются в директивы `@imports`, поэтому программист может писать как угодно, и существующие программы, в которых используются директивы `#import`, будут работать по-прежнему, но используя преимущества модулей.

Автосвязывание

Использование каркасов — двухэтапный процесс. Необходимо не только импортировать заголовок, но и связать код с каркасом. Модули позволяют пропустить второй этап. Это удобство можно использовать или не использовать (с помощью настройки сборки Link Frameworks Automatically), но по умолчанию оно используется. В результате, если код импортирует заголовок каркаса, ваша цель не обязательно будет связана с каркасом. Связывание будет выполняться автоматически во время процесса сборки.

Модули — это искусный и удобный механизм, но у них есть и недостатки по сравнению со старым способом явного импортирования и связывания. Например, когда каркас явно связывается с вашим проектом, вы знаете его, потому что он перечислен в фазе сборки Link Binary With Libraries и в навигаторе проекта. Применяя модули, вы не знаете, какие каркасы используете; у вас нет списка каркасов, которые автоматически связываются с кодом.

Более того, поскольку автоматически связываемые каркасы не перечислены в навигаторе проекта, их заголовки невозможно искать и просматривать в навигаторе проекта (в то время как явно связываемые каркасы это допускали с помощью области видимости "within workspace and frameworks" ("в рабочем пространстве и каркасах")).

К счастью, если вы не воспользуетесь этой возможностью, то сможете связаться с каркасом вручную и включить его в список каркасов в навигаторе проекта, даже если он связывается автоматически.

ГЛАВА 7

Управление nib-файлами

Термин *nib* — это сокращение от "NeXTStep Interface Builder", которое использовалось в качестве расширения файлов `.nib`. Каждое приложение для системы iOS содержит хотя бы один nib-файл. Этот файл генерируется либо из файла `.xib`, либо из файла `.storyboard` в проекте Xcode (см. главу 6). Если вы редактируете файл `.xib` или `.storyboard` в среде Xcode, то оказываетесь в графическом редакторе; это выглядит так, будто часть своего интерфейса вы разрабатываете графически, подобно тому, как вы рисуете диаграммы с помощью программ Adobe Illustrator, OmniGraffle, Canvas или любого другого аналогичного приложения. Я называю этот компонент среды Xcode nib-редактором.

До появления версии Xcode 3.2.x nib-редактирование осуществлялось в отдельном приложении, которое называлось Interface Builder. Начиная с версии Xcode 4 функциональные возможности программы Interface Builder были реализованы в самой среде Xcode. Тем не менее nib-редактор в среде Xcode часто по-прежнему называют Interface Builder.

Работая с nib-редактором, вы на самом деле не создаете рисунков. Вы кодируете инструкции по созданию, инициализации и конфигурированию объектов. Как было сказано в главе 5 (см. раздел "Создание экземпляров на основе nib-файлов), когда вы перетаскиваете кнопку на представление в nib-редакторе, отпускаете эту кнопку в определенном месте, задаете ее размеры и дважды щелкаете на ее заголовке, вводя строку "`Howdy!`", вы вводите инструкции в nib-файл, который будет генерировать экземпляр класса `UIButton` и задавать свойства его рамки и заголовка, а также создавать дочернее представление в представлении, которое вы перетащили в него, выполняя весь процесс создания, инициализации и конфигурирования кнопки.

```
UIButton* b =
   [UIButton buttonWithType:UIButtonTypeSystem];       // создаем
[b setTitle:@"Howdy!" forState:UIControlStateNormal]; // задаем заголовок
[b setFrame: CGRectMake(100,100,52,30)];               // задаем рамку
[self.view addSubview:b];                              // помещаем в представление
```

Инструкции, которые вы создали, работая в nib-редакторе, включаются в nib-файл в собранном приложении. Эти инструкции, по существу, являются коллекцией потенциальных объектов, которые называются nib-объектами. Эти потенциальные объекты, закодированные в nib-файле, не превращаются в реальные объекты, пока nib-файл не будет загружен в ходе выполнения приложения. В этот момент инструкции, записанные в nib-файле, выполняются механизмов загрузки nib-файлов и объекты превращаются в реальные экземпляры, готовые стать частью выполняемого приложения. Большей частью это объекты интерфейса, поэтому

чаще всего они проявляются как представления, являющиеся дочерними по отношению к представлению, которое видит пользователь.

Таким образом, программирование nib-интерфейса связано с тремя вопросами.

- Как редактировать nib-интерфейс с помощью файлов `.xib` и `.storyboard` в среде Xcode?
- Как загрузить nib-интерфейс во время выполнения приложения?
- Как приложение может использовать экземпляры, которые генерирует nib-интерфейс после загрузки?

Эти связанные друг с другом вопросы являются предметом рассмотрения данной главы.

Нужны ли nib-файлы?

Поскольку nib-файлы являются всего лишь источниками экземпляров, может возникнуть вопрос: "А нельзя ли обойтись без них?" Если эти же экземпляры можно генерировать в коде, значит, можно обойтись без nib-файлов? Ответ простой: "Да, можно". Можно написать сложное приложение, в котором не будет файлов `.storyboard` или `.xib` (я писал такие приложения). Однако с практической точки зрения это вопрос баланса. Большинство приложений используют nib-файлы как источник по крайней мере нескольких объектов интерфейса; однако существуют аспекты объектов интерфейса, которые можно настроить только в коде, и иногда проще сгенерировать эти объекты без помощи nib-файлов. В реальной жизни ваши проекты, вероятно, будут часто использовать связь между кодом и объектами, сгенерированными на основе nib-файлов.

Обзор интерфейса nib-редактора

Рассмотрим интерфейс nib-редактора в среде Xcode. В главе 6 мы создали проект Empty Window для устройства iPhone на основе шаблона Single View Application; он содержит файл раскадровки, поэтому мы будем его использовать. Откройте в среде Xcode проект Empty Window, найдите файл `Main.storyboard` в навигаторе проекта и щелкните на нем, чтобы отредактировать.

На рис. 7.1 показано окно проекта после выбора файла `Main.storyboard` и внесения некоторых корректировок. Панель навигатора скрыта; на рисунке показана панель утилит, содержащая инспектор размеров и библиотеку объектов. Интерфейс состоит из четырех частей.

1. Левая часть редактора представляет собой структуру документа (document outline), в которой в иерархическом порядке по именам перечисляется содержимое раскадровки. Ее можно скрыть, перетащив ее правый край или щелкнув на треугольнике в правом нижнем углу.
2. Остальная часть редактора определяет канву (canvas), на которой физически проектируется интерфейс приложения. На этой канве отображаются представления интерфейса приложения, а также сущности, которые могут содержать представления. Представление (view) — это объект интерфейса, который прорисовывается в прямоугольной области. Фраза "сущности, которые могут содержать представления", с моей точки зрения, относится и к контроллерам представления, которые существуют на канве, хотя и не прорисовываются в интерфейсе; контроллер представления — это не разновидность представления, но он владеет представлением (и управляет им).

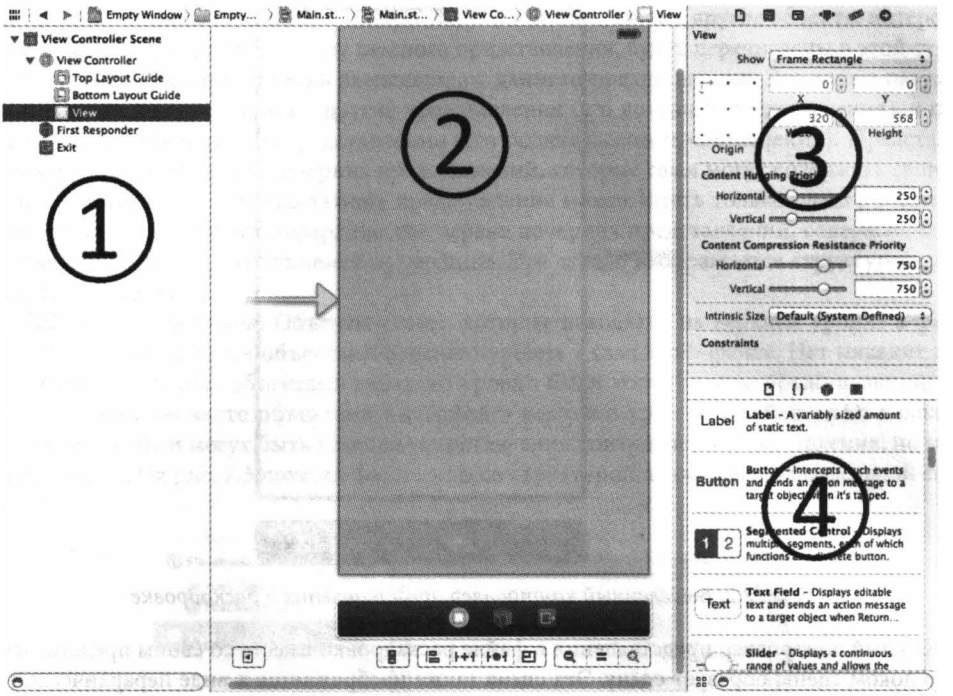


Рис. 7.1. Редактирование nib-файла

3. Инспекторы на панели утилит, позволяющие редактировать детали выбранных объектов.
4. Библиотеки на панели утилит, особенно библиотека объектов, служащая источником объектов интерфейса, которые добавляются в nib-файл.

Структура документа

Структура документа иерархически отображает отношения между объектами в nib-файле и немного отличается в зависимости от того, какой файл редактируется — `.storyboard` или `.xib`.

Основным содержанием файла раскадровки являются сцены (scenes). Грубо говоря, сцена — это контроллер одного представления и некоторый вспомогательный материал; на верхнем уровне каждой сцены находится контроллер одного представления.

Контроллер представления — это не объект интерфейса, но он управляет объектом интерфейса, а именно его представлением (или главным представлением). Контроллер представления в nib-файле не обязан иметь свое главное представление в том же самом nib-файле, но обычно так и происходит, и в этом случае на канве в nib-редакторе представление обычно появляется в контроллере представления. Так, на рис. 7.1 большой голубой прямоугольник на канве — это главное представление контроллера, которое находится в этом контроллере. Сам контроллер изображается на канве в виде прямоугольника, содержащего представление вместе с черным закругленным прямоугольником, расположенным под ним. Этот закругленный прямоугольник называется доком сцены (scene dock). Его легче увидеть, если выделить контроллер представления, как показано на рис. 7.2: контроллер представления и его док сцены выделяются вместе и окаймляются жирной голубой линией.

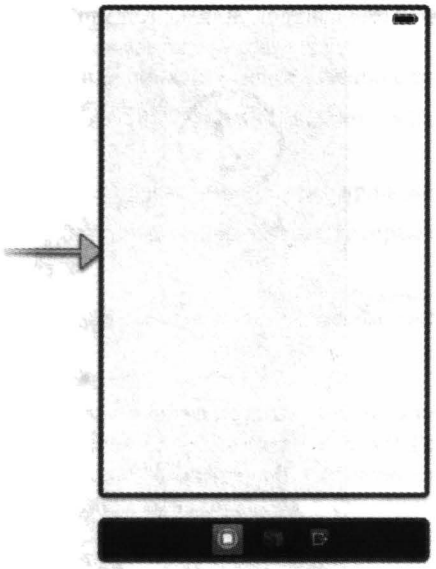


Рис. 7.2. Выделенный контроллер представления в раскадровке

Каждый контроллер представления в файле раскадровки вместе со своим представлением и его доком сцены образует сцену. Эта сцена также изображается в виде иерархической коллекции имен в структуре документа. На верхнем уровне структуры документа находятся сами сцены. Верхний уровень каждой сцены состоит из имен, обозначающих объекты в доке сцены контроллера представления: сам контроллер представления, два прокси-объекта, токены First Responder и Exit. Эти объекты представлены в доке сцены в виде пиктограмм и отображаются на верхнем уровне сцены в структуре документа.

Объекты, перечисленные в структуре документа, разделяются на два вида.

Nib-объекты

Контроллер вместе с его главным представлением и всеми дочерними представлениями, которые должны находиться в этом представлении, — это потенциальные объекты, которые превращаются в реальные объекты при загрузке nib-файла во время выполнения приложения. Такие реальные объекты, которые должны создаваться на основе nib-файла, также называются nib-объектами.

Прокси-объекты

Прокси-объекты (в данном случае токены First Responder и Exit) не представляют собой экземпляры, которые появляются из nib-файла при его загрузке. Они представляют вспомогательные объекты, предназначенные для обеспечения связи между nib-объектами и другими объектами (примеры будут приведены чуть позже). Прокси-объекты нельзя создавать и удалять, так как nib-редактор делает это автоматически.

Помимо объектов верхнего уровня, большинство объектов, перечисленных в структуре документа раскадровки, иерархически зависят от контроллера представления сцены. Например, на рис. 7.1 контроллер имеет главное представление (оно показано как представление, иерархически зависящее от контроллера представления). Это разумно, потому что это

представление принадлежит этому контроллеру. Более того, любые другие объекты интерфейса, которые мы перетащим на канву главного представления, будут перечислены в этой структуре документа как иерархически зависящие от данного представления. Это также разумно. Представление может содержать другие представления (его дочерние представления), и само может содержаться в другом представлении (его родительском представлении). Представление может содержать много дочерних представлений, которые сами могут содержать свои дочерние представления. Однако каждое представление может иметь только одного родителя. Таким образом, существует иерархическое дерево дочерних представлений, содержащихся в единственном объекте, находящемся на вершине. Это дерево изображается структурой документа (как структура!).

В файле `.xib` нет сцен. Объекты сцены, которые находятся на верхнем уровне в файле `.storyboard`, становятся объектами верхнего уровня в самом nib-файле. Нет никаких правил, требующих, чтобы объектами верхнего уровня были контроллеры представлений; они могут ими быть, но часто объектами интерфейса верхнего уровня в файле `.xib` являются представления. Ими могут быть главные представления контроллера представления, но и это не обязательно. На рис. 7.3 показан файл `.xib` со структурой, параллельной отдельной сцене на рис. 7.1.

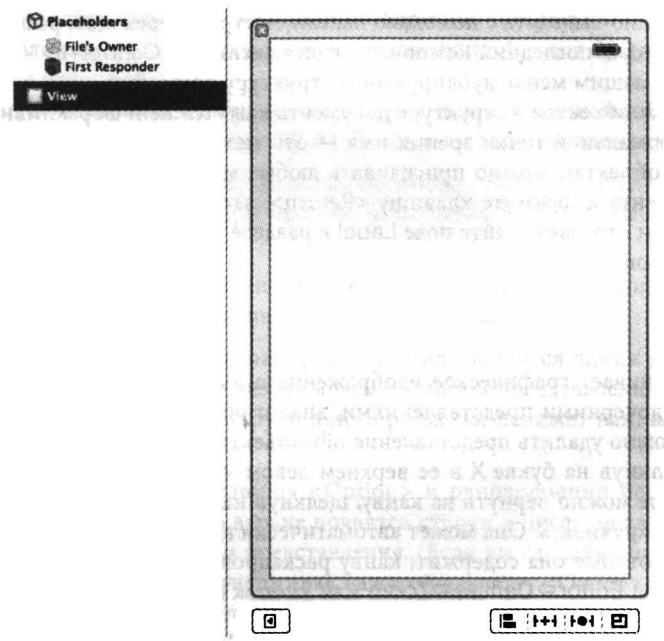


Рис. 7.3. Файл .xib, содержащий представление

Структура документа, показанная на рис. 7.3, содержит три объекта верхнего уровня. Два из них являются прокси-объектами, которые называются шаблонами File's Owner и First Responder. Третий объект является реальным. Это представление, которое будет создано при загрузке nib-файла во время выполнения приложения. Структуру документа в файле `.xib` невозможно скрыть полностью; она сворачивается в набор пиктограмм, представляющих nib-объекты верхнего уровня, аналогично доку сцены в файле раскадровки. Этот набор часто просто называют доком (рис. 7.4).

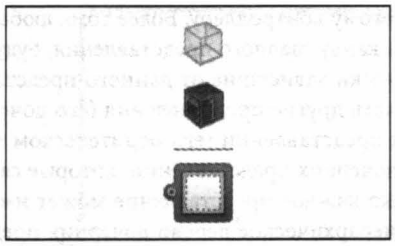


Рис. 7.4. Док в файле .xib

Пока структура документа выглядит необязательной, потому что ее иерархия слишком простая; все объекты на рис. 7.1 и 7.3 легко доступны на канве. Однако, если раскадровка содержит много сцен, а представление содержит много иерархически организованных объектов (вместе с их структурными ограничениями), структура документа становится очень полезной, позволяя делать обзор содержимого nib-файла в виде удобной иерархической структуры, с помощью которой можно найти и выбрать требуемый объект. Иерархию можно перестраивать; например, если объект дочернего представления относится к неправильному родительскому представлению, его можно переместить в структуре, просто перетащив его имя. Кроме того, объекты можно выделять с помощью панели быстрых переходов, расположенной в верхней части редактора: последний компонент этой панели — <Control+6> — является иерархическим всплывающим меню, дублирующим структуру документа.

Если имена nib-объектов в структуре документа кажутся неинформативными, можете изменить их. С формальной точки зрения имя — это метка, не имеющая специального смысла, поэтому nib-объектам можно присваивать любые метки. Выберите метку nib-объекта в структуре документа и нажмите клавишу <Return>, чтобы сделать его редактируемым, или выберите объект и отредактируйте поле Label в разделе Document инспектора идентичности (<Command+Option+3>).

Канва

Канва обеспечивает графическое изображение nib-объекта интерфейса верхнего уровня вместе с его дочерними представлениями, аналогично любому графическому редактору. В файле `.xib` можно удалить представление nib-объекта верхнего уровня с канвы (не удаляя сам объект), щелкнув на букве X в ее верхнем левом углу (рис.7.3). Кроме того, графическое представление можно вернуть на канву, щелкнув на nib-объекте в структуре документа. Канву можно прокручивать. Она может автоматически адаптироваться с учетом графических представлений, которые она содержит; канву раскадровки также можно масштабировать с помощью команды Editor⇨Canvas⇨Zoom или кнопок масштабирования в нижнем правом углу канвы (см. рис. 7.1).

Файл `Main.storyboard` нашего проекта Empty Window содержит только одну сцену, поэтому она представляется графически на канве в виде nib-объекта верхнего уровня — контроллера представления сцены. В контроллере находится его главное представление, которое обычно невозможно отличить от самого контроллера на канве. Во время выполнения приложения этот контроллер представления станет корневым контроллером представления окна; следовательно, его представление будет занимать все окно и фактически будет начальным интерфейсом приложения (см. главу 6). Это обстоятельство позволяет провести эксперимент: любое видимое изменение, которое вносится нами в представление, должно быть видимым во время выполнения приложения. Для того чтобы убедиться в этом, добавим дочернее представление.

1. Начнем с канвы, которая показана на рис. 7.1.
2. Откройте библиотеку объектов (<Command+Option+Control+3>). Если она открывается в виде набора пиктограмм без текста, щелкните на кнопке в левой части панели фильтров, чтобы представить ее в виде списка. Наберите на панели фильтров строку "`button`", чтобы в списке отображались только объекты кнопок. Объект Button указан в списке первым.
3. Перетащите объект Button из библиотеки объектов на главное представление контроллера на канве (рис. 7.5) и отпустите кнопку мыши.

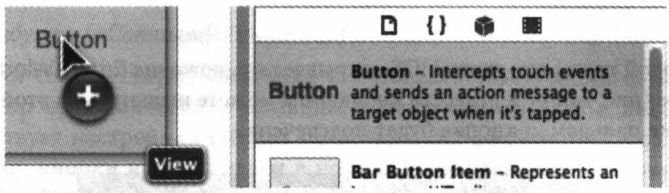


Рис. 7.5. Перетаскивание кнопки на представление

Теперь в представлении на канве есть кнопка. Действие, которые мы выполнили, — перетаскивание из библиотеки объектов на канву — очень типичное; мы будем часто выполнять его при разработке интерфейса.

Как и любой графический редактор, nib-редактор обладает функциональными возможностями, помогающими разрабатывать интерфейс. Попробуем использовать некоторые из них.

- Выбор кнопки. Появляются дескрипторы изменения размеров. (Если вы случайно выберете ее дважды и дескрипторы изменения размеров исчезнут, выберите представление и кнопку снова.)
- Используя дескрипторы изменения размеров, измените размер кнопки, чтобы сделать ее больше. На экране появится информация о размерах.
- Перетащите кнопку ближе к краю представления. Появится линия разметки, демонстрирующая стандартное поле между краем кнопки и представления. Аналогично перетащите кнопку ближе к центру, и линия разметки покажет вам, как центрировать кнопку.
- Выбрав кнопку, нажмите клавишу <Option> и одновременно переместите курсор мыши за пределы кнопки. На экране появятся строки и числа, показывающие расстояние между кнопкой и краями представления. (Если вы случайно щелкнули мышью и стали перетаскивать кнопку, удерживая нажатой клавишу <Option>, то на экране появятся две кнопки. Это объясняется тем, что перетаскивание объекта при нажатой клавише <Option> приводит к дублированию объекта. Выберите нежелательную кнопку и нажмите клавишу <Delete>, чтобы удалить ее.)
- Щелкните мышью на кнопке, удерживая нажатыми клавиши <Control+Shift>. На экране появится меню, позволяющее выбрать кнопку или то, что находится под ней (в данном случае представление, а также контроллер представления, потому что он интерпретируется как фон высокого уровня для всего, что мы делаем).
- Дважды щелкните на заголовке кнопки. Теперь этот текст можно редактировать. Введите новый заголовок, например "`Howdy!`"

Для того чтобы убедиться, что мы действительно проектируем интерфейс приложения, запустите приложение.

1. Проверьте пункт меню Debug⇨Activate / Deactivate Breakpoints. Если вы видите команду Deactivate Breakpoints, выберите ее; мы не хотим прерывать выполнение программы ни в одной точке.
2. Проверьте, чтобы целью во всплывающем меню Scheme был экран iPhone Retina (неважно, какое он имеет разрешение — 3,5 или 4 дюйма).
3. Выберите команду Product⇨Run (или щелкните на кнопке Run инструментальной панели).

После некоторой паузы симулятор iOS открывает приложение Empty Window, и наше окно больше не пустое (рис. 7.6) — в нем есть кнопка! Вы можете щелкнуть на этой кнопке мышью, имитируя касание пальцем, и кнопка будет подсвечена.

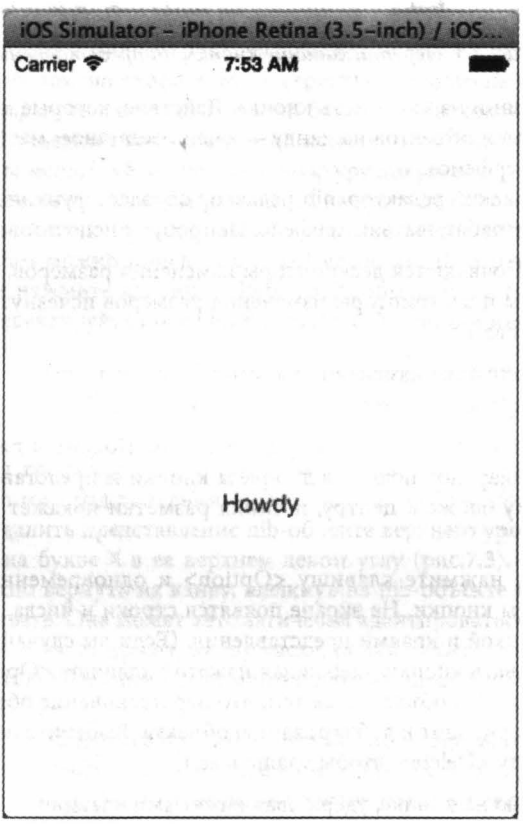


Рис. 7.6. Окно приложения Empty Window больше не пустое

Инспекторы и библиотеки

Вместе с nib-редактором открываются четыре инспектора, которые применяются к любому объекту, выбранному в структуре документа, в доке или на канве.

Инспектор идентичности (<Command+Option+3>)

Самым важным является первый раздел инспектора, Custom Class. Здесь можно изучить и изменить выбранный класс объекта. Некоторые ситуации, в которых может возникнуть необходимость изменить класс nib-объекта, будут рассмотрены в этой главе позднее.

Инспектор атрибутов (<Command+Option+4>)

Его настройки соответствуют свойствам и методам конфигурации объекта в коде. Например, выбор представления и команды в меню Background в инспекторе атрибутов соответствует настройке свойства представления `backgroundColor` в коде. Аналогично выбор кнопки и ввод строки в поле Title эквивалентны вызову метода кнопки `setTitle:forState:`.

Инспектор атрибутов имеет разделы, соответствующие родительским классам класса объекта. Например, инспектор UIButton Attributes имеет три раздела: кроме раздела Button в нем есть разделы Control (потому что класс UIButton является наследником класса UIControl) и View (потому что класс UIButton является наследником класса UIView).

Инспектор размеров (Command+Option+5)

Поля X, Y, Width и Height определяют позицию объекта и размер в родительском представлении, соответствующие свойству рамки в коде. Это же можно сделать на канве, перетаскивая и изменяя размеры, но иногда требуется более высокая точность настройки.

Если функция Autolayout отключена (сброшен флаг Use Autolayout в инспекторе файлов), в инспекторе размеров отображается флаг Autosizing, соответствующий свойству `autoresizingMask`, и анимация, демонстрирующая визуально результаты применения настройки autoresizingMask. Кроме того, раскрывающийся список Arrange содержит полезные команды для позиционирования выбранного объекта.

Если функция Autolayout отключена (по умолчанию для новых файлов `.xib` и `.storyboard`), то остальная часть инспектора размеров выводит ограничения Autolayout выбранного объекта и четырехкнопочную панель в нижнем правом нижнем углу канвы, помогающую выравнивать, позиционировать и ограничивать элементы.

Инспектор связей (Command+Option+6)

Свойства этого инспектора будут рассмотрены ниже в этой главе.

При редактировании nib-файла особенно важными являются две библиотеки.

Библиотека объектов (Command+Option+Control+3)

Является источником объектов, которые можно добавлять в nib-файл.

Библиотека мультимедиа (Command+Option+Control+4)

Перечисляет элементы мультимедиа в вашем проекте, например, изображения, которые можно перетаскивать в класс `UIImageView` или непосредственно в интерфейс (и тогда объект класса `UIImageView` будет создан автоматически).

Загрузка nib-файлов

Nib-файл — это коллекция потенциальных экземпляров, которые называются nib-объектами. Эти экземпляры станут реальными, только если nib-файл загружается при выполнении приложения. В этот момент nib-объекты, содержащиеся в nib-файле, трансформируются в экземпляры, доступные для приложения.

Такая архитектура обеспечивает отличную производительность. Nib-файл обычно содержит интерфейс, а интерфейс — это довольно громоздкая вещь. Nib-файл не загружается, пока это не станет необходимым; в принципе, он может вообще никогда не загрузиться. В этом случае громоздкий интерфейс никогда не будет реализован, пока не станет необходимым. Это позволяет минимизировать требования к памяти, что на мобильных устройствах является главным моментом. Кроме того, загрузка nib-файла требует времени, поэтому для быстрого запуска приложения следует загружать небольшое количество nib-файлов, которые нужны только для начального интерфейса приложения.

Загрузка nib-файла напоминает одностороннее движение: "выгрузки" nib-файла не существует. После того как nib-файл загружен, возникают его экземпляры и на этом работа nib-файла пока закончена. Начиная с этого момента работающее приложение самостоятельно решает, что делать с возникшими экземплярами. Оно должно обращаться к ним, когда они нужны, и удалять их, когда они больше не нужны.

Nib-файл можно интерпретировать как набор инструкторов по генерированию экземпляров; эти инструкции выполняются при каждой загрузке nib-файла. Таким образом, один и тот же nib-файл можно загружать несколько раз, каждй раз генерируя новый набор экземпляров. Например, nib-файл может содержать фрагмент интерфейса, который используется в разных частях приложения. Nib-файл, представляющий отдельную строку таблицы, может загружаться десятки раз, чтобы генерировать десятки видимых строк этой таблицы.

Как указано в главе 5, экземпляры возникают тремя путями: как шаблонные экземпляры (путем вызова метода, создающего экземпляр), путем создания с нуля (с помощью вызова функции `alloc`) и с помощью загрузки nib-файла. Теперь пора перейти к обсуждению третьего способа создания экземпляров.

Кратко перечислим несколько ситуаций, в которых nib-файл загружается во время выполнения приложения.

Контроллер представления создается из файла раскадровки

Раскадровка — это коллекция сцен. Каждая сцена начинается с контроллера представления. Когда этот контроллер представления становится необходимым, он создается из раскадровки. Это значит, что загружается nib-файл, содержащий данный контроллер представления.

Обычно экземпляр контроллера представления создается из раскадровки автоматически. Например, при запуске приложения, имеющего главную раскадровку, система выполнения приложения ищет контроллер начального предствления и создает его экземпляр (см. главу 6). Аналогично раскадровка обычно содержит несколько сцен, связанных переходами (segues); при выполнении перехода создается экземпляр требуемого контроллера представления.

Кроме того, ваш код может создавать контроллер представления из раскадровки вручную. Контроллер начального представления из раскадровки можно создать, вызвав метод

`instantiateInitialViewController`, а любой контроллер представления, указанный в раскадровке строкой идентификатора, — с помощью метода `instantiateViewControllerWithIdentifier:`.

(Отметим, что это не единственный способ создания экземпляра контроллера представления. Это совсем не так. Контроллер представления — это экземпляр, такой же, как любой другой, поэтому его можно создать, отдав инструкцию классу контроллера представления создать свой экземпляр. Эта процедура описана в главе 6, в которой мы создали контроллер представления в приложении Truly Empty с помощью команды `[ViewController new]`. В этот момент не загружается ни один nib-файл; в этом случае экземпляр контроллера представления создается с нуля, а не с помощью nib-файла.)

Контроллер загружает свое главное представление из nib-файла

У контроллера есть главное представление. Однако контроллер — довольно простой объект (это просто некий код), а его главное представление — относительно громоздкий объект. Следовательно, в момент создания экземпляра контроллера ему недостает главного представления. Он генерирует его позже, когда оно понадобится, потому что его необходимо включить в интерфейс. Контроллер может получить свое главное представление несколькими путями; один из них — загрузить его из nib-файла.

Если контроллер является частью сцены в раскадровке и (как правило) имеет свое представление на канве этой раскадровки (как в нашем проекте Empty Window), то используются два nib-файла: nib-файл, содержащий контроллер, и nib-файл, содержащий его главное представления. Как указано выше, nib-файл, содержащий контроллер представления, был загружен в момент создания его экземпляра; теперь, когда экземпляру контроллера потребуется его главное представление, загружается соответствующий nib-файл, и весь интерфейс, связанный с этим контроллером, начинает работать.

Если экземпляр контроллера создается как-то иначе, то может существовать nib-файл, сгенерированный на основе файла `.xib`, связанного с контроллером и содержащего его главное представление. Контроллер представления автоматически загрузит этот nib-файл и извлечет требуемое главное представление. Связь между контроллером и его главным представлением обеспечивается через имя nib-файла. Это можно сделать двумя способами.

Автоматически, основываясь на имени класса контроллера представления

Мы видели этот способ в главе 6 при создании приложения Truly Empty. Класс контроллера представления назывался `ViewController`. Файл `.xib`, а значит и nib-файл, также назывался `ViewController` (игнорируя расширение файла). Этого достаточно для того, чтобы сообщить механизму загрузки nib-файла, где искать nib-файл, содержащий главное представление контроллера при необходимости.

Явно, с помощью свойства контроллера представления nibName

Когда экземпляр контроллера создается с помощью метода `initWithNibName:bundle:`, имя nib-файла, содержащего его главное представление, может быть задано явно с помощью аргумента `nibName:`. Этот аргумент задает свойство `nibName` экземпляра контроллера, которое используется для поиска nib-файла, когда требуется главное представление. В качестве альтернативы контроллер представления может задавать свое собственное свойство `nibName` в настройках.

Ваш код явно загружает nib-файл

Пока мы рассматривали ситуации, в которых nib-файл загружается автоматически. Однако, если nib-файл создается из файла `.xib`, код также может загружать его вручную, вызвав один из следующих методов.

`loadNibNamed:owner:options:`

Это метод экземпляра класса `NSBundle`. Обычно он вызывается так: `[NSBundle mainBundle]`.

`instantiateWithOwner:options:`

Это метод экземпляра класса `UINib`. Требуемый nib-файл был указан, когда экземпляр класса `UINib` создавался и инициализировался с помощью метода `nibWithNibName:bundle:`.

Для того чтобы указать nib-файл во время выполнения приложения, требуются два фрагмента информации — его имя и содержащий его комплект. Действительно, контроллер представления имеет не только свойство `nibName`, но и свойство `nibBundle`, а методы, задающие nib-файл, такие как `initWithNibName:bundle:` и `nibWithNibName:bundle:`, имеют параметры `bundle:` и имя `name:`. Однако в реальной жизни искомым комплектом будет комплект приложения (или, что то же самое, `[NSBundle mainBundle]`); это правило установлено по умолчанию, поэтому указывать комплект не обязательно.

Выходы и владелец nib-файла

При загрузке nib-файла, когда возникают его экземпляры, существует одна проблема: эти экземпляры бесполезны. Экземпляр не используется, пока вы не сошлетесь на него (этому вопросу посвящен раздел в главе 5). Если на экземпляры нет ссылок, то они никак не используются: их нельзя включить в интерфейс; их невозможно модифицировать или конфигурировать; фактически их невозможно использовать, и они могут быстро исчезнуть, не принеся никакой пользы.

Один из основных способов решения этой проблемы — использование выходов. Выход (outlet) — это сущность в nib-файле, которая представляет собой связь от одного объекта в nib-файле к другому. Эта связь имеет направление: именно поэтому я использую предлоги "от" и "к". Один из этих объектов называется источником (source), а другой — целью (destination) выхода. Выход имеет два аспекта.

Имя выхода

В nib-файле выход имеет имя, которое по существу представляет собой строку.

Переменная экземпляра в классе источника

Класс объекта источника выхода имеет переменную экземпляра (или set-метод), соответствующую имени выхода (в соответствием с правилами кодирования ключ–значение, обсуждавшимися в главе 5).

При загрузке nib-файла происходит нечто невероятно интеллектуальное. Объект источника и объект цели больше не являются потенциальными объектами в nib-файле; они

становятся реальными, полноценными экземплярами. Имя выхода немедленно используется по правилам кодирования ключ–значение для сравнения с переменной экземпляра (или set-методом) в объекте источника, и ей присваивается объект цели.

Например, допустим, что мы создали nib-файл, в котором класс `Nib Object A` имеет имя `Dog`, и предположим, что класс `Dog` имеет главную переменную экземпляра класса `Person`. Пусть класс `Nib Object B` — это класс `Person`. Тогда произойдет следующее.

1. Допустим, в nib-редакторе вы рисуете линию, соединяющую объект класса `Nib Object A` (потенциальный экземпляр класса `Dog`) с объектом класса `Nib Object B` (потенциальным экземпляром класса `Person`), — этот выход называется `master`. Эта связь теперь становится частью nib-файла.
2. Запустим приложение, которое загружает этот nib-файл (одним из описанных ранее способов).
3. Создаются объекты классов `Nib Object A` и `Nib Object B`. Теперь у нас есть реальные экземпляры классов `Dog` и `Person`.
4. Однако загрузка nib-файла не завершена. В данный момент nib-файл содержит выход от объекта класса `Nib Object A` к объекту класса `Nib Object B` с именем `master`. Соответственно, он вызывает метод `setValue:forKey:` из экземпляра класса `Dog`, в котором ключом является строка `@"master"`, а значением — экземпляр класса `Person`. Теперь экземпляр класса `Dog` имеет ссылку на экземпляр класса `Person` — а именно, ссылается на значение переменной экземпляра `master`!

Подведем итоги: выход в nib-файле — это просто имя, связывающее два потенциальных объекта. Однако при загрузке nib-файла эти объекты становятся реальными, а выход превращается в реальную ссылку одного объекта на другой с помощью переменной экземпляра (рис. 7.7).

Рис. 7.7. Как выход обеспечивает ссылку на объект, созданный в nib-файле

Механизм загрузки nib-файла не создает переменную экземпляра — т.е. после создания объекта источника он не содержит переменную экземпляра с корректным именем, если она не существовала ранее. Класс, которому принадлежит объект источника, должен заранее определить переменную экземпляра (или set-метод). Таким образом, для того чтобы выход работал, необходимо выполнить предварительную работу в двух местах: в классе источника и в nib-файле. Это довольно сложно. Как мы увидим, среда Xcode существенно облегчает процесс создания выхода, если класс источника не имел необходимой переменной экземпляра. Однако в этом процессе все же легко запутаться, поэтому я объясню его позже.

Выходы — это интеллектуальный способ, позволяющий одному объекту в nib-файле получить ссылку на другой объект в nib-файле, когда оба эти объекта уже не находятся в nib-файле, а были преобразованы в реальные экземпляры. Однако первоначальная задача еще не решена. Nib-файл лишь готовится к загрузке, а его объекты лишь готовятся стать реальными. Проблема состоит не в том, как один объект, созданный из nib-файла, может ссылаться на другой (хотя очень хорошо, что он это может делать); проблема заключается в том, как экземпляр, который уже существовал до загрузки nib-файла, может получить ссылку на объект, который станет реальным только после загрузки nib-файла. В противном случае объекты, сгенерированные на основе nib-файла, смогут ссылаться на другой объект, но никто другой не сможет ссылаться на них, и оба объекта останутся бесполезными.

Нам нужен механизм, позволяющий выходу пересекать метафизический барьер между миром экземпляров до загрузки nib-файла и множеством экземпляров, генерируемых при его загрузке. Этим механизмом является прокси-объект владельца. Я уже упоминал, что файлы `.storyboard` или `.xib` автоматически заполняются некоторыми прокси-объектами, цель которых я не разглашал. Сейчас я собираюсь рассказать вам об одном из них, а именно о прокси-объекте владельца nib-файла. Во-первых, вам необходимо знать, где его искать.

- В сцене раскадровки прокси-объект владельца nib-файла является контроллером представления верхнего уровня. Это первый объект, указанный для сцены в структуре документа, и первый объект, который показан в доке сцены.
- В файле `.xib` прокси-объект владельца nib-файла — это первый объект, показанный в структуре документа или в доке; он указывается под шаблонами Placeholders как File's Owner.

Прокси-объект владельца nib-файла — это прокси-экземпляр, который уже существует за пределами nib-файла в момент его загрузки. При загрузке nib-файла этот объект не создается; он уже существует. Таким образом, реальный, уже существующий экземпляр заменяется прокси-экземпляром, который осуществляет все его связи.

Например, допустим, что в нашем nib-файле существует nib-объект класса `Person`, а не nib-объект класса `Dog`. Классом прокси-объекта владельца nib-файла является класс `Dog` — причем, как вы помните, класс `Dog` содержит переменную экземпляра `master`. В nib-редакторе можно провести связь от объекта владельца nib-файла (потому что он относится к классу `Dog`) к nib-объекту класса `Person`. При загрузке nib-файла механизм его загрузки сравнит прокси-объект владельца nib-файла класса `Dog` с уже существующим экземпляром класса `Dog` и назначит объект класса `Person` в качестве значения его переменной экземпляра `master`.

Откуда же механизм загрузки nib-файлов знает о существовании экземпляра класса `Dog`, с которым следует сравнивать прокси-объект владельца nib-файла в момент его загрузки? Вы никогда не догадаетесь. При каждой загрузке nib-файла этот уже существующий объект назначается его владельцем. Именно этому уже существующему объекту соответствует прокси-объект владельца nib-файла, содержащийся в нем самом (рис. 7.8).

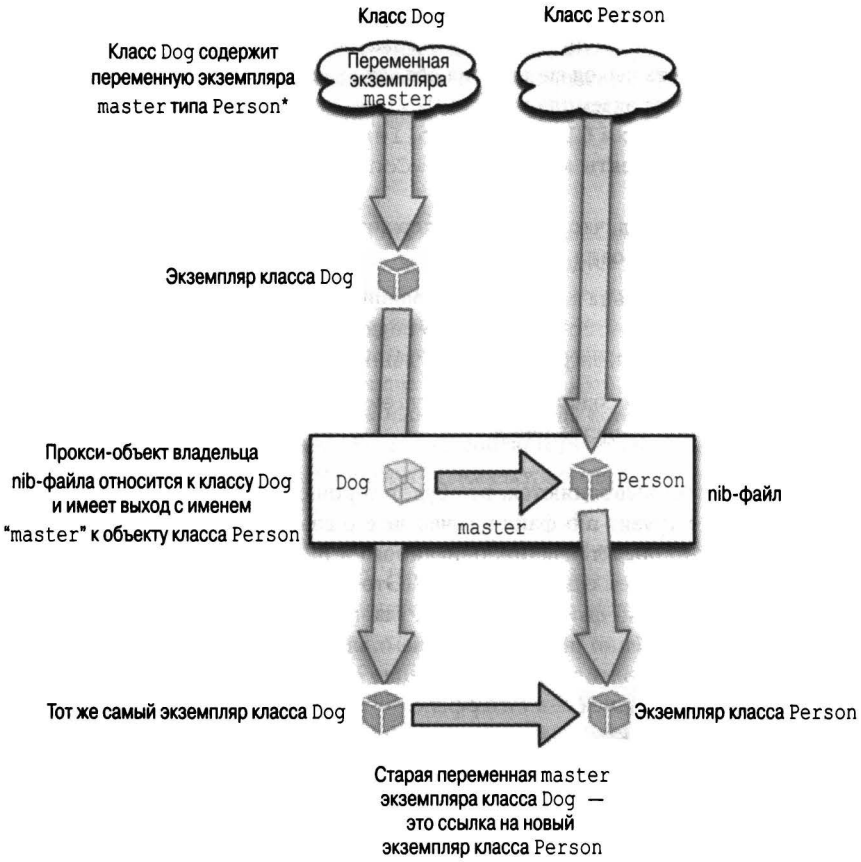


Рис. 7.8. Выход из объекта прокси владельца nib-файла

Каким образом назначается владелец nib-файла? Существуют два способа.

- Если код загружает nib-файл, вызывая методы `loadNibNamed:owner:options:` или `instantiateWithOwner:options:`, то объект владельца задается аргументом `owner:`.
- Если экземпляр контроллера представления загружает nib-файл автоматически, чтобы получить свое главное представление, то контроллер представления сам назначается объектом владельца nib-файла.

Рассмотрим второй вариант, потому что мы уже создали два проекта, использовавших этот способ. Контроллер имеет главное представление. Формально это выражается тем фактом, что класс `UIViewController` имеет свойство `view`, т.е. он имеет переменную экземпляра представления или соответствующий метод доступа (или и то и другое). Когда контроллер загружает nib-файл, чтобы получить свое главное представление, он играет роль владельца этого nib-файла.

Итак, теперь понятно, каким образом контроллер, загружающий nib-файл с его главным представлением, может ссылаться на него и что-то с ним делать (например, включать в интерфейс) — он делает это с помощью переменной экземпляра представления, соответствующего

выходу представления в nib-файле. Эта ситуация похожа на рис. 7.8, на котором показан экземпляр класса Dog, владеющий nib-файлом, содержащим прокси-объект владельца nib-файла класса Dog, у которого есть выход master на объект класса Person, — за исключением того, что теперь у нас существует экземпляр класса UIViewController, содержащий прокси-объект владельца nib-файла класса UIViewController, у которого есть выход view на объект класса UIView. (Говоря "объекты класса UIViewController и UIView", я, разумеется, имею в виду также их подклассы.)

Итак, если предполагается, что nib-файл содержит главное представление контроллера, то должны выполняться два условия.

- Прокси-объект владельца nib-файла, находящийся в этом nib-файле, должен относиться к классу UIViewController или его подклассу — предпочтительно к классу контроллера представления, загружаемого из данного nib-файла.
- Этот nib-файл должен содержать выход view из прокси-объекта владельца nib-файла к объекту главного представления.

Если оба эти условия выполняются, все будет хорошо. Когда контроллеру понадобится его представление, он загрузит nib-файл в качестве его владельца; в этом случае будет создан полноценный экземпляр представления, описанного в nib-файле, и переменная view контроллера представления станет ссылкой на него. Это решает нашу проблему. Все другие экземпляры, создаваемые из nib-файла, зависят от главного представления (как его вложенные представления и т.п.), и именно переменная экземпляра view в контроллере теперь служит ссылкой на главное представление и все зависящие от него экземпляры. Теперь контроллер готов загрузить свое представление в интерфейс.

Если посмотреть на проекты, созданные в главе 6, то вы увидите, что их nib-файлы действительно организованы так, как я описал выше! Сначала посмотрим на проект Empty Window. Откроем файл Main.storyboard. Он содержит одну сцену, в которой прокси-объектом владельца nib-файла является объект View Controller. Выберите пункт View Controller в структуре документа. Переключитесь в окно инспектора идентичности. Он сообщает нам, что прокси-объект владельца nib-класса относится к классу ViewController. Теперь перейдите в окно инспектора связей. Он сообщает нам, что существует выход Outlet на объект View (если вы остановите курсор мыши над линией выхода, то объект View на канве будет выделен, чтобы облегчить его идентификацию).

Теперь посмотрим на проект Truly Empty. Откройте файл ViewController.xib. Прокси-объектом владельца nib-файла является шаблон File's Owner. Выберите его. Перейдите в окно инспектора идентичности. Он сообщает нам, что прокси-объект владельца nib-класса относится к классу ViewController. Перейдите в окно инспектора связей. Он сообщает нам, что существует выход Outlet на объект View.

Создание выхода

Часто возникает необходимость самостоятельно создать выход. Для того чтобы продемонстрировать, как это делается, будем использовать проект Empty Window. Ранее мы добавили кнопку в его начальное представление главного представления контроллера. Теперь создадим выход от контроллера представления к этой кнопке и назовем его button.

1. Начнем с анализа источника и цели предполагаемого выхода. Выберите файл Main.storyboard в проекте Empty Window и с помощью инспектора идентичности идентифицируйте класс используемых объектов. Объект источника относится

к классу ViewController. Объект цели относится к классу UIButton. Это значит, что класс ViewController будет использовать переменную экземпляра типа UIButton* (или одного из родительских классов класса UIButton).

2. Теперь создадим выход. Откройте файл ViewController.h. В разделе @interface введите строку

```
@property IBOutlet UIButton* button;
```

3. Это объявление свойства. До сих пор мы не обсуждали объявления свойств (эту тему мы рассмотрим в главе 12), так что мои слова пока следует принимать на веру. Объявление свойства по существу эквивалентно объявлению переменной экземпляра и методов доступа к ней в одной строку. В реальном проекте я не использую именно такое объявление, но пока можно использовать и такую конструкцию. Ключевым термом здесь является имя IBOutlet; это запрос к среде Xcode, который означает, что мы хотели бы использовать такое свойство как имя выхода, когда источником является экземпляр класса ViewController. Имя IBOutlet соответствует синтаксическим правилам, и хотя среда Xcode может увидеть в этом подсказку, компилятор будет рассматривать ее как эквивалент пустой строки.

4. Откройте файл Main.storyboard еще раз. Выберите класс View Controller и перейдите в окно инспектора связей. Произойдет интересная вещь — в этом окне появился выход кнопки! Это объясняется тем, что мы использовали волшебное слово IBOutlet в объявлении свойства в классе ViewController. Однако этот выход пока не создан. Иначе говоря, инспектор связей сообщает нам, что выбранный объект (View Controller) может служить источником выхода кнопки, но пока им не является. Для того чтобы он стал реальным источником, проведите линию от кружочка у правого края слова button в инспекторе связей к самой кнопке — либо кнопке на канве, либо к ее имени в структуре документа. При этом на экране появится растягивающаяся линия, символизирующая связь, которая тянется за курсором мыши при его перетаскивании, а кнопка подсвечивается, когда курсом мыши оказывается над ней. Отпустите кнопку мыши (рис. 7.9).

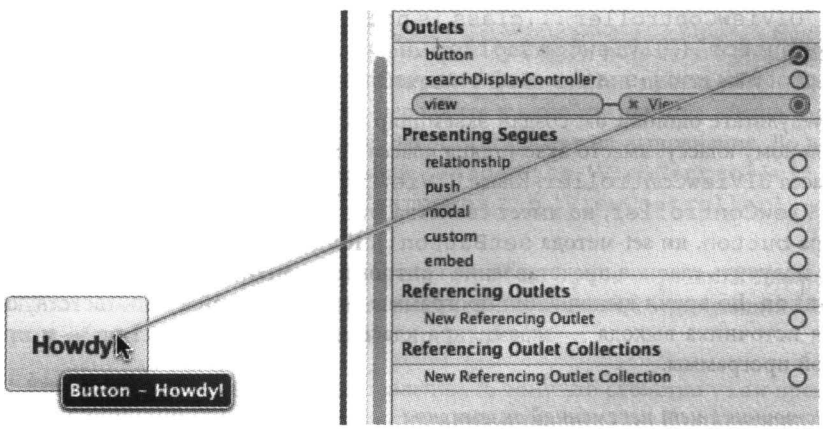


Рис. 7.9. Связывание выхода с кнопкой из инспектора связей

Инспектор связей сообщает, что выход кнопки связан с нашей кнопкой. Это хорошо, но недостаточно. Мы создали выход, но никак его не используем.

Как использовать выход? Если выход правильно настроен, то при выполнении кода после загрузки главного представления контроллера `ViewController` из nib-файла его свойство `button` станет ссылкой на кнопку, созданную из nib-файла, и интерфейс приложения станет видимым. Итак, напишем фрагмент кода, использующий свойство `button` для изменения свойств кнопки. Поскольку кнопка — часть интерфейса, мы должны видеть эти изменения на экране. Давайте изменим заголовок кнопки.

1. Откройте файл `ViewController.m`.
2. После того как экземпляр контроллера получит свое главное представление, будет выполнен метод `viewDidLoad`. В данном случае это значит, что он загрузит nib-файл, содержащий его главное представление. Это nib-файл, в котором мы только что создали выход кнопки, ссылающийся на нашу кнопку. При выполнении кода ссылка `self.button` будет указывать на реальную кнопку в нашем интерфейсе. Вставим следующую строку в метод `viewDidLoad method`:

```
[self.button setTitle:@"Hi!" forState:UIControlStateNormal];
```

3. Соберите и выполните проект. Как видим, несмотря на то, что в раскадровке на кнопке написано "`Howdy!`", в видимом интерфейсе приложения на ней написано "`Hi!`". Наш выход работает!

Неправильная конфигурация выхода

При настройке выхода необходимо одновременно выполнить несколько условий. Если вы не сделаете этого, то рано или поздно ваш выход станет работать неправильно. Не обижайтесь и не бойтесь; будьте готовы! Это случается с каждым. Важно распознать симптомы, чтобы понять, в чем дело.

Неправильный класс источника

- Начнем с примера Empty Window. Откройте раскадровку. Используя инспектор идентичности, измените класс контроллера представления в сцене на `UIViewController`. Запустите проект. На этапе выполнения программы вы получите сообщение об ошибке: "`UIViewController ...class is not key value coding-compliant for the key button`" ("`UIViewController ... — класс не соответствует правилам кодировки ключ-значение для ключа button.`")
- В результате ошибки бы создан экземпляр источника выхода, относящийся к неправильному классу: вместо экземпляра класса `ViewController` мы создали экземпляр класса `UIViewController`. Класс `UIViewController`, встроенный суперкласс класса `ViewController`, не имеет свойства `button` (в нем нет ни переменной экземпляра `button`, ни set-метода `setButton:`). Когда мы загрузим второй nib-файл, чтобы получить главное представление контроллера, он будет содержать выход с именем `button`. Во время выполнения программы система не найдет соответствующей цели для источника выхода — экземпляра класса `UIViewController` — и произойдет сбой программы.

В классе источника нет переменной экземпляра

- Исправим ошибку, сделанную в предыдущем примере, используя инспектор идентичности, и изменим класс контроллера представлений в сцене, вернув имя `ViewController`. Запустим проект на выполнение, чтобы убедиться в успехе. Теперь закомментируем объявление `@property` в файле `ViewController.h` и строку `self.button` в

строке ViewController.m. Запустим проект на выполнение. Во время выполнения приложения произойдет сбой: "ViewController ... class is not key value coding-compliant for the key button." ("ViewController ... — класс не соответствует правилам кодировки ключ-значение для ключа button.")

- Очевидно, что это та же самая проблема, но в другом варианте. Мы правильно указали класс источника — ViewController. Однако мы создали выход button в nib-файле тайком от nib-редактора и удалили соответствующую переменную экземпляра в классе источника выхода. И снова при загрузке nib-файла система выполнения приложения не сможет найти цель для источника выхода — экземпляр класса ViewController — и произойдет сбой.

В nib-файле нет выхода

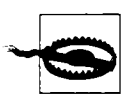
- Исправим ошибку, сделанную в предыдущем примере, сняв символы комментария со строк, закомментированных в файле ViewController.h и ViewController.m. Запустим проект на выполнение, чтобы убедиться, что он работает. Теперь откроем раскадровку. Выберем контроллер представления и в инспекторе связей удалим связь выхода кнопки, щелкнув на букве X на левом конце второго закругленного прямоугольника. Запустим проект на выполнение. Приложение работает, но заголовок кнопки остается строкой "Howdy!", т.е. наша попытка изменить его на "Hi!" в методе viewDidLoad класса ViewController молча провалилась.

- Это очень коварная и невероятно распространенная ошибка. Мы указали правильный класс объекта источника с правильной переменной экземпляра, и nib-редактор содержит выход, но сам выход не имеет объектов цели. В результате после загрузки nib-файлов значение переменной экземпляра button в объект источника будет равно nil. Мы можем ссылаться на переменную self.button, но она равна nil. Мы передаем ей сообщение setTitle:forState:, но это сообщение отправляется в пустоту. Сообщения, отправленные по адресу nil, не вызывают никаких ошибок, но при этом ничего не происходит (см. главу 3).

Отсутствует выход представления

- На этот раз мы будем использовать пример Truly Empty, потому что редактор раскадровок для нашей цели не подходит. Откройте файл .xib в проекте Truly Empty. Выберите прокси-объект File's Owner, перейдите в окно инспектора инспектора связей и отсоедините выход представления. Запустите проект на выполнение. Во время выполнения приложения произойдет сбой: "loaded the 'ViewController' nib but the view outlet was not set" ("загружен nib 'ViewController', но выход представления не задан").

- В этом сообщении сказано все. Nib-файл, играющий роль источника главного представления контроллера, должен иметь выход представления, соединяющий контроллер (прокси-объект владельца nib-файла) с представлением.



Контроллер имеет выход представления, потому что свойство view представления класса UIViewController отмечено как outlet. К сожалению, вы не можете увидеть эту отметку. В документации о стандартных классах каркаса Cocoa ничего не говорится об их свойствах, которые могут быть выходами! В общем, единственный способ узнать, какие выходы есть во встроенном классе, — проверить один из таких классов в nib-редакторе.

Удаление выхода

Для того чтобы правильно удалить выход, т.е. не вызвать ни одной проблемы. перечисленной в предыдущем разделе, требуется внести несколько изменений в разных местах, как это было и при создании выхода. Я рекомендую выполнять эти действия в следующем порядке.

1. Отсоединить выход в nib-файле.
2. Удалить объявление выхода из кода.
3. Попытаться скомпилировать проект и предоставить компилятору выявить остальные проблемы.

Допустим, например, что вы решили удалить выход `button` из проекта Empty Window. Вы можете выполнить следующую трехэтапную процедуру.

1. Отсоедините выход в nib-файле. Для этого откройте раскадровку, выделите объект источника (контроллер представления) и отсоедините выход `button` в инспекторе связей, щелкнув на кнопке X.
2. Удалите объявление выхода из кода. Для этого откройте файл `ViewController.h` и удалите или закомментируйте строку объявления `@property`.
3. Удалите другие ссылки на это свойство. Проще всего попытаться собрать проект; компилятор обнаружит ошибку в строке, ссылающейся на свойство `self.button` в файле `ViewController.m`, потому что такого свойства больше нет. Удалите иди закомментируйте эту строку и соберите проект снова, чтобы убедиться, что все в порядке.

Другие способы создать выходы

Ранее мы создали выход вручную, объявив свойство в заголовочном файле класса и проведя соединительную линию от кружка выхода в инспекторе связей в nib-редакторе к кнопке. Среда Xcode предоставляет много других способов для создания выходов — слишком много, чтобы перечислять их все здесь. Сделаем обзор наиболее интересных способов.

Начнем с удаления выхода из проекта Empty Window (если вы этого еще не сделали). Вместо использования инспектора связей для задания выхода в nib-файле мы будем использовать индикаторы HUD (heads-up display), присоединенные к целевому объекту.

1. В файле `ViewController.h` создайте (или закомментируйте) объявление свойства:
```
@property IBOutlet UIButton* button;
```
2. Находясь в раскадровке, нажмите клавишу <Control> и проведите соединительную линию[1] от контроллера представления к кнопке. Контроллер представления представляется либо меткой View Controller в структуре документа, либо первой пиктограммой на доке сцены. Кнопка представляется либо меткой в структуре документа, либо ее графическим представлением на канве.
3. Когда вы отпустите кнопку мыши, на экране появится индикатор HUD, в котором кнопка указана как возможный выход (рис. 7.10). Щелкните на кнопке.

[1] Выражение "проведите соединительную линию" между точками A и B означает, что пользователь должен навести курсор мыши на точку A, нажать кнопку мыши и перетащить курсор в точку B. Когда речь идет о выражении Control-drag, эта процедура выполняется при нажатой клавише <Control>. — *Примеч. ред.*

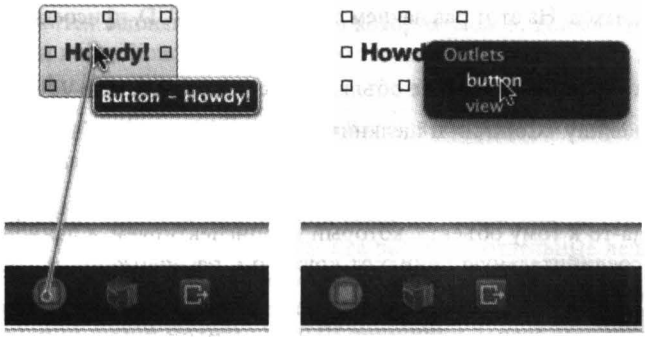


Рис. 7.10. Соединение выхода с помощью проведения соединительной линии от объекта-источника

Снова удалите выход. На этот раз начнем в индикатора HUD, присоединенного к объекту-источника.

1. Создайте или раскомментируйте объявление свойства в файле `ViewController.h`.
2. Нажмите клавишу <Control> и щелкните мышью на представлении контроллера. Контроллер представления представлен либо меткой **View Controller** в структуре документа, либо первой пиктограммой на доке сцены. На экране появится индикатор HUD, который похож на аналогичный индикатор в инспекторе связей.
3. Проведите соединительную связь из кружка справа от кнопки к самой кнопке (рис. 7.11). Отпустите кнопку мыши.

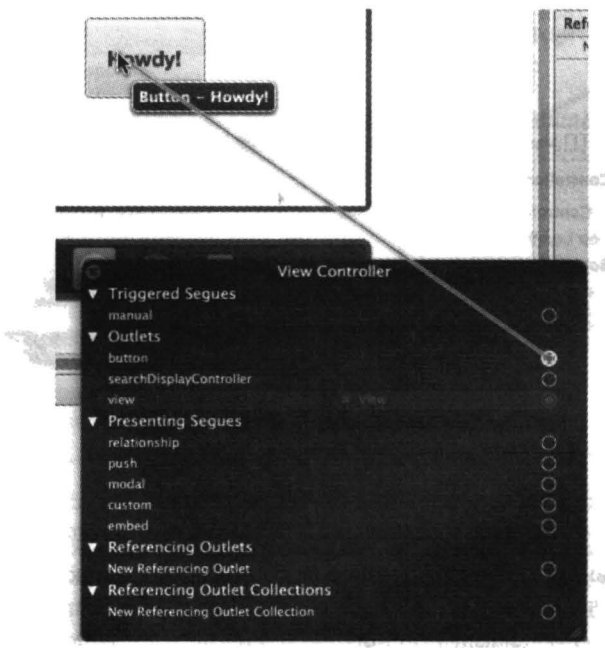


Рис. 7.11. Соединение выхода с помощью проведения соединительной линии от объекта-источника

Снова удалите выход. На этот раз начнем с индикатора HUD, присоединенного к целевому объекту.

1. Создайте или раскомментируйте объявление свойства в файле `ViewController.h`.
2. Нажмите клавишу <Control> и щелкните мышью на кнопке. На экране появится индикатор HUD, который похож на аналогичный индикатор в инспекторе связей.
3. Найдите листинг, в котором есть строка `New Referencing Outlet`. Она обозначает выход откуда-то к тому объекту, который мы инспектируем, в данном случае к кнопке. Проведите соединительную линию от кружочка, расположенного справа от кнопки, к контроллеру представления.
4. На экране появится второй индикатор HUD, в котором перечислены имена из контроллера представления. Щелкните на кнопке.

Снова удалите выход. Теперь мы собираемся создать выход, проведя соединительную линию между кодом и nib-редактором. Для этого необходимо выполнить определенную работу в двух местах: нам необходимо окно помощника. Откройте файл `ViewController.h` в главном окне редактирования. Откройте раскадровку в окне помощника, чтобы кнопка была видна.

1. Создайте или раскомментируйте объявление свойства в файле `ViewController.h`.
2. Слева от объявления свойства на поле появится кружок.
3. Проведите соединительную линию от кружка на краю окна к кнопке в nib-редакторе (рис. 7.12) и отпустите кнопку мыши.

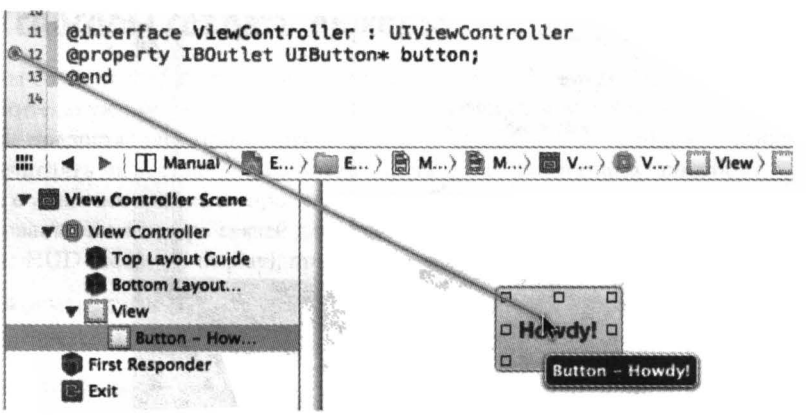


Рис. 7.12. Соединение выхода с помощью проведения соединительной линии от кода к nib-редактору

Снова удалите выход. На этот раз мы собираемся создать код и связь с выходом за один шаг! Для этого будем использовать двухоконный режим из предыдущего примера.

1. Нажмите клавишу <Control> и проведите соединительную линию от кнопки к nib-редактору через границу окна к пустому разделу `@interface` в файле `ViewController.h`.
2. Индикатор HUD предлагает выбрать команду Insert Outlet, Action или Outlet Collection (рис. 7.13). Отпустите кнопку мыши.

3. На экране появится всплывающее меню, в котором можно конфигурировать объявление для вставки в код. Конфигурируйте его так, как показано на рис. 7.14. Щелкните на кнопке Connect.
4. В ваш код вставляется объявление свойства, и выход соединяется с nib-файлов на один шаг.

Создание выхода с помощью непосредственной связи между кодом и nib-редактором очень удобно, но не стоит обольщаться: прямой связи на самом деле не существует. Для того чтобы выход работал правильно, необходимы две разные и отдельные вещи — переменная экземпляра в классе и выход в nib-файле, with the same name and coming from an instance of that class. Именно идентичность имен и классов позволяет сопоставлять их при загрузке nib-файла, чтобы переменная экземпляра была правильно настроена в конкретный момент времени. Среда Xcode пытается помочь настроить все правильно, но это не волшебное соединение кода с nib-файлом.

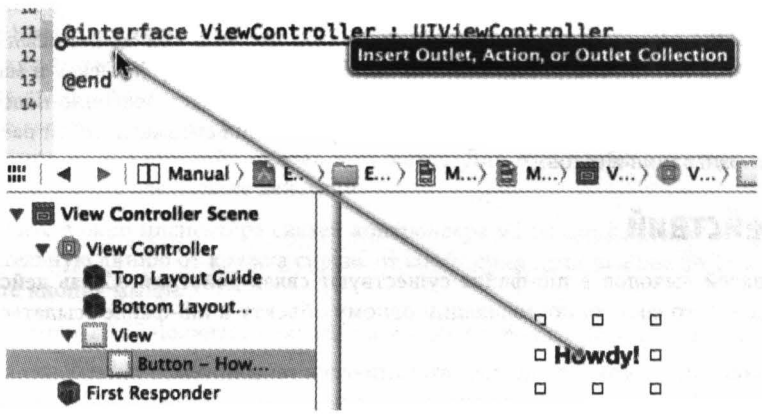


Рис. 7.13. Соединение выхода с помощью проведения соединительной линии от nib-редактора к коду

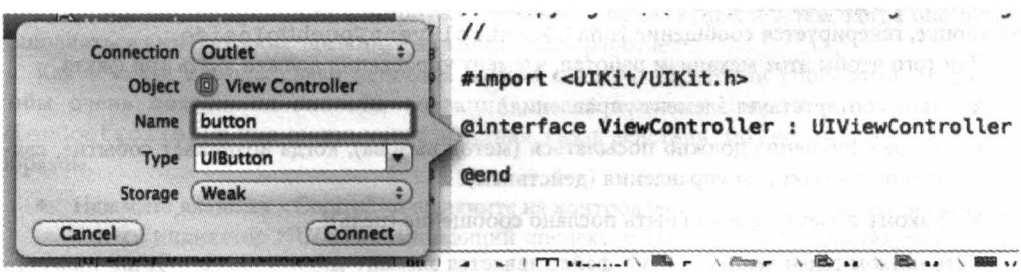


Рис. 7.14. Конфигурирование объявления свойства

Если среда Xcode полагает, что выход настроен правильно, с правильным объявлением выхода в коде и правильной связью в nib-файле, то кружок слева от объявления в коде будет заполнен. Этот кружок служит не только индикатором правильности выхода, но и позволяет открыть всплывающее меню после щелчка, в котором перечисляется то, что может находиться на другом конце связи: щелкните на соответствующем пункте меню и перейдите в nib-редактор, выбрав целевой объект.

Связи выхода

Связь выхода — это переменная экземпляра класса NSArray (в коде), имеющая (в nib-файле) многочисленные связи с объектами того же типа.

Например, предположим, что класс содержит следующее объявление свойства:

```
@property IBOutletCollection(UIButton) NSArray* buttons;
```

Обратите внимание на довольно странный синтаксис: терм IBOutletCollection сопровождают скобки, содержащие имя класса без кавычек. Само свойство объявлено как объект класса NSArray.

С помощью экземпляра этого класса, играющего роль объекта источника в nib-редакторе, можно формировать многокнопочные выходы, каждый из которых связан с другим объектом класса UIButton в nib-файле. При загрузке nib-файла эти экземпляры класса UIButton становятся элементами массива типа NSArray, состоящего из кнопок; порядок, в котором формируются эти выходы, определяются порядком элементов в этом массиве.

Преимущество такого подхода заключается в том, что ваш код может ссылаться на несколько объектов интерфейса, созданных из nib-файла, по номеру (индексу в массиве), а не придумывать для каждого из них отдельное имя. Это оказывается особенно полезным, когда выходы формируются для таких сущностей, как ограничения автоматической разметки и механизмы распознавания жестов.

Связи действий

Кроме связей выходов в nib-файле существуют связи действий. Связь действия, как и связь выхода, — это способ, позволяющий одному объекту в nib-файле ссылаться на другой объект.

Действие (action) — это сообщение, автоматически генерируемое интерфейсным объектом класса UIControl (элементом управления) из каркаса Cocoa и посылаемое другому объекту, когда пользователь что-то делает, например щелкает на кнопке. В ответ на разные действия пользователя заставляют элемент управления генерировать сообщения, которые называются событиями. Для того чтобы увидеть список возможных событий, прочитайте описание класса UIControl в документации в разделе "Control Events." Например, если пользователь щелкает на кнопке, генерируется сообщение типа UIControlEventTouchUpInside.

Для того чтобы этот механизм работал, элемент управления должен знать три факта.

- Чему соответствует элемент управления?
- Какое сообщение должно посылаться (метод вызова), когда возникает событие, связанное с элементом управления (действием)?
- Какому объекту должно быть послано сообщение (цель)?

Источником связи действия в nib-файле является элемент управления; его целью является объект, которому источник посылает сообщение, связанное с действием. Конфигурирование класса целевого объекта, чтобы он имел метод, принимающий сообщение, связанное с действием, должен выполнять программист.

Например, сделаем контроллер представлений в нашем проекте Empty Window целью сообщений, связанного с действием и генерируемого событием UIControlEventTouchUpInside (означающим что кнопка была нажата). Нам нужен метод в контроллере представления, который будет вызываться кнопкой после щелчка на ней. Для того чтобы этот метод был более

ясным, поместим на котроллер представления окно предупреждения. Вставим этот метод в раздел реализации в файле ViewController.m.

```
- (IBAction) buttonPressed: (id) sender {
   UIAlertView* av = [[UIAlertView alloc] initWithTitle:@"Howdy!"
                                                 message:@"You tapped me."
                                                delegate:nil
                                       cancelButtonTitle:@"Cool"
                                       otherButtonTitles:nil];
  [av show];
}
```

Терм IBAction напоминает IBOutlet — это подсказка для самой среды Xcode, которая является синтаксически корректной, потому что, хотя среда Xcode может увидеть подсказку в вашем коде, компилятор будет интерпретировать ее как эквивалент void. Он просит среду Xcode сделать этот метод в nib-редакторе селектором сообщения, связанного с действием, если объект класса, в котором этот метод определен, является целью связи действия. Действительно, если посмотреть в nib-редактор, мы увидим, что он теперь доступен: выберите объект View Controller и перейдите в окно инспектора связей, вы увидите, что метод buttonPressed: теперь перечислен в разделе Received Actions.

Как и для связи выхода, создание связи действия представляет собой двухэтапный процесс. Мы уже выполнили первый этап: определили метод действия. Однако мы еще ничего не сделали для того, чтобы nib-файл вызвал этот метод. Для того чтобы это стало возможным, необходимо выполнить следующие действия.

1. Перейдите в окно инспектора связей контроллера View Controller и проведите соединительную линию от кружка справа от сообщения buttonPressed: к кнопке. Отпустите кнопку мыши.
2. В индикаторе HUD появится список, связанных с элементами управлениями. Щелкните на кнопке Touch Up Inside.

Теперь связь действия создана. Это значит, что при выполнении приложения в любой момент, когда кнопка получает событие Touch Up Inside, т.е. когда она будет нажата, она будет посылать сообщение buttonPressed: цели, которая представляет собой контроллер представления. Мы знаем, что этот метод должен делать: он должен выводить на экран окно предупреждения. Попробуйте! Соберите и выполните приложение, а затем, когда оно будет выполняться симулятором, щелкните на кнопке. Она работает!

Как и связи выхода, связи действия можно создавать многими способами. Эти способы очень похожи на способы создания связей выхода. Например, если в файле ViewController.m уже создан метод действия, связь действия можно создать следующим образом.

- Нажмите клавишу <Control> и щелкните на контроллере представления. На экране появится индикатор HUD, напоминающий инспектор связей. Действуйте так, как указано в предыдущем разделе.
- Выберите кнопку и перейдите в окно инспектора связей. Проведите соединительную линию от кружка Touch Up Inside к контроллеру представления. На экране появится индикатор HUD, в котором перечислены известные методы действия в контроллере представления; щелкните на пункте buttonPressed:.
- Нажмите клавишу <Control> и щелкните кнопкой мыши. На экране появится индикатор HUD, напоминающий инспектор связей. Действуйте так, как указано в предыдущем разделе.

Кроме того, можно создать связь между редактором кода и nib-редактором. Откройте файл `ViewController.m` в одном окне и раскадровку в другом. Слева от метода `buttonPressed:` в файле `ViewController.m` на поле расположен кружок. Вы можете провести соединительную линию от этого кружка через границу окна на кнопку в nib-редакторе.

Как и в случае связи выхода, самым впечатляющим является создание связи действия с помощью проведения соединительной линии из nib-редактора в ваш код, вставки метода действия и создания связи действия в nib-файле за один шаг. Попробуйте выполнить эту процедуру, предварительно удалив метод `buttonPressed:` из своего кода, чтобы в nib-файле не было связи действия.

1. Нажмите клавишу <Control> и проведите соединительную линию из nib-редактора на пустую область в разделе `@implementation` файла `ViewController.m`. На экране появится индикатор HUD, содержащий команду Insert Action. Отпустите кнопку мыши.
2. На экране появится всплывающее меню, в котором можно конфигурировать метод действия, который мы создаем. Назовите его `buttonPressed:`, убедитесь, что пункт Event настроен на Touch Up Inside (по умолчанию), и щелкните на кнопке Connect.

Среда Xcode создала шаблонный метод и формирует соответствующее действие в nib-файле за один шаг:

```
- (IBAction)buttonPressed:(id)sender { }
```

Этот метод — просто шаблон (среда Xcode не может прочитать ваши мысли и угадать, что этот метод должен делать), поэтому в реальной жизни вам придется заполнить пространство между фигурными скобками определенным кодом.

Как и при создании связи выхода, заполненный кружок справа от кода в методе действия означает, что среда Xcode считает конфигурацию связи правильной. Вы можете щелкнуть на этом кружке, чтобы увидеть, какой объект является источником связи (и перейти к нему).

Связи между nib-файлами

Невозможно провести связь выхода или действия между объектом в одном nib-файле и объектом в другом nib-файле или (в раскадровке) между объектом в одной сцене и объектом в другой сцене. Если вы думаете, что это возможно, значит, вы не понимаете, что такое nib-файл (а также сцена или связь). Причина проста: в момент загрузки nib-файла все объекты в nib-файле одновременно становятся экземплярами, поэтому имеет смысл связывать их друг с другом в рамках одного и того же nib-файла, потому что мы знаем, какие экземпляры возникают при загрузке nib-файла. Если попробовать провести связь выхода или действия от объекта в одном nib-файле к объекту в другом nib-файле, невозможно понять, какие реально существующие экземпляры будут связаны друг с другом, потому что разные nib-файлы загружаются в разные моменты времени (если вообще загружаются). Проблема связи между экземплярами, сгенерированными из разных nib-файлов, является частным случаем более общей проблемы коммуникации экземпляров в рамках программы, которая обсуждается в главе 13.

Меню Related Files в nib-редакторе содержит действия и выходы, подключенные к объекту, выделенному в данный момент. Это позволяет осуществлять быстрый переход к другому концу связи или просто получать информацию о нем. Когда главным окном является окно nib-редактора, окно помощника содержит те же пункты, что и меню Tracking. Таким образом, вы можете, например, выбрать объект интерфейса в nib-редакторе, чтобы увидеть в окне помощника метод действия, с которым он связан.

Дополнительная инициализация экземпляров, созданных из Nib-файлов

После завершения загрузки nib-файла его экземпляры являются абсолютно полноценными; они инициализированы и настроены с помощью всех атрибутов, указанных в окнах инспекторов атрибутов и размеров, а их выводы были использованы для задания значений соответствующих переменных экземпляров. Тем не менее программист может добавить свой код для процесса инициализации из загружаемого nib-файла. В этом разделе мы опишем, как это сделать.

Мы применим привычный подход, в рамках которого контроллер представления, функционирующий как владелец загружаемого nib-файла, содержащего его главное представление (а значит, представленный в nib-файле прокси-объектом владельца nib-файла), имеет выход к объекту интерфейса, созданного с помощью nib-файла. В этой архитектуре контроллер представления может выполнять дальнейшую настройку объекта интерфейса, потому что он имеет ссылку на него после загрузки nib-файла. Прежде всего мы можем сделать это с его методом `viewDidLoad`. Это объясняется тем, что мы уже использовали метод `viewDidLoad` как место для вставки кода, изменяющего заголовок "`Howdy!`" на "`Hi!`".

Другая возможность — конфигурировать сам nib-объект в дополнение к его настройке в nib-файле. Часто это делают из-за желания использовать готовый подкласс объекта интерфейса. В принципе программист может сам написать такой подкласс, чтобы создать место для самонастройки кода. Дело в том, что nib-редактор не позволяет выполнять конфигурирование после загрузки nib-файла. Кроме того, может существовать много объектов, которые конфигурируются одинаково и сложно, поэтому имеет смысл предоставить им возможность конфигурироваться самостоятельно с помощью общего подкласса, а не конфигурировать их индивидуально в nib-редакторе.

Например, в пользовательском классе можно реализовать сообщение `awakeFromNib`. Оно рассылается всем объектам, созданным на основе nib-файла, сразу после их создания при загрузке nib-файла the object has been initialized and configured and its connections are operational.

Например, пусть фон кнопки всегда будет красным, независимо от того, как она сконфигурирована в nib-файле. (Это довольно необычно, но очень эффективно.) Создадим в проекте Empty Window подкласс кнопки `MyRedButton`.

1. В навигаторе проекта выберите команду File⇨New⇨File. Выберите класс iOS Cocoa Touch Objective-C и щелкните на кнопке Next.
2. Назовите новый класс `MyRedButton`. Создайте подкласс класса `UIButton` и щелкните на кнопке Next.

3. Сохраните файл в папке проекта в группе Empty Window и выберите целевое приложение Empty Window. Щелкните на кнопке Create. Среда Xcode создает файлы MyRedButton.h и MyRedButton.m.

4. В файле MyRedButton.m в разделе реализации вставьте следующие строки:

```
awakeFromNib:- (void) awakeFromNib {
  [super awakeFromNib];
  self.backgroundColor = [UIColor redColor];
}
```

Теперь у нас есть подкласс UIButton, который при создании объекта кнопки из nib-файла делает ее фон красным. Однако ни в одном nib-файле у нас нет ни одного экземпляра этого подкласса. Исправим это. Откройте раскадровку, выберите кнопку и используйте инспектор идентичности для изменения класса кнопки на MyRedButton.

Теперь соберите и выполните проект. Убедитесь сами, кнопка стала красной!

Предупреждение для программистов в системе OS X

Если вы опытный программист в системе OS X, то, возможно, привыкли к тому, что метод super редко или никогда не вызывается из метода awakeFromNib; это вызывает исключение. В системе iOS вы всегда должны вызывать метод super в методе awakeFromNib. Другое важное отличие от системы OS X состоит в том, что владелец awakeFromNib вызывается во время загрузки nib-файла, поэтому существует возможность, что объект будет посылать сообщение awakeFromNib много раз. В системе iOS сообщение awakeFromNib посылается объекту только один раз, когда этот объект создается из nib-файла.

Еще одна возможность — воспользоваться преимуществами атрибутов User Defined Runtime Attributes в окне инспектора идентичности nib-файла. Это позволяет посылать сообщение setValue:forKey: объекту во время его создания из nib-файла. (На самом деле это сообщение setValue:forKeyPath:; ключевые пути (key paths) обсуждаются в главе 12.) Это позволяет конфигурировать в nib-редакторе аспекты nib-объекта, для которых в nib-редакторе нет встроенного интерфейса. Естественно, объект должен быть готов к реакции на заданный ключ, иначе ваше приложение при загрузке nib-файла завершится аварийно.

Например, один из недостатков nib-редактора заключается в том, что он не позволяет конфигурировать атрибуты уровня. Допустим, мы хотим с помощью nib-редактора закруглить углы нашей красной кнопки. Для этого в коде можно задать настройку кнопки layer.cornerRadius. Nib-редактор не дает доступа к этому свойству. Вместо этого мы можем выбрать кнопку в nib-редакторе и использовать раздел User Defined Runtime Attributes. Установим атрибут Key Path равным layer.cornerRadius, атрибут Type равным Number, а атрибут Value равным значению, которое мы хотим, — 10 (рис. 7.15). Соберите и выполните приложение; убедитесь, что углы кнопки теперь закруглены.

Для того чтобы вмешаться в процесс инициализации nib-объекта еще раньше, если объект относится к классу UIView (или его подклассу) или UIViewController (или его подклассу), можно реализовать метод initWithCoder:. Обратите внимание на то, что метод initWithCoder: не является стандартным инициализатором, предусмотренным шаблоном; вместо него в подклассе UIView есть метод initWithFrame:. Например, наш класс MyRedButton содержит шаблонный метод initWithFrame:. Однако это не дает нам никакой выгоды, потому что метод initWithFrame: не вызывается при создании объекта класса

UIView во время загрузки nib-файла, а вместо него вызывается метод initWithCoder:. (Новички часто делают типичную ошибку — реализуют метод initWithFrame:, а затем пытаются понять, почему их код не работает при создании представления из nib-файла.)

Возможная причина для реализации метода initWithCoder: совпадает с обоснованием для реализации любого другого инициализатора — для инициализации дополнительного экземпляра, объявленного в вашем подклассе. Структура метода initWithCoder: типична для любого инициализатора (см. главу 5): сначала вызывается метод super, а в конце возвращается объект self.

```
- (id) initWithCoder:(NSCoder *)aDecoder {
   self = [super initWithCoder:aDecoder];
   if (self) {
       self->_myIvar = // whatever
   }
   return self;
}
```

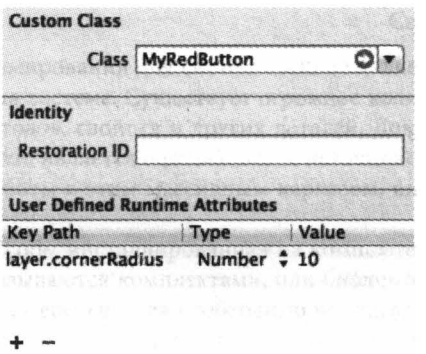


Рис. 7.15. Закругление углов кнопки с помощью атрибута в ходе выполнения приложения

ГЛАВА 8

Документация

Знание бывает двух видов. Мы либо знаем предмет, либо знаем, где найти информацию о нем.
Сэмюэль Джонсон, Биография

Ни один аспект программирования для системы iOS не является более важным, чем гибкая и реактивная справочная система. Существует огромное количество встроенных классов, содержащих множество методов, свойств и других деталей. Документация компания Apple, несмотря на все ее недостатки, является официальным источником информации о поведении Cocoa Touch и правилах работы с этим массивным каркасом, внутреннюю работу которого наблюдать невозможно.

Документация о среде Xcode, инсталлированная на компьютере, подразделяется на крупные фрагменты, которые называются комплектами, или библиотекой. Комплект невозможно просто установить на компьютере; сначала необходимо подписаться на него, чтобы вы могли получать его обновления, время от времени выпускаемые компанией Apple.

При первой инсталляции в среду Xcode огромная часть документации не устанавливается на компьютере; для просмотра документации в справочном окне (см. следующий раздел) потребуется интернет-соединение, чтобы можно было просматривать документацию на сайте компании Apple. Такая ситуация является неприемлемой; обычно пользователи хотят скопировать документацию на свою машину, чтобы работать с ней локально.

Следовательно, необходимо запустить среду Xcode немедленно после инсталляции и установить первоначальные комплекты документации. До некоторой степени этим процессом можно управлять и наблюдать за его выполнением на панели Downloads в окне Preferences (в разделе Documentation); в этом окне также можно указать, хотите ли вы автоматически инсталлировать обновления или время от времени щелкать на кнопке Check and Install Now вручную. Здесь также можно указать, какие комплекты документации вам нужны; я считаю, что для разработки программ для системы iOS необходимы комплекты iOS 7 и Xcode 5. Однако я слышал жалобы о том, что справочное окно (которое рассматривается в следующем разделе) не работает правильно, пока вы не установите комплект OS X 10.8 или 10.9. Кроме того, при первой установке комплекта документации необходимо ввести пароль администратора вашего компьютера.

В среде Xcode 5 комплекты документации инсталлируются в каталоге `Library/Developer/Shared/Documentation/DocSets`.

В отличие от предыдущих версий среды Xcode, версия Xcode 5 не позволяет работать со старыми комплектами документации. В этом пользователю может помочь приложение Dash (`http://kapeli.com/dash`).

Справочное окно

Основным механизмом доступа к документации в среде Xcode является справочное окно (Help⇨Documentation and API Reference, <Command+Option+Shift+?>). В справочном окне выполняется поиск информации; например, нажав клавиши <Command+Option+Shift+?> (или <Command+L>, если вы уже открыли справочное окно), наберите строку `NSString` и нажмите клавишу <Return>, чтобы выбрать главную подсказку NSString Class Reference. При желании, нажав пиктограмму с изображением увеличительного стекла, можно ограничить вывод информации о комплектах документации.

Существуют два способа просмотра результатов для поиска в справочном окне.

Всплывающее окно результатов

Если вы часто набираете строки в поле поиска, в окне результатов будут выводиться около десятка пунктов. Для выбора требуемого результата щелкните кнопкой мыши или воспользуйтесь клавишами навигации и нажмите клавишу <Return>. Если поле поиска находится в фокусе, вплывающее окно результатов можно открывать и сворачивать с помощью клавиши <Esc>.

Полное окно результатов

Если поле поиска находится в фокусе и на экране не открыто всплывающее окно результатов, нажмите клавишу <Return>, чтобы открыть окно со всеми результатами поиска. Эти результаты выводятся на четырех отдельных страницах, разделенных по категориям: API Reference, SDK Guides, Tools Guides и Sample Code.

Поиск в справочном окне можно начать, редактируя код. Эта ситуация возникает часто: вы смотрите на символ, который должен использоваться в вашем коде (имя класса, имя метода, имя свойства и т.д.), и хотите узнать о нем больше. Нажмите клавишу <Option> и наведите курсор мыши на слово в вашем коде, пока на экране не появится голубая пунктирная линия подчеркивания; затем (удерживая клавишу <Option>) дважды щелкните на этом слове. На экране откроется справочное окно, и вы получите прямой доступ к этому слову на странице документации об этом классе. Аналогично при выполнении кода (см. главу 9) вы можете щелкнуть на ссылке More и перейти к справке о текущем символе.

В качестве альтернативы можно выбрать текст в своем коде (или где-то еще) и выбрать команду Help⇨Search Documentation for Selected Text (<Command+Option+Control+/>). Это действие эквивалентно набору текста в поле поиска в справочном окне и просмотру страницы результатов.

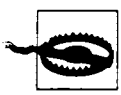
Справочное окно работает как веб-браузер, потому что по существу документация состоит из веб-страниц. Действительно, большинство страниц справочника можно найти на сайте разработчиков Apple `http://developer.apple.com`. Вы можете открыть в своем веб-браузере страницу, которую в данный момент просматриваете в справочном окне, выбрав команду Editor⇨Share⇨Open in Browser или щелкнув на кнопке Share панели окна и выбрав команду Open in Safari. Многочисленные страницы могут одновременно открываться как вкладки в справочном окне. Для этого удерживайте нажатой клавишу <Command> при переходе с вкладки на вкладку — например, нажмите клавишу <Command> и щелкните на ссылке, или нажмите клавишу <Command> и щелкните на пункте во всплывающем окне результатов, или выберите команду Open Link in New Tab в контекстном меню. Пользователь

может переходить с вкладки на вкладку (Window⇨Select Next Tab), причем каждая вкладка помнит свою историю переходов (выберите команду Navigate⇨Go Back или щелкните на кнопке Back инструментальной панели окна, которое служит всплывающим меню).

Страницу документации может сопровождать дополнительная таблица содержания или уточнений, или и того и другого. Эти таблицы отображаются на панели Info справа от справочного окна; если она не открыта, ее можно открыть с помощью команды Editor⇨Show Info или крайней правой кнопки инструментальной панели. Таблицы Table of Contents и Details — это разные панели в окне Info; для того чтобы переключаться между ними, щелкните на одной из пиктограмм на верхнем крае окна Info. Например, страница NSString Class Reference имеет вложенное окно с таблицей содержания, в которой записаны ссылки на все темы в этой странице, а вложенное окно Details ссылается на иерархию наследования класса `NSString`, его адаптированные протоколы и т.д. Мы еще обсудим этот класс в этой главе. Некоторые страницы документации могут использовать вложенное окно с таблицей содержания, чтобы показать место этой страницы среди более крупной группы страниц. Например, группа String Programming Guide состоит из множества страниц, и когда вы просматриваете одну из них, в окне Table of Contents перечисляются все страницы группы String Programming Guide вместе с главной темой каждой страницы.

Если вы собираетесь вернуться на какую-то страницу документации, сделайте закладку таким образом: выберите команду Editor⇨Share⇨Add Bookmark, или щелкните на кнопке Share инструментальной панели и выберите команду Add Bookmark, или (что еще проще) щелкните на пиктограмме закладки на левом поле страницы документации. Закладки отображаются на панели слева от справочного окна; для того чтобы ее открыть, выберите команду Editor⇨Show Bookmarks или щелкните на второй справа кнопке инструментальной панели. Управление закладками документации простое, но эффективное: закладки можно упорядочивать или удалять, и это все.

Для поиска текста на странице документации используйте команды меню Find. Команда Find⇨Find (<Command+F>) активизирует панель поиска, как в браузере Safari.



В отличие от предыдущих версий среды Xcode, версия Xcode 5 не имеет общей таблицы содержания документации. Это не позволяет пользователю работать с "домашней страницей" комплекта документации, на которой были бы перечислены все комплекты документов. Однако такая страница все же существует, и ее можно просматривать с помощью браузера. Я рекомендую найти комплект iOS 7 в папке `Library/Developer/Shared/Documentation/DocSets`, открыть его в окне Finder с помощью контекстного меню Show Package Contents, перейти в файл `Contents/Resources/Documents/navigation/index.html`, перетащить его в свой браузер и сделать закладку.

Страницы документации о классах

В подавляющем большинстве случаев искомой страницей документации является справка о классе. Для пользователей важно, чтобы страница справки о классе удобно и понятно описывала свойства класса, поэтому целесообразно остановиться на нем подробнее (рис. 8.1).

На справочной странице о классе особенно важную роль играет вложенное окно Details справочного окна Info, поэтому важно внимательно следить за информацией, которая отображается в окне Details при изучении класса (в презентации веб-браузера эта информация появляется в верхней части справочной страницы о классе).

Tasks

Creating Buttons

+ buttonWithType:

Configuring the Button Title

titleLabel *property*
reversesTitleShadowWhenHighlighted *property*
- setTitle:forState:
- setAttributedTitle:forState:
- setTitleColor:forState:
- setTitleShadowColor:forState:
- titleColorForState:
- titleForState:
- attributedTitleForState:
- titleShadowColorForState:

Configuring Button Presentation

adjustsImageWhenHighlighted *property*
adjustsImageWhenDisabled *property*
showsTouchWhenHighlighted *property*
- backgroundImageForState:
- imageForState:
- setBackgroundImage:forState:

Inherits From
UIControl
UIView
UIResponder
NSObject
Conforms To
UIAppearance
UIAppearanceContainer
NSObject
UIDynamicItem
NSCoding
Framework
UIKit
Availability
iOS 2.0 and later
Declared In
UIButton.h
Sample Code
avTouch
Popovers
TicTacToe
UICatalog
AddMusic

Рис. 8.1. Начало справочной страницы о классе `UIButton`

Раздел Inherits From

Содержит ссылки на цепочки подклассов. Одна из самых больших ошибок новичков заключается в пренебрежении документацией о цепочке суперклассов. Класс является наследником своих суперклассов, поэтому функциональные свойства или информация, которую вы ищете, могут находиться в суперклассе. На странице класса `UIButton` не следует искать информацию о методе `addTarget:action:forControlEvents:`, так как она находится на странице класса `UIControl`. Не стоит также искать свойство `frame` в классе `UIButton` на странице класса `UIButtone`, — эта информация находится на странице класса `UIView`.

Раздел Conforms To

Содержит ссылки на протоколы, адаптированные классом. Протоколы обсуждаются в главе 10.

Раздел Framework

Сообщает, какому каркасу принадлежит этот класс. Для того чтобы использовать этот класс, ваш код должен ссылаться на этот каркас и импортировать заголовок каркаса (см. главу 6).

Раздел Availability

Указывает самую раннюю версию операционной системы, в которой реализован данный класс. Например, класс `UIDynamicAnimator` (вместе с каркасом UIKit Dynamics) не был

изобретен до появления системы iOS 7. Таким образом, если вы хотите использовать эту функциональную возможность в своем приложении, то должны задать как цель только iOS 7 или более позднюю версию или указать, что ваше приложение никогда не использует этот класс при выполнении в системах более ранних версий.

Раздел Declared In

Заголовок (или заголовки), в котором объявлен класс. К сожалению, это не ссылка; я не нашел быстрого способа для просмотра заголовка в документации. Это неудобно, поскольку часто возникает необходимость просмотреть содержимое заголовочного файла, который может содержать полезные комментарии или другие детали. Заголовочный файл можно открывать в окне проекта, как будет показано далее.

Раздел Sample Code

Если справочная страница о классе ссылается на пример кода, может возникнуть желание просмотреть этот код. (См. мои замечания о примерах класса в следующем разделе.)

Раздел Related

Если справочная страница класса содержит список ссылок, щелкните на них и прочитайте справку. Например, справочная страница класса `UIView` содержит ссылки на справочник View Programming Guide for iOS. Справки представляют собой обзоры тем, содержат важную информацию (часто вместе с примерами кода) и могут служить ориентиром для принятия решений.

Справочная страница класса разделяется на разделы, перечисленные во вложенном окне Table of Contents в окне Info.

Раздел Overview

Некоторые справочные страницы класса содержат чрезвычайно важную информацию в разделе Overview, включая ссылки на связанные справки и другую информацию. (Например, см. справочную страницу класса `UIView` .)

Раздел Tasks

В этом разделе в категориальном порядке перечисляются ссылки на свойства и методы, упоминаемые на странице. Часто даже простой просмотр этого списка может дать полезную подсказку.

Разделы Properties, Class Methods, Instance Methods

Эти разделы содержат полную документацию о свойствах и методах класса. (Свойство — это метод, как правило, даже два метода доступа, get- и set-методы, — но в списке перечисляются именно свойства, а не методы доступа.)

Раздел Constants

Многие классы определяют константы, используемые в конкретных методах. Например, чтобы создать экземпляр класса `UIButton` в коде, можете вызвать метод класса `buttonWithType:`; значение аргумента будет константной, указанной в разделе Constants справочной страницы класса `UIButtonType`. (Для того чтобы облегчить ее поиск, в этом разделе есть ссылка из метода `buttonWithType:` в подраздел UIButtonType из раздела Constants.) Существует формальное определение константы, которое обычно

не интересует программиста (см. главу 1, в которой описано, как прочитать формальное определение). Затем каждое значение объясняется, а имя значения можно скопировать и вставить в код.

В заключение поговорим о том, как на справочной странице класса перечисляются и объясняются индивидуальные свойства и методы. В последние годы эта часть документации достигла совершенства и снабжена прекрасными ссылками. Отметим следующие подразделы, следующие за именами свойств или методов.

Раздел Description

Краткое описание предназначения свойства или метода.

Раздел Formal Declaration

Описываются параметры метода и тип возвращаемого значения. (Объявление свойства описывается в главе 12.)

Раздел Parameters and Return Value

Точная информация о смысле и предназначении параметров и возвращаемого значения.

Раздел Discussion

Часто содержит чрезвычайно важные детали о поведении метода. Этот раздел необходимо внимательно изучить!

Раздел Availability

При обновлении операционной системы в старый класс можно включать новые методы; если новый метод критически важен для вашего приложения, вы можете запретить ее выполнение под управлением устаревших операционных систем, в которых этот метод не реализован.

Раздел See Also

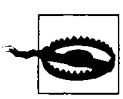
Содержит ссылки на связанные методы. Очень полезен для понимания места метода в классе.

Раздел Related Sample Code

Иногда целесообразно посмотреть на образец кода, чтобы понять, как работает метод в реальной жизни.

Раздел Declared In

Соответствующий заголовочный файл.



Методы, включенные в класс с помощью категории (глава 10), часто не упоминаются на справочной странице, и их бывает трудно найти. Например, метод awakeFromNib (см. главу 7) не упоминается ни в документации о классе UIButton, ни в описаниях его суперклассов или протоколов. Это главный недостаток справочной системы компании Apple. В этом случае вам может помочь справочная система сторонней организации, например AppKiDo (`http://appkido.com`).

Образцы кода

Компания Apple предоставляет большое количество образцов кода для проектов. Такой код можно просмотреть непосредственно в справочном окне; иногда этого достаточно, но в этом случае можно увидеть только реализацию одного класса или заголовочный файл, а для обзора этого мало. В качестве альтернативы можно открыть образец кода для проекта в среде; щелкните на кнопке Open Project в верхней части страницы, содержащей образец кода в справочном окне. Если просматривать образец кода в браузере на странице `http://developer.apple.com`, на экране появится кнопка Download Sample Code. Открыв образец кода в окне проекта, вы можете читать, просматривать, редактировать и, разумеется, выполнять проект.

Как справка образец кода и хорош и плох. Он может быть превосходным источником работоспособного кода, который можно копировать, вставлять и использовать с небольшими изменениями в своих проектах. Обычно он содержит многочисленные комментарии, потому что разработчики компании Apple понимали, что пишут этот код с целью обучения. Пример кода также иллюстрирует концепции, которые пользователю трудно извлечь из документации. (Например, пользователи, не разобравшиеся с классом `UITouch`, часто начинают понимать его, анализируя пример `MoveMe`.) Однако логика проекта часто разбросана по множеству файлов, и нет ничего труднее, чем пытаться понять, что они означают (разумеется, кроме вашего собственного кода). Более того, ученикам более всего необходимо понять не столько сам законченный проект, сколько процесс его разработки, описанный в комментариях. Именно этого в образцах кода и недостает.

По-моему, образцы кода, предоставляемые компанией Apple, неравноценны. Некоторые из них написаны небрежно и даже содержат ошибки, а другие просто превосходны. Впрочем, обычно эти программы поучительны и определенно являются важной частью документации, а также значительно облегчают работу. Однако наибольшую ценность они приобретают только после того, как вы выйдете на определенный уровень компетентности.

Другие ресурсы

Перечислим другие полезные ресурсы, дополняющие документацию.

Справка Quick Help

Справка Quick Help — это лаконичная документация о конкретных темах, обычно об имени символа (класса или метода). Если на экране открыто окно инспектора Quick Help (<Command+Option+2>), в нем отображается информация о текущем выборе или точке вставки. Например, если вы редактируете код и точка вставки или выделение относятся к слову `CGPointMake`, то в окне инспектора Quick Help будет отображаться справка о структуре `CGPointMake`.

Кроме того, справку Quick Help можно получить в окне инспектора Quick Help для объектов интерфейса, выделенных в nib-редакторе, для настроек сборки при редактировании проекта или цели и т.д.

Документация Quick Help может также отображаться во всплывающем окне, а не в окне инспектора Quick Help. Выберите слово и команду Help⇨Quick Help for Selected Item (<Command+Control+Shift+?>). В качестве альтернативы нажмите клавишу <Option> и установите курсор мыши на выделенное слово, пока он не превратится в знак вопроса (а само слово будет подчеркнуто голубой пунктирной линией), и, удерживая клавишу <Option>, щелкните кнопкой мыши.

Документация Quick Help содержит ссылки. Например, если щелкнуть на ссылке Reference, в справочном окне откроется полная документация; если щелкнуть на ссылке на заголовочный файл, то в окне откроется соответствующий заголовочный файл.

В среде Xcode 5 появилась возможность вставлять документацию о ваших собственных методах в документацию Quick Help. Для этого необходимо написать комментарий в формате doxygen (см. `http://www.stack.nl/~dimitri/doxygen/`) или HeaderDoc (найдите справку о формате "HeaderDoc" на веб-странице `http://developer.apple.com`). Я рекомендую формат doxygen, который стал фактически стандартом. Перед объявлением или определением метода (или свойства) вставляется комментарий. Например:

```
/*! Many people would like to dog their cats. So it is \e perfectly
reasonable to supply a convenience method to do so.

\param cats A string containing cats
\return A string containing dogs */

- (NSString*) dogMyCats: (NSString*) cats {
  return @"Dogs";
}
```

Знак восклицания в начале комментария означает, что далее следует комментарий в формате doxygen, а местоположение этого комментария автоматически связывает его с методом `dogMyCats:`, определение которого следует сразу за комментарием. Выражения с обратной косой чертой, `\e`, `\param` и `\return`, являются командами системы doxygen. В результате, если где-то в коде будет выделен метод `dogMyCats:`, справка о нем будет отображаться в окне Quick Help (рис. 8.2). Описание метода в формате doxygen (или его краткое описание, которое обозначается командой `\brief`) также отображается как часть кода (подробнее об этом — в главе 9).

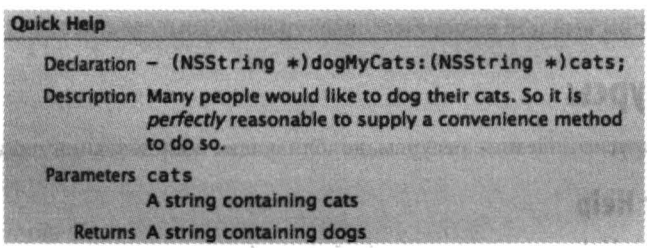


Рис. 8.2. Документация о пользовательском методе, вставленная в окно Quick Help

Символы

Символ — это глобально определенный термин, например имя класса, метода или переменной экземпляра. Если вы можете увидеть имя символа в своем коде в среде Xcode, то можете быстро перейти к описанию этого символа. Выделите текст и выберите команду Navigate⇨Jump to Definition (<Command+Control+J>). В качестве альтернативы нажмите клавишу <Command> и наведите курсор мыши на искомый термин, пока курсор не примет вид указательного пальца (а термин будет подчеркнут голубой пунктирной линией); нажмите клавишу <Command> и щелкните на термине мышью, чтобы перейти на этот символ.

Если символ определен в каркасе Cocoa, вы переходите к соответствующему объявлению в заголовочном файле. Если символ определен в вашем коде, вы переходите в определение

класса или метода; это может быть очень полезным не только для понимания своего кода, но и для перемещения по нему.

Точный смысл слова "переход" зависит от клавиш модификации, которые используются в комбинации с клавишей <Command> и настройками на панели Navigation среди предпочтений среды Xcode. По умолчанию комбинация клавиши <Command> и щелчка мышью приводит к переходу в рамках того же окна редактора, комбинация <Command+Option+щелчок> выполняет переход в окно помощника, а комбинация <Command+двойной щелчок> переносит вас в новое окно. Аналогично комбинация <Command+Option+Control+J> переносит пользователя в окно помощника в определение выделенного термина.

Другой способ просмотра списка символов, существующих в проекте, и перехода к определению символа основан на использовании окна навигатора символов (см. главу 6).

Важным, но часто несправедливо забываемым, способом перехода к определению символа, имя которого вам известно, даже если вы не видите его в коде на экране, является команда File⇨Open Quickly (<Command+Shift+O>). Наберите в поле поиска ключевые буквы имени, которые система будет интерпретировать, например, для поиска метода `applicationDidFinishLaunching:` можно набрать буквы "appdid". В прокручиваемом списке под полем поиска появятся возможные совпадения; этот список можно прокручивать с помощью мыши или клавиш. Определения в вашем коде перечисляются перед определениями из заголовков каркаса Cocoa. Этот способ может обеспечить быстрый способ навигации по коду.

Заголовочные файлы

Иногда полезной формой документации может стать заголовочный файл. Он компактно описывает переменные и методы экземпляров класса и может содержать комментарии с полезной информацией, которую больше нигде не найти. Отдельный заголовочный файл может содержать объявления нескольких интерфейсных и протокольных классов. Таким образом, он может быть прекрасным справочником.

Существует несколько способов просмотра заголовочных файлов в редакторе среды Xcode.

- Если класс написан вами и вы редактируете файл реализации, выберите команду Navigate⇨Jump to Next Counterpart (<Command+Control+Up>).
- Активизируйте меню Related Files на панели быстрых переходов (<Control+1>). Это меню позволяет переходить в любой заголовочный файл, импортируемый в текущий файл (и в любые файлы, которые импортируются в текущий файл), а также в заголовочные файлы суперклассов и подклассов текущего класса и т.д. Для того чтобы перейти в окно помощника, нажмите и удерживайте клавишу <Option>.
- Выделите текст и выберите команду File⇨Open Quickly (<Command+Shift+O>), как описано в предыдущих разделах.
- Нажмите клавишу <Command> и щелкните на символе, выберите команду Navigate⇨Jump to Definition или перейдите в справочное окно Quick Help, как описано в предыдущих разделах.
- Используйте навигатор символов.

Все эти подходы требуют, чтобы окно проекта было открытым; для того чтобы операция выполнялась эффективно, команда File⇨Open Quickly требует, чтобы комплект SDK был активным, а другие команды работают с конкретным окном или словами в открытом проекте. Альтернатива, которая работает всегда, даже если ни один проект не открыт, заключается в

переходе в окно Terminal и использовании команды `-h` для открытия заголовочного файла в среде Xcode. Аргумент команды может быть частью имени заголовочного файла. Если возникает неоднозначность, то эта команда становится интерактивной; например, команда `-h NSString` может открыть как файл `NSString.h`, так и файл `NSStringDrawing.h` (или оба, или ни одного). Я бы хотел, чтобы эта команда была встроена в саму среду Xcode.

Ресурсы Интернета

Интернет и поисковая система Google значительно облегчили процесс программирования. Интересно, что можно найти с помощью системы Google search. Весьма вероятно, что вашу задачу уже кто-то решил и написал об этом в Интернете. Часто можно найти образец кода, который можно вставить в свой проект и адаптировать.

Справочные ресурсы компании Apple доступны по адресу `http://developer.apple.com`. Эти ресурсы обновляются, прежде чем внести изменения в комплекты документации, загружаемые разработчиками. Кроме того, существуют определенные материалы, которые не являются частью документации среды Xcode на вашем компьютере. Если вы зарегистрированный пользователь системы iOS, то имеете доступ к видеозаписям iTunes, включая видеозаписи всех сессий конференции WWDC 2013 (и предыдущих конференций) и форумов разработчиков Apple (`https://devforums.apple.com`). К тому же большинство документации компании Apple имеет альтернативный формат PDF, удобный для хранения и просмотра на устройстве iPad. Что еще лучше, приложение Docsets, разработанное Оле Цорном (Ole Zorn), позволяет загружать и просматривать все комплекты документов на устройствах iPad или iPhone. Это приложение не бесплатное, но относится к программам с открытым исходным кодом (`https://github.com/omz/DocSets-for-iOS`), так что вы можете свободно собрать ее самостоятельно.

Компания Apple поддерживает несколько почтовых рассылок (`http://lists.apple.com/mailman/listinfo`). Долгое время я был подписчиком группы Xcode-users (чтобы задавать вопросы об использовании инструментов Xcode) и группы Cocoa-dev (чтобы задавать вопросы по программированию с помощью каркаса Cocoa). В этих списках можно выполнять поиск, но поисковая система компании Apple работает не очень хорошо; лучше использовать поисковую систему Google с помощью термина `site:lists.apple.com` или адреса `http://www.cocoabuilder.com`, на котором заархивированы эти списки. Компания Apple не добавила списки рассылки, посвященные программированию для системы iOS; для этого предназначены форумы разработчиков, но их интерфейс крайне корявый, к тому же их невозможно открыть через Google и вообще извне, что снижает их полезность.

Со временем спонтанно возникли и стали популярными другие интернет-источники, посвященные программированию для системы iOS, а многие программисты ведут блог, на котором описывают свой опыт программирования в системе iOS с использованием каркаса Cocoa. В частности, я люблю сайт Stack Overflow (`http://www.stackoverflow.com`); конечно, он не посвящен исключительно программированию для системы iOS, но большинство программистов общаются здесь, лаконично и правильно отвечают на вопросы, а интерфейс позволяет быстро и легко найти правильный ответ на свой вопрос.

ГЛАВА 9

Жизненный цикл проекта

В этой главе мы рассмотрим некоторые из основных этапов жизненного цикла проекта в среде Xcode: от задумки до представления в магазин App Store. Этот обзор позволит обсудить некоторые дополнительные возможности интегрированной среды разработки Xcode. Вы уже знаете, как создать проект, определить класс и установить связь с каркасом (см. главу 6), а также как создать и отредактировать nib-файл (см. главу 7) и как использовать документацию (см. главу 8).

Архитектура устройства и условный код

Во время создания проекта (File⇨New⇨Project), после выбора шаблона проекта, на экране, в котором вводится имя проекта, появляется раскрывающийся список Devices, содержащий пункты iPad, iPhone или Universal. Эту настройку можно изменить позднее (используя раскрывающийся список Devices на вкладке General при редактировании целевого приложения), но ваша жизнь будет значительно легче, если вы сразу сделаете правильный выбор, потому что он влияет на детали шаблона, лежащего в основе нового проекта.

Устройства iPhone и iPad отличаются своими физическими характеристиками, а также программными интерфейсами. Устройство iPad имеет более крупный экран, а также некоторые встроенные возможности интерфейса, которых нет на устройстве iPhone, например, разделенные представления и всплывающие меню. Универсальные приложения работают как на iPhone, так и на iPad, причем на каждом устройстве обычно используется интерфейс специального типа.

Выбор пункта в раскрывающемся списке Devices также влияет на настройку сборки Targeted Device Family.

iPad

Приложение будет выполняться только на iPad.

iPhone

Приложение будет работать на iPhone или iPod touch; оно также может работать на iPad, но не как естественное приложение для iPad (оно выполняется в уменьшенном масштабируемом окне, которое я называю iPhone Emulator; компания Apple иногда называет это "режимом совместимости").

iPhone/iPad

Приложение будет естественным образом выполняться на обоих типах устройств и должно быть структурировано как универсальное.

Если вы обновляете приложение для системы iOS 7 и хотите, чтобы оно естественным образом выполнялось на 64-битовом устройстве, измените настройку сборки Architectures для целевого приложения на "Standard architectures (including 64-bit)".

Две дополнительные настройки уровня проекта определяют, какая система установлена на вашем устройстве.

Base SDK

Новейшая система, в которой может работать ваше приложение. Когда я писал книгу, в среде Xcode 5.0 существовало только две возможности: iOS 7.0 и Latest iOS (iOS 7.0). Они выглядят одинаково, но второй выбор лучше (и он предлагается по умолчанию для нового проекта). Если вы обновите среду Xcode для разработки приложения, которое будет работать в последующих версиях операционной системы, любые существующие проекты, которые уже настроены на Latest iOS, будут использовать этот новейший комплект SDK в качестве своего комплекта Base SDK автоматически, без изменения настройки Base SDK.

iOS Deployment Target

Самой ранней операционной системой, на которой может работать ваше приложение в среде Xcode 5.0, может быть любая система iOS, вплоть до версии 4.3. Для того чтобы легко изменить настройку проекта iOS Deployment Target, откройте проект и перейдите на вкладку Info, а затем выберите пункт в раскрывающемся списке iOS Deployment Target. (Для того чтобы легко изменить настройку цели iOS Deployment Target, откройте цель, перейдите на вкладку General и выберите пункт в раскрывающемся списке Deployment Target. Однако, как правило, эту настройку изменяют на уровне проекта и предоставляют цели автоматически адаптировать настройки проекта.)

Написать приложение, в котором настройка Deployment Target отличалась бы от Base SDK, довольно сложно. С этим связаны две основные проблемы.

Неподдерживаемые функциональные возможности

В каждой новой системе компания Apple добавляет новые функциональные возможности. Среда Xcode охотно позволяет вам компилировать любые функции из комплекта Base SDK, даже если их нет в системе, заданной настройкой Deployment Target. Но ваше приложение даст сбой, если система выполнения приложений обнаружит функцию, не поддерживаемую системой, в которой приложение в действительности выполняется. Таким образом, если вы установили настройку проекта Deployment Target равной iOS 6, ваш проект будет скомпилирован и приложение будет работать в системе iOS 6, даже если она содержит функциональные свойства, доступные только в системе iOS 7, но ваше приложение даст сбой в системе iOS 6, если одна из таких функций будет действительно обнаружена.

Модифицированное поведение

В каждой новой системе компания Apple позволяет себе изменять механизм работы некоторых функций. В результате такие функции в разных системах работают по-разному. В некоторых случаях один и тот же метод может делать две разные вещи, в зависимости от того, в какой системе выполняется приложение. В других случаях функции в разных системах можно обрабатывать по-разному, реализуя или вызывая разные наборы методов.

Таким образом, обратная совместимость может потребовать написать дополнительный код — иначе говоря, такой код, в котором в разных системах выполняются разные наборы инструкций. Но дело не только в коде. Проект может содержать другие источники, например nib-файл, не совместимые с более ранними системами. (Например, nib-файл, использующий автоматическую разметку, вызовет сбой при загрузке в системе iOS 5.1 или более ранней, потому что автоматическая разметка использует класс `NSLayoutConstraint`, которого раньше не было.)

Даже если вы не стремитесь к обратной совместимости, вам все равно, возможно, придется решать проблемы, связанные с условным кодом, если вы хотите написать универсальное приложение. Несмотря на то что вы, возможно, стремитесь сократить дублирование общего кода для версий приложения, предназначенных для устройств iPhone и iPad, какой-то код придется написать отдельно, потому что ваше приложение должно работать иначе на разных устройствах. Как я уже говорил, вы не можете открыть всплывающее окно на iPhone; сложности могут значительно возрасти, потому что интерфейсы могут сильно отличаться и работать совершенно по-разному — нажатие на ячейке таблицы на устройстве iPhone может активизировать новый экран, а на более крупном устройстве iPad это может просто изменить часть экрана.

Разные программируемые устройства могут по-разному интерпретировать код в зависимости от системы или типа устройства, на котором выполняется приложение; таким образом, существует возможность предотвратить сбой или неправильную работу кода, используя настройки среды выполнения приложений.

Явное тестирование среды

Класс `UIDevice` позволяет запрашивать текущее устройство (`currentDevice`), чтобы выяснить версию (`systemVersion`) и тип (`userInterfaceIdiom`) его системы.

Для примера создайте проект Universal на основе шаблона Master–Detail Application и загляните в файл `AppDelegate.m`. Вы увидите код в методе `application:didFinishLaunchingWithOptions:`, который конфигурирует начальный интерфейс по-разному в зависимости от типа устройства, на котором выполняется приложение.

Суффикс имени ключа `Info.plist`

Настройки `Info.plist` можно конфигурировать только на один тип устройства, добавляя суффикс `~iphone` или `~ipad` к имени ключа. Если суффикс уже есть, эта настройка заменяет общую настройку (без суффикса) на подходящее устройство.

Откройте проект Universal, основанный на шаблоне Master–Detail Application. Вы увидите, что файл `Info.plist` содержит два набора настроек "Supported interface orientations": общий набор (`UISupportedInterfaceOrientations`) и набор настроек только для устройства iPad, который заменяет общий набор, когда приложение выполняется на iPad (`UISupportedInterfaceOrientations~ipad`).

Аналогично проект Universal, основанный на шаблоне Master-Detail Application, содержит две раскадровки: одну, обеспечивающую интерфейс для iPhone, и другую для iPad. Выбор между ними осуществляет настройка `Info.plist` "`Main storyboard file base name`", которая появляется дважды: в общем случае (`UIMainStoryboardFile`) и только для iPad (`UIMainStoryboardFile~ipad`), которая заменяет общую, когда приложение выполняется на устройстве iPad.

Суффикс имени ресурса

Многие вызовы функций, загружающих ресурсы по имени из комплекта приложения, подчиняются тем же правилам использования суффиксов в именах, что и ключи в файле `Info.plist`, автоматически выбирая альтернативные ресурсы, имена которых до расширения заканчиваются суффиксом `~iphone` или `~ipad`, соответствующим типу устройства. Например, если задать имя изображения `@"linen.png"`, то метод `imageNamed:` в классе `UIImage` загрузит изображение `linen~ipad.png`, если он найдет его и приложение работает на устройстве iPad.

(Однако в среде Xcode 5 есть новшество: если изображение находится в каталоге ресурсов, соглашение об именах можно не выполнять. Так как используемое изображение определяется его местом в этом каталоге. Выберите множество изображений в этом каталоге и пункт Device Specific в раскрывающемся списке Devices в окне инспектора атрибутов для создания разных версий изображения для устройств iPhone и iPad. Это одна из многих причин использования каталогов ресурсов!)

Слабо связанные каркасы

Если ваше приложение связано с каркасом и пытается работать в системе, в которой этого каркаса нет, то во время его выполнения произойдет сбой. Для решения этой проблемы можно установить необязательную связь с каркасом, изменив пункт Required в листинге на фазе сборки Linked Frameworks and Libraries на `Optional`. С технической точки зрения это называется слабой связью с каркасом (weak-linking the framework).

(Этот метод работает, даже если вы используете новые модули в среде Xcode 5 (см. главу 6), — но в этом случае вы должны ясно установить слабую связь с требуемым каркасом, сделав так, чтобы его имя появилось в списке Linked Frameworks and Libraries; автоматического способа задать слабую связь не существует.)

Проверка существования метода

Существует возможность проверить, существует ли требуемый метод, используя метод `respondsToSelector:` и вызовы объекта класса `NSObject`

```
if ([UIButton respondsToSelector: @selector(appearance)]) {
// ok — вызываем метод класса appearance} else {
// не вызываем метод класса appearance
}
```

Проверка существования класса

Существует возможность проверить, существует ли класс, используя функцию `NSClassFromString`, которая возвращает `nil`, если класс не существует. Кроме того, если используется версия Base SDK 5.0 или более поздняя и каркас, которому принадлежит класс, существует или имеет слабую связь с приложением, можно послать этому

классу любое сообщение (как классу) и проверить, равен ли результат nil; этот механизм работает, потому что классы, начиная с системы iOS 5, являются слабо связанными.

```
// считаем, что каркас Core Image является слабо связанным
if ([CIFilter class]) { // ok - можно работать с классом CIFilter
```

Проверка существования констант и функций

Существует возможность проверить, существует ли имя константы, включая имя функции в языке C, взяв адрес имени и сравнив его с нулем. Например:

```
if (&UIApplicationWillEnterForegroundNotification) {
    // OK - можно ссылаться на UIApplicationWillEnterForegroundNotification
```

Управление версиями

Рано или поздно в жизни реального приложения наступает момент, когда приходится включать проект в систему управления версиями. Управление версиями — это способ периодического снятия снимков (формально называемых фиксациями (commits)) вашего проекта. Оно имеет следующие цели.

Безопасность

Система управления версиями может помочь вам хранить копии в автономном репозитарии, чтобы ваш код не был потерян в случае локального сбоя на компьютере или другого "несчастного случая".

Сотрудничество

Управление версиями позволяет нескольким разработчикам работать с одним и тем же кодом.

Свобода от страха

Проект — сложная вещь; часто изменения должны вноситься экспериментально, иногда сразу во многие файлы, иногда в течение многих дней, прежде чем новая функциональная возможность будет протестирована. Управление версиями означает, что программист может легко отследить свои действия (до предыдущей фиксации), если что-то получится плохо; это придает ему смелости предпринимать попытки, в успехе которых он не уверен. Кроме того, если программист сомневается в правильности выбранного пути, он может попросить систему управления версиями перечислить изменения, которые сделаны им недавно. Если обнаружится ошибка, он может вернуться к нужной версии и выявить проблему.

Среда Xcode имеет несколько механизмов управления версиями, основанных на системах Git (http://git-scm.com) и Subversion (http://subversion.apache.org, которая также называется SVN). Это не значит, что любую другую систему управления версиями в своем проекте использовать невозможно! Это лишь значит, что нельзя использовать любую другую систему управления версиями как интегрированную часть среды Xcode. Это не страшно, так как для управления версиями существует много других способов, и даже системы Git и Subversion можно использовать, игнорируя среду Xcode и работая с системой управления версиями в режиме командной строки в окне Terminal или использовать графический пользовательский интерфейс сторонних разработчиков, таких как svnX for Subversion (http://www.lachoseinteractive.net/en/products) или SourceTree for git (http://www.sourcetreeapp.com).

Если вы не хотите использовать систему управления версиями, интегрированную в среду Xcode, можно просто ее отключить (частично или полностью). Если снять флажок Enable Source Control в окне настроек Source Control, то можно лишь выбрать команду Check Out в меню Source Control, чтобы загрузить код с удаленного сервера. Если установить флажок Enable Source Control, то три дополнительных флажка позволят выбрать автоматический режим. Лично я предпочитаю устанавливать флажки Enable Source Control и Refresh local status automatically, чтобы среда Xcode отображала статус файла в окне навигатора проекта; остальные два флажка я оставляю сброшенными, потому что управляю версиями вручную. (Возможность сбросить флажок Add and remove files automatically в среде Xcode 5 особенно полезна; в среде Xcode 4 добавление файлов в индекс системы Git в момент добавления в проект было неудобным.)

При создании нового проекта в диалоговом окне Save имеется флажок, предусматривающий возможность с самого начала разместить репозитарий системы Git в папке проекта. Этот репозитарий можно разместить как на вашем компьютере, так и на удаленном сервере. Если у вас нет иных причин, предлагаю установить этот флажок!

Открывая существующий проект, использующий системы Subversion или Git, среда Xcode обнаруживает этот факт и может мгновенно отображать информацию о версии в окне интерфейса. Если используется удаленный репозитарий, среда Xcode автоматически включает информацию об этом в окно настроек Accounts, которое в среде Xcode 5 является унифицированным интерфейсом для управления репозитарием. Для того чтобы использовать удаленный сервер без проверки рабочей версии, введите эту информацию в окне настроек Accounts.

Если вы установили флажок Refresh local status automatically, файлы в окне навигатора проекта отображаются вместе со своим статусом. Например, если вы используете систему Git, то можете различать модифицированные файлы (M), новые неотслеженные файлы (?) и новые файлы, добавленные в индекс (A).

Действия, связанные с реальным управлением версиями, можно выполнить в меню Source Control (впервые включенное в верхний уровень в среде Xcode 5) и в контекстное меню в навигаторе проекта. Для того чтобы проверить и открыть проект, хранящийся на удаленном сервере, выберите команду Source Control⇨Check Out. Другие пункты меню Source Control очевидны, например Commit, Push, Pull (or Update), Refresh Status и Discard Changes. Особенно отметим первый пункт в меню Source Control (новшество в среде Xcode 5), в котором перечислены все открытые рабочие версии по именам и ветвям; это иерархическое меню позволяет осуществлять управление ветвями.

Выбрав команду Source Control⇨Commit, вы увидите представление сравнения всех измененных файлов. Из этой фиксации можно исключить как отдельные, так сразу все изменения, чтобы группа связанных файлов образовала совокупность осмысленных фиксаций. Аналогичное представление сравнений для любой фиксации можно открыть с помощью команды Source Control⇨History. (Правда, среда Xcode все еще не поддерживает представление ветвей с помощью инструмента `gitk`.) Конфликты слияний также изображаются с помощью удобного графического интерфейса, демонстрирующего сравнения.

Существует также возможность в любой момент времени сравнивать разные версии редактируемого файла с помощью редактора версий; для этого необходимо выбрать команду View⇨Version Editor⇨Show Version Editor или щелкнуть на третьей кнопке Editor инструментальной панели окна проекта. Редактор версий может работать в трех режимах: в режиме сравнения (Comparison view), режиме осуждения (Blame view) и режиме регистрации (Log view). Для выбора режима необходимо выбрать команду View⇨Version Editor или использовать раскрывающийся список, который открывается после щелчка на третьей кнопке Editor инструментальной панели в окне редактора версий).

Например, как показано на рис. 9.1, в этом окне можно увидеть, что в последней версии данного файла (слева) я больше не использую явно объект класса `NSStringDrawingContext`: в качестве аргумента метода `context:`. Если выбрать команду Editor⇨Copy Source Changes, соответствующий файл `diff` помещается в буфер. Если переключиться в режим осуждения, то можно увидеть мое собственное сообщение фиксации: "`eliminated NSStringDrawingContext`". Панель быстрых переходов внизу окна редактирования версий позволяет просматривать в редакторе любую версию фиксации текущего файла.

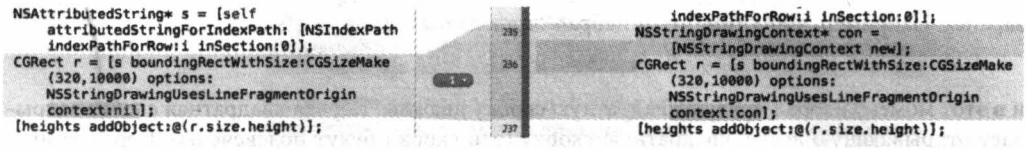


Рис. 9.1. Сравнение версий

Другой способ узнать, как изменилась строка, который впервые появился в среде Xcode 5, — выделить часть этой строки (в обычном редакторе) и выбрать команду Editor⇨Show Blame For Line. На экране появится всплывающее окно, описывающее фиксацию, в которой эта строка приняла текущий вид. Используя кнопки в этом всплывающем окне, можно провести непосредственное сравнение версий этой фиксации или переключиться в режим осуждения или сравнения.

В среде Xcode также есть собственный механизм создания и хранения копий вашего проекта в целом. Для этого необходимо выбрать команду File⇨Create Snapshot (в соответствии с вашими настройками некоторым массовым операциям, таким как Найти и заменить или Выполнить рефакторинг, может предшествовать создание копий). Несмотря на то что эти копии нельзя считать полноценной системой управления версиями, фактически они поддерживаются репозитариями системы Git и могут подстраховывать программиста перед внесением изменений, о которых он впоследствии может пожалеть. Копиями можно управлять на вкладке Projects в окне Organizer; здесь можно экспортировать копии, возвращаясь к предыдущему состоянию папки проекта.

Редактирование кода

Многие аспекты среды редактирования в системе Xcode можно настроить по своему усмотрению. На первом шаге необходимо выбрать начертание и размер шрифта в настройке Source Editor на панели настроек Fonts & Colors. Это очень важно для того. чтобы писать и читать код было удобно! Я предпочитаю крупные шрифты (13, 14 и даже 16) и приятный моноширинный шрифт, такой как Monaco, Menlo или Consolas (или свободно распространяемый шрифт Inconsolata).

В среде Xcode существуют возможности форматирования, автозаполнения и выбора текста, адаптированные для языка Objective-C. Их реализация зависит от настроек на вкладках Editing и Indentation панели настроек Text Editing. Я не собираюсь подробно описывать эти настройки детально, но советую вам воспользоваться ими. На вкладке Editing я бы установил все флажки, включая Line Numbers; отображение номеров строк полезно при редактировании. На вкладке Indentation я бы также установил все флажки; я считаю, что макет кода на языке Objective-C в среде Xcode при этих настройках превосходен.

Если вам нравится интеллектуальная система выравнивания текста в среде Xcode и вы обнаружили, что выравнивание вашей строки кода оказалось неправильным, выберите команду

Editor⇨Structure⇨Re-Indent (<Control+I>), которая выполняет автоматическую подстановку текущей строки. (Проблемы с автоматическим выравниванием могут быть вызваны нарушением синтаксических правил в файле, так что это также может служить признаком ошибки.)

Обратите внимание на флажок Automatically balance brackets in Objective-C method calls. на вкладке Editing. Если этот флажок установлен, то, когда вы поставите закрывающую квадратную скобку после определенной части текста, среда Xcode автоматически вставит открывающую квадратную скобку перед этим фрагментом текста. Мне нравится эта возможность, поскольку она позволяет мне набирать вложенные квадратные скобки, не планируя их заранее. Например, допустим, что я набрал строку

```
UIAlertView* av = [UIAlertView alloc
```

и в этот момент набрал правую квадратную скобку дважды. Первая квадратная скобка закрывает открывающую левую квадратную скобку (эти скобки будут подсвечены). Перед второй закрывающей квадратной скобкой будет вставлен пробел, а перед текстом — недостающая открывающая квадратная скобка. Курсор вставки будет находиться перед второй закрывающей квадратной скобкой, позволяя мне вставлять сюда текст `init:`.

```
UIAlertView* av = [[UIAlertView alloc] ]
// точка вставки находится здесь      ^
```

Если флажок Enable type-over completions установлен, среда Xcode делает еще больше. Когда я начинаю набирать ту же строку кода

```
UIAlertView* av = [U
```

среда Xcode автоматически добавляет закрывающую квадратную скобку, а точка вставки располагается перед ней:

```
UIAlertView* av = [U]
```

Однако эта закрывающая квадратная скобка является гипотетической и поэтому окрашивается серым цветом. Теперь я заканчиваю набирать первый вложенный вызов метода, и закрывающая квадратная скобка остается серой:

```
UIAlertView* av = [UIAlertView alloc]
// Я набрал строку до этого места  ^
```

Для того чтобы подтвердить, что закрывающая квадратная скобка вставлена правильно, существует несколько способов. Можно просто набрать закрывающую фигурную скобку или нажать клавишу <Tab> или <Right>. Гипотетическая закрывающая квадратная скобка станет реальной, а точка вставки переместится после нее, позволяя набирать код дальше.

Автоматическое дополнение

Когда вы пишете код, вы можете использовать возможность автоматического дополнения в среде Xcode. Язык Objective-C довольно многословный, и любой способ ускорить и облегчить набор текста следует приветствовать. Однако лично я не устанавливаю флажок Suggest completions while typing на вкладке Editing; вместо этого я устанавливаю флажок Use Escape key to show completion suggestions и кода хочу использовать возможность автоматического дополнения, включаю этот механизм вручную, нажимая клавишу <Esc>.

Для примера рассмотрим код, набранный выше, а точка вставки расположена перед второй закрывающей квадратной скобкой. Теперь я набираю слово `init`, а затем нажимаю клавишу <Esc>. На экране появляется маленькое всплывающее меню, в котором перечислены четыре

метода `init`, подходящие для класса `UIAlertView` (рис. 9.2). Вы можете перемещаться по этому меню, закрывать его и подтверждать выбор, используя только клавиатуру. Итак, если бы он не был выбран по умолчанию, я бы перешел к пункту `initWithTitle:...` с помощью клавиши <Down> и нажал клавишу <Return> для подтверждения выбора.

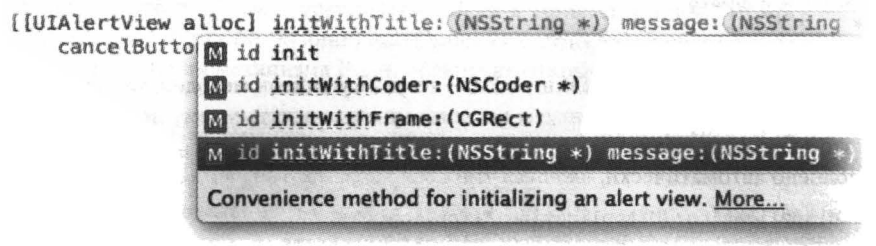


Рис. 9.2. Меню автоматического дополнения

В качестве альтернативы я мог бы нажать комбинацию клавиш <Control+Period> вместо <Esc>. Комбинация клавиш <Control+Period> позволяет переключать альтернативы. Нажмите клавишу <Return>, чтобы подтвердить сделанный выбор. Другая возможность — нажать клавишу <Tab>, выполняющую частичное дополнение, не закрывая меню автоматического дополнения. Если бы в ситуации, изображенной на рис. 9.2, в этот момент я нажал бы клавишу <Tab>, то в мой код было бы вставлено имя кода `initWith` — система подчеркивает его пунктирной линией, — а пункт `init`, который стал бы неприемлемым, был бы исключен из меню.

Заметим, что в меню автоматического дополнения существует сокращенная форма быстрой справки; щелкните на ссылке More, чтобы увидеть (в справочном окне) полную справку о выбранном методе (см. главу 8). Если выбранный метод определен в вашем коде и вы использовали формат doxygen для описания этого метода в комментарии, как описано в главе 8, то здесь появится описание `\brief`, если оно существует, или полное описание в противном случае. По этой причине целесообразно определить описание `\brief`, чтобы сохранить компактность окна для дополнения кода.

Если выбрать альтернативу в меню автоматического дополнения, то в мой код будет вставлен шаблон вызова метода (я разделил его на несколько строк):

```
[[UIAlertView alloc] initWithTitle:<#(NSString *)#>
                           message:<#(NSString *)#>
                          delegate:<#(id)#>
                 cancelButtonTitle:<#(NSString *)#>
                 otherButtonTitles:<#(NSString *), ...#>, nil]
```

Выражения в угловых скобках <#...#> являются полями для заполнения, в которых набирается каждый параметр. Вы можете выделить следующее поле для заполнения с помощью клавиши <Tab> (если точка вставки предшествует полю для заполнения) или выбрать команду Navigate⇨Jump to Next Placeholder (<Control+/>). Таким образом, я могу выбрать поле для заполнения и набрать в нем реальный аргумент, который хочу передать, выбрать следующее поле для заполнения, набрать в нем следующий аргумент и т.д.

Поля для заполнения разделены скобками, <#...#>, но в среде Xcode они отображаются как текстовые лексемы, чтобы предотвратить их непреднамеренное редактирование. Для того чтобы преобразовать поле для заполнения в обычную строку без этих разделителей, выберите его и нажмите клавишу <Return> или дважды щелкните на этом поле.

Механизм автоматического дополнения распространяется и на объявления методов. Вы не обязаны заранее знать или вводить тип значения, возвращаемого методом. Просто наберите начальные символы, - или + (чтобы отметить метод экземпляра или метод класса), за которыми следуют первые несколько букв имени метода. Например, в моем делегате приложения я мог бы набрать

```
- appli
```

Если после этого я нажму клавишу <Esc>, то увижу список методов, таких как `application:didChangeStatusBarFrame:`; которые можно послать моему делегату приложения (почему это возможно, объясняется в главе 11). Если я выберу один из методов, объявление будет заполнено автоматически, включая тип возвращаемого значения и имена параметров.

```
- (void)application:(UIApplication *)application
    didChangeStatusBarFrame:(CGRect)oldStatusBarFrame
```

В этот момент я готов набрать левую фигурную скобку, за которой следует оператор return; в результате на экране появится соответствующая закрывающая фигурная скобка, а точка вставки будет расположена между ними, позволяя ввести тело метода.

Сниппеты

Механизм автоматического дополнения кода сопровождается сниппетами кода. Сниппет кода — это фрагмент текста с аббревиатурой. Сниппеты кода хранятся в библиотеке Code Snippet (<Command+Option+Control+2>), но аббревиатура сниппета является глобально доступной, поэтому их можно использовать, не открывая библиотеку. Вы набираете аббревиатуру, и имя сниппета появляется среди всех возможных дополнений.

Например, для вставки блока `if` можно набрать слово `if` и нажать клавишу <Esc>, получить автоматическое дополнение и выбрать пункт If Statement. После нажатия клавиши <Return> в коде появится блок `if`, поле для условного выражения (между круглыми и скобками) и поле для инструкций (между фигурными скобками).

Для того чтобы выяснить аббревиатуры сниппетов, необходимо открыть окно редактирования, дважды щелкнув на сниппете в библиотеке Code Snippet и щелкнув на кнопке Edit. Если аббревиатура сниппета слишком сложная, просто перетащите ее из библиотеки Code Snippet в свой текст.

Программисты могут добавлять свои сниппеты, которые включаются в категорию User snippets; проще всего это сделать, перетащив текст в библиотеку Code Snippet. Отредактируйте его по своему вкусу, задайте имя, описание и аббревиатуру; для формирования полей для вставки используйте скобки `<#...#>`.

Механизм fix-it и прямая синтаксическая проверка

В среде Xcode есть чрезвычайно мощный механизм Fix-it, который может создавать и реализовывать предложения по предотвращению проблем. Для его активизации щелкните на значке, символизирующем проблему. Он находится на полях. Такой символ появляется после компиляции, если обнаружена ошибка.

Например, на рис. 9.3 я пропустил символ `@` перед литералом `NSString` языка Objective-C, и компилятор выдал сообщение об ошибке (потому что я набрал литерал языка C, а это совершенно разные вещи). Щелкнув на символе предупреждения на поле, я открыл небольшое диалоговое окно, в котором не только описывается ошибка, но и содержится предложение по

ее исправлению. Но это не все: в этом окне есть гипотетическое (выделенное серым цветом) решение проблемы; в данном случае предлагается вставить в код символ @. И это еще не все: если нажать клавишу <Return> или дважды щелкнуть на кнопке Fix-it в диалоговом окне, то среда Xcode действительно вставит пропущенный символ @ в мой код — и предупреждение исчезнет, потому что проблема будет решена. Если я согласен с этим, то могу выбрать команду Editor⇨Fix All in Scope (<Command+Option+Control+F>), и среда Xcode реализует все ближайшие предложения механизма Fix-it, больше не открывая диалоговое окно.

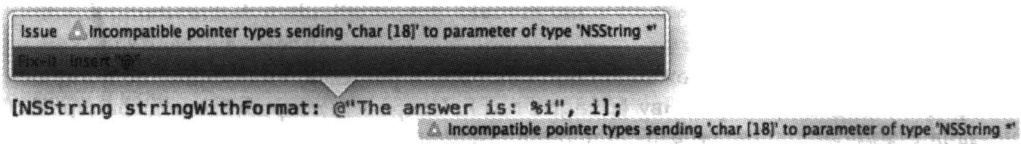


Рис. 9.3. Предупреждение с предложением механизма Fix-it

Прямая синтаксическая проверка — это форма постоянной компиляции. Даже если вы не компилировали и не сохраняли код, механизм прямой синтаксической проверки может выявить наличие проблемы и предложить решение с помощью механизма Fix-it. Эта функция может быть включена или выключена с помощью флажка Show live issues на панели настроек General. Лично я предпочитаю отключать этот механизм, поскольку он слишком навязчивый. Когда я набираю код, он почти никогда не бывает синтаксически правильным, потому что термины и скобки всегда набраны лишь наполовину; это неизбежно при наборе текста. Например, простой набор открывающей круглой скобки мгновенно активизирует проверку синтаксиса и сообщение об ошибке (которая исчезает только тогда, когда я набираю закрывающую круглую скобку).

Навигация по коду

Разработка проекта в среде Xcode подразумевает редактирование кода во многих файлах одновременно. К счастью, среда Xcode предусматривает множество способов для навигации по коду, многие из которых уже упоминались в предыдущих главах.

Перечислим некоторые из основных способов навигации в среде Xcode.

Навигатор проекта

Если вы знаете нечто об имени файла, то можете быстро его найти в окне навигатора проекта (<Command+1>), набрав его имя в поле поиска на панели фильтров в нижней части навигатора (Edit⇨Filter⇨Filter in Navigator, <Command+Option+J>). Например, наберите имя `story`, и вы увидите свои файлы `.storyboard`. Более того, использовав панель фильтров, вы можете нажать клавишу <Tab>, а затем клавишу <Up> или <Down> для перемещения в окне навигатора проекта. Таким образом вы можете открыть искомый файл, используя только клавиши.

Навигатор символов

Если выделить первые две пиктограммы на панели фильтров (первые две имеют синий цвет, а третья — темная), то навигатор символов перечислит классы вашего проекта и их методы. Теперь вы можете перейти в поисковый метод. Как и в навигаторе проекта, здесь есть панель фильтров, помогающая найти то, что вы ищете.

Панель быстрых переходов

Каждый компонент пути к коду образует пункт меню.

Нижний уровень

На нижнем уровне (крайнем правом) в панели быстрых переходов находится список объявлений и определений методов и функций в вашем файле. Они перечислены в порядке появления (для того чтобы упорядочить их в алфавитном порядке, нажмите и удерживайте клавишу <Command>). Для того чтобы перейти к одному из них, выберите его в этом меню.

В меню нижнего уровня можно вставить заголовки разделов, выделенные полужирным шрифтом, используя директиву `#pragma mark`. Например, попробуйте модифицировать файл `ViewController.m` в проекте Empty Window.

```
#pragma mark - View lifecycle
- (void)viewDidLoad
{
   [super viewDidLoad];
   // Дополнительная настройка после загрузки представления...
}
```

В результате пункт viewDidLoad в меню нижнего уровня будет включен в раздел View lifecycle.

Для того чтобы в меню появилась разделительная линия, наберите директиву `#pragma mark`, значением которой является знак переноса. В предыдущем примере использованы и знак переноса (чтобы создать разделительную линию), и заголовок (чтобы создать заголовок, набранный полужирным шрифтом). Аналогично в меню нижнего уровня появятся комментарии, расположенные за пределами методов и начинающиеся символами `TODO:`, `FIXME:`, `???:` или `!!!:`.

Более высокие уровни

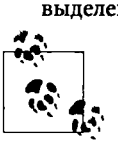
Компоненты пути высокого уровня представляют собой иерархические меню. Таким образом, любой из них можно использовать для перемещения вниз по иерархии файлов.

История

Каждая панель редактора запоминает имена файлов, которые в ней редактировались. Для просмотра этой истории надо нажать на треугольниках Back и Forward, которые являются кнопами и раскрывающимися списками (или выбрать команды Navigate⇨Go Back и Navigate⇨Go Forward, или нажать клавиши <Command+Control+Left> и <Command+ +Control+Right>.

Связанные элементы

Крайняя левая кнопка на панели быстрого перехода активизирует меню Related Items, иерархическое меню файлов, связанных с текущим файлом, например сопряженные файлы, суперклассы и включаемые файлы. В этот список входят даже методы, которые вызывают выделенный метод или вызываются им.



Меню компонентов пути на панели быстрых переходов можно фильтровать! Начните набирать строку при открытом меню панели быстрого перехода, чтобы фильтровать то, что отображается в этом меню. Эта фильтрация использует

"интеллектуальный" поиск, а не точный поиск текста; например, набрав строку "adf", мы найдем метод `application:didFinishLaunchingWithOptions:` (если он не отображается в меню).

Окно помощника

Помощник позволяет находиться одновременно в двух местах. Нажмите и удерживайте клавишу <Option> во время навигации, чтобы открыть содержимое файла в окне помощника, а не в основном окне редактора. Меню Tracking в панели быстрых переходов в окне помощника автоматически устанавливает его связь с основным окном (механизм слежения обсуждался в главе 6).

Вкладки и окна

Находиться одновременно в двух местах можно, открыв вкладку или отдельное окно (см. главу 6).

Быстрый переход к определению

Команда Navigate⇨Jump to Definition (<Command+Control+J>) позволяет быстро перейти к объявлению или определению символа, уже выделенного в коде.

Быстрое открытие

Команда File⇨Open Quickly (<Command+Shift+O>) выполняет поиск символа в коде или заголовочном файле Cocoa, открывая диалоговое окно.

Точки прерывания

Навигатор прерываний содержит все точки прерывания в вашем коде. В среде Xcode нет закладок, но существует возможность преднамеренно использовать точки прерывания как закладки. Точки прерывания обсуждаются в этой главе в последующих разделах.

Поиск

Поиск — это форма навигации. В среде Xcode 5 меню Find стало не подменю меню Edit, а меню верхнего уровня. Среда Xcode имеет средства как для глобального поиска (Find⇨Find in Project/Workspace, <Command+Shift+F>), который эквивалентен навигатору поиска, так и для поиска на уровне редактора (Find⇨Find, <Command+F>); не путайте их.

Возможности поиска очень важны. Если щелкнуть на пиктограмме с изображением увеличительного стекла в поле поиска, в раскрывающемся списке появятся дополнительные пункты. Команды глобального поиска включают область видимости (на рис. 9.3 область видимости определена пунктом In project), позволяя задавать конкретные способы поиска файла: щелкните на текущей области видимости, чтобы увидеть диалоговое окно и даже создать свою собственную область видимости. Существует также возможность поиска с помощью регулярных выражений. В этом механизме скрыта масса возможностей.

Для того чтобы заменить текст, щелкните на слове Find в левом конце панели поиска, чтобы активизировать всплывающее меню, и выберите команду Replace. Вы можете заменить все вхождения (Replace All) или выбрать конкретные результаты поиска в навигаторе поиска и заменить только их (Replace). В среде Xcode 5 есть новшество, позволяющее удалять результаты поиска из навигатора поиска, чтобы защитить их от действия команды Replace All. Команда Preview в навигаторе поиска открывает диалоговое окно, в котором демонстрируется возможный результат каждой замены и появляется возможность заранее

выбрать или отменить выбор конкретных замен. На уровне редактора необходимо нажать и удерживать клавишу <Option> до щелчка на кнопке Replace All, чтобы найти и заменить текст только в текущем выделении.

Существует более сложная форма поиска на уровне редактора — команда Editor⇨Edit All In Scope, которая выполняет одновременный поиск всех вхождений выделенного термина (обычно имени символа) в текущем наборе фигурных скобок. Этот механизм можно использовать для изменения имени переменной в ее области видимости или просто выяснить, где оно используется.

Для того чтобы изменить имя символа в коде, а также получить автоматическую интеллектуальную помощь при перестройках кода, которые часто возникают при программировании на языке Objective-C, используйте команду Refactoring среды Xcode (см. раздел "Внесение изменений в проекте" в справочнике Xcode).

Выполнение приложения в симуляторе

Для того чтобы собрать и выполнить приложение в симуляторе, создается целевое приложение типа IOS Simulator application. Окно симулятора имитирует устройство. В зависимости от настроек сборки целевого приложения Base SDK, Deployment Target и Targeted Device Family можно выбрать устройство и систему, которые будут имитироваться в симуляторе (см. главу 6).

Тип устройства можно переключить и в самом симуляторе, выбрав команду Hardware⇨Device, которая прекращает выполнение приложения в симуляторе. Вы можете заново запустить приложение, собрав и выполнив его в среде Xcode снова или щелкнув на пиктограмме приложения в окне симулятора. Во втором случае любая связь со средой Xcode разрывается, поскольку механизм отладки в этом случае не используется, вы не можете останавливать выполнение в точках прерывания, и сообщения об ошибках не выводятся на консоль Xcode. Кроме того, этот механизм можно использовать для быстрой проверки работы приложения на разных устройствах.

Окно симулятора может открываться наполовину, на треть или в полном размере (выберите команду Window⇨Scale). Эта возможность касается только дисплея и напоминает масштабирование окна, поэтому при изменении этого параметра работа симулятора не прекращается. Например, на симуляторе можно имитировать устройство с экраном Retina в полном размере, чтобы увидеть каждый пиксель при удвоенном разрешении экрана, или в половинном виде для экономии памяти.

С симулятором можно работать так, будто он представляет собой реальное устройство. Используя мышь, можно имитировать касание экрана устройства; нажмите и удерживайте клавишу <Option>, чтобы курсор мыши стал похожим на два пальца, симметрично перемещающихся вокруг их общего центра, иди клавиши <Option+Shift>, чтобы эти два пальца перемещались параллельно. В некоторых представлениях симулятора имеется кнопка Home, на которой можно щелкнуть мышью, но самым надежным способом является щелчок мышью на кнопке Home, чтобы выбрать команду Hardware⇨Home (<Command+Shift+H>). (Вследствие многозадачности щелчок на кнопке Home для переключения выполнения приложения со среды Xcode на главное окно не приводит к остановке выполнения приложения ни в среде Xcode, ни в симуляторе. Для того чтобы прекратить выполнение приложения в симуляторе, выйдите из него или переключитесь на среду Xcode или выберите команду Product⇨Stop.) Команды в меню Hardware позволяют также имитировать жесты на устройстве, например, вращение

или тряску устройства и блокировку экрана. Кроме того, можно протестировать приложение, имитируя редкие события, например, нехватку памяти.

Меню Debug в симуляторе помогает идентифицировать проблемы, связанные с анимацией и рисованием. Команды этого меню можно выбирать в ходе выполнения приложения в симуляторе, не прерывая его. Команда Toggle Slow Animations замедляет анимацию, чтобы можно было увидеть, что именно происходит. Четыре следующих команды (их имена начинаются со слова Color) похожи на функции, доступные при выполнении приложения на устройстве с помощью механизмов Instruments или Core Animation, выявляя возможные источники проблем при рисовании на экране.

Меню Debug также позволяет открывать журнал в приложении Console и задавать координаты имитируемого устройства (которые используются в приложении Core Location).

Отладка

Отладка — это искусство обнаружения ошибок во время выполнения приложений. Это искусство можно разделить на два основных метода: грубая отладка и остановка во время выполнения.

Грубая отладка

Грубая отладка (caveman debugging) предусматривает изменение кода, обычно временное, как правило, путем добавления кода для выдачи информативных сообщений на консоль. Консоль можно просматривать на панели Debug (в главе 6 описан метод вывода информации на консоль на отдельной вкладке).

Стандартной командой отсылки сообщения на консоль является команда `NSLog`. Это функция языка C, получающая аргумент типа `NSString`, представляющий собой строку, за которой следуют аргументы форматирования.

Строка форматирования — это строка (в данном случае объект класса `NSString`), содержащая символы, называемые спецификаторами формата, значения которых (аргументы формата) подставляются во время выполнения приложения. Все спецификаторы формата начинаются с символа (`%`), так что единственный способ ввести литерал процента — использовать двойной знак процента (`%%`). Символ(ы), следующий за знаком процента, задает тип значения, которое будет подставлено во время выполнения программы. Наиболее распространенными спецификаторами формата являются символы `%@` (ссылка на объект), `%d` (целое число) и `%f` (число типа `double`). Рассмотрим следующий пример:

```
NSLog(@"the window: %@", self.window);
```

В данном примере `self.window` — это первый (и единственный) аргумент формата, поэтому его значение заменит первый (и единственный) спецификатор формата `%@`, когда строка форматирования будет выводиться на консоль. Таким образом, вывод на консоль будет выглядеть примерно следующим образом (для ясности я его слегка отформатировал):

```
the window: <UIWindow: 0x8a68a60;
              frame = (0 0; 320 480);
              hidden = YES;
              gestureRecognizers = <NSArray: 0x8a69fd0>;
              layer = <UIWindowLayer: 0x8a697b0>>
```

Мы выводим на экран класс объекта, его адрес в памяти (важно убедиться, что эти два экземпляра фактически являются одним и тем же экземпляром), а также значения некоторых дополнительных параметров. Эта отличная сводка является результатом реализации метода

описания в классе UIWindow: этот метод вызывается, когда объект используется вместе со спецификатором формата %@. По этой причине вы, возможно, захотите реализовать описание в своих собственных классах, чтобы формировать краткое описание с помощью вызова функции NSLog.

Для того чтобы увидеть полный список спецификаторов формата, доступных для строки форматирования, прочитайте документ компании Apple String Format Specifiers (в справочнике String Programming Guide). Спецификаторы формата в большой степени основаны на стандартной функции printf из библиотеки языка C; ее описание можно найти на справочной странице sprintf в стандарте K&R B1.2 и спецификацию функции printf, разработанную IEEE. В справочном руководстве есть ссылки на эти страницы.

Основной ошибкой при работе с функцией NSLog (и любой строкой форматирования) является передача неправильного количества аргументов форматирования или значений аргумента неправильного типа, который отличается от типа спецификатора формата. Новички жалуются на то, что система регистрации ошибок часто объявляет определенное значение бессмысленным, хотя на самом деле бессмысленным является сам вызов функции NSLog; например, если спецификатор формата был равен %d, а значение соответствующего аргумента имело тип float. Другой распространенной ошибкой является работа с классом NSNumber, как будто он является типом содержащегося в нем числа; тип NSNumber — это не число, а объект (%@). Кроме того, могут возникнуть сложности при работе с целыми числами, имеющими и не имеющими знак. К счастью, компилятор помогает программисту, выдавая соответствующие предупреждения.

Структуры языка C не являются объектами, поэтому, для того чтобы увидеть значение структуры с помощью функции NSLog, вы должны как-то раскрыть эту структуру. Обычная структура каркаса Cocoa имеет для этой цели свои удобные функции. Например:

```
NSLog(@"%@", NSStringFromCGRect(self.window.frame)); // {{0, 0}, {320, 480}}
```

Пуристы насмехаются над грубой отладкой, но я ее использую очень часто: она удобная, информативная и простая. Иногда этот способ является единственным. В отличие от отладчика, функция NSLog работает при любой конфигурации сборки (Debug или Release) и независимо от того, где выполняется приложение (на симуляторе или на устройстве). Эта функция работает, когда невозможно сделать паузу (из-за проблем с потоками, например). Иногда она работает на другом устройстве, например на тестовом приборе, на котором может выполняться приложение. Тестовый прибор может не иметь консоли, поэтому выдать информацию бывает трудно, но все же это возможно: например, тестовый прибор может подсоединить устройство к компьютеру, и вы увидите журнал регистрации в окне Organizer среды Xcode или в окне инструмента iPhone Configuration Utility.

Не забудьте удалить или закомментировать вызовы функции NSLog перед поставкой своего приложения, потому что вы, наверное, не хотите, чтобы при выполнении программы на устройстве клиента она выводила на экран массу ненужной информации. Есть один полезный трюк (позаимствованный у Дженса Альфке (Jens Alfke)) — вызвать функцию MyLog вместо функции NSLog и определить функцию MyLog в своем заголовочном файле (и когда наступит момент для прекращения регистрации, изменить значение с 0 на 1):

```
#define MyLog if(0); else NSLog
```

Полезно знать, что переменная с именем _cmd содержит селектор для текущего метода. Таким образом, с помощью одной инструкции можно выяснить, где вы находитесь:

```
NSLog(@"Logging %@ in %@", NSStringFromSelector(_cmd), self);
```

(Аналогично в языке C инструкция `NSLog(@"%s", __FUNCTION__)` регистрирует имя функции.)

Другая разновидность вызова для регистрации ошибок — операторы контроля ошибок `assert`. Операторы контроля ошибок — это условия, которые означают, что ваше утверждение (`assert`) в данный момент является истинным, — а если это не так, то приложение испытает сбой. Операторы контроля ошибок — очень хороший способ подтвердить, что ситуация соответствует вашим ожиданиям, не только когда вы только что написали код, но в будущем.

Простейшая форма операторов контроля ошибок — это функция языка C (на самом деле это макрос), которой передается один аргумент — условие, которое может быть ложным (0) или истинным (любое другое значение). Если условие ложное, ваше приложение испытает сбой, когда поток управления достигнет соответствующей строки, и в журнал регистрации ошибок будет выведено подробное описание. Например, допустим, что мы утверждаем NO, и это условие ложное и ведет к сбою. Когда поток управления достигнет соответствующей строки, возникнет сбой и появится сообщение:

```
Assertion failed: (NO),
function -[AppDelegate application:didFinishLaunchingWithOptions:],
file /Users/mattleopard/Desktop/testing/testing/AppDelegate.m, line 20.
```

Эта информация позволяет нам найти причину ошибки: мы знаем условие оператора контроля ошибок, метод, в котором содержится оператор контроля ошибок, файл, содержащий этот метод и номер строки.

Операторы контроля ошибок верхнего уровня находятся в макросах `NSAssert` (используются в методах языка Objective-C) и `NSCAssert` (используются в функциях языка C). Это позволяет формировать собственные сообщения об ошибках, которые появляются на консоли в дополнение к обычным сообщениям операторов контроля ошибок. Сообщение об ошибках может представлять собой строку форматирования, содержащую спецификаторы формата и соответствующие значения, как в методе `NSLog`.

Некоторые разработчики полагают, что операторы контроля ошибок должны допускать выполнение оставшегося кода, а не прекращать работу приложения. Однако по умолчанию высокоуровневые макросы `NSAssert` и `NSCAssert` в сборке Release недоступны из-за настройки сборки Enable Foundation Assertions, которая имеет значение No для конфигурации Release в шаблонных проектах. Для того чтобы сохранить работоспособные операторы контроля ошибок в сборке Release, это значение в настройках целевого приложения следует изменить на `Yes`.

Отладчик среды Xcode

Собирая и выполняя приложение в среде Xcode, вы можете остановить его выполнение с помощью отладчика и использовать его возможности.

Отладчик в среде Xcode 5 называется LLDB. Полное техническое описание отладчика можно найти на веб-странице `http://lldb.llvm.org`.

Между выполнением приложение и его отладкой в среде Xcode существует большая разница; основное различие между ними состоит в том, активны ли точки прерывания или игнорируются. Активность точек прерывания имеет два уровня.

Глобально

Точки прерывания в совокупности могут быть либо активными, либо неактивными. Если точки прерывания неактивные, то выполнение программы не будет приостановлено ни на одной из точек прерывания.

Индивидуально

Отдельная точка прерывания может быть включена или отключена. Даже если точки прерывания в совокупности активны, выполнение программы не приостанавливается на точке прерывания, если она отключена. Отключение точки прерывания позволяет оставить точку, в которой впоследствии можно приостановить выполнение программы.

Точка прерывания игнорируется, если она отключена или если все точки прерывания в совокупности не активны.

Важно помнить, что при использовании отладчика приложение должно быть собрано в конфигурации сборки Debug (она используется по умолчанию для схемы действия Run). Отладчик почти бесполезен, если приложение собрано в конфигурации сборки Release, в том числе потому, что оптимизация, выполняемая компилятором, может разрушить связи между инструкциями выполняемого кода и строками исходного кода.

Для создания точки прерывания (рис. 9.4) выберите в редакторе точку, в которой вы хотите сделать паузу, и команду Debug⇨Breakpoints⇨Add Breakpoint at Current Line (<Command+\>). Комбинация клавиш, указанная в скобках, добавляет точку прерывания в текущую строку и удаляет ее соответственно. Точка прерывания символизируется стрелкой на поле. В качестве альтернативы можно просто щелкнуть на поле, чтобы добавить точку прерывания; для того чтобы удалить точку прерывания жестом, перетащите ее в пределы поля.

Рис. 9.4. Точка прерывания

Рис. 9.5. Отключенная точка прерывания

Если в вашем коде есть точки прерывания, то вам потребуется механизм для их просмотра и управления. Эту задачу выполняет навигатор точек прерывания. Здесь можно перемещаться по точкам прерывания, включать и отключать их, щелкая на стрелках в навигаторе, а также удалять точки прерывания.

Кроме того, существует возможность изменять поведение точки прерывания. Нажмите клавишу <Control>, щелкните мышью на символе точки прерывания, расположенном на поле или в навигаторе проекта, и выберите команду Edit Breakpoint или нажмите комбинацию команд <Command+Option+щелчок на точке прерывания>. Это очень мощный инструмент: точку прерывания можно активизировать при определенных условиях или после определенного количества прерываний. Кроме того, с точкой прерывания можно связать одно или несколько действий, например, выполнение команды отладчика, регистрацию ошибок, воспроизведение звука, озвучивание текста или выполнение сценария.

Точку прерывания можно настроить так, чтобы программа автоматически продолжала свою работу после выполнения определенного действия. Это может стать прекрасной альтернативой грубой отладке: вместо вставки вызовов функции `NSLog`, которые должны компилироваться вместе с кодом и позднее должны быть удалены из готового приложения, существует возможность вставить точку прерывания, которая регистрирует информацию и позволяет

программе продолжать свою работу. По определению такие точки прерывания работают только в режиме отладки проекта. Готовое приложение, работающее на реальном устройстве, не должно выводить на экран горы служебной информации.

В навигаторе точек прерывания можно создавать определенные виды точек прерывания, которые невозможно создать в редакторе исходного кода. Щелкните на кнопке Plus в нижней части навигатора и выберите команду из ее всплывающего меню. К наиболее важным относятся следующие типы точек прерывания.

Точки прерывания исключения

Точка прерывания исключения приостанавливает выполнение приложения в момент генерирования или перехвата исключения, независимо от того, вызывает ли это исключение сбой впоследствии. Я рекомендую создавать точки прерывания исключения для приостановки приложения в момент генерирования всех исключений, поскольку это дает наилучшее представление о стеке вызовов и значениях переменных в момент появления исключения (это лучше делать сейчас, чем в момент сбоя). Вы видите, в каком месте кода находитесь, и можете проверить значения переменных, что, возможно, позволит понять причины проблемы. Если вы создаете такую точку прерывания исключения, я предлагаю использовать контекстное меню и команду Move Breakpoint To⇨User, которая делает точку прерывания постоянной и глобальной для всех ваших проектов.

Единственная проблема, связанная с точкой прерывания исключения, состоит в том, что иногда код, написанный разработчиками компании Apple, генерирует и перехватывает исключения произвольно. Это не сбой и не значит ничего плохого, но в любом случае ваше приложение приостанавливает выполнение для отладки, что может быть досадным.

Символическая точка прерывания

Символическая точка прерывания приостанавливает выполнение приложения при вызове определенного метода или функции, независимо от того, какой объект осуществляет вызов и какому объекту отправляется сообщение. Такой метод можно указать с помощью символа метода экземпляра или символа метода класса (- или +), за которым следуют квадратные скобки, содержащие имя класса или метода (см. раздел документации A Useful Shorthand). Например, для того чтобы выяснить, откуда было послано сообщение `beginReceivingRemoteControlEvents` общему экземпляру в моем приложении, я задаю символическую точку прерывания следующим образом:

```
-[UIApplication beginReceivingRemoteControlEvents]
```

В среде Xcode 5 можно указать имя метода, набрав только имя, и оно будет преобразовано в отдельные точки прерывания во всех соответствующих методах класса; затем можно активизировать только ту точку, которую вы хотите. Эта возможность является полезной также для подтверждения, что вы правильно ввели имя метода. Таким образом, я могу создать символическую точку прерывания для метода `beginReceivingRemoteControlEvents`, а среда Xcode сама создаст выражение `-[UIApplication beginReceivingRemoteControlEvents]`.

Для изменения состояния всех точек прерывания щелкните на кнопке Breakpoints на панели, расположенной в верхней части панели Debug, или выберите команду Debug⇨Activate/Deactivate Breakpoints (<Command+Y>). Активное состояние всех точек прерывания в совокупности не влияет на включение или выключение отдельных точек прерывания; если точки прерывания были неактивными, они просто все игнорируются, и никаких прерываний

не возникает. Стрелки, символизирующие точки прерывания, окрашены голубым цветом, если точки прерывания являются активными, и серым, если нет.

Когда приложение выполняется с активными точками прерывания и обнаруживает включенную точку прерывания (или выполняются условия ее включения и т.д.), оно приостанавливает выполнение. В активном окне проекта редактор показывает файл, содержащий точку выполнения, которым обычно является файл, содержащий точку прерывания. Точка выполнения изображается серой стрелкой; это строка, которая должна быть выполнена (рис. 9.6). В зависимости от настроек команды Running⇨Pauses в окне настроек Behaviors на экране может открыться окно навигатора отладки и панель Debug.

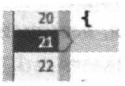


Рис. 9.6. Пауза в точке прерывания

Перечислим некоторые действия, которые можно выполнить во время паузы в точке прерывания.

Выяснить, где вы находитесь

Одна из наиболее распространенных причин установки точки прерывания — желание убедиться в том, что поток выполнения проходит через определенную строку. Вы можете выяснить, в каком из ваших методов вы находитесь, щелкнув на имени метода в стеке вызовов в навигаторе отладки.

Методы, перечисленные в стеке вызовов вместе с пиктограммой User и выделенные черным шрифтом, — это ваши методы; щелкните на одном из них, чтобы увидеть, в каком месте этого метода возникла пауза. Другие методы, текст которых окрашен в серый цвет, — это методы, для которых вы не написали код, поэтому щелкать на них нецелесообразно, если вы не знаете ассемблера. Ползунок на панели фильтра скрывает часть цепочки вызовов, чтобы сэкономить место, начиная с методов, для которых у вас нет кода.

Вы можете также просматривать и перемещаться по стеку вызовов, используя панель быстрых переходов, расположенную в верхней части панели Debug.

Изучение значений переменных

Это очень распространенная причина для приостановки. На панели Debug выводится список значений переменных в текущей области видимости (в соответствии с выделением в стеке вызовов). Щелкнув на треугольнике открытия, вы увидите дополнительные свойства объекта, например элементов коллекции, переменных экземпляра и даже часть закрытой информации. Здесь можно увидеть значения локальных переменных, даже если до приостановки выполнения они не были инициализированы; эти значения не имеют смысла и должны игнорироваться.

Переключите всплывающее меню под списком переменных в состояние Auto, чтобы просматривать только те переменные, которые по мнению среды Xcode должны вас интересовать (например, потому что их значения недавно изменились). Если вас интересует полная информация, лучше всего установить настройку Local. Для фильтрации имен и значений можно использовать поле поиска. Если форматированный краткий отчет недостаточно информативный, вы можете послать переменной объекта описание (или, если вы

реализовали его, описание `debugDescription`) и просмотреть вывод на консоль. Для этого выберите команду Print Description of [Variable] в контекстном меню или выберите переменную и щелкните на кнопке Info под списком переменных.

В среде Xcode 5 появился новый мощный инструмент для просмотра значения переменной в графическом представлении: выделите переменную и щелкните на кнопке Quick Look (пиктограмма в виде глаза) под списком переменных или нажмите клавишу <Spacebar>. Например, графическое представление структуры данных `CGRect` является масштабируемым прямоугольником.

Кроме того, в среде Xcode 5 всплывающие подсказки (data tips) сделаны более информативными. Для того чтобы увидеть всплывающую подсказку, наведите курсор мыши на имя переменной в вашем коде. Всплывающая подсказка напоминает маленький экран для вывода данного значения в списке переменных: в нем есть изогнутый треугольник, щелкнув на котором можно получить более полную информацию, кнопка Info, щелкнув на которой можно увидеть описание значения в этом окне и на консоли, а также кнопка Quick Look для графического представления значения (рис. 9.7).

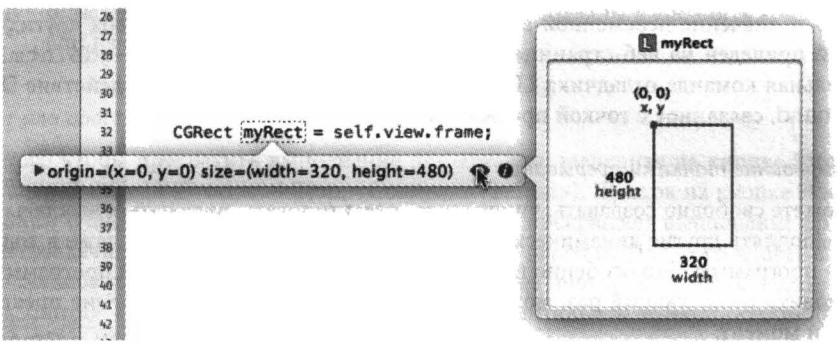


Рис. 9.7. Подсказка

Кроме того, существует возможность изменить значения переменных во время паузы для отладки. Выберите строку в списке переменных и нажмите клавишу <Return>, чтобы сделать ее редактируемой, или дважды щелкните на подсказке; так можно изменить простые скалярные значения (например, числа типов float и int, а также указатели). Можно также использовать команду expr на консоли, чтобы задать значение. Эту возможность следует использовать осторожно (или не использовать вообще).

Установка точки наблюдения

Точка наблюдения похожа на точку прерывания, но вместо привязки к определенной строке программы она зависит от значения переменной: как только значение этой переменой изменяется, отладчик приостанавливает выполнение приложения. Точку наблюдения можно установить только во время паузы в работе отладчика. Нажмите клавишу <Control>, щелкните на переменной в списке переменных и выберите команду Watch [Variable]. Созданные точки наблюдения перечисляются и управляются в окне навигатора точек прерывания. Поскольку для работы с точками наблюдения требуются дополнительные ресурсы, их количество не должно быть большим.

Управление выражениями

Выражение — это код, который можно добавить в список переменных и вычислять каждый раз во время паузы. Выберите команду Add Expression в контекстном меню в списке переменных. Выражение вычисляется в текущем контексте вашего кода, поэтому остерегайтесь побочных эффектов.

Общение с отладчиком

Используя консоль, можно общаться непосредственно с отладчиком. Интерфейс отладчика среды Xcode — это средство общения с программой LLDB; для того чтобы общаться непосредственно с отладчиком LLDB, можно делать все, что вы обычно делаете с интерфейсом отладчика в среде Xcode, и даже больше.

Распространенной командой является `po` (сокращение от "print object"— "вывести объект"), за которой следует имя переменной объекта или вызов метода, возвращающего объект. Эта команда вызывает метод описания объекта (или, если вы реализовали его, описание `debugDescription`). Другая важная команда — `expr`, которая вычисляет выражение на языке Objective-C в текущем контексте, т.е. помимо прочего, вы можете вызвать метод или изменить значение переменной в области видимости! Полезный список других возможностей приведен на веб-странице `http://lldb.llvm.org/lldb-gdb.html`. Любая консольная команда отладчика LLDB может также выполняться как действие Debugger Command, связанное с точкой прерывания.

Манипулирование точками прерывания

Вы можете свободно создавать, уничтожать, редактировать, включать и выключать, а также выполнять другие динамические действия с точками прерывания даже в ходе выполнения программы. Это особенно полезно, потому что место, в которой программу следует остановить в следующий раз, может зависеть от того, где ее выполнение прекращено в данный момент.

Действительно, это одно из главных преимуществ точек прерывания над грубой отладкой. Для того чтобы изменить грубую отладку, необходимо остановить выполнение приложения, отредактировать его, собрать заново и запустить снова. Однако для манипуляций с точками прерывания программу останавливать не обязательно! Операцию, которая выполняется неправильно, но не вызывает краха программы, можно повторить в реальном времени; вы можете просто добавить точку прерывания и попытаться снова. Например, если кнопка при нажатии работает неправильно, вы можете добавить точку прерывания и снова ее нажать; в этот раз будет выполнен тот же самый код, и вы сможете понять, в чем заключается проблема.

Пошаговая отладка или продолжение

Для того чтобы продолжить выполнение приостановленного приложения, вы можете либо возобновить выполнение, пока не встретится следующая точка прерывания (Debug⇨Continue), либо сделать один шаг и снова остановиться. Кроме того, вы можете выделить строку и выбрать команду Debug⇨Continue to Current Line (или Continue to Here из контекстного меню), которая устанавливает точку прерывания на указанной строке, продолжает выполнение и удаляет эту точку прерывания.

Перечислим команды пошаговой отладки в меню Debug.

Step Over

Пауза на следующей строке.

Step Into

Пауза в методе, который вызывается в текущей строке, если он есть; в противном случае пауза на следующей строке.

Step Out

Пауза после возвращения из текущего метода.

Вы можете выбрать эти команды с помощью удобных кнопок на панели в верхней части окна Debug. Даже если окно Debug свернуто, панель, содержащая эти кнопки, появляется во время выполнения программы.

Если команда Step Over не достигает следующей строки из-за точки прерывания в методе, вызванном в текущей строке, команда Continue автоматически остановит выполнение программы, — вы не обязаны выполнять команду Step Out из-за неожиданной точки прерывания.

Start over или abort

Для того чтобы прекратить выполнение приложения, щелкните на кнопке Stop инструментальной панели (Product⇨Stop, <Command+Period>). Щелчок на кнопке Home в окне Simulator (Hardware⇨Home) или на устройстве не прекращает выполнение приложения в многозадачной системе iOS 4 и в более поздних версиях. Для того чтобы остановить выполняемое приложение и запустить его заново без повторной сборки, нажмите клавишу <Control> и щелкните на кнопке Run инструментальной панели (Product⇨Perform Action⇨Run Without Building, <Command+Control+R>).

Вы можете вносить изменения в свой код во время выполнения или остановки приложения, но эти изменения не будут волшебным образом известны работающему приложению. Существуют языки и среды программирования, в которых это возможно, но среда Xcode и язык Objective-C к ним не относятся. Для того чтобы увидеть результат внесенных изменений, необходимо остановить приложение и запустить его в нормальном режиме (что подразумевает сборку).

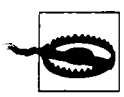
Модульное тестирование

Модульный тест — это код, не являющийся частью целевого приложения. Он предназначен для проверки работоспособности фрагмента целевого приложения. Например, модульный тест может вызвать какой-нибудь метод из целевого приложения, задать значения его параметров и проверить, всегда ли полученный результат соответствует ожидаемому, т.е. выполнить проверку не только в обычных, но и в неправильных или экстремальных условиях. Модульные тесты полезно разрабатывать до программирования реального кода, рассматривая эту процедуру как часть процесса разработки работоспособного алгоритма. Кроме того, зная, что ваш код уже прошел тестирование, вы можете время от времени повторять его, чтобы убедиться, что не внесли новых ошибок.

Модульные тесты не предназначены для проверки работоспособности целевого приложения в целом путем выполнения разнообразных сценариев нажатия кнопок виртуальным пользователем для тестирования интерфейса. Конечно, такую проверку выполнить можно. Например, можно обеспечить доступность вашего приложения, так что элементы пользовательского интерфейса в коде станут видимыми, а затем использовать программу Automation Instrument для выполнения сценариев JavaScript, имитирующих работу виртуального пользователя, но такая проверка не относится к модульному тестированию. Модульные тесты — это инструменты для проверки бизнес-логики, а не интерфейса.

В прошлом конфигурирование приложения для модульного тестирования было крайне сложной процедурой, но в среде Xcode 5 модульные тесты стали полноценными компонентами: шаблоны приложений генерируют проекты, имеющие не только целевое приложение, но и цель тестирования (test target), а удобное управление и выполнение тестов обеспечивает как навигатор тестов (<Command+5>), так и файл тестовых файлов.

Тестовый класс в среде Xcode 5 является подклассом класса `XCTestCase` (который в свою очередь является подклассом класса `XCTest`). Тестовый метод — это метод экземпляра тестового класса, не возвращающий никаких значений (`void`) и не имеющий параметров. Его имя начинается со слова `test`. Тестовая цель зависит от целевого приложения, т.е. до компиляции и сборки тестового класса целевое приложение уже должно быть скомпилировано и собрано. Выполнение теста подразумевает запуск приложения; продуктом тестовой цели является комплект, загружаемый в приложение при его запуске. Каждый тестовый метод будет вызван с одним или несколькими операторами контроля ошибок; в среде Xcode 5 их имена начинаются со слова `XCTAssert`. (Для того чтобы узнать о них больше, откройте файл `XCTestAssertions.h`.)



Операторы контроля ошибок, имена которых начинаются со слова XCTAssert, на самом деле являются макросами. Не усложняйте их синтаксис, чтобы не создавать проблем. Например, массив литералов, начинающихся с символа @, здесь не работает.

Кроме тестовых методов, тестовый класс может содержать вспомогательные методы, которые вызываются тестовыми методами. Кроме того. они могут содержать любой из четырех специальных методов, унаследованных от класса `XCTestCase`.

Метод класса `setUp`

Вызывается один раз перед всеми тестовыми методами, содержащимися в классе.

Метод класса `setUp`

Вызывается перед каждым тестовым методом.

Метод класса `tearDown`

Вызывается после каждого тестового метода.

Метод класса `tearDown`

Вызывается один раз после всех тестовых методов, содержащихся в классе.

Тестовая цель — это разновидность цели, ее продуктом является комплект (bundle), а фазы сборки напоминают фазы сборки целевого приложения. Это значит, что в комплект можно включать ресурсы, такие как тестовые данные. Для загрузки таких ресурсов можно использовать метод setUp; для ссылки на комплект тестирования из кода можно использовать следующее выражение.

```
[NSBundle bundleForClass:[MyTestClass class]]
```

В качестве примера используем проект Empty Window. Включим в класс ViewController (пустой) метод экземпляра dogMyCats:, например:

```
- (NSString*) dogMyCats: (NSString*) cats {
    return nil;
}
```

Предполагается, что метод dogMyCats: получает любую строку и возвращает строку @"dogs". Впрочем, сейчас он этого не делает, а вместо этого возвращает значение nil. Это ошибка. Напишем тестовый метод, который будет обнаруживать эту ошибку.

Проект Empty Window содержит один тестовый класс Empty_WindowTests. В отличие от обычного класса, класс Empty_WindowTests объявлен целиком в отдельном файле Empty_WindowTests.m — у него нет соответствующего заголовочного файла (.h), но он содержит разделы @interface и @implementation в .m-файле. (Как сказано в главе 4, это совершенно законно.)

Удалите из файла Empty_WindowTests.m сушествующий тестовый метод testExample. Мы заменим его тестовым методом, вызывающим метод dogMyCats: и применяющим оператор контроля ошибок к результату. Поскольку dogMyCats: — это метод экземпляра класса ViewController, нам понадобится экземпляр класса ViewController. Для того чтобы обращаться к экземпляру класса ViewController и вызывать его метод, необходимо импортировать заголовочный файл этого класса:

```
#import "ViewController.h"
```

Мы будем вызывать метод dogMyCats:, чтобы получить возможность пропустить его через компилятор, следовательно, нам нужна информация о методе dogMyCats:. Это значит, что метод dogMyCats: должен быть объявлен в импортированном файле ViewController.h.

Создадим переменную экземпляра в классе Empty_WindowTests для хранения в ней объекта класса ViewController. Объявим свойство в разделе @interface.

```
@interface Empty_WindowTests : XCTestCase
@property ViewController* viewController;
@end
```

Зададим значение этого свойства в методе setUp.

```
- (void)setUp {
   [super setUp];
   self.viewController = [ViewController new];
}
```

Теперь можно написать тестовый метод. Назовем его testDogMyCats. Он имеет доступ к экземпляру класса через ссылку self.viewController, потому что метод because setUp будет выполнен до тестового метода.

```
- (void)testDogMyCats {
   NSString* input = @"cats";
   XCTAssertEqualObjects([self.viewController dogMyCats:input], @"dogs",
      @"ViewController dogMyCats: fails to produce dogs from \"%@\"",
      input);
}
```

Теперь мы готовы выполнить наш тест. Это можно сделать разными способами. Навигатор тестов содержит нашу тестовую цель, тестовый класс и тестовый метод. Наведите курсор мыши на любое из этих имен, и справа от него появится кнопка. Щелкнув на соответствующей кнопке, вы можете выполнить все тесты в каждом классе, все тесты в классе `Empty_WindowTests` или просто тест `testDogMyCats`. Однако есть еще кое-что! Вернитесь в файл `Empty_WindowTests.m`, и вы увидите индикаторы в виде ромбиков на левом поле возле строк, содержащих директиву `@interface`, и имя тестового метода. Вы можете щелкнуть одном из них, и в результате будут выполнены все тесты в данном классе или отдельный тест соответственно. Кроме того, для того чтобы выполнить все тесты, можно выбрать команду Product⇨Test.

Проверьте, что во всплывающем меню схемы выбрана цель Simulator, и выполните метод `testDogMyCats`. Будут последовательно скомпилированы и собраны целевое приложение и целевой тест. (Если хотя бы один из перечисленных этапов не будет выполнен, тестирование невозможно, и вы столкнетесь с хорошо знакомыми ошибками компиляции или сборки.) После этого происходит запуск приложения в окне Simulator и теста.

Тест не выполняется! (Хорошо, мы знали, что это должно случиться, не так ли?) Выполнение приложения в окне Simulator прекращается. Эта ошибка описывается на баннере рядом с оператором контроля ошибок, на котором произошел сбой, а также в навигаторе проблем; возможно, проще всего прочитать о том, что произошло, в навигаторе журнала (в нем есть кнопка More, открывающая полное описание ошибки). Более того, везде появляется красный индикатор X — в навигаторе тестов рядом с методом `testDogMyCats`, в навигаторе проблем, в навигаторе журнала и в файле `Empty_WindowTests.m` рядом со строкой `@implementation` и первой строкой метода `testDogMyCats`. Большинство из этих индикаторов X являются кнопками! Щелкнув на одной из них, можно выполнить этот тест снова. (После выполнения отдельного теста можно также выбрать команду Product⇨Perform Action⇨Test [TestMethod] Again.)

Впрочем, все это можно сделать только после того, как мы исправим ошибку в коде. Измените в файле `ViewController.m` метод `dogMyCats`, чтобы он возвращал `@"dogs"`, а не `nil`. Теперь снова выполните тест. Тест пройден!

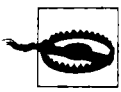
Когда возникает сбой теста, вы, возможно, захотите остановить выполнение приложения в той точке, где не выполняется оператор контроля ошибки. Для этого в навигаторе точек прерывания щелкните на кнопке Plus, расположенной внизу, и выберите команду Add Test Failure Breakpoint. Эта точка прерывания напоминает точку прерывания Exception, приостанавливая выполнение приложения на строке, содержащей оператор контроля ошибок в вашем методе, перед тем как выдать сообщение о сбое. После этого можно переключиться на тестируемый метод, например, и проверить его переменные или сделать что-то еще, чтобы устранить причину сбоя.

Существует полезная функциональная возможность, позволяющая перемещаться между методом и тестом, который его вызывает: если выделенная точка находится в методе, меню Related Files в панели быстрого перехода содержит пункт Test Callers. Это же относится к меню Tracking в окне помощника.

Код теста выполняется в комплекте, который по существу включается в выполняемое приложение. Это значит, что он видит глобальные сущности в приложении, например `[UIApplication sharedApplication]`. Таким образом, например, вместо создания нового экземпляра класса `ViewController` для инициализации ссылки `self.viewController` в классе `Empty_WindowTests` мы могли бы обращаться к уже существующему экземпляру класса `ViewController`.

```
self.viewController =
 ( ViewController*)[[[[UIApplication sharedApplication]
     delegate] window] rootViewController];
```

Организация тестовых методов в рамках целевых тестов (тестовых наборов) и тестовых классов по большей части является вопросом удобства: она оказывает влияние на макет навигатора тестов и позволяет выделить тесты, выполняемые совместно. Кроме того, каждый тестовый класс имеет свои собственные переменные экземпляра, свой собственный метод `setUp` и т.д. Для того чтобы создать новый целевой тест или новый тестовый класс, необходимо щелкнуть на кнопке Plus в нижней части окна навигатора тестов.



Когда вы переименовываете проект, содержащий целевой тест (команда Renaming Parts of a Project), этот целевой тест будет разрушен: некоторые из его настроек сборки по-прежнему ссылаются на старое имя приложения, поэтому целевой тест невозможно собрать и тесты невозможно выполнить. Если вам неудобно редактировать настройки целевого теста вручную, проще всего сделать копию тестового кода, удалить целевой тест, создать новый целевой тест (который уже будет иметь правильные настройки) и восстановить код теста.

Статический анализатор

Время от времени следует использовать статический анализатор для выявления возможных источников ошибок в вашем коде. Для этого следует выбрать команду Product⇨Analyze (<Command+Shift+B>), которая выполняет принудительную компиляцию кода, а статический анализатор глубоко исследует его, выдавая сообщения в окне навигатора и в самом коде. Как и при обычной компиляции, среда Xcode 5 может анализировать отдельный файл (выберите команду Product⇨Perform Action⇨Analyze [Filename]).

Статический анализатор анализирует код, а не выполняет его отладку в реальном времени, но это очень интеллектуальный и совершенный инструмент. Благодаря этому он может сообщить о потенциальных проблемах, которые могли бы ускользнуть от вашего внимания. Вам может показаться, что для этого достаточно компилятора. Действительно, некоторые из возможностей статического анализатора перекочевали в него из компилятора, который в среде Xcode 5 стал еще более интеллектуальным и полезным. Более того, одна из основных причин использования статического анализатора — желание облегчить ручное управление памятью в экземплярах языка Objective-C, — по большей мере обусловлено использованием механизма ARC. Однако не все вопросы управления памятью учитываются механизмом; например, он не выполняет управление памятью ARC с помощью ссылок `CFTypeRef` (глава 12), в то время как анализатор предупреждает о возможных ошибках. Статический анализатор затрачивает время на глубокое исследование возможных значений и путей выполнения кода и может выявлять возможные источники проблем, связанных с нарушением логики программы, которые компилятор игнорирует.

Например, недавно я выполнил статический анализ следующего кода:

```
-(void) newGameWithImage:(id)imageSource song:(NSString*)song {
   CGImageSourceRef src;
   if ([imageSource isKindOfClass:[NSURL class]])
      src = CGImageSourceCreateWithURL(
               (__bridge CFURLRef)imageSource, nil);
   if ([imageSource isKindOfClass:[NSData class]])
      src = CGImageSourceCreateWithData(
```

```
                (__bridge CFDataRef)imageSource, nil);
    // ...
    if (nil != src)
       CFRelease(src); }
```

Во-первых, анализатор сообщил, что переменная `src` имеет мусорное значение. Это показалось мне неправдоподобным; ведь переменная `src` должна была быть инициализирована одним из двух условий метода `isKindOfClass:`. Потом я понял, что анализатор не знал об этом, как и я. Я хотел, чтобы этот метод вызывался со значением `imageSource`, которое должно быть объектом класса `NSURL` или `NSData`; а что, если какой-нибудь идиот вызвал бы этот метод со значением другого типа? Тогда очевидно, что переменная `src` останется неинициализированной. По этой причине я инициализировал ее значением `nil`:

```
CGImageSourceRef src = nil;
```

Я запустил статический анализатор снова, но он все еще выдавал сообщения о том, что переменная `src` приводит к потенциальной утечке памяти по завершении метода. Как это возможно? Правила управления памятью с помощью ссылок `CFTypeRef` требуют, чтобы программист создал объект класса `CGImageSourceRef`, вызывающий метод `CGImageSourceCreateWithURL` или `CGImageSourceCreateWithData`, а затем вызвал метод `CFRelease` для удаления этого объекта. Я так и сделал.

Когда статический анализатор выдает предупреждение, можно щелкнуть на пиктограмме в самом начале предупреждения. В ответ анализатор нарисует диаграмму, изображающую пошаговую логику программы, и нарисует стрелки, демонстрирующие путь выполнения программы. Я сделал это и увидел, что анализатор считает, что оба условия метода `isKindOfClass:` могут выполняться одновременно (рис. 9.8). Если это так, то первое значение переменной `src` может исчезнуть: она будет заменена вторым значением без каких-либо операций по управлению памятью!

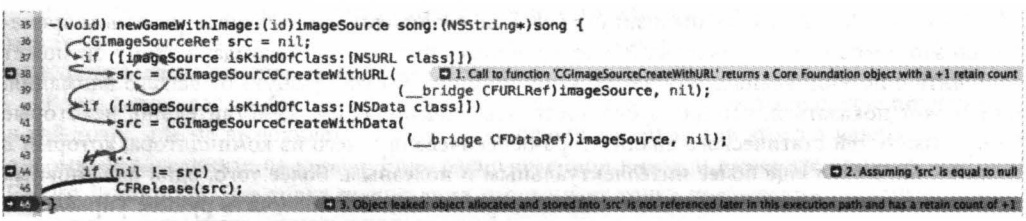


Рис. 9.8. Статический анализатор рисует диаграмму

Из принципов наследования классов следует, что оба условия метода `isKindOfClass:` не могут выполняться одновременно; объект `imageSource` должен относиться к классу `NSURL` или `NSData` (или ни к одному из них). Однако статический анализатор интересует логика и поток выполнения программы. Если я считаю, что может выполняться только одно из этих условий, я должен выразить этот факт логически и структурно. Я так и сделал, заменив второй оператор `if` на `else if`:

```
if ([imageSource isKindOfClass:[NSURL class]])
    src = CGImageSourceCreateWithURL(
        (__bridge CFURLRef)imageSource, nil);
else if ([imageSource isKindOfClass:[NSData class]])
    src = CGImageSourceCreateWithData(
        (__bridge CFDataRef)imageSource, nil);
```

В этом все и дело! Выполнив статический анализ еще раз, я не получил никаких сообщений об ошибках. Статический анализатор более глубоко исследовал логическую структуру моего когда, чем и помог мне уточнить логическую структуру, сделав ее более ясной и безопасной.

Для того чтобы статический анализатор автоматически запускался как часть обычной компиляции при сборке, можно использовать настройку сборки Analyze During 'Build'. Если установить ее значение равным `Yes`, то целесообразно установить параметр Mode of Analysis for 'Build' равным `Shallow`; полный анализ (`Deep`) затрачивает слишком много времени.

Более подробное описание статического анализатора можно найти на веб-странице `http://clang-analyzer.llvm.org`.

Чистка

Периодически во время повторяющихся сеансов тестирования и отладки и перед сборкой (при переключении с режима Debug на режим Release или при запуске приложения на устройстве, а не в симуляторе) целесообразно очистить цель. Это значит, что существующие сборки будут удалены и кеш будет очищен, так что весь код необходимо будет компилировать снова и собирать приложение заново.

Первая сборка приложения после чистки может выполняться дольше, чем обычно. Однако это того стоит, потому что при чистке происходит буквальное удаление мусора. Например, допустим, что вы включили в свое приложение какой-нибудь ресурс, а затем решили, что он вам больше не нужен. Вы можете удалить его из фазы сборки Copy Bundle Resources (или вообще из проекта), но это не значит, что он будет удален из собранного приложения. Такие забытые ресурсы могут вызывать разнообразные загадочные неприятности. Например, в интерфейсе может появиться неправильная версия nib-файла, а отредактированный вами код может вести себя иначе, чем до редактирования. Чистка удаляет собранное приложение и очень часто решает возникшие проблемы.

Чистка имеет несколько уровней.

Фоновая чистка

Выберите команду Product⇨Clean, которая удаляет собранное приложение и часть промежуточной информации из папки сборки.

Углубленная чистка

Нажав и удерживая клавишу <Option>, выберите команду Product⇨Clean Build Folder, которая удаляет всю папку сборки.

Полная чистка

Закройте проект. Откройте окно Organizer (с помощью команды Window⇨Organizer), найдите ваш проект в списке, расположенном в левой части окна Projects, и щелкните на нем. Щелкните на кнопке Delete, расположенной справа. В результате будет удалена вся папка проекта, вложенная в пользовательскую папку `Library/Developer/Xcode/DerivedData`. В следующий раз, когда проект будет открыт, его индекс будет собран заново. Для этого потребуется время, но некоторые проблемы можно устранить только так.

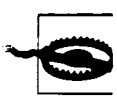
Тотальная чистка

Выйдите из среды Xcode. Откройте пользовательскую папку `Library/Developer/Xcode/DerivedData` и отправьте все ее содержимое в корзину. В этом случае произойдет полная чистка всего проекта, который вы недавно открывали (кроме того, будет удалена информация о модулях, если вы их использовали). При этом можно существенно сэкономить память.

Кроме чистки проекта, необходимо также удалить приложение из симулятора. Цель этого действия точно такая же, как у чистки проекта: при сборке и копировании приложения в симулятор ресурсы, существующие в приложении, могут остаться (для экономии времени), и приложение может повести себя странно. Для чистки симулятора в ходе его работы выберите команду iOS Simulator⇨Reset Content and Settings. В качестве альтернативы можно удалить приложение из симулятора, работая в окне Finder. Если симулятор работает, выйдите из него. Затем откройте пользовательскую папку `Library/Application Support/iPhone Simulator` и загляните в папку, название которой соответствует текущей версии SDK (например, эта может называться `7.0`); найдите в ней папку `Applications` и переместите ее содержимое в корзину.

Выполнение приложения на устройстве

Рано или поздно вы перейдете от тестирования, отладки и выполнения приложения с помощью симулятора на реальное устройство. Симулятор — это хорошо, но он всего лишь имитирует выполнение приложения; между ним и реальным устройством существует масса различий. На самом деле симулятор — это ваш компьютер, который работает быстро и имеет много памяти, так что проблемы, связанные с управлением памятью и скоростью выполнения приложения, остаются скрытыми, пока вы не перейдете на реальное устройство. Взаимодействие пользователя с симулятором ограничено работой с мышью: вы можете щелкать, перетаскивать объекты, удерживая клавишу <Option>, имитировать жесты двух пальцев, но более сложные жесты можно выполнить только на реальном устройстве. Многие функциональные возможности операционной системы iOS, например акселерометр и доступ к музыкальной библиотеке в симуляторе, вообще не представлены, поэтому тестирование приложения, которое их использует, возможно только на реальном устройстве.



Нечего и думать о разработке приложений без тестирования на реальном устройстве. Вы не имеете представления о том, как на самом деле будет выглядеть и работать ваше приложение, пока не выполните его на устройстве. Если вы пошлете в интернет-магазин App Store приложение, не проверенное на реальном устройстве, ждите проблем.

Перед выполнением своего приложения на устройстве, даже для простого тестирования, вы обязаны стать членом программы iOS Developer Program и платить годовой взнос. (Да, это приводит в ярость. Но пока возьмите себя в руки.) Только так можно получить и ввести в среде Xcode регистрационные данные для работы на устройстве. Зайдите на веб-сайт iOS Developer Program (`http://developer.apple.com/programs/ios`), заполните анкету и заплатите годовой взнос. Для начала достаточно программы Individual. Стоимость программы Company такая же, но она позволяет разработчикам получать разные привилегии, исполняя разные роли. (Для того чтобы распространять собранное приложение для тестирования другими пользователями, становиться членом программы Company не обязательно.)

Членство в программе iOS Developer Program подразумевает две вещи.

Идентификатор Apple ID

Идентификатор пользователя на сайте компании Apple (вместе с соответствующим паролем).

Ваш идентификатор Developer Program Apple ID можно использовать с разными целями. Кроме того, что он необходим для запуска вашего приложения на устройстве, этот же идентификатор открывает вам доступ к форумам разработчиков компании Apple, позволяет загружать бета-версии среды, смотреть видеофайлы конференции WWDC и многое другое.

Название команды

С одним и тем же идентификатором Apple ID вы можете принадлежать разным командам. В каждой их них вы будете играть роль, определенную вашими привилегиями.

Например, в команде Matt Neuburg, которая состоит из меня одного, я играю роль Агента, т.е. могу делать все что угодно: я могу разрабатывать приложения, выполнять их на своем устройстве, представлять приложения в интернет-магазине App Store и получать деньги за продажу копий приложения в этом магазине. В то же время я вхожу в команду TidBITS, для которой пишу приложение TidBITS News; в этой команде я просто Администратор, т.е. не могу представлять приложение в интернет-магазин App Store или просматривать детали приложения в службе iTunes Connect — в этой команде роль Агента играет кто-то другой.

Получив свой идентификатор Developer Program Apple ID, вы должны ввести его в окне настроек Accounts в среде Xcode. (Это новшество в среде Xcode 5.) Щелкните на кнопке Plus, расположенной в левом нижнем углу, и выберите команду Add Apple ID. Введите идентификатор Apple ID и пароль. С этого момента среда Xcode будет идентифицировать вас по имени команды, связанному с этим идентификатором; вводить пароль заново в среде Xcode не обязательно.

Для того чтобы выполнить приложение на устройстве, во время сборки его необходимо подписать. Неподписанное приложение не будет выполнено на устройстве (если вы его не взломаете). Для того чтобы подписать приложение, необходимы две вещи.

Идентификация

Идентификация — это разрешение компании Apple, выданное команде для разработки на конкретном компьютере приложений, которые могут выполняться на устройстве. Идентификация состоит из двух частей

Закрытый ключ

Закрытый ключ хранится на вашем компьютере в связке ключей. Таким образом он идентифицирует компьютер, на котором данная команда имеет право разрабатывать приложения для устройств.

Сертификат

Сертификат — это виртуальное разрешение от компании Apple. Он состоит из открытого ключа, соответствующего закрытому ключу (потому что вы сообщаете компании Apple свой открытый ключ, запрашивая сертификат). Ключ этого сертификата необходим для того, чтобы любой компьютер, на которой записан закрытый ключ, мог быть действительно использован для разработки приложений, предназначенных для устройств от имени команды.

Профиль обеспечения

Профиль обеспечения (provisioning profile) — это еще одно виртуальное разрешение от компании Apple. Он объединяет следующие четыре сущности.

- Идентификация.
- Приложение, идентифицированное по идентификатору его комплекта.
- Список допустимых устройств.
- Список вознаграждений. Вознаграждение — это специальные привилегии, которые имеют не все приложения, например доступ к службе iCloud. Вам не следует думать о вознаграждениях, пока вы не напишете программу, для которой необходимы такие привилегии.

Таким образом, профиля обеспечения достаточно для подписания приложения во время его сборки. Он сообщает, что на данном компьютере Mac разрешается собирать данное приложение, которое будет выполняться на указанных устройствах.

Существуют два типа идентификации и, соответственно, два типа сертификатов и два типа профилей: обеспечения разработки и обеспечения распространения (сертификат распространения также называется сертификатом продукции). В этой главе нас интересуют идентификация, сертификат и профиль обеспечения разработки; о распространении речь пойдет далее.

Единственным владельцем всей информации является компания Apple: ваши сертификаты, профили обеспечения, зарегистрированные приложения и зарегистрированные устройства. Для того чтобы проверить или получить копию этой информации, следует обращаться в компанию Apple. Для этого существуют два инструмента.

The Member Center

Member Center представляет собой набор веб-страниц. Для того чтобы зарегистрироваться в этом центре, необходимо быть членом программы Developer Program. Зайдите на веб-страницу Member Center (`https://developer.apple.com/membercenter`) или на веб-страницу iOS Dev Center (`https://developer.apple.com/devcenter/ios/`) и щелкните на ссылке Certificates, Identifiers, & Profiles. Вы получите доступ ко всем функциональным возможностям и информации, которая доступна вам в соответствии с типом вашего членства и вашей ролью. Ранее эта часть веб-сайта компании Apple называлась Portal; в апреле 2013 года ее интерфейс был существенно модернизирован.

Xcode

Кроме профиля обеспечения распространения, все, что можно получить в центре Member Center, можно получить и с помощью среды Xcode. Когда все хорошо, использование среды Xcode намного упрощает работу! Если же возникла проблема, следует обращаться в центр Member Center.

Получение сертификата

Настройка идентификации и получение сертификата выполняются только один раз (или не чаще одного раза в год; эти процедуры, возможно, придется повторить при возобновлении членства в программе Developer Program). Напомним, что сертификат зависит от пары закрытый ключ–открытый ключ. Закрытый ключ хранится в связке ключей, а открытый ключ передается компании Apple для включения в сертификат. Открытый передается компании

Apple в запросе на сертификат. Таким образом, процедура получения сертификата сводится к следующему.

1. Сгенерируйте пару закрытый ключ–открытый ключ на вашем компьютере с помощью программы Keychain Access. Закрытый ключ будет сохраняться в связке ключей на вашем компьютере.
2. Включите открытый ключ в запрос на сертификат и пошлите запрос в центр Member Center, идентифицируя себя и (при необходимости) свою команду с помощью идентификатора Apple ID, и укажите, какой сертификат вам нужен — для разработки или для распространения.
3. Компания Apple создаст сертификат, который также содержит открытый ключ.
4. Этот сертификат загружается и импортируется в связку ключей, которая использует открытый ключ для поиска соответствующего закрытого ключа. Сертификат также хранится в связке ключей.
5. С этого момента компания Xcode может видеть сертификат в связке ключей и осуществлять идентификацию для разработки или распространения под соответствующим названием команды.

Все это выглядит довольно сложно, потому что так оно и есть. Однако в среде Xcode 5 все эти этапы при отсылке запроса на сертификат выполняются автоматически! Для этого следует сделать следующее.

1. Откройте окно настроек Accounts в среде.
2. Если вы еще не ввели свой идентификатор разработчика Apple ID, сделайте это сейчас.
3. Слева выберите свой Apple ID. Справа выберите свою команду. Щелкните на кнопке View Details.
4. Если у вас уже был сертификат и он был аннулирован порталом, но его срок действия не истек, вы можете увидеть диалоговое окно, предлагающее ввести запрос и загрузить сертификат. Щелкните на кнопке Request.

   В противном случае щелкните на кнопке Plus и выберите команду iOS Development (под заголовком столбца Signing Identities).

После этого все происходит автоматически: генерируется пара закрытый ключ–открытый ключ, и сертификат запрашивается, генерируется, загружается, сохраняется в связке ключей и перечисляется под заголовком Signing Identities в диалоговом окне View Details.

(Более того, разрешение на получение профиля обеспечения командной разработки также можно сгенерировать, как показано на рис. 9.10. Таким образом, вы можете получить все, что требуется для выполнения приложения на устройстве!)

Если все работает правильно, можете пропустить оставшуюся часть главы. Если же нет, то сейчас мы опишем более сложную процедуру генерирования пары закрытый ключ–открытый ключ и запроса на сертификат. Инструкции, позволяющие начать этот процесс, можно также получить в центре Member Center (зайдите на веб-страницу Certificates и щелкните на кнопке Plus в правом верхнем углу).

Запустите программу Keychain Access и выберите команду Keychain Access⇨Certificate Assistant⇨Request a Certificate from a Certificate Authority. Используя свое имя и адрес электронной почты в качестве идентификаторов, сгенерируйте и сохраните на диске 2048-битовый

файл запроса на сертификат RSA. С этого момента ваш закрытый ключ будет храниться в вашей связке ключей; запрос на сертификат, содержащий ваш открытый ключ, будет временно храниться на вашем компьютере. (Например, вы можете сохранить его на рабочем столе.)

В центре Member Center вам предоставят интерфейс, позволяющий отправить сохраненный файл запроса на сертификат. Вы отправляете его, и сертификат генерируется; щелкните на листинге центра Member Center, чтобы увидеть кнопку Download, и щелкните на ней. Найдите файл, который вы только что загрузили, и дважды щелкните на нем; программа Keychain Access автоматически импортирует сертификат и сохранит его в связке ключей. Вам не обязательно хранить файл запроса на сертификат или загруженный файл сертификата; ваша связка ключей теперь содержит все необходимые регистрационные данные. Если все работает, вы можете увидеть сертификат в своей связке ключей, прочитать его детали, выяснить, что он действует и связан с вашим закрытым ключом (рис. 9.9). Более того, вы будете иметь возможность убедиться, что среда Xcode теперь знает об этом сертификате: в окне настроек Accounts щелкните на кнопке Apple ID, расположенной слева, и на названии вашей команды, расположенном справа, а затем щелкните на кнопке View Details. Откроется диалоговое окно, в верхней части которого вы увидите идентификатор подписи iOS Development со статусом `Valid`.

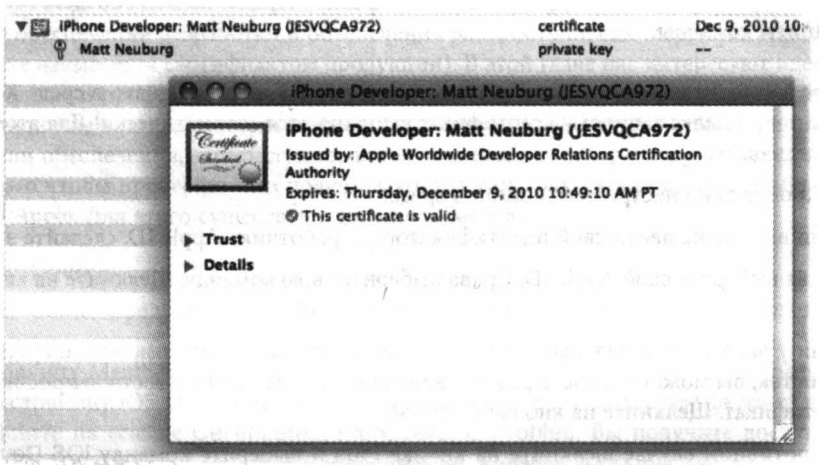


Рис. 9.9. Действительный сертификат разработки в окне Keychain Access

Если вы впервые получаете сертификат от центра Member Center, то вам потребуется другой сертификат — WWDR Intermediate Certificate. Это сертификат, который подтверждает, что сертификаты, выпущенные службой WWDR (Apple Worldwide Developer Relations Certification Authority), заслуживают доверия. (Здесь ничего нельзя выдумывать.) Среда Xcode должна автоматически инсталлировать его в вашей связке ключей; если же этого не произойдет, вы можете получить копию этого сертификата вручную, щелкнув на ссылке в нижней части веб-страницы Member Center, с которой начинался процесс добавления сертификата.

Получение профиля обеспечения разработки

Как уже говорилось, профиль обеспечения объединяет идентификацию, устройство и идентификатор комплекта приложения. Если никаких проблем нет, то в простейшем случае в среде Xcode вы сможете получить профиль обепечения для разработки за один шаг. Если

приложение не требует специальных разрешений или возможностей, то одного профиля для командной разработки будет достаточно для всех ваших приложений, поэтому эту процедуру достаточно выполнить только один раз.

Если вы сделали то, что сказано в предыдущем разделе, то у вас уже есть идентификатор для разработки (development identity). Возможно, у вас есть также универсальный профиль обеспечения разработки! Если нет, то сделайте следующее.

1. Подсоедините устройство к вашему компьютеру.
2. Откройте окно Organizer в среде Xcode (Window⇨Organizer).
3. Откройте вкладку Devices.
4. Выберите присоединенное устройство.
5. Щелкните на кнопке Add to Member Center или Use for Development в нижней части окна.

Устройство будет зарегистрировано в компании Apple и получит уникальный идентификационный номер, так что в результате будет создан и загружен универсальный профиль обеспечения разработки.

Для того чтобы подтвердить, что данное устройство добавлено в Member Center, откройте свой браузер и щелкните на кнопке Devices.

Для того чтобы подтвердить, что у вас есть универсальный профиль обеспечения разработки, щелкните на кнопке View Details в окне настроек Accounts (для соответствующей команды). Здесь перечислены сертификаты и профили. Универсальный профиль для разработки помимо заголовка "iOS Team Provisioning Profile" имеет неспецифический идентификатор комплекта приложения, отмеченный звездочкой (рис. 9.10).

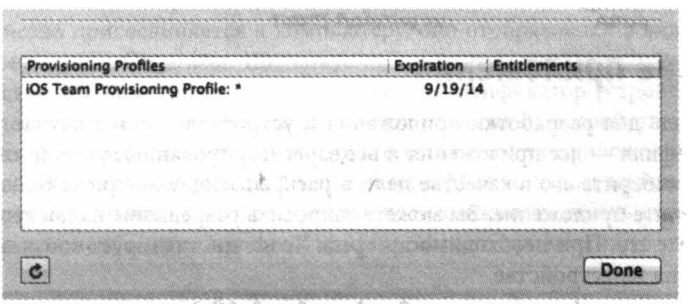


Рис. 9.10. Универсальный профиль для разработки

Универсальный профиль для разработки позволяет тестировать ваше приложение на целевом устройстве при условии, что ваше приложение не требует специальных прав (например, не использует службу iCloud).

С помощью среды Xcode можно подключить дополнительные устрйоства. Присоедините устройство к вашему компьютеру. Щелкнув на кнопке Add to Member Center или Use for Development, вы автоматически зарегистрируете устройство в центре Member Center и добавите его ко всем командным профилям для разработки. Эти профили будут автоматически сгенерированы и загружены в среду Xcode.

Иногда среда Xcode генерирует индивидуальные профили обеспечения разработки для каждого идентификатора приложения, закрепленного в центре Member Center за командой. Затем они перечисляются в окне настроек Accounts в диалоговом окне View Details,

соответствующем конкретной команде, под заглавием "iOS Team Provisioning Profile" и с конкретным идентификатором комплекта приложения. Для запуска на устройстве эти дополнительные командные профили обеспечения могут оказаться избыточными.

Профиль для разработки можно также получить от центра Member Center вручную.

1. Убедитесь, что ваше целевое устройство перечислено в разделе Devices в центре Member Center. Если нет, щелкните на кнопке Plus и введите имя данного устройства вместе с его идентификатором UDID. Вы можете скопировать идентификатор устройства UDID из этого списка на вкладку Devices в окне Organizer.

2. Убедитесь, что ваше приложение зарегистрировано в центре Member Cente, выполнив команду Identifiers⇨App IDs. Если нет, добавьте его. Щелкните на кнопке Plus и введите имя этого приложения. Не беспокойтесь о бессмысленных буквах и цифрах, которые Member Center добавит в качестве префикса к идентификатору вашего комплекта; используется идентификатор Team ID. Введите идентификатор комплекта в поле Explicit App ID, взяв его из поля Bundle Identifier в разделе General в среде Xcode, которое вы заполняли при редактировании целевого приложения. (К сожалению, если идентификатор комплекта сгенерирован автоматически, вы не сможете скопировать его из поля Bundle Identifier.)

3. Щелкните на кнопке Plus, выбрав пункт Provisioning Profiles. Запросите профиль iOS App Development. На следующем экране выберите идентификатор App ID. На следующем экране выберите ваш сертификат разработки. На следующем экране выберите устройство (или устройства), на котором должно выполняться приложение. На следующем экране введите имя профиля и щелкните на кнопке Generate. Щелкните на кнопке Download. Найдите загруженный профиль и дважды щелкните на нем, чтобы открыть его в среде Xcode. Теперь можете удалить загруженный профиль, поскольку среда Xcode сделала его копию.

Выполнение приложения

Имея профиль для разработки приложения и устройство (или в случае универсального командного профиля — все приложения и все зарегистрированные устройства), подсоедините устройство, выберите его в качестве цели в раскрывающемся списке Scheme, а затем соберите и выполните приложение. Вы можете запросить разрешение на доступ к вашей связке ключей; получите его. При необходимости среда Xcode инсталлирует соответствующий профиль обеспечения на устройстве.

Допустим, приложение собрано, загружено на устройство и выполняется на нем. Когда вы запускаете приложение из среды Xcode, все происходит так же, как и в симуляторе: вы можете выполнять или отлаживать приложение, и при этом выполняемое приложение сохраняет связь со средой Xcode, так что можно останавливать выполнение в точках прерывания, читать сообщения на консоли и т.д. Внешнее отличие заключается в том, что для физического взаимодействия с приложением вы используете устройство (физически подсоединенное к вашему компьютеру), а не симулятор.

Выполнение приложения из среды Xcode на устройстве можно использовать как простой способ копирования текущей версии приложения на устройство. Вы можете остановить работу приложения (в среде Xcode), отсоединить устройство от компьютера, запустить приложение на устройстве и работать с ним. Это хороший способ тестирования. Но это не отладка, поэтому вы не имеете обратной связи со средой Xcode, хотя функция `NSLog` автоматически записывает сообщения на консоль, и позднее их можно прочитать.

Управление профилем и устройством

В среде Xcode 5 центральным местом для обзора идентификаторов и профилей обеспечения является окно настроек Accounts. Выберите Apple ID и команду, а затем команду View Details.

Важным свойством окна настроек Accounts является возможность экспортировать информацию об учетной записи. Эта возможность нужна, если вы хотите разрабатывать приложения на другом компьютере. Выберите Apple ID и используйте меню Gear в нижней части экрана, чтобы выбрать команду Export Accounts. Вы должны ввести имя файла, указать место его хранения, а также пароль; этот пароль ассоциируется исключительно с этим файлом и нужен только для открытия этого файла на другом компьютере. Перейдя на другой компьютер, на который вы скопировали экспортированный файл, запустите программу Xcode и дважды щелкните на экспортированном файле; среда Xcode спросит у вас пароль. Когда вы введете его, как по волшебству будет активизирован весь набор команд, идентификаторов, сертификатов и профилей обеспечения для второй копии среды Xcode, включая записи в вашей связке ключей.

В качестве альтернативы вам, возможно, потребуется просто экспортировать идентификатор без профилей обеспечения. Это можно сделать с помощью меню, на пиктограмме которого изображена шестеренка, в диалоговом окне View Details в окне настроек Accounts.

Если профили обеспечения, перечисленные в диалоговом окне View Details в окне настроек Accounts, потеряют синхронизацию с центром Member Center, щелкните на кнопке Refresh в левом нижнем углу. Если это не поможет, выйдите из среды Xcode и в окне Finder откройте папку `Library/MobileDevice/Provisioning Profiles`, а затем удалите оттуда все. Заново запустите программу Xcode. В окне Accounts ваши профили обеспечения исчезли! Теперь щелкните на кнопке Refresh. Среда Xcode загрузит свежие копии всех ваших профилей обеспечения, и синхронизация с центром Member Center будет восстановлена.

Когда устройство присоединяется к компьютеру, оно отображается в виде зеленой точки в разделе Devices в окне Organizer. Щелкните на его имени, чтобы получить информацию об этом устройстве. Вы можете увидеть уникальный идентификатор устройства. Кроме того, вы можете увидеть профили обеспечения, инсталлированные на устройстве, а также журнал консоли устройства в реальном времени, так, будто вы запустили приложение Console, чтобы увидеть журналы регистрации на вашем компьютере. Можно также просмотреть отчеты о сбоях, возникших на устройстве. Существует также возможность делать снимки экранов вашего устройства (это необходимо для представления вашего приложения в интернет-магазин App Store).

Индикаторы и инструменты

В среде Xcode есть инструменты для графического и числового описания внутреннего поведения приложения, за которыми необходимо следить. В навигаторе отладки среды Xcode 5 впервые появились индикаторы (gauges), позволяющие отслеживать загрузку центрального процессора и степень использования памяти в любой момент работы приложения. Кроме того, сложное и мощное служебное приложение Instruments собирает профильные данные и помогает отслеживать проблемы, связанные с управлением памятью, а также собирает числовую информацию, необходимую для улучшения производительности приложения и его динамичности. Возможно, на этапе завершения своего приложения вы захотите использовать приложение Instruments для повышения его эффективности (преждевременная оптимизация — это лишь пустая трата времени и сил).

Индикаторы в навигаторе отладки работают во время сборки и выполнения вашего приложения. За ними необходимо следить, чтобы знать степень загрузки центрального процессора и использования памяти. Щелкните на индикаторе, чтобы увидеть подробную информацию в окне редактора. Индикатор не предоставляет полной информации, но чрезвычайно динамичен и всегда активен, поэтому с его помощью в любой момент можно легко понять, что происходит с вашим приложением. В частности, если возникла проблема, например, неожиданно длинный период высокой нагрузки центрального процессора или завышенное потребление памяти, ее можно выявить с помощью индикатора и отследить, используя утилиту Instruments.

На рис. 9.11 я продемонстрировал несколько моментов работы моего приложения, повторно выполняя наиболее типичные действия, ожидаемые от пользователя. Столбцы индикатора Memory в навигаторе отладки выглядят совершенно одинаковыми, что является хорошим признаком, но просто для подстраховки я щелкнул на нем, чтобы открыть более крупную диаграмму использования памяти в окне редактора. Некоторые действия, выполняемые впервые, вызывают всплески и падения нагрузки на память, но при повторении нагрузка на память выравнивается, поэтому нет никаких признаков проблемы.

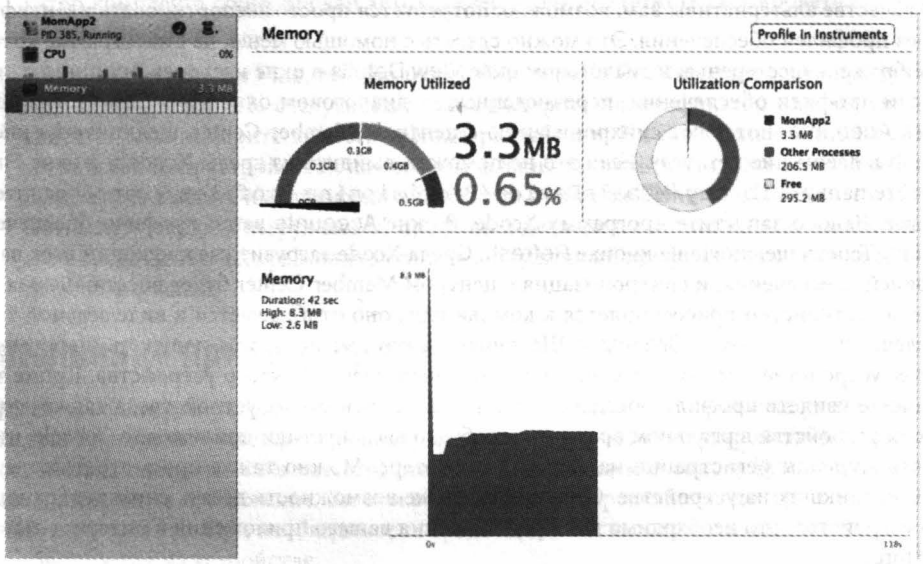


Рис. 9.11. Индикаторы навигатора отладки

Утилиту Instruments можно использовать как в симуляторе, так и на устройстве. Именно на устройстве будет выполняться окончательное тестирование, причем некоторые инструменты (такие как Core Animation) доступны только на устройстве.

Для того чтобы начать работу с утилитой Instruments, укажите целевое устройство в раскрывающемся списке Scheme на инструментальной панели в окне проекта и выберите команду Product⇨Profile. Допустим, что в схеме вашего приложения указано действие Profile; по умолчанию это значит, что ваше приложение будет использовать конфигурацию настроек Release. Если приложение выполняется на устройстве, вы можете увидеть некоторые сообщения о проверке, но их можно спокойно игнорировать. Утилита Instruments начинает работу; если в раскрывающемся списке Instrument для действия Profile установлено свойство `Ask on Launch` (по умолчанию), утилита Instruments откроет диалоговое окно, в котором можно выбрать шаблон слежения.

В качестве альтернативы можно щелкнуть на кнопке Profile In Instruments в редакторе индикатора в навигаторе отладки; в этом случае будет запущена утилита Instruments и автоматически выбран соответствующий шаблон слежения. Диалоговое окно предлагает две возможности: команда Restart останавливает выполнение вашего приложения и запускает его заново с помощью утилиты Instruments, а команда Transfer не препятствует выполнению приложения, подключая к нему утилиту Instruments. Обычно программисты пользуются командой Restart: это значит, что вы заметили проблему на индикаторах и хотите воспроизвести ее более подробно под наблюдением утилиты Instruments.

Когда утилита Instruments работает, вы должны взаимодействовать с приложением как пользователь; утилита Instruments будет записывать статистические показатели. После запуска утилиты Instruments можно уточнить профиль данных, которые вас особенно интересуют, и сохранить окно Instruments как пользовательский шаблон.

Использование утилиты Instruments требует знаний, которые выходят за рамки рассмотрения данной книги. Этой утилите можно (и нужно) посвятить отдельную книгу. За подробной информацией обращайтесь к документации компании Apple, особенно к справочнику Instruments User Reference and Instruments User Guide. Кроме того, этой утилите посвящено много видеозаписей конференции WWDC; обратите внимание на сессии, в именах которых используется слово "Instruments" или "Performance". Здесь я просто демонстрирую работу этой утилиты без пространных объяснений.

Начнем с построения диаграммы для степени использования памяти при выполнении приложения TidBITS News при его запуске и во время работы пользователя. Память мобильного устройства — это очень скудный ресурс, поэтому важно не слишком сильно ее захватывать. Я установил в качестве цели симулятор и выбрал команду Product⇨Profile. После запуска утилиты Instruments я выбрал шаблон слежения Allocations и щелкнул на кнопке Profile. В симуляторе началось выполнение моего приложения, я какое-то время с ним поработал, а затем остановил утилиту Instruments, которая тем временем построила диаграмму загрузки памяти (рис. 9.12). Изучая эту диаграмму, я обнаружил несколько пиков: первый — при загрузке приложения, второй — в момент, когда приложение загружало и анализировало RSS-канал; но максимум достиг всего лишь 5,40 Мбайт, после чего приложение практически все время использовало примерно 2 Мбайт. Эти числа свидетельствуют об очень умеренном потреблении памяти, а также о том, что после запуска приложение работает стабильно, чему я очень рад.

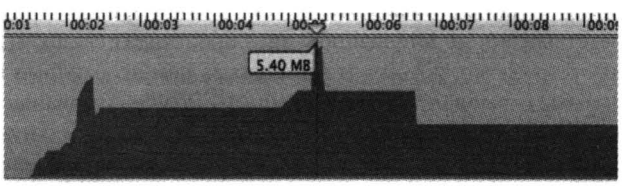


Рис. 9.12. Диаграмма загрузки памяти во времени, построенная утилитой Instruments

Другое направление анализа с помощью утилиты Instruments — выявление утечки памяти. Как сказано в главе 12, утечка памяти возможна даже при использовании механизма ARC. В тривиальном примере, рассмотренном ниже, используются два класса: `MyClass1` и `MyClass2`; класс `MyClass1` имеет переменную экземпляра, которая относится к классу `MyClass2`, а класс `MyClass2` имеет переменную экземпляра, которая относится к классу `MyClass1`. Рассмотрим следующий код:

```
MyClass1* m1 = [MyClass1 new];
MyClass2* m2 = [MyClass2 new];
m1.ivar = m2;
m2.ivar = m1;
```

Как указано в главе 12, можно предпринять некоторые меры, препятствующие утечке памяти; но я пока не делал этого, чтобы утечка памяти все же произошла. Установим целью приложения симулятор и выберем команду Product⇨Profile; происходит запуск приложения Instruments. Затем выберем шаблон слежения Leaks и щелкнем на кнопке Profile. Приложение начинает выполняться в симуляторе, и через 10 секунд (именно такой интервал установлен по умолчанию для анализа утечек к утилите Instruments) обнаруживается утечка памяти. Щелкнув на нескольких кнопках, можно вывести на экран диаграмму ошибки, вызывающую утечку памяти (рис. 9.13)!

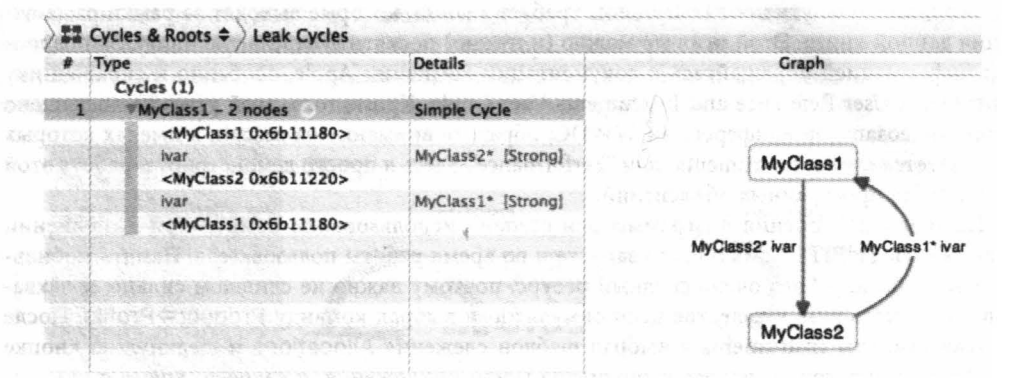


Рис. 9.13. Описание утечки памяти утилитой Instruments

Если анализ использования памяти осуществляется с помощью утилиты Instruments, то, возможно, целесообразно изменить схему, чтобы действие Profile использовало конфигурацию настроек Debug. По умолчанию она использует конфигурацию Release, которая оптимизирует ваш код и запутывает анализ использования памяти.

В последнем примере я пытался выяснить, почему в моем приложении Albumen переключение с главного представления на детализированное происходит слишком медленно. В качестве цели я установил устройство, потому что именно на нем необходимо измерять скорость выполнения программы, и выбрал команду Product⇨Profile. Произошел запуск утилиты Instruments, в качестве шаблона слежения я выбрал Time Profiler и щелкнул на кнопке Profile. На экране появилось главное представление; я коснулся ячейки в табличном представлении, и после значительной задержки на экране появилось детализированное представление. В этот момент я остановил работу утилиты Instruments и посмотрел на ее сообщение.

Как показано на рис. 9.14, этот переход занял примерно три секунды. Если щелкнуть на треугольниках открытия в нижней части окна, окажется, что большая часть этого времени уходит на выполнение метода `CA::Layer::layout_and_display_if_needed`. Это не мой код; он скрыт глубоко в недрах каркаса Cocoa. Однако, щелкнув на маленькой стрелке, которая появляется справа от строки, на которую указывает курсор мыши, можно увидеть стек вызовов и выяснить, каким образом мой код связан с этим методом (рис. 9.15).

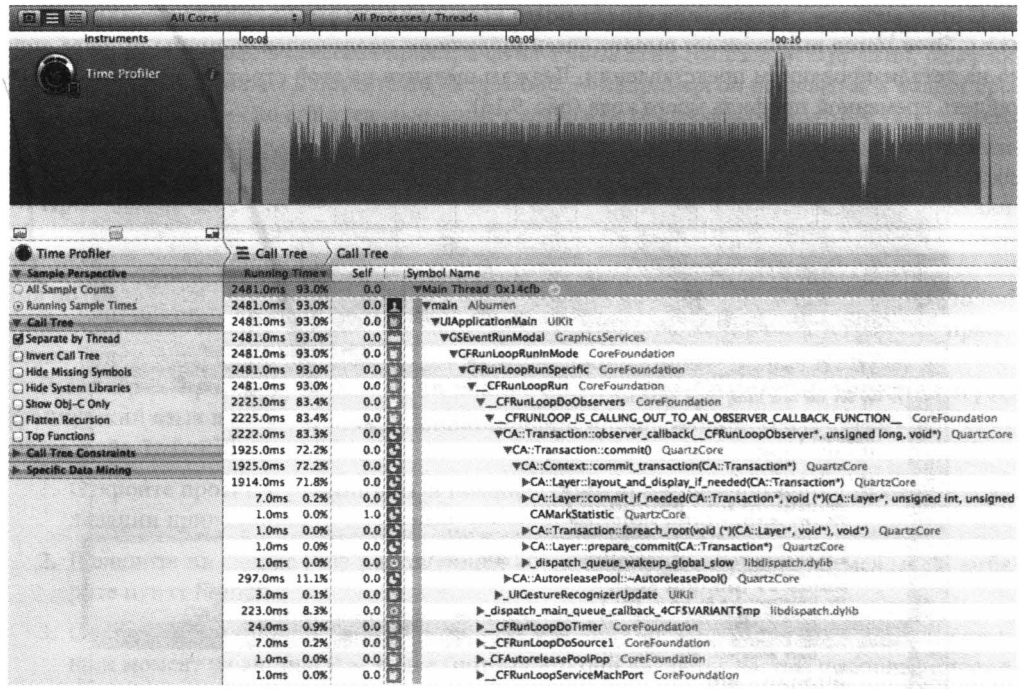


Рис. 9.14. Временной профиль в утилите Instruments

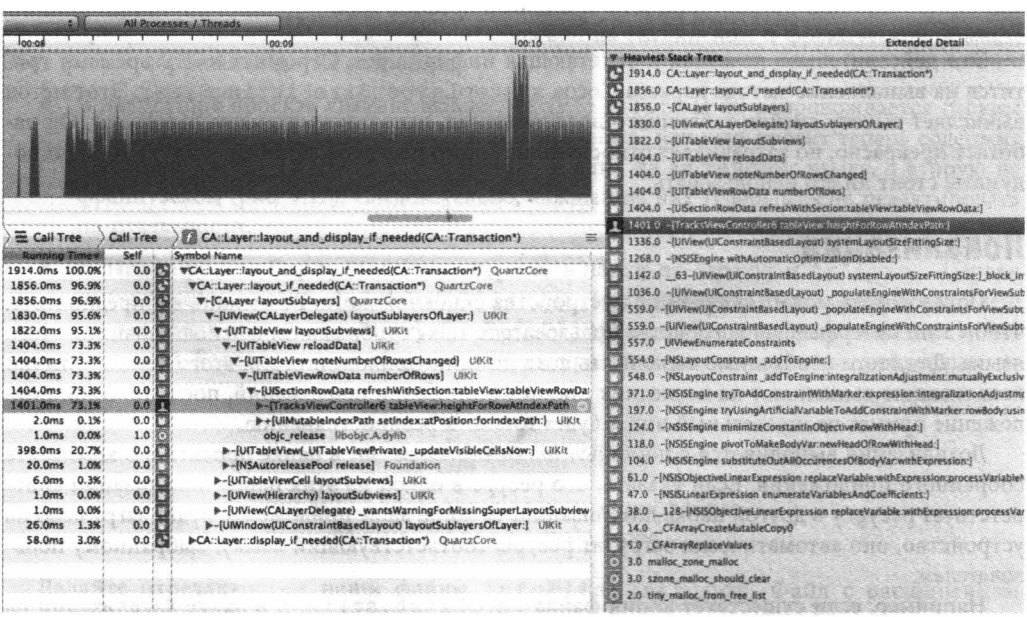


Рис. 9.15. Разбор временного профиля

С этим методом связана одна строка моего кода: `tableView:heightForRowAtIndexPath:`. Этот метод выравнивает высоту ячеек табличного представления, чтобы их было видно на детализированном представлении. Дважды щелкнув на этой строке в листинге, можно увидеть временной профиль моего кода (рис. 9.16).

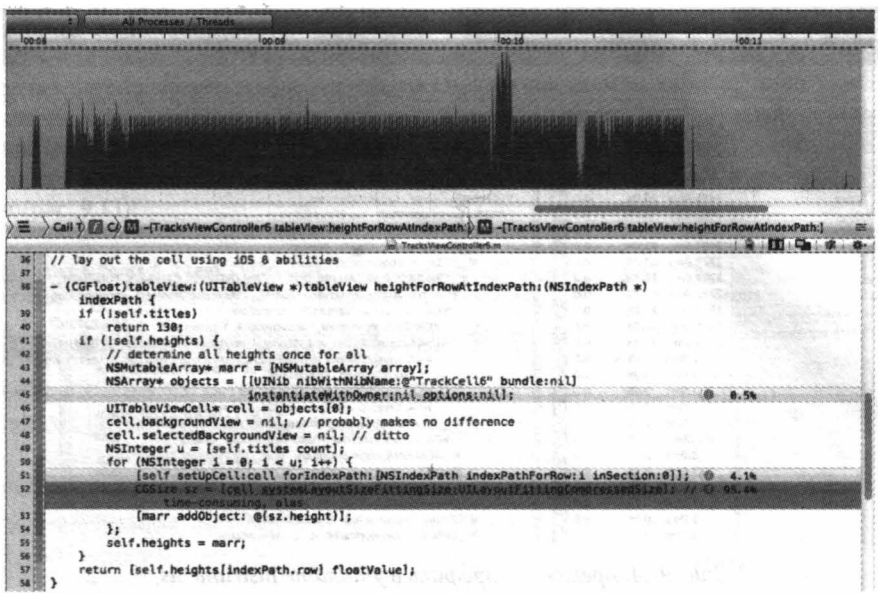


Рис. 9.16. Временной профиль кода

Это действительно полезная и угнетающая информация. Огромная часть времени тратится на выполнение метода каркаса Cocoa `systemLayoutSizeFittingSize:`. Этот метод вычисляет высоту ячейки табличного представления, используя механизм Autolayout. Он работает прекрасно, но затрачивает относительно много ресурсов, поэтому следует хорошо подумать, стоит ли его использовать.

Локализация

Пользователь может выбрать для устройства основной язык. Возможно, вы предпочтете, чтобы ваш интерфейс на устройстве пользователя тоже отображался с использованием этого языка. Для этого необходимо выполнить локализацию приложения для этого языка. Обычно локализацию выполняют на последних стадиях разработки приложений, после того как приложение примет окончательный вид и будет готово к распространению.

Локализация выполняется с помощью папок локализации в папке проекта и в комплекте собранного приложения. Если указать, что ресурс в одной из этих папок локализации соответствует ресурсу в другой папке локализации, то когда ваше приложение будет загружено на устройство, оно автоматически загрузит ресурс, соответствующий языку, выбранному пользователем.

Например, если существует копия файла `InfoPlist.strings` в папке локализации на английском языке и копия файла `InfoPlist.strings` в папке локализации на французском языке, то второй вариант будет использован, когда приложение затребует копию файла

InfoPlist.strings на устройстве, на котором основным языком является французский. Я недаром использовал в качестве примера файл InfoPlist.strings. Это файл, содержащий настройки, принятые в проекте по умолчанию, — например, он появляется в нашем проекте Empty Window, — но его предназначение в главе 6 не обсуждалось, поэтому, скорее всего, вы остались в недоумении. Для того чтобы снять вопросы, отвечу, что это файл с расширением .strings и любой файл с расширением .strings, предназначенный для локализации.

Предназначением данного файла InfoPlist.strings является хранение локализованных версий значений ключей из файла Info.plist. Например, значение ключа CFBundleDisplayName, как указано в файле Info.plist вашего проекта, выводится как имя под пиктограммой приложения на устройстве пользователя. Вы можете изменять это имя в зависимости от выбора основного языка. Например, на франкоязычном устройстве приложение Empty Window можно назвать Fenetre Vide.

Рассмотрим процедуру локализации нашего приложения. Сначала мы должны выбрать французский язык в качестве основного языка нашего устройства, затем необходимо локализовать файл InfoPlist.strings.

1. Откройте проект. В разделе Info в таблице Localizations перечислены возможные локализации приложения.
2. Щелкните на кнопке Plus под таблицей Localizations. В раскрывающемся меню выберите пункт French.
3. Откроется диалоговое окно, в котором перечисляются файлы, локализованные в текущий момент на английском языке (потому что они являются частью шаблона приложения). Нас интересует строка InfoPlist.strings, поэтому выберем ее, не выбирая никаких других файлов. Щелкните на кнопке Finish.

Теперь файл InfoPlist.strings локализован на английском и французском языках. Этот факт отображается двумя способами.

- В навигаторе проекта листинг файла InfoPlist.strings сопровождается гибким треугольником. Щелкните на нем, чтобы выяснить, что наш проект теперь содержит две копии файла InfoPlist.strings — одну на английском языке, а вторую на французском (рис. 9.17). Следовательно, каждый из них теперь можно редактировать по-отдельности.
- В папке проекта Empty Window теперь есть папки en.lproj и fr.lproj. Первая папка содержит копию файла InfoPlist.strings для англоязычных пользователей; вторая папка содержит копию файла InfoPlist.strings для франкоязычных пользователей. Более того, при сборке эта структура папок встраивается в приложение.

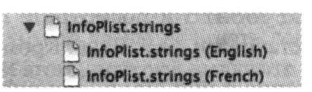


Рис. 9.17. Отображение локализации файлов с расширением .string в среде Xcode

Давайте отредактируем наши файлы InfoPlist.strings. Файл с расширением .strings — это просто коллекция пар ключ–значение в следующем формате:

```
/* Optional comments are C-style comments */
"key" = "value";
```

В случае файла `InfoPlist.strings` ключ — это просто имя ключа из файла `Info.plist` — реальное имя ключа, а не его название на английском языке. Например, англоязычный файл `InfoPlist.strings` должен выглядеть так:

```
"CFBundleDisplayName" = "Empty Window";
```

Франкоязычный файл `InfoPlist.strings` должен выглядеть так:

```
"CFBundleDisplayName" = "Fenetre Vide";
```

Попробуем проверить!

1. Соберите и запустите приложение Empty Window в программе iPhone Simulator.
2. Остановите выполнение проекта в среде Xcode. В симуляторе появится главное окно.
3. Проверьте название своего приложения, выведенное в главном окне симулятора. Это `Empty Window` (возможно, в сокращенном виде).
4. Запустите в симуляторе приложение Settings и измените язык на французский (General⇨International⇨Language⇨Francais). Теперь имя приложение отображается как Fenetre Vide.

Это приятно или нет? Когда вы закончите восхищаться своим космополитизмом, измените язык симулятора обратно на английский.

Поговорим теперь о nib-файлах. До появления среды Xcode 4.5 и операционной системы iOS 6 необходимо было локализовать копию всего nib-файла. Например, если вы хотели иметь франкоязычную версию nib-файла, вам приходилось поддерживать два разных nib-файла. Если вы создали кнопку в одном nib-файле, то должны были создать такую же кнопку в другом — за исключением того, что в одном nib-файле она имела англоязычное название, а в другом — франкоязычное. И так для каждого объекта интерфейса и каждого языка локализации. Это уже не так приятно, правда?

К счастью, существует более удобный способ — базовая интернационализация (base internationalization). Если проект использует базовую интернационализацию, то можно установить соответствие между nib-файлом в папке `Base.lproj` и файлом `.strings` в папке локализации. Таким образом, разработчик должен поддерживать только один экземпляр nib-файла. Если приложение выполняется на устройстве, локализованном для языка, которому соответствует файл с расширением `.strings`, то строки в этом файле будут заменены строками из nib-файла.

По умолчанию наш проект Empty Window использует базовую интернационализацию, и его файл `Main.storyboard` хранится в папке `Base.lproj`. Это позволяет нам локализовать файл раскадровки на французском языке.

1. Откройте файл `Main.storyboard` и посмотрите на инспектор файлов. В разделе Localization должен быть установлен флаг Base. Кроме того, установите флаг French.
2. Обратите внимание на листинг файла `Main.storyboard` в навигаторе проекта. Теперь возле него нарисован гибкий треугольник. Щелчок на этом треугольнике открывает файл. Теперь у нас есть файл `Main.storyboard` для базовой локализации и файл `Main.strings` для франкоязычной локализации.
3. Откройте файл `Main.strings` для франкоязычной локализации. Он был создан автоматически с ключами, соответствующими каждому элементу интерфейса, имеющему название в файле `Main.storyboard`. По комментариям и именам ключей можно

догадаться, как работает этот механизм. В нашем случае существует один элемент интерфейса в файле `Main.storyboard`, и легко догадаться, какому элементу соответствует ключ в файле. Он выглядит примерно так:

```
/* Class = "IBUIButton"; normalTitle = "Howdy!"; ObjectID = "Df5-YJ-JME"; */
"Df5-YJ-JME.normalTitle" = "Howdy!";
```

4. Во второй строке, содержащей пару ключ–значение, измените значение на "`Bonjour!`". Ключ не изменяйте! Он был сгенерирован автоматически и правильно, чтобы обеспечивать соответствие между значением и названием кнопки.

Теперь запустим проект в симуляторе и протестируем его англоязычную и франкоязычную версии. Перед этим закомментируйте строку `self.button` в файле `ViewController.m`! Эта строка изменяет название кнопки на ее название из nib-файла, но именно заголовок в nib-файле мы и собираемся локализовать.

Запустите проект. Вы должны увидеть, что на англоязычном устройстве заголовок кнопки имеет вид "`Howdy!`", а на франкоязычном — "`Bonjour!`"

Если теперь модифицировать nib-файл, например добавить еще одну кнопку на представлении в файл `Main.storyboard`, то автоматического изменения соответствующих файлов с расширением `.strings` не произойдет, потому что их нужно редактировать вручную. Следовательно, в реальной жизни не следует начинать локализацию nib-файлов, пока не будет закончена разработка интерфейса. Итак, вот что следует сделать.

1. Выберите файл `Main.storyboard` и команду File⇨Show в окне Finder.
2. Запустите приложение Terminal. Наберите в командной строке команду `ibtool --export-strings-file output.strings`, после которой должен стоять пробел, и перетащите, файл `Main.storyboard` из окна Finder в окно Terminal. Нажмите клавишу <Return>.

В результате на основе файла `Main.storyboard` в текущем каталоге будет сгенерирован новый файл `output.strings`. Объединение этой информации с существующими файлами `.strings` на основе раскадровки `Main.storyboard` программист должен сделать самостоятельно.

Для полноты картины покажем, как заставить проект использовать базовую интернационализацию, если он этого еще не сделал. В качестве примера будем использовать проект Truly Empty, созданный на основе шаблона Empty Application и не имеющий nib-файла. У нас есть файл `ViewController.xib`. Давайте его локализуем.

1. Откройте файл `ViewController.xib` и локализуйте его на английском языке, щелкнув на кнопке Localize в окне инспектора файлов. На экране откроется диалоговое окно, спрашивающее вас, хотите ли вы начинать создание файла `.lproj` для англоязычной версии. Мы хотим этого, поэтому щелкните на кнопке Localize.
2. Откройте проект и установите флаг Use Base Internationalization под таблицей Localization.
3. На экране откроется диалоговое окно, в котором перечислены nib-файлы, которые в данный момент локализованы только на английском языке. Сейчас этот список содержит только файл `ViewController.xib`, и он выбран. Хорошо! Щелкните на кнопке Finish.

После этого проект готов к новой локализации, которую можно провести так, как было показано в предыдущих примерах.

В заключение следует вспомнить о строках, которые появляются в интерфейсе приложения, но являются значениями, сгенерированными в коде? В приложении Empty Window примером такой строки может служить сообщение, которое активизируется после щелчка на кнопке. Здесь применяется точно такой же подход — файл с расширением `.strings`, — но ваш код необходимо модифицировать явным образом. Это можно сделать по-разному, но проще всего использовать макрос `NSLocalizedString` (который вызывает метод `localizedStringForKey:table:` из экземпляра класса `NSBundle`). Итак, например, мы можем модифицировать метод `buttonPressed:`.

```
UIAlertView* av = [[UIAlertView alloc]
                    initWithTitle:NSLocalizedString(@"AlertGreeting", nil)
                    message:NSLocalizedString(@"YouTappedMe", nil)
                    delegate:nil
                    cancelButtonTitle:NSLocalizedString(@"Cool", nil)
                    otherButtonTitles:nil];
```

Строка, представляющая собой первый аргумент метода `NSLocalizedString`, играет роль ключа в файлах с расширением `.strings`. (Второй аргумент, который в данном случае равен `nil`, в реальной жизни был бы сообщением механизму локализации, объясняющим, какая информация должна содержаться в этой строке.) Однако наш код теперь стал неработоспособным, потому что в проекте нет соответствующего файла с расширением `.strings`! По умолчанию в этом коде предполагается существование файла `Localizable.strings`. Такого файла нет. Это не ошибка, но ключи не имеют значений, и хотя сам ключ можно использовать при выводе сообщения, это не то, чего мы хотели. Необходимо создать недостающий файл с расширением `.strings`.

1. Выберите команду File⇨New⇨File.
2. Откроется диалоговое окно, предлагающее выбрать шаблон. Слева в разделе iOS выберите пункт Resource, справа — пиктограмму Strings File. Щелкните на кнопке Next.
3. Назовите файл `Localizable.strings`. Выберите соответствующий пункт в разделе Group и убедитесь, что этот файл является частью целевого приложения Empty Window. Щелкните на кнопке Create.
4. Теперь необходимо локализовать новый файл. Выберите файл `Localizable.strings` в навигаторе проекта. Щелкните на кнопке Localize под таблицей Localization в инспекторе файлов. На экране откроется диалоговое окно, предлагающее переместить существующий файл в англоязычную или базовую локализацию; в нашем проекте они совпадают.
5. Перейдите в инспектор файлов и добавьте в нем желаемые локализации. Например, установите флаг French.

Кроме того, необходимо наполнить файлы `Localizable.strings` содержимым в соответствии с ключами локализованных строк, заданных в вашем коде. Это можно сделать вручную с помощью инструмента командной строки `ibtool`, генерирующего файл с расширением `.strings` из nib-файла, или инструмента `genstrings` для генерирования файла с расширением `.strings` из исходного файла. Например, на моем компьютере я набрал в окне Terminal команду

```
$ genstrings /Users/matt/Desktop/Empty\ Window/Empty\ Window/ViewController.m
```

В результате в текущем каталоге возник файл `Localizable.strings`, имеющий следующее содержание:

```
/* No comment provided by engineer. */
"AlertGreeting" = "AlertGreeting";

/* No comment provided by engineer. */
"Cool" = "Cool";

/* No comment provided by engineer. */
"YouTappedMe" = "YouTappedMe";
```

Скопируйте и вставьте это содержимое в англоязычную и франкоязычную версии файлов `Localizable.strings` в нашем проекте, пройдитесь по парам, изменяя значение в каждой паре в соответствии с указанной локализацией. Например, в англоязычном файле `Localizable.strings` должна быть пара ключ-значение:

```
"AlertGreeting" = "Howdy!";
```

а во франкоязычном файле `Localizable.strings` —

```
"AlertGreeting" = "Bonjour!";
```

И так далее.

Архивирование и распространение

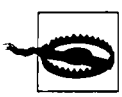
Под распространением (distribution) подразумевается предоставление возможности лицам, не являющимся разработчиками из вашей команды, собирать приложение для выполнения на своем устройстве. Существуют два вида распространения

Ситуативное распространение

Вы предоставляете копию вашего приложения ограниченному кругу известных пользователей, чтобы они могли испытать его на своих конкретных устройствах и сообщить об ошибках, сделать предложения и т.д..

Распространение через App Store

Вы поставляете приложение в интернет-магазин App Store, чтобы любой мог загрузить его (возможно, бесплатно) и выполнить.



Существует регистрационный лимит — 100 устройств в год на одного разработчика (не на приложение), который ограничивает количество ситуативных тестеров. Устройства, использованные для разработки, также учитываются в этом лимите.

Для того чтобы создать копию приложения для распространения, сначала необходимо собрать и заархивировать приложение. Именно этот архив впоследствии будет экспортироваться для ситуативного распределения или распределения через App Store. Архив — это сохраненная сборка. Он имеет три цели.

Распространение

Архив служит основой для ситуативного распространения или распространения через App Store.

Воспроизведение

При каждой сборке условия могут изменяться, поэтому получившееся приложение может работать немного иначе. Однако архив сохраняет конкретную бинарную сборку; каждое приложение, извлеченное из конкретного архива, гарантированно будет идентично этой бинарной сборке и будет работать точно так же. Этот факт важен для тестирования: если источником ошибки является конкретный архив, вы можете раздать его в рамках ситуативного распространения и выполнить, зная, что вы тестируете одно и то же приложение.

Символизация

Архив содержит файл с расширением `.dSYM`, позволяющий программе Xcode получать регистрационные записи о сбоях и сообщать о месте сбоя в вашем коде. Это позволяет работать с отчетами об ошибках, полученными от пользователей.

Опишем процедуру создания архива.

1. Укажите в качестве цели в раскрывающемся списке Scheme на инструментальной панели в окне проекта устройство `iOS Device`. (Пока вы не сделаете это, команда Product⇨Archive будет отключена. Устройство подключать не обязательно; мы не собираемся собирать приложение для выполнения на конкретном устройстве, а хотим сохранить архив, который будет запускаться на каком-то устройстве.)
2. Если хотите, отредактируйте схему, чтобы подтвердить, что при выполнении действия Archive будет использоваться конфигурация сборки Release. (Эта конфигурация используется по умолчанию, но проверить не мешает.)
3. Выберите команду Product⇨Archive. Она компилирует и собирает приложение. Сам архив хранится в папке данных в папке пользователя `Library/Developer/Xcode/Archives`. Кроме того, он указывается в окне Organizer в списке Archives, который можно спонтанно открыться, чтобы показать, что архив действительно создан. Здесь можно добавлять комментарии; кроме того, можно изменить имя архива (это не влияет на имя приложения).

В прошлом я рекомендовал модифицировать настройки сборки проекта и, возможно, добавлять конфигурацию, чтобы в процессе сборки архив был подписан с профилем для распространения. Оказалось это совсем не обязательно! По умолчанию архив будет подписан с профилем для разработки; позднее, при экспорте архива, этот профиль можно заменить на профиль для распространения.

Для того чтобы выполнить распространение любого вида с помощью архива, необходимо иметь идентификатор для распространения (закрытый ключ и сертификат распространения в связке ключей на вашем компьютере) и профиль для распространения, предназначенный специально для данного приложения. Если вы осуществляете ситуативное распространение или распространение через App Store, то для каждого из этих видов распространения вам понадобится отдельный профиль.

Идентификатор для распространения можно получить, не выходя из среды Xcode, точно так же, как было указано при описании процедуры получения идентификатора для разработки: в окне настроек Accounts в диалоговом окне View Details выберите свою команду, щелкните на кнопке Plus и выберите команду iOS Distribution. Если этот способ не сработает, получите сертификат вручную, как описано ранее.

В среде Xcode невозможно создать профиль для распространения; это можно сделать только в центре Member Center с помощью вашего браузера.

1. Если хотите создать профиль для ситуативного распространения, соберите уникальные идентификаторы всех устройств, на которых данная сборка будет выполняться, и добавьте их в раздел Devices в центре Member Center. (Если вы планируете распространять приложение через интернет-магазин App Store, можете пропустить этот этап.)
2. Убедитесь, что приложение зарегистрировано в центре Member Center, как было описано ранее.
3. Находясь в центре Member Center в разделе Provisioning Profiles, щелкните на кнопке Plus, чтобы запросить новый профиль. В форме Add iOS Provisioning Profile укажите профиль `Ad Hoc` или `App Store`. На следующем экране выберите свое приложение в раскрывающемся списке. На следующем экране выберите свой сертификат распространения. На следующем экране только для профиля ситуативного распределения укажите устройства, на котором вы хотите выполнить свое приложение. На следующем экране присвойте профилю имя.

   Имя профиля следует выбирать осторожно, потому что позднее вам нужно будет узнать его в среде Xcode! Мой опыт подсказывает, что лучше давать имя, содержащее название приложения и слова "AdHoc" или "AppStore".
4. Щелкните на кнопке Generate, чтобы сгенерировать профиль. Для того чтобы получить профиль, надо либо щелкнуть на кнопке Download, а затем найти загруженный профиль и дважды щелкнуть на нем в среде Xcode, либо открыть диалоговое окно View Details в окне настроек Accounts в среде Xcode и щелкнуть на кнопке Refresh в левом нижнем углу.

Ситуативное распространение

Документация компании Apple утверждает, что сборка, предназначенная для ситуативного распространения (Ad Hoc distribution), должна включать в себя пиктограмму, которая будет появляться в магазине iTunes, но мой опыт подсказывает, что это не обязательно. Если вы хотите включить эту пиктограмму в сборку, она должна представлять собой файл в формате PNG или JPEG размером 512×512 с именем `iTunesArtwork` и без расширения. Убедитесь, что пиктограмма включена в сборку на фазе Copy Bundle Resources.

Для создания файла, предназначенного для ситуативного распространения, необходимо выполнить следующие действия.

1. Соберите архив вашего приложения, как описано в предыдущем разделе.
2. В окне Organizer в разделе Archives выберите архив и щелкните на кнопке Distribute в правом верхнем углу окна. В открывшемся диалоговом окне необходимо указать процедуру; выберите команду Save for Enterprise или Ad-Hoc Deployment. Щелкните на кнопке Next.
3. Теперь вам предложат ввести кодовый знак приложения. Вы должны увидеть список профилей обеспечения распространения. Выберите профиль распространения Ad Hoc для нашего приложения — теперь вы увидите, насколько важно давать профилю распространения осмысленное имя!

4. Откроется диалоговое окно Save. Дайте файлу информативное имя; оно не влияет на имя приложения. Сохраните файл на диске. Он будет иметь расширение `.ipa` ("iPhone app").
5. Найдите в окне Finder только что сохраненный файл. Отправьте его своим пользователям вместе с инструкцией.

Пользователь должен скопировать файл с расширением `.ipa` в безопасное место, например в папку Desktop, а затем запустить программу iTunes и перетащить файл с расширением `.ipa` из окна Finder на пиктограмму iTunes, расположенную на панели Dock. Затем пользователь должен присоединить устройство к компьютеру, включить конкретное приложение в список приложений, выполняемых на данном устройстве, и инсталлировать его в ходе следующего сеанса синхронизации, а в заключение синхронизировать устройство, чтобы скопировать на него приложение. (Если это не первая версия приложения, распространяемого вами среди ситуативных тестеров, пользователю, возможно, придется сначала удалить текущую версию с устройства; в противном случае новая версия может не скопироваться на устройство во время сеанса синхронизации.)

Если в списке устройств, для которых предназначен профиль обеспечения ситуативного распространения, указано и ваше устройство, возможно, вы также будете обязаны выполнить все эти инструкции, чтобы ситуативное распространение было выполнено правильно. Для начала удалите с вашего устройства все предыдущие копии вашего приложения (например, созданные при разработке) и все профили, которые можно ассоциировать с этим приложением (в приложении Settings после выбора команды General⇨Profiles). Затем скопируйте приложение на свое устройство, синхронизируя его с программой iTunes, как описано выше. Приложение должно выполняться на вашем устройстве, и вы должны увидеть на своем устройстве профиль ситуативного распространения (в приложении Settings). Поскольку у вас нет привилегий по сравнению с другими ситуативными тестерами, приложение будет работать на вашем устройстве точно так же, как и на устройствах других тестеров.

Последние приготовления приложения

Поскольку приближается знаменательный день, когда вы планируете представить свое приложение в интернет-магазин App Store, не давайте соблазну большой славы и огромной прибыли сбить вас с правильного пути и не пропускайте чрезвычайно важные этапы заключительной подготовки приложения. Компания Apple предъявляет к вашему приложению массу требований; например, она требует наличия пиктограмм и заставок, а игнорирование их приведет к тому, что ваше приложение будет отклонено. Не спешите. Напишите памятку и строго ей следуйте. Прочитайте раздел "App-Related Resources" в справочнике iOS App Programming Guide компании Apple, где вы найдете все подробности.

На разных этапах вы можете получать оценку вашего приложения, чтобы убедиться, что не пропустили какие-то требования. Например, по умолчанию в конфигурации сборки Release для нового проекта параметр Validate Build Product по умолчанию установлен равным `Yes`. Таким образом, если приложение Empty Window, разработанное в предыдущих главах, использует конфигурацию сборки Release, то при его сборке программа Xcode выдаст предупреждение, что у приложения нет пиктограммы. Когда вы представите приложение в интернет-магазин App Store, оно будет подвергнуто еще более строгой проверке.

К счастью, среда Xcode 5 (в отличие от предыдущих версий) сообщает, какие размеры должны иметь пиктограмма и заставка. Если вы не используете каталог ресурсов, эти размеры

указываются, когда вы редактируете целевое приложение на вкладке General (рис. 9.18). Добавьте в проект рисунок в формате PNG правильного размера и щелкните на кнопке папки в правом конце строки, чтобы выбрать его в диалоговом окне.

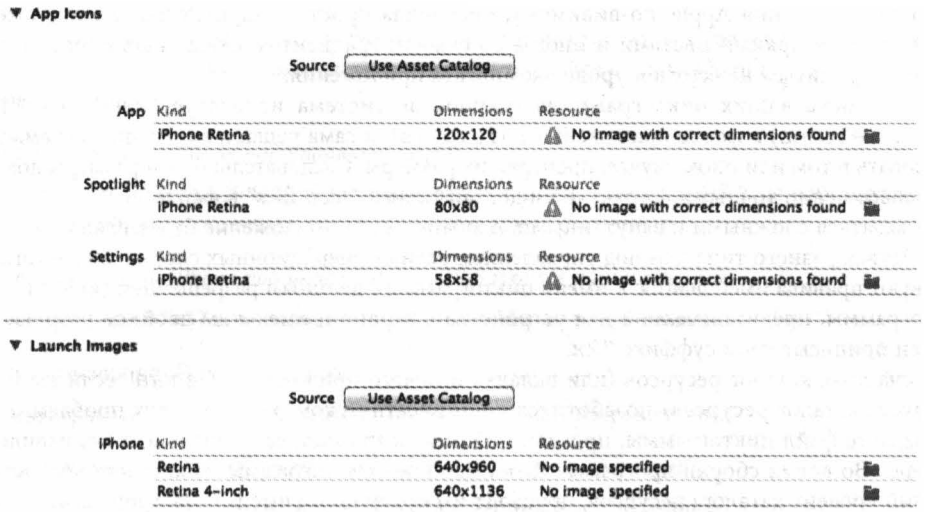


Рис. 9.18. Пиктограммы и заставки без каталога ресурсов

Если вы не используете каталог ресурсов для пиктограмм или заставок, но хотите его использовать, щелкните на кнопке Use Asset Catalog (рис. 9.18). Кнопка Use Asset Catalog открывает список, в котором указано имя каталога активов и набора пиктограмм в этом каталоге, доступных для приложения.

Если вы используете каталог ресурсов, то размеры изображений должны быть указаны в самом каталоге ресурсов. Выберите область изображения и обратите внимание на раздел Expected Size в окне инспектора атрибутов. (По непонятной причине, "2x" значит, что изображение должно быть в два раза больше, чем пиктограмма, а не заставка.) Определите, какие области должны отображаться на экране, установив флаги в окне инспектора атрибутов (рис. 9.19). Для того чтобы добавить изображение, перетащите его из окна Finder в соответствующую область.

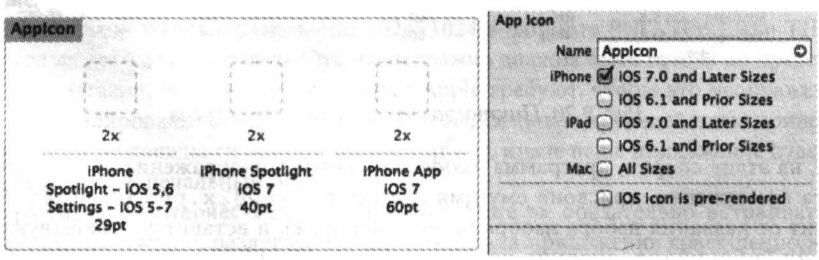


Рис. 9.19. Области пиктограмм в каталоге ресурсов

Я советую использовать каталог ресурсов! Он существенно упрощает конфигурирование пиктограмм, позволяя присваивать имена и правильно задавать параметры в файле `Info.plist` во время сборки. Примеры будут приведены в следующих разделах.

Пиктограммы в приложении

Файл пиктограммы должен иметь формат PNG без показателя прозрачности альфа. Пиктограмма должна быть квадратной; округление углов будет выполнено автоматически. В системе iOS 7 компания Apple, по-видимому, предпочла простые, карикатурные изображения с несколькими яркими цветами и иногда со слабым градиентом фона. Файлы пиктограмм должны находиться на верхнем уровне комплекта приложения.

Для поиска ваших пиктограмм на устройстве система использует ключ "Icon files" (`CFBundleIcons`) в файле приложения `Info.plist` и сама решает, какую пиктограмму использовать в том или ином случае, проверяя их размеры. Следовательно, эти размеры должны быть совершенно точными. Структура и детали записи "Icon files" в файле `Info.plist` могут показаться сложными и запутанными, особенно если приложение будет выполняться на устройствах разного типа или под управлением разных операционных систем. Более того, существуют правила именования: к имени пиктограммы с двойным разрешением (иначе говоря, пиктограммы, предназначенной для устройства с экраном, имеющим двойное разрешение) должен приписываться суффикс `@2x`.

К счастью, каталог ресурсов (или вкладка целевого приложения General, если вы не используете каталог ресурсов) позаботился об автоматическом решении всех проблем. Когда вы создаете файл пиктограммы, программа Xcode возражает, если она имеет неправильные размеры. Во время сборки программа Xcode запишет пиктограммы из каталога ресурсов на верхний уровень каталога ресурсов, присвоит им правильные имена и настроит файл с параметрами сборки `Info.plist`, чтобы ссылки на пиктограммы были правильными.

Рассмотрим для примера простейший случай: приложение, которое будет выполняться только на устройстве iPhone, только под управлением системы iOS 7. Для этого требуется одна пиктограмма с двойным разрешением 120×120 — вариант с одинарным разрешением не нужен, потому что система iOS 7 не работает ни на одном из устройств iPhone с одинарным разрешением экрана. Допустим, я перетащил такую пиктограмму из окна Finder в соответствующую область каталога ресурсов, как показано на рис. 9.20.

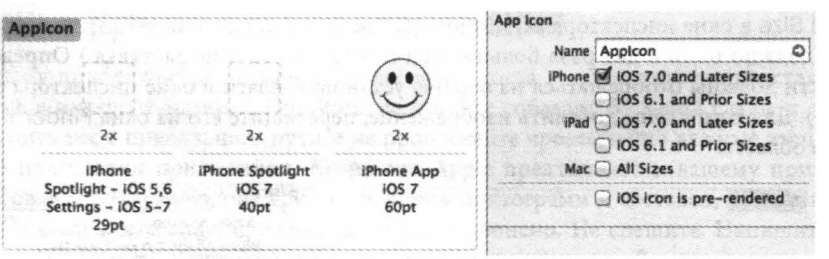


Рис. 9.20. Пиктограмма в каталоге ресурсов

Затем, на этапе сборки, программа Xcode запишет это изображение на верхний уровень комплекта приложения, присвоив ему имя `AppIcon60x60@2x.png` (префикс "`AppIcon`" происходит от названия набора изображений в каталоге), и вставит соответствующий материал в файл `Info.plist`, ссылаясь на этот графический файл, чтобы система на устройстве могла его найти.

```
<key>CFBundleIcons~iphone</key>
<dict>
    <key>CFBundlePrimaryIcon</key>
     <dict>
         <key>CFBundleIconFiles</key>
```

```
        <array>
            <string>AppIcon60x60</string>
        </array>
    </dict>
</dict>
```

Целесообразно перечислить требуемые размеры пиктограмм на разных системах и устройствах, потому что они также перечислены в интерфейсе системы Xcode 5. Итак, ключевые требования выглядят так.

Для приложения на iPhone под управлением iOS 7

Одна пиктограмма 120×120 с двойным разрешением. Система iOS 7 не работает на устройствах iPhone с экранами, имеющими одинарное разрешение.

Для приложения на iPhone под управлением iOS 6 и более ранних версий

Пиктограмма 57×57 и ее вариант 114×114 с двойным разрешением.

Для приложения на iPad под управлением iOS 7

Пиктограмма 76×76 и ее вариант 152×152 с двойным разрешением.

Для приложения на iPad под управлением iOS 6 и более ранних версий

Пиктограмма 72×72 и ее вариант 144×144 с двойным разрешением.

Приложение, предназначенное для разных устройств или нескольких систем (или и того и другого), должно иметь все варианты пиктограмм, перечисленные выше. Вот почему каталог ресурсов настолько удобен!

При желании можно включить в приложение уменьшенные пиктограммы, которые должны появляться на экране, когда пользователь выполняет поиск на устройстве (а также в приложении Settings, если вы включаете комплект настроек). В системе iOS 7 уменьшенная пиктограмма имеет размер 29×29 (58×58 при двойном разрешении) для комплекта Settings 40×40 (80×80 при двойном разрешении) для результатов поиска. Однако я никогда не включаю в приложения эти пиктограммы.

Другие пиктограммы

Когда вы будете представлять свое приложение в интернет-магазин App Store, вас попросят предъявить пиктограмму размером 1024×1024 в формате PNG, JPEG или TIFF, чтобы изобразить ее на веб-сайте магазина. Эта пиктограмма должна быть готова до представления приложения в магазин. Инструкции компании Apple требуют, чтобы это изображение было не просто масштабированной версией пиктограммы, символизирующей ваше приложение, и в то же время она не должна сильно отличаться от нее, иначе ваше приложение будет отвергнуто (у меня есть этот печальный опыт).

Пиктограмму для интернет-магазина The App Store не обязательно встраивать в ваше приложение; действительно, раздувать размер комплекта приложения совершенно не нужно. С другой стороны, ее целесообразно хранить в проекте (и в папке проекта), чтобы ее было легко найти и отредактировать. Итак, я рекомендую включить эту пиктограмму в свой проект и скопировать ее в папку проекта, но не включать ее в целевое приложение.

Если вы создали пиктограмму `iTunesArtwork` для ситуативного распространения, то, возможно, теперь захотите удалить ее из фазы Copy Bundle Resources.

Заставки

Между моментом, когда пользователь касается пиктограммы на экране устройства для запуска приложения, и началом его работы существует определенная задержка. Для того чтобы заполнить эту паузу и дать пользователю визуальное представление о происходящем, необходимо предусмотреть заставку в виде файла в формате PNG.

Заставкой могут быть пустые рамки основных элементов или областей интерфейса, которые появятся, когда процедура запуска приложения будет завершена. В этом случае после завершения процедуры запуска приложения переход от заставки к реальному приложению представляет собой заполнение этих рамок реальными элементами интерфейса. Для того чтобы создать такую заставку, лучше всего сделать снимок экрана реального начального интерфейса вашего приложения. Обычно я вставляю в свое приложение временный код, который выполняется, пока на экране отображается заставка с пустыми рамками обычного интерфейса; затем я делаю снимок экрана с этим интерфейсом и удаляю временный код. Процедура снятия снимка экрана описана в следующем разделе.

В системе iOS 7 заставка должна занимать весь экран. Для приложения, предназначенного для устройства iPhone, высота заставки должна совпадать с высотой экрана, даже если приложение запускается в альбомной ориентации (в этом случае следует скорректировать границы заставки, чтобы они соответствовали начальной ориентации). Для приложения, предназначенного для устройства iPad, заставка обычно имеет два варианта: для книжной и альбомной ориентации (причем как для одинарного, так и для двойного разрешения).

В системе iOS 6 и более ранних версиях заставки приложения для устройства iPad должны были иметь строку состояния на верхней границе экрана, если приложение не запускалось в полноэкранном режиме. Ширина строки состояния совпадала с шириной заставки, а высота составляла 20 пикселей (т.е. 40 пикселей для изображения с двойным разрешением. (Для того чтобы выполнять операцию обрезания этой строки легко и точно, я использую свободно распространяемую программу GraphicConverter, `http://www.lemkesoft.com`.)

После появления устройства iPhone 5 (и пятого поколения устройства iPod touch), имеющего более высокие экраны по сравнению с более ранними устройствами, заставка получила второе предназначение: она сообщает системе, должно ли приложение работать на этих устройствах "в исходном формате". Как правило, если приложение содержит заставку, специально предназначенную для удлиненного экрана, приложение будет обрезано ("letterboxed") до размеров устройства iPhone 4, так что сверху и снизу его будут обрамлять широкие черные полосы. Более высокий экран в каталоге ресурсов называется "R4".

(Вот почему по умолчанию, как я писал в главе 6, каталог ресурсов на этапе сборки генерирует черную заставку с именем `LaunchImage-700-568h@2x~iphone.png`: если этого не сделать, то ваше приложение на более высоком экране будет обрезано, как при просмотре широкоформатных фильмов.)

Основным именем графического файла с заставкой может быть либо `Default`, либо другое имя, указанное в файле `Info.plist`. Кроме того, существует множество квалификаторов имен: `-568h` для более высоких изображений на устройстве iPhone, `@2x` для изображений с двойным разрешением, и `~iphone` или `~ipad` для соответствующего типа устройства. Более того, в системе iOS 7 существует новый ключ `UILaunchImagesInfo.plist` со сложной структурой (см. справочник Information Property List Key Reference компании Apple). Для пиктограмм приложения проще всего использовать каталог ресурсов и предоставить ему выбирать имена файлов и настраивать файл `Info.plist` во время сборки.

Для современных типов устройств приняты следующие размеры экрана в пикселях.

iPhone 4S и младше

640×960 (двойное разрешение; система iOS 7 не работает на устройствах iPhone с одинарным разрешением).

iPhone 5 и старше

640×1136 (двойное разрешение; устройств iPhone 5 с одинарным разрешением не существует.)

iPad

1024×768 и 2048×1536

Снимки экрана

Когда вы представите свое приложение в интернет-магазин App Store, вас попросят прислать один или несколько снимков экрана вашего приложения для того, чтобы выставить их в магазине. Эти снимки экрана следует сделать заранее.

Требуемые размеры снимков экрана указаны в справочнике iTunes Connect Developer Guide компании Apple в разделе "Adding New Apps"; найдите там таблицу "Upload file sizes and format descriptions". Размеры снимков зависят от ориентации и размера экрана. Вы обязаны предоставить хотя бы один снимок экрана, соответствующий размерам экранов устройств каждого типа, на которых может работать ваше приложение, — не потому, что целевое устройство обязательно будет иметь двойное разрешение экрана, а потому, что существует возможность, что ваш снимок будет просматриваться на экране с двойным разрешением.

В системе iOS 6 и более ранних версиях, если приложение скрывало строку состояния, потому что выполнялось в полноэкранном режиме, область строки состояния перед передачей в магазин следовало отрезать от снимка (заставки для устройств iPad обсуждались в предыдущем разделе). В системе iOS 7 это не обязательно, поскольку в ней все приложения работают в полноэкранном режиме.

Снимки экрана можно сделать либо в симуляторе, либо на устройстве, подсоединенном к компьютеру.

Симулятор

Запустите приложение в симуляторе, используя список целевых устройств (и, при необходимости, меню симулятора Hardware), чтобы установить тип целевого устройства и разрешение. Выберите команду Choose File⇨Save Screen Shot.

Устройство

В окне Organizer среды Xcode найдите присоединенное устройство в списке Devices и щелкните на кнопке Screenshots. Щелкните на кнопке New Screenshot, расположенной в правом нижнем углу. Снимок экрана можно сделать в окне Organizer. Щелкнув на кнопке Export, вы получите снимок на рабочем столе и сделаете его доступным для загрузки в интернет-магазин App Store.

Снимок экрана можно сделать на устройстве, одновременно нажав клавиши блокировки экрана и клавишу <Home>. В этом случае снимок экрана появится в разделе Camera Roll приложения Photos, и вы сможете переслать его на компьютер любым удобным способом (например, послав электронное письмо самому себе).

Если снимок экрана есть в окне Organizer, его можно непосредственно преобразовать в заставку (см. предыдущий раздел): выберите его и щелкните на кнопке Save as Launch Image. В диалоговом окне вас попросят присвоить ему имя и указать, в какой проект его следует добавить.

Параметры в списке свойств

В файле `Info.plist` есть много настроек, играющих важную роль для правильного поведения вашего приложения. Их полное описание содержится в справочнике Information Property List Key Reference компании Apple. Большинство требуемых ключей создается как часть шаблона и имеет разумные значения, заданные по умолчанию, но их все равно следует перепроверить. Особое внимание следует уделить следующим ключам.

Имя комплекта на экране (`CFBundleDisplayName`)

Имя, которое выводится под пиктограммой приложения на экране устройства; это имя должно быть коротким, чтобы избежать усечения.

Поддерживаемые варианты ориентации интерфейса (`UISupportedInterfaceOrientations`)

Этот ключ задает спектр возможных вариантов ориентации интерфейса приложения. Эту настройку можно задать с помощью флагов на вкладке General в редакторе цели. Иногда приходится редактировать файл `Info.plist` вручную, чтобы изменить порядок следования вариантов ориентации, потому что вид приложения при запуске на устройстве iPhone определяет ориентация, указанная первой.

Требуемые возможности устройства (`UIRequiredDeviceCapabilities`)

Этот ключ необходимо устанавливать, когда приложение требует возможностей, которыми обладают не все устройства. Внимательно проверьте список возможных значений. Не используйте этот ключ, если он не влияет на работу вашего приложения на устройстве, не имеющем указанных возможностей.

Номер версии (`CFBundleVersion`)

Приложение должно иметь номер версии. Лучше всего задавать его на вкладке General в редакторе цели. Здесь существует небольшая путаница из-за наличия двух полей: Version и Build. Первое поле соответствует ключу "`Bundle versions string, short`" (`CFBundleShortVersionString`) в файле Info.plist, а второе — ключу "`Bundle version`" (`CFBundleVersion`). Насколько я понял, компания обращает внимание на второй ключ, только если не задан первый. Я считаю наиболее безопасным при предоставлении приложения в интернет-магазин Apple Store задать оба значения одинаковыми. Значение должно иметь вид строки версии, например "1.0". Именно этот номер будет фигурировать на веб-страницах интернет-магазина App Store. Невозможность увеличить значение строки при предоставлении новой версии приложения является причиной для отказа.

Стиль строки состояния (`UIStatusBarStyle`)

В течение многих лет компания Apple изменяла свои правила, касающиеся строки состояния, как перчатки. Существовали правила "серый или черный", "автоматический выбор цвета", "черный для iPad" и т.д. Новое правило для системы iOS 7 само по себе простое, но оно также отражает остатки отмененных правил.

В системе iOS 7 все приложения являются полноэкранными как на iPhone, так и на iPad, и строка состояния, если она видна, представляет собой прозрачную накладку. Существуют два стиля строк состояния: `UIStatusBarStyleDefault`, т.е. черный текст строки состояния, и `UIStatusBarStyleLightContent`, т.е. белый текст строки состояния. Способ, которым стиль строки состояния задается в системе iOS 7, изменился. Теперь он задается не на уровне приложения, а на уровне контроллера представления: считается, что в контроллерах представлений вашего приложения установлен стиль строки состояния `UIStatusBarStyleDefault`, который реализуется методом `preferredStatusBarStyle`.

Таким образом, современный подход подразумевает задание стиля строки состояния контроллерами представлений во время выполнения приложения, а не в файле `Info.plist`.

Тем не менее возможность задать стиль строки состояния в файле `Info.plist` остается либо для обеспечения обратной совместимости, потому что ваше приложение также может выполняться под управлением системы iOS 6 и более ранних версий, либо из-за того, что ваш старый код невозможно адаптировать для системы iOS 7. В этом случае для настройки стиля строки состояния можно продолжать использовать файл `Info.plist`; по умолчанию в системе iOS 7 этот параметр игнорируется.

Кроме того, можно попросить систему iOS 7 не игнорировать этот параметр. Для этого в файле `Info.plist` установите ключ "`View controller-based status bar appearance`" (`UIViewControllerBasedStatusBarAppearance`) равным `NO`. Это значит, что ваше приложение не будет использовать вызовы контроллера представлений, такие как `preferredStatusBarStyle` для определения стиля строки состояния; вместо этого оно установит общее свойство `statusBarStyle` класса `UIApplication`, как это делалось в системе iOS 6 и более ранних версиях. Теперь настройка `UIStatusBarStyle` в файле `Info.plist` не будет игнорироваться системой iOS 7; старые настройки `Black Translucent` ("`Transparent black style`", `UIStatusBarStyleBlackTranslucent`) и `Black Opaque` ("`Opaque black style`", `UIStatusBarStyleBlackOpaque`) будут считаться эквивалентными новой настройке `UIStatusBarStyleLightContent`. Эти настройки можно удобно изменять с помощью раскрывающегося списка **Status Bar Style** на вкладке **General** в редакторе цели.

Строка состояния вначале скрыта (`UIStatusBarHidden`)

Это свойство является параллельным стилю строки состояния. В системе iOS 7 визуализацией и сокрытием строки состояния управляют контроллеры представлений, реализующие метод `prefersStatusBarHidden`. Таким образом, настройка из файла `Info.plist` в системе iOS 7 игнорируется, если в этом файле ключ "`View controller-based status bar appearance`" (`UIViewControllerBasedStatusBarAppearance`) не задан равным `NO`. Ключ `UIStatusBarHidden` удобно задать, если он еще нужен, с помощью флага "`Hide during application launch`" на вкладке `General` в редакторе цели.

Параметры в списке свойств могут принимать разные значения в зависимости от устройств, на которых будет выполняться ваше приложение. Для того чтобы указать, что данный список свойств содержит настройки, предназначенные для конкретного типа устройств, добавьте к его ключам суффикс `~iphone`, `~ipod` или `~ipad`. Это свойство обычно является полезным для универсальных приложений, описанных ранее.

Представление приложения в интернет-магазин App Store

Перед тем как представить ваше приложение в интернет-магазин App Store, целесообразно сделать его архив, как было описано ранее, и в последний раз проверить как сборку Ad Hoc. Заархивированную сборку в окне Organizer можно использовать для генерирования сборки Ad Hoc или App Store. Сборку App Store тестировать нельзя, поэтому для тестирования можно использовать заархивированную сборку Ad Hoc. Генерируя сборку App Store, вы используете ту же самую заархивированную сборку; это гарантирует, что ее поведение будет совпадать с поведением протестированной сборки.

Если вы удовлетворены работой своего приложения и уже инсталлировали или собрали все необходимые ресурсы, значит, вы готовы представить свое приложение в интернет-магазин App Store для распространения. Для этого необходимо выполнить подготовительную работу на веб-сайте iTunes Connect. Ссылку на него можно найти на страницах разработчиков приложения для системы iOS при регистрации на веб-сайте компании Apple. Кроме того, это можно сделать непосредственно на веб-сайте `http://itunesconnect.apple.com`, но и здесь вам придется ввести пользовательское имя и пароль разработчика приложения для системы iOS.

При первом визите на веб-сайт iTunes Connect вы обязаны зайти в раздел Contracts и заполнить форму вашего контракта. Вы не имеете права предлагать на продажу никакие приложения, пока не заключите этот контракт, и даже для свободно распространяемых приложений необходимо заполнить определенную форму контракта.

Я не собираюсь описывать все этапы, которые вам придется пройти на веб-сайте iTunes Connect, чтобы выставить на продажу свое приложение, поскольку все это уже описано в справочнике iTunes Connect Developer Guide компании Apple, которому принадлежит последнее слово в этом вопросе. Я просто напомню основные сведения, которые вы должны предоставить.

Название вашего приложения

Это название, под которым оно появится в интернет-магазине App Store; оно не обязано совпадать с именем пиктограммы приложения на устройстве, которое определяется ключом "`Bundle display name`", заданным в файле `Info.plist`. Это имя может содержать до 255 символов, хотя компания Apple рекомендует не превышать 70 символов, а в идеале длина имени должна быть меньше 35. Вы можете удивиться, когда, предоставив информацию о приложении на веб-сайт iTunes Connect, узнаете, что выбранное вами имя уже кем-то используется. Заранее это узнать невозможно, поэтому вам придется потратить несколько минут на исправление.

Описание

Вы должны предоставить описание, длина которого не превышает 4000 символов (компания Apple рекомендует не превышать 580 символов). Особенно важным является первый абзац, поскольку почти все пользователи читают именно его, посещая интернет-магазин App Store. Это должен быть гладкий текст, без HTML-разметки и разных стилей символов.

Ключевые слова

Это список, разделенный запятыми, размер которого не превышает 100 символов. Эти ключевые слова наряду с именем вашего приложения помогут пользователям найти ваше приложение с помощью функции Search в интернет-магазине App Store.

Поддержка

Это URL-адрес веб-сайта, на котором пользователь может получить дополнительную информацию о вашем приложении; такой веб-сайт целесообразно создать заранее.

Авторские права

Не включайте в эту строку символ авторских прав; он будет добавлен самим интернет-магазином App Store.

Идентификатор товарной позиции

Он не имеет значения, поэтому беспокоиться о нем не стоит. Это просто уникальный идентификатор, выделяющий ваше приложение среди других. Он удобен, когда необходимо что-нибудь сделать с именем вашего приложения. Это не обязательно число; на самом деле этим идентификатором может быть любая строка.

Цена

Вы не можете произвольно установить цену. Вы должны выбрать ее из соответствующего списка.

Дата доступа

Существует возможность открыть доступ к приложению сразу, как только оно будет одобрено. Обычно это зависит от желания разработчика. В качестве альтернативы компания Apple пришлет вам электронное письмо с одобрением приложения, и тогда вам придется вернуться на веб-сайт iTunes Connect и сделать свое приложение доступным вручную.

Вы не имеете права проверять или загружать свое приложение, пока формально не сообщите на веб-сайте iTunes Connect об окончании своих приготовлений. Для этого необходимо щелкнуть на кнопке Ready to Upload Binary в правом верхнем углу веб-страницы iTunes Connect. Программисты часто делают ошибку, забывая это сделать, а потом интересуются, почему им не удается проверить или представить свое приложение.

Завершив ввод информации о своем приложении, вы можете выполнить последнюю проверку: вернитесь в окно Organizer, выберите заархивированную сборку и щелкните на кнопке Validate. (Впрочем, в прошлом эта функция работала не очень хорошо.)

В заключение, когда будете готовы загрузить приложение, которое вы только что описали, и статус вашего приложения на веб-сайте iTunes Connect изменится на "`Waiting for Upload`", вы можете выполнить загрузку с помощью программы Xcode. Выберите заархивированную сборку в окне Organizer и щелкните на кнопке Distribute. В диалоговом окне выберите команду "Submit to the iOS App Store". После этого будет выполнена загрузка и приложение будет подтверждено.

В качестве альтернативы можно использовать программу Application Loader. Экспортируйте архив как файл с расширением `.ipa`, как для ситуативного распространения, но при этом используя профиль распространения App Store. Запустите программу Application Loader, выбрав команду Xcode⇨Open Developer Tool⇨Application Loader и передайте ей файл `.ipa`.

Впоследствии вы получите электронное письмо от компании Apple, в котором вас проинформируют о состоянии вашего приложения на разных этапах: "`Waiting For Review`" "`In Review`" и, наконец, если все будет хорошо, "`Ready For Sale`" (даже если это свободно распространяемое приложение). Затем ваше приложение появится в интернет-магазине App Store.

ЧАСТЬ III

Cocoa

Для программирования в операционной системе iOS можно использовать совокупность каркасов, предоставляемых компанией Apple. Эти каркасы образуют среду Cocoa, зарегистрированную под торговой маркой Cocoa Touch и обеспечивающую интерфейс программирования приложений для системы iOS. Таким образом, среда Cocoa играет важную и принципиальную роль в программировании для системы iOS. В конечном итоге разрабатываемый вами прикладной код будет практически полностью связан со средой Cocoa. Каркасы Cocoa Touch предоставляют базовые функциональные возможности, требующиеся любому приложению для системы iOS. В приложении можно создать окно, отобразить пользовательский интерфейс, содержащий кнопку, отреагировать на ее нажатие пользователем и выполнить прочие действия благодаря тому, что в среде Cocoa заранее известно, как все это делается. Однако работа с каркасом приложений дает не только большие преимущества, но и накладывает не меньшие обязательства. Вам придется научиться мыслить категориями каркаса приложений, размещать свой код там, где он ожидает его обнаружить, а также выполнять много других обязательств, накладываемых каркасом. В главах этой части рассматриваются следующие вопросы.

- В главе 10, служащей продолжением главы 5, описываются некоторые языковые средства Objective-C, применяемые в среде Cocoa, в том числе категории и протоколы, а также дается краткий обзор наиболее важных базовых классов.
- В главе 11 описывается событийно-ориентированная модель наряду с основными шаблонами проектирования. *Событие* представляет собой сообщение, посылаемое из среды Cocoa в прикладной код. Среда Cocoa является событийно-ориентированной, и если она не посылает событие прикладному коду, то он не выполняется. Для того чтобы выполнить прикладной код в подходящий момент, нужно знать, какие именно события следует ожидать от среды Cocoa и когда они могут быть отправлены.
- В главе 12 описываются обязанности, связанные с получением экземпляров, аккуратно инкапсулированных с соблюдением правил управления памятью, принятых для объектов в среду Cocoa.
- В главе 13 даются ответы на некоторые вопросы, связанные с организацией взаимодействия объектов в среде, ориентированной на каркасы Cocoa.

ГЛАВА 10

Классы Cocoa

Для правильного применения каркасов Cocoa Touch требуется ясно понимать, каким образом в них организованы классы. Организация классов в среде Cocoa зависит от определенных языковых средств Objective-C, представленных в этой главе. Кроме того, в ней дается краткий обзор некоторых из наиболее употребительных служебных классов Cocoa, а также обсуждается корневой класс Cocoa.

Наследование

В среде Cocoa, по существу, предоставляется большое разнообразие объектов с заранее известным поведением, определенным требуемым образом. Например, объекту типа `UIButton` известно, как отобразить кнопку и как отреагировать на нажатие пальцем, производимое пользователем, а объекту типа `UITextField` известно, как отобразить поле редактируемого текста, обратиться к клавиатуре и принять вводимые с нее данные.

Зачастую стандартное поведение или внешний вид объекта, обеспечиваемые средой Cocoa, не вполне отвечают искомым, и поэтому объект требует специальной настройки. Однако это совсем не означает, что непременно необходимо наследование! Классы Cocoa наделены немалым количеством методов, которые можно вызывать, а также свойств, которые можно устанавливать таким образом, чтобы специально настроить экземпляр класса. Именно к этой мере и следует прибегать в первую очередь. Для того чтобы выяснить, готовы ли экземпляры классов выполнить то, что от них требуется, следует всегда обращаться за справкой к документации. Например, в документации на класс `UIButton` поясняется, что надпись на кнопке, цвет надписи, внутреннее изображение и много других свойств и видов поведения кнопки можно установить, не прибегая к наследованию.

Тем не менее установки свойств и вызова методов иногда оказывается недостаточно для специальной настройки экземпляра класса требуемым образом. В подобных случаях среда Cocoa может предоставить методы, вызываемые автоматически при выполнении экземпляром своих функций, и тогда его поведение можно специально настроить, прибегнув к наследованию и замещению (см. главу 4). Для этого не придется программировать встроенные в среду Cocoa классы, но можно выполнить их наследование, создав новый класс, действующий подобно встроенному классу, за исключением внесенных в него изменений.

Как ни странно, особенно для тех, кто имеет опыт работы с другим объектно-ориентированным каркасом приложений, наследование относится едва ли не к самым маловажным способам связывания прикладного кода со средой Cocoa. Определить точно момент, когда следует прибегнуть к наследованию, оказывается не так-то просто. Однако, как правило,

выполнять наследование не требуется, если только оно не предлагается явным образом. Некоторые классы Cocoa Touch, в том числе типичный класс UIViewController, подвергаются наследованию довольно часто. В то же время большинству классов, встроенных в среду Cocoa Touch, наследование вообще не требуется, а в документации на некоторые из них просто запрещается это делать.

Рассмотрим для примера класс UIView. В среде Cocoa Touch имеется немало встроенных подклассов, производных от класса UIView (UIButton, UITextField и т.д.), воспроизводящих и ведущих себя нужным образом и редко требующих своего наследования. С другой стороны, можно создать свой подкласс UIView, воспроизводящий себя совершенно по-новому. Для того чтобы воспроизвести представление типа UIView, достаточно вызвать метод drawRect: из класса UIView. Для того чтобы сделать то же самое совершенно по-другому, придется выполнить наследование класса UIView и реализовать метод в drawRect: в его подклассе. По этому поводу в документации говорится следующее: "Реализуйте этот метод, если в представлении воспроизводится особое содержимое". Слово "реализуйте" может означать лишь одно: реализацию в подклассе, т.е. наследование и замещение. Следовательно, в документации подчеркивается, что наследование класса UIView требуется лишь для того, чтобы воспроизвести содержимое по-своему и совершенно иначе.

Допустим, требуется создать окно, содержащее горизонтальную линию. В среде Cocoa отсутствует интерфейсный элемент управления для горизонтальной линии, и поэтому ее придется начертить самостоятельно, прибегнув к наследованию класса UIView, как поясняется ниже.

1. Перейдя к примеру проекта Empty Window, выберите команду меню File⇨New⇨File и укажите сначала класс Cocoa Touch Objective-C, а затем конкретный подкласс, производный от класса UIView. Присвойте новому классу имя MyHorizLine. При этом в среде Xcode автоматически создаются файлы MyHorizLine.m и MyHorizLine.h. Непременно сделайте их частью цели приложения.

2. Замените в файле MyHorizLine.m содержимое раздела следующим фрагментом кода без дополнительных пояснений:

```
- (id)initWithCoder:(NSCoder *)decoder {
   self = [super initWithCoder:decoder];
   if (self) {
       self.backgroundColor = [UIColor clearColor];
  }
   return self;
}

- (void)drawRect:(CGRect)rect {
    CGContextRef c = UIGraphicsGetCurrentContext();
    CGContextMoveToPoint(c, 0, 0);
    CGContextAddLineToPoint(c, self.bounds.size.width, 0);
    CGContextStrokePath(c);
}
```

3. Отредактируйте раскадровку. Найдите класс UIView в библиотеке Object, где он просто называется View, и перетащите его на объект View, находящийся на холсте. При желании можете изменить его размеры, чтобы сделать менее высоким.

4. Если класс UIView, который вы только что перетащили на холст, все еще выделен, воспользуйтесь инспектором идентичности, чтобы изменить этот класс на MyHorizLine.

Соберите и выполните приложение в симуляторе. В итоге должна появиться горизонтальная линия, соответствующая местоположению верхнего края экземпляра класса MyHorizLine в nib-файле. Таким образом, представление воспроизвело себя в виде горизонтальной линии, поскольку для этого было выполнено наследование его класса UIView.

Данный пример был начат с пустого класса UIView, не имевшего функциональных возможностей для рисования. Именно поэтому не потребовалось делать вызов метода суперкласса. По умолчанию в реализации метода drawRect: из класса UIView ничего не делается, но можно было бы также выполнить вывод подкласса встроенного класса UIView, чтобы изменить уже имеющийся порядок его воспроизведения. Например, из документации на класс UILabel следует, что для одной и той же цели в нем имеются два разных метода, drawTextInRect: и textRectForBounds:limitedToNumberOfLines:, причем вызывать их непосредственно не следует, а достаточно лишь заместить их в подклассах. Следовательно, эти методы будут вызываться автоматически из Cocoa при воспроизведении метки. Итак, выполнив наследование класса UILabel, можно реализовать эти методы в подклассе, чтобы изменить порядок воспроизведения метки.

Рассмотрим еще один пример из одного из моих приложений, где я выполняю наследование класса UILabel и замещаю метод drawTextInRect:, чтобы воспроизвести метку в собственной прямоугольной рамке. Как поясняется в документации: "В замещаемом методе можно дополнительно настроить текущий [графический] контекст и затем сделать вызов метода суперкласса для конкретного воспроизведения [текста]". Ниже поясняется, как это делается.

1. Создайте новый файл класса в проекте Empty Window, выполните наследование класса UILabel и присвойте новому подклассу имя MyBoundedLabel.
2. Введите следующий фрагмент кода в разделе реализации файла MyBoundedLabel.m:

```
- (void)drawTextInRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextStrokeRect(context, CGRectInset(self.bounds, 1.0, 1.0));
    [super drawTextInRect:CGRectInset(rect, 5.0, 5.0)];
}
```

3. Отредактируйте раскадровку. Введите класс UILabel в интерфейс, изменив его имя на MyBoundedLabel в инспекторе идентичности.

Соберите и выполните приложение. В итоге появится метка с текстовой надписью в прямоугольной рамке. (Для удобства сравнения воспроизводимых по-разному меток полезно ввести в интерфейс похожую метку из класса UILabel.)

Наследование редко выполняется в среде Cocoa еще и потому, что во многих встроенных в эту среду классах применяется механизм делегатов (см. главу 11) как один из способов специальной настройки их поведения. Вряд ли стоит выполнять наследование класса UIApplication (т.е. класса одиночного разделяемого экземпляра приложения) только для того, чтобы отреагировать на окончание запуска приложения, поскольку механизм делегатов предоставляет возможность сделать это (с помощью метода application:didFinishLaunchingWithOptions:). Именно поэтому в шаблонах предоставляется класс AppDelegate, который не является подклассом, производным от класса UIApplication. С другой стороны, если требуется произвести сложную в некотором роде специальную настройку основного поведения приложения при передаче сообщений о происходящих событиях, можно выполнить наследование класса UIApplication, чтобы заместить метод sendEvent:. О том же говорится и в документации, но там же подчеркивается, что делается это крайне редко.

Для наследования класса `UIApplication` требуется заменить значение `nil` третьего аргумента в вызове функции `UIApplicationMain()` из файла `main.m` на имя типа `NSString` получаемого в итоге подкласса. В противном случае экземпляр подкласса, производного от класса `UIApplication`, не будет получен как разделяемый экземпляр приложения (см. главу 6).

Категории

Категория — это языковое средство Objective-C, позволяющее перейти непосредственно к существующему классу и внедрить в него дополнительные методы. Это можно сделать даже в отсутствие кода для класса, как, например, для классов Cocoa. Методы экземпляра могут обращаться по ссылке `self`, а это, как обычно, будет означать экземпляр, которому первоначально послано сообщение. В отличие от подкласса, в категории нельзя определить дополнительные переменные экземпляра. В ней можно заместить методы, но вряд ли стоит пользоваться такой возможностью. Категория определяется практически так же, как и класс (см. главу 4). Для этого потребуются разделы интерфейса и реализации, которые, как правило, распределяются по парам файлов с расширениями `.h` и `.m`. Нужно, однако, иметь в виду следующее.

- В разделе интерфейса не следует объявлять суперкласс, поскольку он уже существует, а его суперкласс — уже объявлен. Разумеется, в разделе интерфейса должно бы доступно объявление исходного класса, для чего, как правило, импортирует заголовочный файл этого класса (или заголовочный файл каркаса, в котором он определен).
- Имя категории следует ввести в круглых скобках в начале обоих разделов интерфейса и реализации, где задается имя класса.

Если вы пользуетесь, как обычно, парой файлов с расширениями `.h` и `.m`, импортируйте файл с расширением `.h`. Если из любого другого файла с расширением `.m` в вашем проекте требуется вызвать какой-нибудь из методов, внедренных в категорию, то эти методы должны быть объявлены открытыми в файле с расширением `.h` данной категории, тогда как в файле с расширением `.m` должен быть организован импорт файла с расширением `.h` из данной категории.

Для того чтобы установить пару файлов с расширениями `.h` и `.m`, проще всего обратиться к среде Xcode, где это будет сделано автоматически. С этой целью выберите сначала команду меню File⇨New⇨File, а затем вариант Objective-C category среди прочих типов файлов среды Cocoa Touch для системы iOS в открывшемся диалоговом окне Choose a template. Вам будет предложено присвоить имя новой категории и классу, в котором она определяется.

С другой стороны, если методы, внедренные в данной категории, требуется вызывать только из одного класса, то вполне благоразумно разместить оба раздела интерфейса и реализации данной категории в заголовочном файле класса с расширением `.h`. В этом случае раздел интерфейса может быть пустым, а методы, определенные в разделе реализации данной категории, могут быть доступны для применения в файле с расширением `.m`.

Например, в одном из моих приложений я обнаружил, что целый ряд строковых преобразований выполняются с целью вывести путь к различным файлам ресурсов в пакете приложения по имени и назначению ресурса. В итоге оказалось, что для этой цели применяется полдесятка служебных методов. Поскольку все эти методы оперировали объектами класса `NSString`, то их целесообразно было реализовать в виде категории типа `NSString`. Таким образом, любой объект типа `NSString` мог реагировать на них в любом месте кода моего приложения.

Прикладной код был структурирован следующим образом (он приводится здесь лишь для одного метода):

```
// StringCategories.h:
#import <Foundation/Foundation.h>

@interface NSString (MyStringCategories)
- (NSString*) basePictureName;
@end

// StringCategories.m:
#import "StringCategories.h"

@implementation NSString (MyStringCategories)
- (NSString*) basePictureName {
    return [self stringByAppendingString:@"IO"];}
@end
```

Если бы метод `basePictureName` был реализован как служебный метод в каком-нибудь другом классе, ему нужно было бы принимать параметр, в качестве которого пришлось бы передавать объект типа `NSString`. Если бы это был метод экземпляра, то пришлось бы дополнительно получать ссылку на экземпляр данного класса. В этом отношении категория более аккуратна и компактна. Достаточно было расширить сам класс `NSString`, чтобы получить `basePictureName` в качестве метода его экземпляра. Таким образом, из любого файла с расширением `.m`, импортирующего файл `StringCategories.h`, можно отправить сообщение типа `basePictureName` непосредственно любому объекту типа `NSString`, который требуется преобразовать, как показано ниже.

```
NSString* aName = [someString basePictureName];
```

Категория особенно пригодна для таких классов, как `NSString`, поскольку в документации предупреждается, что наследование этого класса выполнять нецелесообразно. Дело в том, что класс `NSString` является составной частью более сложных классов, называемых *кластером классов*, а это означает, что настоящим классом объекта типа `NSString` на самом деле может оказаться совсем другой класс. Категория оказывается намного более удобной для видоизменения класса в кластере классов, чем наследование.

Метод, определяемый с помощью категории, может быть в равной степени методом класса. Таким образом, служебные методы можно внедрить в любой подходящий класс и затем вызывать их без дополнительных издержек на получение каких бы то ни было экземпляров. Классы доступны глобально, а следовательно, их методы, по существу, становятся глобальными (см. главу 13).

Например, в одном из своих приложений я обнаружил часто используемый определенный цвет (`UIColor`). Вместо того чтобы повторять инструкции для формирования этого цвета всякий раз, когда он понадобится, я разместил эти инструкции в одном месте: в разделе реализации категории в файле с расширением `.m`, как показано ниже.

```
@implementation UIColor (MyColors)
  + (UIColor*) myGolden {
      return [self colorWithRed:1.000 green:0.894 blue:0.541 alpha:.900]; }
@end
```

Метод `myGolden` я объявил в разделе интерфейса категории соответствующего файла с расширением `.h` и организовал импорт этого файла в файле своего проекта с расширением `.pch` (предварительно скомпилированном файле заголовка). Поскольку предварительно скомпилированный файл заголовка автоматически импортируется через мой проект, то я могу теперь делать вызов `[UIColor myGolden]` откуда угодно.

Разделение класса

С помощью категории можно разделить класс на несколько пар файлов с расширением `.h` и `.m`. Если класс угрожает стать слишком длинным и громоздким, но в то же время он явно должен оставаться единым, то в одной паре файлов можно определить основную его часть, включая и переменные экземпляра, а в другой паре — отдельную категорию класса, содержащую дополнительные методы.

Такое разделение классов происходит в самой среде Cocoa. Характерным тому примером служит класс `NSString`. Он определен как часть каркаса Foundation, а его основные методы объявлены в файле `NSString.h`. В этом файле находится сам класс `NSString` без категории, но с двумя методами `length` и `characterAtIndex:`, поскольку они считаются тем минимумом, который требуется для того, чтобы строка была символьной. Дополнительные методы для создания символьных строк, их кодирования, разбиения и поиска в них сосредоточены в категориях. Интерфейс для некоторых из этих категорий находится в том же самом файле `NSString.h`. Однако символьная строка может служить в качестве имени пути к файлу, и поэтому соответствующая категория класса `NSString` находится в файле `NSPathUtilities.h`, где объявлены методы для разбиения символьной строки имени класса на составляющие и тому подобных операций. Далее в файле `NSURL.h` находится еще одна категория класса `NSString`, в которой объявляются методы для экранирования знаком процента специальных символов в символьных строках URL. И наконец в файле `NSStringDrawing.h` из совершенно другого каркаса (UIKit) вводятся еще две категории с методами для воспроизведения символьных строк в графическом контексте.

Для программиста такая организация класса `NSString` не имеет особого значения, поскольку он так и остается классом `NSString`, независимо от того, каким образом он получает свои методы. Однако она может иметь значение, когда вы обращаетесь за справкой к документации. Методы класса `NSString`, объявленные в файлах `NSString.h`, `NSPathUtilities.h` и `NSURL.h`, описываются на странице документации на класс `NSString`, но там отсутствуют методы этого же класса, объявленные в файле `NSStringDrawing.h`. Это можно объяснить тем, что они происходят из другого каркаса и поэтому описываются в отдельном документе "NSString UIKit Additions Reference" (Справочник дополнений класса `NSString` из каркаса UIKit). В итоге найти описание методов воспроизведения символьных строк не так-то просто, особенно если учесть, что документация на класс `NSString` никак не связана с другими документами. Я считаю это главным изъяном в структуре документации о среде Cocoa. На помощь здесь можно призвать стороннюю утилиту вроде AppKiDo.

Расширения классов

Расширение класса — это безымянная категория, отдельно существующая как раздел интерфейса, например, следующим образом:

```
@interface MyClass ()
// здесь следует конкретное наполнение
end
```

Как правило, расширение класса доступно *только* расширяемому классу или производному от него подклассу. В таком случае расширение класса обычно находится в файле реализации этого класса с расширением `.m`, как показано ниже.

```
// MyClass.m:

@interface MyClass ()
  // здесь следует конкретное наполнение
```

```
@end
@implementation MyClass {
  // переменные экземпляра
}
// методы
@end
```

Это типичный порядок расположения расширений определенных классов, принятый в файлах шаблонов для проектов, выполняемых в Xcode. Например, в рассматриваемом здесь проекте `Empty Window` расширение класса находится в начале файла `ViewController.m`. Можете убедиться в этом сами!

Каким же должно быть "наполнение" расширения класса для того, чтобы сделать его настолько полезным, чтобы оно оказалось в файле шаблона? Прежде всего следует отметить, что расширение класса служило раньше в качестве стандартного решения проблемы, связанной с порядком определения методов.

До внедрения усовершенствований в компилятор LLVM языка Objective-C в версии 3.1 (Xcode 4.3) один метод из раздела реализации не мог вызвать другой метод из того же самого раздела, если только он не был определен или объявлен раньше другого метода. Упорядочить определение методов не так-то просто, и поэтому вполне очевидным выходом из этого затруднительного положения является объявление метода, которое можно сделать только в разделе интерфейса. В то же время для того чтобы разместить объявление метода в разделе интерфейса, придется перейти к другому, заголовочному файлу, что не совсем удобно, а еще хуже — метод становится в итоге открытым и доступным для вызова из любого класса, в котором импортируется этот заголовочный файл. В этом нет ничего страшного, если данный метод предполагается сделать открытым, но что, если требуется оставить его закрытым? Решение этого вопроса состоит в размещении расширения класса в начале его файла реализации. Если разместить объявления методов в этом расширении класса, то их объявления будут доступны всем методам из раздела реализации, а следовательно, оно смогут вызывать друг друга, хотя и не будут доступны для другого класса.

Впрочем, сейчас такой специальный прием уже не нужен, поскольку одни методы (и функции) в реализации класса могут быть доступны для вызова другим методам независимо от порядка их определения. В современной версии языка Objective-C расширения класса могут оказаться полезными, главным образом, для объявлений свойств, о чем речь пойдет в главе 12. Подобно объявлениям методов, объявления свойств должны находиться в разделе интерфейса, и хотя некоторые свойства предназначаются быть открытыми, нередко определенную их часть предпочтительнее оставить закрытыми для класса. Они остаются глобально доступными для всех методов этого класса и удобны для хранения значений, доступных многим методам, но только не за пределами данного класса. Следовательно, закрытые свойства целесообразно объявить в расширении класса.

Например, в главе 7 свойство `IBOutlet` было объявлено в заголовочном файле класса. Однако на практике такое свойство, скорее всего, будет объявлено в расширении класса, находящемся в файле реализации. Выходы класса обычно не имеют другого назначения в классе. Еще одно применение расширений классов рассматривается в следующем разделе.

Протоколы

Во всяком достаточно развитом объектно-ориентированном языке программирования должен учитываться тот факт, что иерархии подклассов и суперклассов явно недостаточно для выражения требующихся отношений между классами. Например, объекты классов `Bee` и `Bird` должны иметь некоторые общие свойства в силу того, что пчела и птица могут летать.

В то же время класс Bee может наследовать от класса Insect, хотя и не всякое насекомое способно летать. Так как же объекту типа Bee приобрести аспекты объекта типа Flier совершенно независимо от того, как их приобретает объект типа Bird? В некоторых объектно-ориентированных языках программирования этот вопрос разрешается с помощью *подмешанных* классов. Например, в языке Ruby можно определить модуль Flier, наполнить его определениями методов и затем внедрить в оба класса, Bee и Bird. В языке Objective-C принят более простой, облегченный подход, называемый *протоколом*. В среде Cocoa протоколы нашли широкое применение.

Протокол — это всего лишь именованный список объявлений методов без всякой реализации. В классе можно формально объявить, что он соответствует протоколу или принимает его, и такое соответствие наследуется подклассами. Подобное объявление удовлетворяет требованиям компилятора при попытке отправить соответствующее сообщение. Так, если в протоколе объявлен метод экземпляра myCoolMethod, а в классе MyClass — соответствие данному протоколу, то экземпляру класса MyClass можно отправить сообщение myCoolMethod, не вызвав никаких возражений со стороны компилятора.

Фактически реализация методов, объявленных в протоколе, возлагается на соответствующий ему класс. Протокольный метод может быть обязательным или необязательным. Если протокольный метод обязательный, а класс соответствует данному протоколу, то компилятор выдаст предупреждение, если класс не выполнит своих обязательств реализовать этот метод. С другой стороны, реализовывать необязательные методы совсем не обязательно. (Разумеется, это справедливо лишь с точки зрения компилятора. Так, если во время выполнения сообщение посылается объекту без реализации соответствующего метода, то в конечном итоге возникает аварийная ситуация, как пояснялось в главе 3.)

Рассмотрим на конкретном примере, каким образом протоколы применяются в среде Cocoa. Одни объекты можно копировать, а другие — нельзя, но это не имеет никакого отношения к наследованию классов этих объектов. Тем не менее хотелось бы иметь единообразный метод, на который мог бы реагировать любой копируемый объект. С этой целью в среде Cocoa определяется протокол NSCopying, в котором объявляется единственный (обязательный) метод copyWithZone:. Класс, явно соответствующий протоколу NSCopying, обязуется реализовать метод copyWithZone:. Ниже показано, каким образом протокол NSCopying определяется в файле NSObject.h, где его можно обнаружить в прикладном коде.

```
@protocol NSCopying
- (id)copyWithZone:(NSZone *)zone;
@end
```

Это все, что требуется для определения протокола. В этом определении используется директива компилятора @protocol, в которой устанавливается название протокола. Оно состоит полностью из определений методов и завершается директивой компилятора @end.

Как правило, определение протокола находится в заголовочном файле, чтобы его можно было импортировать в классах, которым требуется знать об этом протоколе, чтобы принять его или вызвать объявленные в нем методы. Раздел протокола располагается в заголовочном файле отдельно, но не в любом другом разделе, как, например, раздел интерфейса.

Объявления необязательных методов в определении протокола должны следовать после директивы @optional. В определении протокола может быть установлено, что в нем внедряются другие протоколы. Такие протоколы указываются списком через запятую и в угловых скобках после имени основного протокола, как показано в приведенном ниже примере кода из файла UIAlertView.h, принадлежащего компании Apple.

```
@protocol UIAlertViewDelegate <NSObject>
@optional
- (void)alertView:(UIAlertView *)alertView
    clickedButtonAtIndex:(NSInteger)buttonIndex;
// ... объявления других необязательных методов ...
@end
```

Определение протокола NSCopying в файле NSObject.h является всего лишь определением. Оно не устанавливает соответствие класса NSObject протоколу NSCopying. На самом деле класс NSObject *не* соответствует протоколу NSCopying! Для того чтобы убедиться в этом, попробуйте отправить метод copyWithZone: в качестве сообщения своему подклассу, производному от класса NSObject, как показано ниже.

```
MyClass* mc = [MyClass new];
MyClass* mc2 = [mc copyWithZone: nil];
```

При действующем механизме ARC этот фрагмент кода не будет скомпилирован, поскольку ни одна из реализаций метода copyWithZone: не унаследована. Для того чтобы класс формально соответствовал протоколу, имя протокола должно быть указано в угловых скобках после имени суперкласса (или после круглых скобок, если это объявление категории) в разделе интерфейса заголовочного файла данного класса. Это неизбежно повлечет за собой импорт заголовочного файла, в котором объявляется протокол (или другого заголовочного файла, откуда импортируется данный заголовочный файл). Для того чтобы установить соответствие класса нескольким протоколам, их следует указать через запятую в угловых скобках.

Теперь посмотрим, что произойдет, если установить формальное соответствие класса протоколу NSCopying. С этой целью попробуйте внести следующие изменения в первую строку из раздела интерфейса заголовочного файла своего класса:

```
@interface MyClass : NSObject <NSCopying>
```

Теперь ваш код будет скомпилирован, но компилятор предупредит вас, что в классе отсутствует реализация метода copyWithZone:, что он обязывался сделать, поскольку метод copyWithZone: является обязательным для протокола NSCopying.

Имя протокола можно также использовать при указании типа объекта. Чаще всего объект обозначается как id, но при сопутствующем условии, что он соответствует протоколу, имя которого указывается в угловых скобках. Для того чтобы продемонстрировать это положение, рассмотрим еще один типичный пример применения протоколов в среде Cocoa, связанный непосредственно с табличным представлением (UITableView). У класса UITableView имеется свойство dataSource, объявленное следующим образом:

```
@property (nonatomic, assign) id<UITableViewDataSource> dataSource
```

Это свойство представляет переменную экземпляра типа id <UITableViewDataSource>. Это означает, что источник данных должен соответствовать протоколу UITableView DataSource, какому бы классу он ни принадлежал. Такое соответствие накладывает обязательство на источник данных реализовать хотя бы обязательные методы экземпляра tableView:numberOfRowsInSection: и tableView:cellForRowAtIndexPath:, которые будут вызываться из табличного представления, когда ему нужно будет знать, какие именно данные следует отображать.

Если вы попытаетесь установить в свойстве dataSource табличного представления объект, не соответствующий протоколу UITableViewDataSource, то получите предупреждение от компилятора, как показано в приведенном ниже примере кода.

```
MyClass* mc = [MyClass new];
UITableView* tv = [UITableView new];
tv.dataSource = mc; // компилятор выдаст предупреждение
```

При действующем механизме ARC это предупреждение выражено в довольно запутанных терминах следующим образом: "Assigning to 'id<UITableViewDataSource>' from in compatible type 'MyClass *__strong' (Присваивание 'id<UITableViewDataSource>' из несовместимого типа 'MyClass *__strong').

Для того чтобы компилятор не выдавал подобные предупреждения, в объявлении класса `MyClass` должно быть установлено соответствие протоколу `UITableViewDataSource`. Как только это будет сделано, объект класса `MyClass` приобретет идентификатор `id <UITableViewDataSource>`, а при компиляции третьей строки приведенного выше фрагмента кода предупреждение уже не возникнет. Разумеется, в классе `MyClass` нужно также предоставить реализации методов `tableView:numberOfRowsInSection:` и `tableView:cellForRowAtIndexPath:`, чтобы избежать появления другого предупреждения о том, что в данном классе не реализован обязательный метод протокола, соответствовать которому он обязался.

В довольно большом числе случаев объект, который требуется присвоить и от которого ожидается соответствие протоколу, определяется как `self`. В подобных случаях такое соответствие класса протоколу можно объявить в файле реализации (с расширением `.m`) как часть расширения класса, аналогично приведенному ниже примеру.

```
// MyClass.m:
@interface MyClass () <UITableViewDataSource>
@end

@implementation MyClass
- (void) someMethod {
   UITableView* tv = [UITableView new];
   tv.dataSource = self; }
@end
```

Я предпочитаю именно такой порядок, поскольку он означает, что объявление соответствия протоколу находится в том же файле, где применяется этот протокол. Преобладающая в среде Cocoa тенденция применять протоколы связана с объектами делегатов (и разумеется, в первую очередь с реализацией механизма делегирования, который вам, скорее всего, придется определять в своих протоколах). Подробнее о делегатах речь пойдет в главе 11, но и теперь вы можете заметить, что у многих классов имеется свойство `delegate` и что класс этого свойства зачастую обозначается идентификатором `id <SomeProtocol>`. Например, в шаблоне проекта `Empty Window` предоставляется класс `AppDelegate`, объявленный следующим образом:

```
@interface AppDelegate : UIResponder <UIApplicationDelegate>
```

Дело в том, что назначение класса `AppDelegate` — выполнять функции общего для приложения делегата. Общим для приложения является объект типа `UIApplication`, а его свойство `delegate` обозначается как `id <UIApplicationDelegate>`. Таким образом, класс `AppDelegate` объявляет свои функции установлением явного соответствия протоколу `UIApplicationDelegate`.

В заголовочном файле, содержащем раздел с определением протокола и раздел интерфейса класса, возникает затруднение, где трудно отделить причину от следствия, поскольку и то и другое упоминается в каждом из этих разделов. Очевидно, что раздел интерфейса не может быть первым, поскольку в нем упоминается протокол перед его определением. С другой

стороны, раздел с определением протокола не может быть первым потому, что в нем упоминается класс до его определения. Как правило, подобное затруднение разрешается следующим образом: первым следует раздел интерфейса класса, но ему предшествует *предваряющее объявление* протокола в одной строке, где указывается только имя протокола, который определяется далее, т.е. через три строки кода, как показано ниже.

```
@protocol MyProtocol;
@interface MyClass : NSObject
@property (nonatomic, weak)id<MyProtocol> delegate;
@end

@protocol MyProtocol
- (void) doSomething: (MyClass*) m;
@end
```

Для вас как программиста применение протоколов в среде Cocoa имеет значение в двух отношениях.

Соблюдение соответствия

Если значение объекта, которое требуется присвоить в качестве аргумента, обозначается как `id <SomeProtocol>`, вы должны непременно обеспечить соответствие класса этого объекта протоколу `SomeProtocol` (и реализацию в нем обязательных методов данного протокола).

Пользование документацией

У протокола имеется своя страница в документации, где поясняется, что свойство `delegate` обозначается как `id <UIApplicationDelegate>`, но в то же время подразумевается, что если требуется выяснить, какие именно сообщения может получать делегат класса `UIApplication`, то для этого придется обратиться за справкой к документации на протокол `UIApplicationDelegate`.

Аналогично, если в документации на класс упоминается, что класс соответствует протоколу, не забудьте изучить документацию на протокол, поскольку она может содержать важные сведения о поведении класса. Для того чтобы выяснить, какие именно сообщения могут отправляться объекту, как упоминалось в главе 8, вам придется проследить вверх цепочку наследования его класса вплоть до суперкласса, а также найти любые протоколы, которым соответствует класс (или суперкласс) этого объекта.

Неформальные протоколы

В Интернете или в документации можно периодически встретить обозначение *неформальный протокол*. Такой протокол на самом деле вообще не является протоколом. Это лишь способ предоставить компилятору сигнатуру метода, чтобы отправлять сообщения без всяких возражений со стороны компилятора.

Неформальный протокол можно реализовать двумя взаимодополняющими способами. Один из них состоит в том, чтобы определить категорию в классе `NSObject`. Благодаря этому любой объект имеет право получать сообщения от методов, перечисленных в категории. Другой способ состоит в том, чтобы определить протокол, которому формально не соответствует ни один из классов. В этом случае любое сообщение, посылаемое от любых методов, перечисленных в протоколе, достигнет только тех объектов, которые обозначаются как `id`, и благодаря этому устраняются любые возражения со стороны компилятора.

Неформальные протоколы широко применялись, прежде чем в протоколах появилась возможность объявлять методы как необязательные, но теперь особой необходимости в них нет. (Хотя такие протоколы по-прежнему применяются, это делается все реже. Так, в версии iOS 7 уже осталось очень мало неформальных протоколов.) Кроме того, неформальные протоколы несколько опасны, поскольку они позволяют неумышленно определить метод с тем же именем, что и у существующего метода, но с другой сигнатурой и непредсказуемыми последствиями.

Необязательные методы

В протоколе можно явным образом обозначить некоторые или все методы как необязательные. Какой же практический толк от необязательного метода? Как известно, если объекту посылается сообщение, а объект не в состоянии его обработать, то возникает исключение и приложение, скорее всего, завершается аварийно. В то же время ведь объявление метода — это своего рода соглашение, предполагающее возможность обработки сообщения, посылаемого объекту. Если же нарушить это соглашение, объявив метод, который может быть и не реализован, то не послужит ли это побудительной причиной для аварийных отказов?

Ответ на этот вопрос состоит в том, что Objective-C — не только динамический, но и интроспективный язык программирования. Объекту можно дать команду обрабатывать сообщение, фактически не посылая его. Главная роль в этом принадлежит методу `respondsToSelector:` из класса `NSObject`, который принимает селектор в качестве своего параметра и возвращает логическое значение `BOOL`. С помощью этого метода сообщение можно отправить объекту только в том случае, если это будет безопасно, как показано ниже.

```
MyClass* mc = [MyClass new];
if ([mc respondsToSelector:@selector(woohoo)]) {
    [mc woohoo];
}
```

Это вряд ли стоит делать перед отправкой любого прежнего сообщения, поскольку в этом нет никакой необходимости, если только речь не идет о необязательных методах. К тому же это несколько замедляет работу приложения. На самом деле вызов метода `respondsToSelector:` для объектов в приложении считается в среде Cocoa чем-то само собой разумеющимся. Для того чтобы убедиться в этом, реализуйте метод `respondsToSelector:` в классе `AppDelegate` из проекта `Empty Window`, снабдив его средствам регистрации, как показано ниже.

```
- (BOOL) respondsToSelector: (SEL) sel {
     NSLog(@"%@", NSStringFromSelector(sel));
     return [super respondsToSelector:(sel)];
}
```

После запуска приложения `Empty Window` на своем компьютере я получил следующий результат (в нем опущены закрытые методы и многократные вызовы одного и того же метода).

```
application:handleOpenURL:
application:openURL:sourceApplication:annotation:
applicationDidReceiveMemoryWarning: applicationWillTerminate:
applicationSignificantTimeChange:
application:willChangeStatusBarOrientation:duration:
application:didChangeStatusBarOrientation:
application:willChangeStatusBarFrame:
application:didChangeStatusBarFrame:
application:deviceAccelerated:
application:deviceChangedOrientation:
```

```
applicationDidBecomeActive: applicationWillResignActive:
applicationDidEnterBackground: applicationWillEnterForeground:
applicationWillSuspend: application:didResumeWithOptions:
application:shouldSaveApplicationState:
application:supportedInterfaceOrientationsForWindow:
application:performFetchWithCompletionHandler:
application:didReceiveRemoteNotification:fetchCompletionHandler:
application:willFinishLaunchingWithOptions:
application:didFinishLaunchingWithOptions:
```

Таким образом, в среде Cocoa проверяется, какие необязательные (в том числе и недокументированные) методы протокола `UIApplicationDelegate` фактически реализованы в экземпляре класса `AppDelegate`, который явно согласился реагировать на любые сообщения от этих методов, поскольку это делегат объекта типа `UIApplication`, а следовательно, он формально соответствует протоколу `UIApplicationDelegate`. Шаблон делегата полностью полагается на данный способ (см. главу 11). Обратите внимание на правила, которым здесь следует среда Cocoa. Как только ей встретится рассматриваемый объект, в ней проверяются сразу все необязательные методы протокола и, предположительно, сохраняются результаты. Такой первоначальный единовременно совершаемый "обстрел" проверяемого объекта вызовами метода `respondsToSelector:` приводит к некоторому замедлению работы приложения, но, получив однажды ответы на все интересующие вопросы, среда Cocoa больше не повторяет в дальнейшем проверки того же самого объекта.

Некоторые классы из каркаса Foundation среды Cocoa

Классы из каркаса Foundation среды Cocoa предоставляют основные типы данных и служебные средства, образующие основание для большей части того, что приходится делать в среде Cocoa. Очевидно, что перечислить все эти классы, а тем более полностью описать их, невозможно. В то же время можно дать краткий обзор некоторых наиболее употребительных классов, с которым вам следует ознакомиться, прежде чем приступать к написанию самой простой программы в среде Cocoa. Более подробные сведения о классах из каркаса Foundation, начиная с их перечня, приведены в документации *Foundation Framework Reference*, предоставляемой компанией Apple.

Полезные структуры и константы

Структура `NSRange` играет очень важную роль в обращении с некоторыми рассматриваемыми здесь классами. Она состоит из целочисленных (`NSUInteger`) компонентов: переменных `location` и `length`. Например, диапазон, местоположение которого определяется значением `1` переменной `location`, начинается со второго элемента какого-нибудь объекта, поскольку отсчет элементов всегда начинается с нуля. Если его длина, определяемая переменной `length`, равна `2`, то она обозначает текущий и следующий элементы. Для обращения с диапазоном в среде Cocoa предоставляются также различные служебные методы, среди которых чаще всего применяется метод `NSMakeRange`. (Следует иметь в виду, что имя `NSMakeRange` этого метода обратно сравнимо с такими именами, как `CGPointMake` и `CGRectMake`.)

Целочисленная константа `NSNotFound` указывает на то, что некоторый запрашиваемый элемент не найден. Так, если запросить индекс некоторого объекта в массиве типа `NSArray`, где этот объект отсутствует, то в результате будет возвращена константа `NSNotFound`. (Этот результат не может быть равен `0`, чтобы указывать на отсутствие объектов, поскольку `0` будет обозначать первый элемент массива. Он не может быть равен `nil`, поскольку `nil = 0` и в

любом случае не подходит, если ожидается целочисленное значение. Не может он быть равен и -1, потому что значение индекса массива должно быть всегда положительным.) Истинное числовое значение константы NSNotFound особого значения не имеет, поскольку сравнение всегда производится с самой константой NSNotFound, чтобы выяснить, содержит ли результат запроса значащий индекс. Если же в результате поиска по запросу возвращается диапазон, а искомый объект в нем отсутствует, то переменная location в результирующей структуре NSRange будет равна константе NSNotFound.

Класс NSString со товарищи

Класс NSString уже не раз довольно свободно использовался в примерах, приведенных ранее в этой книге. Он является объектным вариантом представления символьной строки в среде Cocoa. Создать объект класса NSString можно с помощью целого ряда методов и инициализаторов этого класса или же с помощью буквенного обозначения @"...", которое на самом деле является директивой компилятора. Особо важная роль принадлежит методу stringWithFormat:, который позволяет преобразовывать числа в символьные строки и объединять последние. Подробнее об этом см. в главе 9, где рассматриваются форматы символьных строк в связи с классом NSLog. Ниже приведены некоторые примеры применения символьных строк.

```
int x = 5;
NSString* s = @"widgets";
NSString* s2 = [NSString stringWithFormat:@"You have %d %@.", x, s];
```

В классе NSString реализован современный, основанный на уникоде подход к содержимому символьной строки. Элементами такой строки являются символы, количество которых определяется свойством length, обозначающим длину строки. Символы строки не представлены байтами, поскольку числовое представление символа в уникоде может состоять из нескольких байтов в зависимости от конкретной кодировки. Они не являются также символическими знаками (так называемыми *глифами*), потому что последовательность составных символов, печатаемых как одна буква, может состоять из нескольких символов. Таким образом, длина диапазона типа NSRange, обозначающего один символ, может быть больше 1. Подробнее об этом можно узнать из главы "Characters and Grapheme Clusters" документации *String Programming Guide*, предоставляемой компанией Apple.

Поиск объектов типа NSString можно организовать с помощью различных методов типа rangeOf..., возвращающих структуру NSRange. Кроме того, класс NSScanner позволяет просмотреть всю символьную строку в поисках отдельных частей по определенному критерию. Например, с помощью класса NSScanner (а также класса NSCharacterSet) можно пропустить в символьной строке все, что предшествует числу, а затем извлечь это число. Семейство методов поиска rangeOfString: дает возможность находить подстроки. С помощью параметра NSRegularExpressionSearch можно организовать поиск, используя регулярное выражение. Регулярные выражения поддерживаются также в виде отдельного класса NSRegularExpression, в котором объект типа NSTextCheckingResult служит для описания совпавших результатов.

Ниже приведен пример кода, взятого из одного из моих приложений, где пользователь нажатием пальцем выбирает кнопку с надписью вроде 5 by 4 или 4 by 3. В данном примере нужно выяснить оба числа. Одно из них обозначает количество строк, а другое — количество столбцов, которое должно быть в компоновке. Для обнаружения обоих чисел в надписи используется класс NSScanner.

```
NSString* s = // надпись на кнопке, например, @"4 by 3"
NSScanner* sc = [NSScanner scannerWithString:s];
int rows, cols;
[sc scanInt:&rows];
[sc scanUpToCharactersFromSet:[decimalDigitCharacterSet]
                    intoString:nil];
[sc scanInt:&cols];
```

После выполнения этого кода требующиеся числа сохраняются в переменных `rows` и `cols`. Ниже показано, как то же самое можно сделать с помощью регулярного выражения.

```
NSString* s = // надпись на кнопке, например, @"4 by 3"
int rowcol[2];
int* prowcol = rowcol;
NSError* err = nil;
NSRegularExpression* r =
    [NSRegularExpression regularExpressionWithPattern:@"\\d"
                                              options:0
                                                error:&err];
// проверка ошибок опущена
for (NSTextCheckingResult* match in
    [r matchesInString:s options:0 range:NSMakeRange(0, [s length])])
        *prowcol++ = [[s substringWithRange: [match range]] intValue];
```

После выполнения этого кода требующиеся числа сохраняются в элементах массива `rowcol[0]` и `rowcol[1]`. В данном примере синтаксис выглядит слишком громоздким, поскольку каждое совпадение с регулярным выражением приходится преобразовывать сначала из объекта типа `NSTextCheckingResult` в диапазон, а затем в подстроку исходной символьной строки и, наконец, в целое значение.

Для поддержки более сложного анализа текста имеется ряд дополнительных классов, в том числе класс `NSDataDetector`, производный от класса `NSRegularExpression` и эффективно находящий определенные типы строковых выражений, например, URL или номер телефона, а также класс `NSLinguisticTagger`, фактически предпринимающий попытку проанализировать текст, разбивая его на грамматические части речи.

Символьная строка, представленная объектом типа `NSString`, считается неизменяемой. С помощью одной символьной строки можно сформировать разными способами другую строку, присоединив, например, к ней другую строку или выделив подстроку, но нельзя изменить саму символьную строку. Для этой цели потребуется подкласс `NSMutableString`, производный от класса `NSString`.

В классе `NSString` имеются служебные средства для работы с символьной строкой, содержащей путь к файлу. Этот класс нередко применяется вместе с `NSURL` — еще одним, также заслуживающим внимания классом из каркаса Foundation. В классе `NSString` и ряде других классов, рассматриваемых в этом разделе, предоставляются методы для ввода-вывода содержимого символьных строк в файл. При вызове этих методов файл может быть указан по пути в виде объекта типа `NSString` или URL в виде объекта типа `NSURL`.

Класс `NSString` не несет никакой информации о типе и размере шрифта. У интерфейсных объектов, отображающих символьные строки (например, класса `UILabel`), имеется свойство `font`, содержащее объект класса `UIFont`, но это свойство определяет только *один* тип и размер шрифта, которым будет отображаться символьная строка. До появления версии iOS 6 отображение оформленного стилями текста, разные фрагменты которого имели отличающие стилевые атрибуты (размер, тип, цвет шрифта и т.д.), было непростым делом. В классе `NSAttributedString`, представлявшем символьную строку со стилевым оформлением, требовалось использовать каркас Core Text и компоновать оформленный стилями текст,

воспроизводя его вручную, поскольку такой текст нельзя было отобразить в любом стандартном интерфейсом объекте. Начиная с версии iOS 6, класс NSAttributedString стал полноценным классом языка Objective-C. Теперь у него имеются методы и вспомогательные классы, позволяющие легко создать самое изощренное стилевое оформление строк и целых абзацев текста, а встроенные интерфейсные объекты, предназначенные для воспроизведения текста, способны отображать оформленный стилями текст.

Воспроизведение символьных строк в графическом контексте может быть выполнено методами, предоставляемыми в категории NSStringDrawing из класса NSString (см. документацию NSString *UIKit Additions Reference*), а также в классе NSAttributedString (см. документацию NSAttributedString *UIKit Additions Reference*).

Класс NSDate со товарищи

Класс NSDate автоматически представляет дату и время в виде количества секунд (типа NSTimeInterval), отсчитываемых относительно некоторой исходной даты. В результате вызова [NSDate new] или [NSDate date] получается объект даты для текущих даты и времени. Для выполнения других операций над датами могут потребоваться классы NSDateComponents и NSCalendar, хотя сделать это будет не так-то просто в силу сложной организации календарей (см. документацию *Date and Time Programming Guide*). Ниже приведен пример составления даты по ее календарным значениям.

```
NSCalendar* greg =
    [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
NSDateComponents* comp = [NSDateComponents new];
comp.year = 2013;
comp.month = 8;
comp.day = 10;
comp.hour = 15;
NSDate* d = [greg dateFromComponents:comp];
```

Аналогично, для выполнения арифметических операций над датами служит класс NSDateComponents. Ниже показано, как добавить один день к текущей дате:

```
NSDate* d = // любая дата
NSDateComponents* comp = [NSDateComponents new];
comp.day = 1;
NSCalendar* greg =
    [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
NSDate* d2 = [greg dateByAddingComponents:comp toDate:d options:0];
```

Не менее важно представление дат в виде символьных строк. Для создания и синтаксического анализа символьных строк с датами служит класс NSDateFormatter, использующий строковый формат, аналогичный возвращаемому методом stringWithFormat: из класса NSString. Дело усложняется тем, что точное строковое представление составляющей или формата даты может зависеть от пользовательских региональных настроек, включая местный язык, формат и календарь. (На самом деле региональные настройки следует учитывать и при форматировании символьных строк средствами класса NSString.)

В следующем примере, взятом из одного из моих приложений, подготавливается содержимое метки типа UILabel, чтобы сообщить в ней дату и время последнего обновления данных. Это приложение не локализовано. В частности, слово "at" (в) появляется в символьной строке не переведенным с английского языка, и поэтому требуется также проконтролировать полностью представление составляющих даты и времени. Для этого придется опереться на конкретные региональные настройки.

```
NSDateFormatter *df = [NSDateFormatter new];
if ([[NSLocale availableLocaleIdentifiers] indexOfObject:@"en_US"]
        != NSNotFound) {
    NSLocale* loc = [[NSLocale alloc] initWithLocaleIdentifier:@"en_US"];
    [df setLocale:loc]; // английские, по возможности, названия месяца
                        // и часового пояса
}
[df setDateFormat:@"'Updated' d MMMM yyyy 'at' h:mm a z"];
NSString* updatedString = [df stringFromDate: [NSDate date]]; // текущая дата
```

С другой стороны, опираясь на пользовательские региональные настройки, можно отформатировать дату с помощью метода `dateFormatFromTemplate:options:locale:` из класса `NSDateFormatter` и текущих региональных настроек. В качестве "шаблона" служит символьная строка, в которой перечисляются используемые составляющие даты, но их порядок следования, знаки препинания и представление на местном языке отдаются полностью на откуп региональным настройкам, как показано ниже.

```
NSDateFormatter *df = [NSDateFormatter new];
NSString* format =
    [NSDateFormatter dateFormatFromTemplate:@"dMMMMyyyyhmmaz"
        options:0 locale:[NSLocale currentLocale]];
[df setDateFormat:format];
NSString* updatedString = [df stringFromDate: [NSDate date]]; // текущая дата
```

На мобильном устройстве, где установлены французские региональные настройки, например, по команде Settings⇨General⇨International⇨Region Format, результат может выглядеть следующим образом: `20 juillet 2013 5:14 PM UTC-7`. Обратите внимание на то, что местный язык региональных настроек не является системным, т.е. результат представлен на французском языке вследствие установки региональных настроек, а не выбора языка на мобильном устройстве. За более подробными сведениями о региональных настройках в целом обращайтесь к документации вашего браузера на библиотеки ICU (International Components for Unicode — Международные компоненты для уникода), откуда происходит поддержка в iOS создания и синтаксического анализа символьных строк данных. Для того чтобы выяснить, какие региональные настройки существуют, обращайтесь по адресу `http://demo.icu-project.org/icu-bin/locexp`.

Начинающие программировать под iOS часто совершают ошибку, забывая о том, что дата содержит часовой пояс. Например, при регистрации объекта типа `NSDate` средствами класса `NSLog` время (и день) может быть интерпретировано неверно, поскольку значение даты представлено в часовом поясе по Гринвичу. Во избежание подобной ошибки следует вызвать метод `descriptionWithLocale:`, предоставляющий требуемые (обычно текущие) региональные настройки, или же воспользоваться средством форматирования даты.

Класс NSNumber

Объект класса `NSNumber` заключает в себе числовое значение, в том числе и логическое значение `BOOL`. Следовательно, его можно использовать для хранения и передачи числа там, где предполагается получить объект. Объект класса `NSNumber` образуется из конкретного числа с помощью метода, обозначающего числовой тип. Например, для того чтобы образовать число из целочисленного значения `int`, можно вызвать метод `numberWithInt:` следующим образом:

```
[[NSUserDefaults standardUserDefaults] registerDefaults:
    [NSDictionary dictionaryWithObjectsAndKeys:
        [NSNumber numberWithInt: 4],
        @"cardMatrixRows",
```

```
        [NSNumber numberWithInt: 3],
        @"cardMatrixColumns", nil]];
```

Как упоминалось в главе 5, в компиляторе LLVM версии 4.0 (в среде Xcode 4.4) используется новый синтаксис для получения нового экземпляра класса NSNumber. Для этого нужно соблюсти следующие условия.

- Предварить числовой литерал (или логическое значение BOOL) знаком @. Для дополнительного обозначения числового типа после числового литерала следует указать U (целое число без знака), L (целое длинное число), LL (целое число двойной длины) или F (число с плавающей точкой). Например, числовое значение @3.1415 равнозначно значению [NSNumber numberWithDouble:3.1415], а логическое значение @YES — значению [NSNumber numberWithBool:YES].
- Если выражение дает числовое значение, заключить его в круглые скобки, предварив открывающую скобку знаком @. Так, если высота и ширина представлены числами с плавающей точкой, то выражение @(height/width) равнозначно [NSNumber numberWithFloat: height/width].

Таким образом, приведенный выше пример кода можно переписать следующим образом:

```
[[NSUserDefaults standardUserDefaults] registerDefaults:
    [NSDictionary dictionaryWithObjectsAndKeys:
        @4,
        @"cardMatrixRows",
        @3,
        @"cardMatrixColumns",
        nil]];
```

(Синтаксис литералов поддерживается также в классе NSDictionary. Как будет показано далее, приведенный выше код можно написать еще компактнее.)

Сам объект типа NSNumber не является числом, и поэтому его нельзя использовать в вычислениях или там, где предполагается получить конкретное число. Вместо этого приходится извлекать число явным образом из оболочки объекта типа NSNumber, используя метод, выполняющий действие, обратное заключению числа в оболочку данного объекта. Выбор этого метода оставляется на ваше усмотрение. Так, если объект типа NSNumber заключает в оболочку числовое значение типа int, можно вызвать метод intValue, чтобы извлечь это значение, как показано ниже.

```
NSUserDefaults* ud = [NSUserDefaults standardUserDefaults];
int therows = [[ud objectForKey:@"cardMatrixRows"] intValue];
int thecols = [[ud objectForKey:@"cardMatrixColumns"] intValue];
```

На самом деле это настолько распространенное преобразование при обращении к классу NSUserDefaults, что в нем для этой цели предоставляются служебные методы. Следовательно, тот же самый код можно написать следующим образом:

```
NSUserDefaults* ud = [NSUserDefaults standardUserDefaults];
int therows = [ud integerForKey:@"cardMatrixRows"];
int thecols = [ud integerForKey:@"cardMatrixColumns"];
```

С другой стороны, подкласс NSDecimalNumber, производный от класса NSNumber, может быть использован в вычислениях благодаря целому ряду имеющихся в нем арифметических методов (или эквивалентных им функциям C, которые действуют быстрее). Это особенно полезно для округления чисел, поскольку имеется удобный способ указать требующийся режим округления.

Класс NSValue

Класс NSValue является суперклассом для класса NSNumber. Он служит для заключения в оболочку нечисловых значений языка С вроде структур там, где предполагается получить объект, например, для сохранения в массиве типа NSArray или для доступа к значениям по ключам.

Служебные методы, предоставляемые из категории NSValueUIGeometryExtensions в классе NSValue, позволяют легко заключать в оболочку и извлекать из нее значения типа CGPoint, CGSize, CGRect, CGAffineTransform, UIEdgeInsets и UIOffset (см. справочник NSValue *UIKit Additions Reference*). Дополнительные категории позволяют легко заключать в оболочку и извлекать из нее значения типа NSRange, CATransform3D, CMTime, CMTimeMapping, CMTimeRange, MKCoordinate и MKCoordinateSpan. Вряд ли вам придется хранить любые другие значения языка С в объектах класса NSValue, но ничто не помешает вам сделать это, если потребуется.

Класс NSData

Класс NSData представляет общую последовательность байтов и служит, по существу, в качестве буфера, занимающего часть оперативной памяти. Это неизменяемый класс, а его изменяемой версией служит подкласс NSMutableData. На практике класс NSData можно применять в двух случаях.

- При загрузке данных из Интернета. Например, классы NSURLConnection и NSURLSession предоставляют то, что они получают из Интернета, в виде объекта типа NSData. Этот объект можно затем преобразовать, например, в символьную строку и обратно, указав подходящую кодировку.
- При сохранении объекта в виде файла или пользовательских настроек. Например, значение типа UIColor нельзя сохранить непосредственно в пользовательских настройках. Следовательно, если пользователь выберет цвет и его нужно сохранить, то значение типа UIColor следует преобразовать в объект типа NSData (средствами класса NSKeyedArchiver) и затем сохранить его, как показано ниже.

```
[[NSUserDefaults standardUserDefaults] registerDefaults:
    [NSDictionary dictionaryWithObjectsAndKeys:
        [NSKeyedArchiver archivedDataWithRootObject:
        [UIColor blueColor]],
        @"myColor",
        nil]];
```

Равенство и сравнение

Упомянутые выше типы могут сразу показаться основными типами данных, но на самом деле они являются типами объектов, а это означает, что они являются указателями (см. главу 3). Следовательно, их нельзя сравнивать, используя операторы языка С для проверки на равенство, как это обычно делается с конкретными числами. Дело в том, что при сравнении типов объектов в операторах языка С, по существу, сравниваются указатели, а не содержимое экземпляров объектов, как показано в приведенном ниже примере.

```
NSString* s1 = [NSString stringWithFormat: @"%@, %@", @"Hello", @"world"];
NSString* s2 = [NSString stringWithFormat:@"%@, %@", @"Hello", @"world"];
if (s1 == s2) // ложно
    // ...
```

Две символьные строки равнозначны (@"Hello, world"), но не являются одинаковыми объектами. (Этот пример намеренно усложнен, поскольку эффективное управление строковыми литералами в среде Cocoa позволяет выявить, что обе символьные строки, непосредственно инициализированные значением @"Hello, world", являются одним и тем же объектом. Это не позволяет наглядно проиллюстрировать обсуждаемый здесь вопрос.)

Проверка на равенство должна быть реализована в отдельных классах. В частности, общая проверка на равенство с помощью метода isEqual: наследуется от класса NSObject и замещается, но в некоторых классах определяются также более конкретные и эффективные проверки. Таким образом, правильная проверка на равенство должна быть произведена следующим образом:

```
if ([s1 isEqualToString: s2])
```

Аналогично отдельные классы должны предоставлять методы для упорядоченного сравнения. Для такого сравнения вызывается стандартный метод compare:, который возвращает одну из следующих трех констант: NSOrderedAscending (получатель меньше аргумента), NSOrderedSame (получатель равен аргументу) или NSOrderedDescending (получатель больше аргумента); см. пример 3.2.

Класс NSIndexSet

Класс NSIndexSet представляет набор однозначных целых чисел и служит для обозначения номеров элементов упорядоченной коллекции вроде массива типа NSArray. Например, для одновременного извлечения нескольких объектов из массива достаточно указать требуемые индексы в виде объекта типа NSIndexSet. Этот класс служит и другим целям. В частности, объект типа NSIndexSet можно передать табличному представлению типа UITableView, чтобы указать, какие именно разделы следует ввести и какие из них удалить.

В качестве практического примера допустим, что требуется обратиться к элементам 1, 2, 3, 4, 8, 9 и 10 массива NSArray. Класс NSIndexSet позволяет выразить этот запрос в компактно реализованном и готовом к употреблению виде. Конкретная реализация достаточно прозрачна, но нетрудно представить, что в данном случае множество индексов может быть составлено из двух структур {1,4} и {8,3} типа NSRange, а методы класса NSIndexSet только приветствуют составление множества типа NSIndexSet из диапазонов.

Класс NSIndexSet является неизменяемым, тогда как изменяемым оказывается его подкласс NSMutableIndexSet. Простое множество типа NSIndexSet можно составить только из одного непрерывного диапазона непосредственно, передав структуру NSRange методу indexSetWithIndexesInRange:, а для составления более сложного множества индексов придется воспользоваться классом NSMutableIndexSet, чтобы присоединить дополнительные диапазоны. Для обхода (перечисления) значений индексов или диапазонов, указанных во множестве типа NSIndexSet, следует вызвать метод enumerateIndexesUsingBlock: или enumerateRangesUsingBlock: или же варианты этих методов.

Классы NSArray и NSMutableArray

Класс NSArray представляет упорядоченную коллекцию объектов. Свойство count определяет длину этой коллекции, а конкретный объект может быть извлечен из нее по номеру индекса с помощью метода objectAtIndex:. Индекс первого объекта в данной коллекции равен нулю, и поэтому индекс последнего ее элемента равен count - 1.

Начиная с версии 4.0 (Xcode 4.5) компилятора LLVM вызывать метод objectAtIndex: больше не нужно, а использовать вместо этого обозначение, напоминающее синтаксис языка C и других языков программирования, в которых определены массивы. Это обозначение

состоит в том, чтобы присоединить квадратные скобки, в которые заключен номер индекса, к ссылке на массив, т.е. осуществить *индексирование*.

Так, если массив pep состоит из элементов @"Manny", @"Moe" и @"Jack", то обозначение элемента массива pep[2] равнозначно обозначению [pep objectAtIndex:2], а точнее — обозначению [pep objectAtIndexedSubscript:2]. Дело в том, что такое обозначение индексирования приводит к тому, что любая ссылка, к которой присоединяется индекс, направляется методу objectAtIndexedSubscript:. Это, в свою очередь, означает, что любой класс, включая и ваши собственные, может реализовать метод objectAtIndexedSubscript: и тем самым стать пригодным для обозначения индексирования. Следует, однако, иметь в виду, что данный метод должен быть объявлен как открытый, чтобы компилятор разрешил такое обозначение индексирования.

Массив типа NSArray можно образовать по-разному, но, как правило, для этого предоставляется список объектов, которые он должен содержать. Как упоминалось в главе 3, синтаксис литералов, доступный с версии 4.0 (Xcode 4.4) компилятора LLVM, позволяет заключить этот список в выражение @[...]. Это еще один способ образовать массив типа NSArray, как показано ниже.

```
NSArray* pep = @[@"Manny", @"Moe", @"Jack"];
```

Класс NSArray является неизменяемым, но это совсем не означает, что объекты, содержащие в массиве типа NSArray, нельзя изменить. Напротив, это означает, что после того, как такой массив будет образован, из него уже нельзя удалить объект, ввести или заменить в нем объект по заданному индексу. Для выполнения подобных операций можно создать из текущего массива производный от него новый массив с исходным содержимым плюс или минус несколько объектов или же воспользоваться подклассом NSMutableArray, производным от класса NSArray.

Методам addObject: и replaceObjectAtIndex:withObject: из класса NSMutableArray предоставляется то же обозначение индексирования, которое применяется в классе NSArray. В данном случае индексированная ссылка является значением, употребляемым в левой части выражения присваивания:

```
pep[3] = @"Zelda";
```

Вследствие этого массив типа NSMutableArray направляется методу setObject: atIndexedSubscript:. В классе NSMutableArray это реализуется таким образом, чтобы выражение pep[3] = @"Zelda" было равнозначно вызову метода addObject:, присоединяющему объект в конце массива, если массив pep состоит из трех элементов. Если массив pep состоит из более трех элементов, то выражение pep[3] = @"Zelda" равнозначно вызову метода replaceObjectAtIndex:withObject:. Если массив pep состоит из менее трех элементов, то данное выражение приводит к исключению, возникающему в связи с выходом за пределы массива.

Организовать обход (перечисление) каждого объекта в массиве можно с помощью конструкции for...in, описанной в главе 1. (Если попытаться изменить содержимое массива типа NSMutableArray при перечислении его элементов, возникнет исключение.) Для поиска объекта в таком массиве можно вызвать метод indexOfObject: или indexOfObjectIdenticalTo:. Первый из них производит проверку на равенство, вызывая метод isEqual:, тогда как во втором на равенство сравниваются указатели.

Те, кто имеет опыт программирования на других языках, могут упустить из виду такие служебные функции обработки массивов, как, например, функция map(), которая составляет новый массив из результатов вызова метода для каждого объекта в текущем массиве. (Для вызова метода makeObjectsPerformSelector: требуется селектор, не возвращающий

значение, а для вызова метода enumerateObjectsUsingBlock: — блочная функция, также не возвращающая значение). В качестве выхода из этого положения можно создать пустой изменяемый массив и организовать перечисление исходного массива, вызывая соответствующий метод и присоединяя каждый раз результат к изменяемому массиву (см. ниже пример 10.1). Иногда вместо функции map() можно также воспользоваться механизмом доступа к значениям по ключам, как поясняется в главе 12.

Пример 10.1. Создание одного массива путем перечисления другого массива

```
NSMutableArray* marr = [NSMutableArray new];
for (id obj in myArray) {
    id result = [obj doSomething];
    [marr addObject: result];
}
```

Имеется немало способов организовать поиск или фильтрацию массива с помощью блока, как показано ниже.

```
NSArray* pep = @[@"Manny", @"Moe", @"Jack"];
NSArray* ems =
    [pep objectsAtIndexes:[pep indexesOfObjectsPassingTest:
       ^BOOL(id obj, NSUInteger idx, BOOL *stop) {
           return ([(NSString*)obj rangeOfString:@"m"
               options:NSCaseInsensitiveSearch].location == 0);
    }]];
```

Имеется также возможность получить отсортированный массив. Для этого можно предоставить разными способами правила сортировки или отсортировать массив непосредственно, если он изменяемый, как показано в примерах 3.1 и 3.2.

Формирование нового массива из некоторых или всех элементов существующего массива не является затратной операцией. Объекты, составляющие элементы первого массива, в этом случае не копируются, а новый массив состоит лишь из нового ряда указателей на уже существующие объекты. Это же справедливо и для других рассматриваемых далее типов коллекций.

Класс NSSet со товарищи

Класс NSSet представляет неупорядоченную коллекцию раздельных объектов. Слово "раздельный" здесь означает, что в коллекции не должно быть двух одинаковых объектов, при сравнении которых с помощью метода isEqual: последний возвращает логическое значение YES. Выяснить, присутствует ли объект во множестве, оказывается намного эффективнее, чем искать его в массиве. Кроме того, можно узнать, является одно множество подмножеством другого или пересекает его; обойти (перечислить) множество с помощью конструкции for...in, хотя порядок его обхода не определен; а также отфильтровать множество подобно массиву. Разумеется, большая часть операций над множеством сродни операциям над массивом, за исключением того, что с множеством, предполагающим обозначение его упорядочения, вряд ли можно что-нибудь сделать.

Для того чтобы преодолеть и это ограничение, можно воспользоваться упорядоченным множеством. Такое множество (типа NSOrderedSet) очень похоже на массив, а методы обращения с ним подобны методам обработки массива. Более того, в классе NSOrderedSet реализован метод objectAtIndexedSubscript:, позволяющий извлекать элементы из

множества посредством индексирования. В то же время элементы упорядоченного множества должны быть раздельными. Упорядоченное множество дает немало преимуществ, присущих множествам. Например, аналогично множеству типа NSSet, выяснить, присутствует ли объект в упорядоченном множестве, оказывается намного эффективнее, чем искать его в массиве. Кроме того, упорядоченные множества можно объединять, пересекать или вычитать одно из другого. Так как раздельность элементов множества зачастую, да и вообще не является ограничением, поскольку они все равно должны каким-то образом отличаться, то при всякой возможности лучше пользоваться множеством типа NSOrderedSet, чем множеством типа NSArray.

Превращение массива в упорядоченное множество делает его *однозначным*. Это означает, что он остается упорядоченным, но только первое вхождение одинакового объекта переносится во множество.

Класс NSSet является неизменяемым. Одно множество типа NSSet можно вывести из другого, введя или удалив элементы или же воспользовавшись подклассом NSMutableSet. Аналогично у класса NSOrderedSet имеется изменяемый эквивалент — класс NSMutable OrderedSet, в котором реализован метод setObject:atIndexedSubscript:. Операция ввода или вставки в изменяемое множество объекта, который уже в нем существует, не накладывает никаких дополнительных издержек, поскольку ничего и не вводится (в силу соблюдаемого правила раздельности элементов множества). Впрочем, это не приводит к ошибке.

Подкласс NSCountedSet является производным от класса NSMutableSet и представляет изменяемое неупорядоченное множество объектов, которые совсем *не* обязательно должны быть раздельными. Как правило, оно называется *множеством с повторяющимися элементами*. Оно реализуется как обычное множество с дополнительным подсчетом количество раз, когда в него был введен каждый элемент.

Классы NSDictionary и NSMutableDictionary

Класс NSDictionary представляет неупорядоченную коллекцию пар "ключ–значение", которые в некоторых языках программирования называются просто *хешем*. Ключ обычно, хотя и не обязательно, является объектом типа NSString. Значение может быть любым объектом. Класс NSDictionary является неизменяемым, тогда как его подкласс NSMutableDictionary — изменяемым.

Ключи словаря являются раздельными и сравниваются методом isEqual:. Если попытаться ввести пару "ключ–значение" в словарь типа NSMutableDictionary, то эта пара будет введена, при условии, что указанный ключ отсутствует в словаре. Если он уже присутствует в словаре, то соответствующее значение заменяется указанным.

Класс NSDictionary применяется в основном для запроса значения из статьи словаря по заданному ключу (с помощью метода objectForKey:). Если такой ключ отсутствует в словаре, то в результате получается значение nil, а следовательно, это еще один способ выяснить наличие ключа в словаре. Таким образом, словарь является простым и удобным средством для хранения данных и объектно-ориентированным аналогом структуры. Словари нередко применяются в среде Cocoa для предоставления дополнительного набора именованных значений. Примерами тому служат свойство userInfo класса NSNotification, параметр options: метода application:didFinishLaunchingWithOptions: и т.д.

Те же самые усовершенствования Objective-C, благодаря которым стали доступны для программирования литералы массивов и индексирование, сделали возможным применение литералов словарей и аналогичное индексирование. Словарь

можно сформировать не только из массива объектов и массива ключей (с помощью метода `dictionaryWithObjects:forKeys:`) или списка чередующихся объектов и ключей с завершающим значением `nil` (с помощью метода `dictionaryWithObjectsAndKeys:`), но и непосредственно в виде списка разделяемых запятой пар "ключ-значение", где после каждого ключа следует двоеточие и значение, а весь этот список заключается в выражение `@{...}`. Итак, вернемся к рассмотренному ранее примеру применения класса `NSUserDefaults` в приведенном ниже фрагменте кода.

```
[[NSUserDefaults standardUserDefaults] registerDefaults:
    [NSDictionary dictionaryWithObjectsAndKeys:
        @4,
        @"cardMatrixRows",
        @3,
        @"cardMatrixColumns",
        nil]];
```

Этот код можно переписать следующим образом:

```
[[NSUserDefaults standardUserDefaults] registerDefaults:
    @{@"cardMatrixRows":@4, @"cardMatrixColumns":@3}];
```

Для того чтобы извлечь значение из словаря по заданному ключу, вместо вызова метода `objectForKey:` теперь можно просто указать индекс ключа в квадратных скобках для ссылка на словарь следующим образом: `dict[key]`. Аналогично для ввода пары "ключ-значение" в словарь типа `NSMutableDictionary` вместо вызова метода `setObject:forKey:` эту пару можно просто присвоить ссылке на проиндексированный словарь. Как и в классе `NSArray`, все это делается подспудно с помощью вызываемых методов `objectForKeyedSubscript:` и `setObject:forKeyedSubscript:`. Эти методы достаточно объявить в своих классах, чтобы сделать последние пригодными для обозначения индексирования по ключу.

Такие структуры данных, как массив словарей, словарь словарей и т.д., весьма распространены и нередко лежат в основе функциональных возможностей приложения. Рассмотрим пример, взятый из одного из моих приложений. Пакет приложения содержит текстовый файл со следующим содержимым:

```
chapterNumber [tab] pictureName [return]
chapterNumber [tab] pictureName [return]
```

При запуске моего приложения этот текстовый файл загружается и подвергается синтаксическому анализу, в результате которого составляется словарь. Каждая статья этого словаря имеет следующую структуру:

```
key: (chapterNumber, as an NSNumber)
value: [Mutable Array]
    (pictureName)
    (pictureName)
    ...
```

В конечном итоге все иллюстрации к главе собираются под одним номером этой главы. Такая структура данных предназначена для того, чтобы значительно упростить и ускорить последующий доступ ко всем иллюстрациям к данной главе.

Ниже приведен конкретный код для синтаксического анализа текстового файла, в результате чего составляется подобная структура данных. Для каждой строки из текстового файла в словаре создается статья, если она не существует под номером данной главы, а в качестве значения служит пустой изменяемый массив. Как бы то ни было, статья теперь существует в словаре, а ее значением является изменяемый массив, к которому присоединяется название

иллюстрации. Обратите внимание, каким образом в данном простом примере совместно используются многие классы из каркаса Foundation, обсуждаемые в этом разделе.

```
NSString* f = [[NSBundle mainBundle] pathForResource:@"index" ofType:@"txt"];
NSError* err = nil;
NSString* s = [NSString stringWithContentsOfFile:f
                                        encoding:NSUTF8StringEncoding
                                           error:&err];
// проверка ошибок опущена
NSMutableDictionary* d = [NSMutableDictionary new];
for (NSString* line in [s componentsSeparatedByString:@"\n"]) {
    NSArray* items = [line componentsSeparatedByString:@"\t"];
    NSInteger chnum = [items[0] integerValue];
    NSNumber* key = @(chnum);
    NSMutableArray* marr = d[key];
    if (!marr) { // такой ключ отсутствует, создать пару "ключ-значение"
      marr = [NSMutableArray new];
      d[key] = marr;
    }
    // теперь массив marr является изменяемым, пустым или заполненным
    NSString* picname = items[1];
    [marr addObject: picname];
}
```

Из словаря типа `NSDictionary` можно получить список ключей, отсортированный список ключей или список значений. Обойти (перечислить) статьи словаря можно по его ключам, используя конструкцию `for...in`, хотя порядок такого обхода, разумеется, не определен. Кроме того, словарь предоставляет метод `objectEnumerator`, который можно использовать в конструкции `for...in` для обхода только значений. Имеется также возможность обойти пары "ключ-значение", используя блок, и даже отфильтровать словарь типа `NSDictionary`, проверяя его значения.

Переход на современный язык Objective-C

Если у вас имеется устаревший прикладной код, который требуется преобразовать, чтобы воспользоваться синтаксисом литералов `NSNumber`, `NSArray` и `NSDictionary` языка Objective-C, а также индексированием массивов и словарей, вы можете сделать это без всякого труда. С этой целью выберите команду меню Edit⇨Refactor⇨Convert to Modern Objective-C Syntax.

Класс NSNull

Класс `NSNull` ничего не делает, а только предоставляет указатель на одиночный объект `[NSNull null]`. Этот одиночный объект можно использовать вместо пустого значения `nil` в тех случаях, когда требуется конкретный объект, а пользоваться значением `nil` запрещено. Его, например, нельзя использовать как значение элемента коллекции (типа `NSArray`, `NSSet` или `NSDictionary`), и тогда приходится обращаться за помощью к объекту `[NSNull null]`.

Несмотря на все сказанное ранее о равенстве, сравнение с объектом `[NSNull null]` можно организовать с помощью оператора равенства языка C, поскольку это одиночный экземпляр. Следовательно, сравнение указателей в данном случае вполне допустимо.

Изменяемые и неизменяемые классы

Начинающие программировать под iOS испытывают определенные трудности, применяя пары изменяемых и неизменяемых классов из каркаса Foundation, поэтому здесь уместно дать

некоторые рекомендации по этому поводу. Из документации может быть не совсем ясно, что в изменяемых классах исполняются, а по возможности и замещаются, методы из неизменяемых классов. Так, выражение `[NSArray array]` формирует неизменяемый массив, тогда как выражение `[NSMutableArray array]` — изменяемый массив. Это же справедливо и для получения экземпляров с помощью всех инициализаторов и методов служебных классов. Они могут включать в свои имена слово "`array`", но при отправке классу `NSMutableArray` они предоставляют изменяемый массив.

Это обстоятельство дает также ответ на следующий вопрос: как сделать неизменяемый массив изменяемым, и наоборот? Если метод `arrayWithArray:` посылается классу `NSArray`, то он предоставляет *изменяемый* массив, содержащий те же самые объекты в том же порядке, что и в оригинале. Следовательно, этот единственный метод может преобразовать неизменяемый массив в изменяемый, и обратно. С помощью метода `copy` можно также получить неизменяемую копию, а с помощью метода `mutableCopy` — изменяемую.

Все сказанное выше в равной степени относится и к парам изменяемых и неизменяемых классов. Часто удобно автоматически работать с временным изменяемым экземпляром, а затем сохранять (а возможно, и поставлять другим классам) неизменяемый экземпляр, защищая тем самым значение от случайного изменения или без вашего ведома. Дело не в том, как объявлен класс, к которому относится переменная, а в том, к какому классу на самом деле относится экземпляр (полиморфизм; см. главу 5). Следовательно, возможность легко переходить от неизменяемого к изменяемому варианту одних и тех же данных является благом.

Для того чтобы проверить, является ли экземпляр изменяемым или неизменяемым, совсем *не* обязательно обращаться к его классу. Все пары изменяемых и неизменяемых классов реализованы в виде *кластеров классов*, а это означает, что в среде Cocoa применяет какой-то скрытый класс, отличающийся от описываемого в документации класса, с которым приходится иметь дело. Этот скрытый класс подлежит изменению без всяких уведомлений, поскольку это не дело программиста и вообще его не касается. Следовательно, код в приведенной ниже форме подвержен ошибкам.

```
if ([NSStringFromClass([n class]) isEqualToString: @"NSCFArray"]) // неверно!
```

Вместо этого выяснить, является ли коллекция изменяемой, можно, проверив, реагирует ли она на модифицирующий метод:

```
if ([n respondsToSelector:@selector(addObject:)]) // верно!
```

(К сожалению, такой прием годится *только* для коллекций. Он не подходит, например, для того, чтобы отличить класс `NSString` от класса `NSMutableString`.)

Если класс коллекции является неизменяемым, это совсем не означает, что объекты, собранные в его коллекции, также являются неизменяемыми. Они по-прежнему остаются объектами и не теряют свое обычное поведение лишь потому, что они указываются посредством неизменяемой коллекции.

Списки свойств

Список свойств — это строковое (в формате XML) представление данных. Только классы `NSString`, `NSData`, `NSArray` и `NSDictionary` из каркаса Foundation могут быть преобразованы в список свойств. Более того, коллекция класса `NSArray` или `NSDictionary` может быть преобразована в список свойств лишь в том случае, если она содержит *только* объекты перечисленных выше классов, а также классов `NSDate` и `NSNumber`. (Именно поэтому объект

типа UIColor следует преобразовать в объект типа NSData, как упоминалось выше, чтобы сохранить его в пользовательских настройках по умолчанию. Ведь пользовательские настройки по умолчанию представляют собой список свойств.)

Список свойств чаще всего служит для хранения данных в файле. В классах NSArray и NSDictionary предоставляются служебные методы writeToFile:atomically: и writeToURL:atomically: для формирования файлов со списками свойств по заданному пути или URL к файлу соответственно. Они предоставляют также обратные служебные методы для инициализации объекта типа NSArray или NSDictionary в зависимости от содержимого списка свойств из заданного файла. Именно по этой причине вам, скорее всего, придется начинать с одного из этих классов, чтобы составить список свойств. (Ведь методы типа writeToFile:... и writeToURL:... из классов NSString и NSData просто записывают данные непосредственно в файл, а не выводят их в виде списка свойств.)

Когда объект типа NSArray или NSDictionary формируется из файла со списком свойств описанным выше образом, то все коллекции, а также объекты символьных строк и прочих данных в коллекции оказываются неизменяемыми. Если же требуется сделать их изменяемыми или преобразовать в список свойств экземпляр одного из других классов из списка свойств, то для этой цели служит класс NSPropertyListSerialization (см. документацию *Property List Programming Guide*).

Скрытые особенности класса NSObject

Каждый класс наследует от класса NSObject, и поэтому здесь имеет смысл уделить немного места его исследованию, чтобы стало понятнее, как им пользоваться. Класс NSObject имеет довольно сложное строение. Ниже перечисляются некоторые особенности его поведения.

- Определение некоторых собственных методов класса и методов экземпляра, имеющих в основном отношение к основным операциям получения экземпляров отправки методов и разрешения (см. документацию *NSObject Class Reference*).
- Принятие протокола класса NSObject. В этом протоколе объявляются методы экземпляра, имеющие в основном отношение к управлению памятью, взаимосвязи экземпляра с его классом и интроспекции. Протокольные методы из класса NSObject являются обязательными, и поэтому все они реализуются в этом классе. (см. документацию *NSObject Class Reference*). Такая архитектура позволяет сделать класс NSProxy корневым, а также принять протокол класса NSObject.
- Реализация служебных методов, связанных с протоколами NSCopying, NSMutableCopying и NSCoding, без формального принятия этих протоколов. Они не приняты в классе NSObject намеренно, поскольку это привело бы к необходимости принять их во всех остальных классах, что было бы неверно. Благодаря именно такой архитектуре можно вызвать соответствующий служебный метод, если в классе все же принят один из этих протоколов. Например, в классе NSObject реализуется метод экземпляра copy, что позволяет вызвать его для экземпляра любого класса. Если в классе не принят протокол NSCopying и не реализован метод copyWithZone:, то исход подобной операции будет неудачным.
- Применение более десятка категорий, распределенных по различным заголовочным файлам, в классе NSObject для внедрения в нем большого числа методов. Например, метод awakeFromNib, упоминавшийся в главе 7, происходит из категории UINibLoadingAdditions класса NSObject, объявленной в заголовочном файле UINibLoading.h. Метод performSelector:withObject:afterDelay:,

рассматриваемый в главе 11, происходит из категории NSDelayedPerforming класса NSObject, объявленной в заголовочном файле NSRunLoop.h.

- Наследование. Как пояснялось в главе 4, объект класса является объектом. Следовательно, все классы, объекты которых относятся к типу Class, наследуют от класса NSObject. Это означает, что *любой метод, определенный в классе* NSObject, *может быть вызван для объекта класса как метод этого класса*! Например, метод respondsToSelector: определен в классе NSObject как метод экземпляра, а следовательно, его можно также рассматривать как метод класса и отправлять объекту класса.

Затруднение для программиста состоит в том, что в документации, предоставляемой компанией Apple, очень жестко регламентирована классификация. Если вы пытаетесь выяснить, что же можно сообщить объекту, то вас интересует не происхождение методов этого объекта, а только то, что можно сообщить этому объекту. В документации методы различаются по их происхождению. Хотя класс NSObject является корневым и самым важным классом, от которого наследуют все остальные классы, тем не менее *ни на одной из страниц документации не предоставляется краткий обзор всех его методов*. Вместо этого приходится одновременно обращаться к документации *NSObject Class Reference* и *NSObject Protocol Reference*, а также к страницам документации с описанием протоколов NSCopying, NSMutableCopying и NSCoding, чтобы понять, каким образом они взаимодействуют с методами, определенными в классе NSObject. Ведь нужно еще мысленно предоставить свой вариант метода класса для каждого метода экземпляра из класса NSObject!

Кроме того, имеются методы, внедренные в класс NSObject по категориям. Методы наиболее общего характера описаны на отдельных страницах документации на класс NSObject. Например, метод cancelPreviousPerformRequestsWithTarget: происходит из категории, объявленной в заголовочном файле NSRunLoop.h. Тем не менее он описывается (и совершенно справедливо) в документации на класс NSObject, поскольку это метод данного класса, а следовательно, он, по существу, является глобальным методом, который можно отправить в любой момент.

Остальные являются методами делегата, применяются в ограниченных случаях (а на самом деле считаются формальными протоколами) и поэтому не требуют централизованной документации. Например, метод animationDidStart: описывается (и совершенно справедливо) в документации на класс CAAnimation, поскольку о нем нужно знать только в том случае, если приходится иметь дело с классом CAAnimation. Однако каждый объект реагирует на метод awakeFromNib и, скорее всего, имеет большое значение для каждого разрабатываемого приложения. Тем не менее для ознакомления с его функциями приходится переходить от документации на класс NSObject к поискам на страницах документации *NSObject UIKit Additions Reference*, где вряд ли можно обнаружить его описание! Это же относится (можно с уверенность утверждать) и ко всем методам доступа к значениям по ключам (см. главу 12), а также к методам наблюдения за значениями по ключам (см. главу 13).

Итак, собрав вместе, так или иначе, все методы класса NSObject, вы обнаружите, что они подпадают под определенную естественную классификацию, как в основном и описывается в документации, предоставляемой компанией Apple (см. также разделы "The Root Class" и "Cocoa Objects" документации *Cocoa Fundamentals Guide*). Ниже приведена эта классификация методов класса NSObject.

Создание, разрушение и управление памятью

К этой категории относятся методы для создания экземпляра, например, `alloc` и `copy`; методы, которые можно заместить, чтобы выяснить что-нибудь происходящее в течение срока действия объекта, например `initialize` (см. главу 11) и `dealloc` (см. главу 12); а также методы для управления памятью (см. главу 12).

Отношения между классами

К этой категории относятся методы, позволяющие выяснить класс объекта и наследование, например `class`, `superclass`, `isKindOfClass:` и `isMemberOfClass:`. Для проверки класса экземпляра (или класса) служат такие методы, как `isKindOfClass:` и `isMemberOfClass:`. Прямое сравнение объектов двух классов редко рекомендуется, как, например, в следующем выражении:

```
[someObject class] == [otherObject class]
```

Дело в том, что класс экземпляра в среде Cocoa может быть закрытым, недокументированным подклассом ожидаемого класса. Об этом упоминалось ранее в связи с кластерами классов, но подобная ситуация может возникнуть и в других случаях.

Интроспекция и сравнение объектов.

К этой категории относятся методы, позволяющие выяснить, что произойдет, если объекту будет послано сообщение (например, `respondsToSelector:`); представить объект в виде символьной строки (метод `description`, применяемый при отладке кода; см. главу 9); а также сравнить объекты (`isEqual:`).

Реагирование на сообщения

К этой категории относятся методы, позволяющие выяснить, что происходит, когда объекту посылается определенное сообщение (например, `doesNotRecognizeSelector:`). Подробнее об этом можно узнать из документации *Objective-C Runtime Programming Guide.*

Отправка сообщений

К этой категории относятся методы, предназначенные для отправки сообщений в динамическом режиме. Например, метод `performSelector:` принимает селектор в качестве параметра и, оправляя его объекту, сообщает ему, что он должен выполнить этот селектор. На первый взгляд, это похоже на отправку сообщения данному объекту, но что, если вплоть до времени выполнения неизвестно, какое именно сообщение следует посылать? Имеются также варианты метода `performSelector:`, позволяющие отправить сообщение в отдельно указанном потоке или по истечении определенного промежутка времени (`performSelector:withObject:afterDelay:` и подобные методы).

ГЛАВА 11

События в среде Cocoa

Ни один прикладной код не выполняется до тех пор, пока он не будет вызван из среды Cocoa. По этой причине искусство программирования в операционной системе iOS состоит, главным образом, в том, чтобы знать, когда и почему в среде Cocoa вызывается прикладной код. Зная это, вы сможете разместить код в нужном месте, правильно указав имя метода, чтобы ваш код был выполнен в подходящий момент, а приложение вело себя именно так, как вы и предполагали.

Например, в главе 7 был рассмотрен пример написания метода, вызываемого в тот момент, когда пользователь нажимает определенную кнопку в интерфейсе приложения. Все было устроено таким образом, чтобы метод *непременно* вызывался именно тогда, когда пользователь нажимает данную кнопку.

```
- (void) buttonPressed: (id) sender {
   // ... отреагировать на нажатие кнопки
}
```

Такая архитектура олицетворяет основы построения программной среды Cocoa. Прикладной код подобен панели кнопок, ожидающей от среды Cocoa нажатия одной из них. Если произойдет нечто такое, что, по мнению среды Cocoa, следует знать прикладному коду или на что он должен отреагировать, она нажмет нужную кнопку на его панели. Следовательно, вам нужно организовать свой код с учетом поведения среды Cocoa. Эта среда берет на себя определенные обязательства относительно того, как и когда она будет осуществлять диспетчеризацию сообщений, направляемых прикладному коду. Такие сообщения в среде Cocoa называются *событиями*. Зная, что собой представляют эти события, вы можете подготовить свой код к их приему от среды Cocoa.

Таким образом, для программирования в операционной системе iOS вам придется, по существу, отдать управление. Ваш код вообще не будет выполняться, когда ему заблагорассудится. Он может выполняться *только* в ответ на некоторое событие. Когда что-нибудь происходит, например, пользователь делает жест на сенсорном экране, или наступает определенная стадия во время действия приложения, в среде Cocoa осуществляется отправка события прикладному коду, если только он готов принять его. Следовательно, вам не нужно, как прежде, писать код и размещать его где-нибудь по старинке. Вместо этого вы пользуетесь каркасом приложений, давая ему возможность воспользоваться вашим кодом. Итак, вы подчиняетесь правилам, обязательствам и ожиданиям среды Cocoa, чтобы ваш код был правильно вызван в нужный момент.

Конкретные события, которые можно получать в прикладном коде, перечислены в документации. Общая архитектура, определяющая способ и момент отправки событий, а также способы организации прикладного кода для своевременного их получения являются предметом рассмотрения этой главы.

Причины для получения событий

Вообще говоря, причины для получения событий можно разделить на четыре категории. Это неофициальные, но придуманные мною категории. Зачастую оказывается не совсем ясно, к какой именно категории следует отнести конкретное событие. Более того, событие может подпасть сразу под две категории. Тем не менее этими категориями удобно пользоваться, чтобы наглядно представить, как и почему среда Cocoa взаимодействует с прикладным кодом. Ниже показано, каким образом события разделяются по этим четырем категориям.

Пользовательские события

Пользователь выполняет какую-нибудь операцию в диалоговом режиме, а событие инициируется непосредственно. Очевидными тому примерами служат события, получаемые, когда пользователь касается или проводит пальцем по экрану или же нажимает клавишу.

События времени действия

Это события, уведомляющие о наступлении определенной стадии во время действия приложения, например, когда приложение запускается на выполнение, готово перейти в фоновый режим работы или только что загруженный его компонент (например, представление типа `UIViewController`) должен быть удален с экрана.

Функциональные события

Такие события наступают в том случае, если в среде Cocoa предполагается сделать что-нибудь, а прикладному коду требуется предоставить дополнительные функциональные возможности, и тогда она обращается к нему. К этой категории относятся события, связанные с воспроизведением представления типа `UIView` с помощью метода `drawRect:` или надписи на метке типа `UILabel` с помощью метода `drawTextInRect:`, как было показано в главе 10.

Запросные события

Такие события наступают в том случае, если среде Cocoa требуется обратиться с запросом к прикладному коду. В этом случае ее поведение зависит от получаемого ответа. Например, порядок появления данных в таблице (типа `UITableView`) таков, что всякий раз, когда среде Cocoa требуется ячейка таблицы, она запрашивает ее у прикладного кода.

Наследование

Во встроенном классе Cocoa можно определить методы, которые эта среда будет вызывать сама и которые желательно (или обязательно) заместить в подклассе. Это обусловливает специальное, а не только стандартное поведение прикладного кода.

В главе 10 был приведен пример, в котором представление типа `UIView` воспроизводилось с помощью метода `drawRect:`. Это был пример так называемого функционального события. Замещая метод `drawRect:` в подклассе, производном от класса `UIView`, можно предписать целую процедуру для воспроизведения представления. Точно неизвестно, когда именно этот

метод будет вызван, да это и не важно, но когда он все же будет вызван, представление воспроизведется. Это гарантирует, что представление всегда будет выглядеть именно так, как требуется. (Для этого совсем не обязательно вызывать метод `drawRect:` самостоятельно. Достаточно вызвать метод `setNeedsDisplay` и предоставить среде Cocoa возможность самой вызвать метод `drawRect:` в ответ.)

Встроенные в класс `UIView` подклассы могут иметь и другие методы функциональных событий, которые потребуется специально настроить путем наследования. Как правило, это требуется для того, чтобы изменить способ воспроизведения представления, не возлагая на себя все бремя самостоятельного выполнения процедуры воспроизведения. В главе 10 был приведен пример воспроизведения надписи на метке типа `UILabel` с помощью метода `drawTextInRect:`. Аналогичным примером служит воспроизведение ползунка типа `UISlider`, где имеется возможность специально настроить положение и размер бегунка, перемещающегося по шкале ползунка, заместив метод `thumbRectForBounds:trackRect:value:`.

Класс `UIViewController` служит подходящим примером для наследования. Среди методов, перечисленных в документации на класс `UIViewController`, следует прежде всего отметить те, для замещения которых могут возникнуть веские основания. Если создать в среде Xcode подкласс, производный от класса `UIViewController`, то можно обнаружить, что в шаблон уже входят два замещенных метода, с которых можно начать разработку нового подкласса. Например, метод `viewDidLoad` вызывается, чтобы уведомить о том, что представление контроллера загружено, а следовательно, можно приступить к инициализации. Это вполне очевидный пример события времени действия.

Такой метод из класса `UIViewController`, как `supportedInterfaceOrientations`, относится к так называемым запросным событиям. В этом случае из прикладного кода нужно возвратить битовую маску, сообщающую среде Cocoa, в какой ориентации представление может оказаться в определенный момент времени, когда бы он ни настал. Обязанность вызвать этот метод в подходящие моменты времени доверяется среде Cocoa, и если пользователь повернет мобильное устройство, то интерфейс приложения должен или не должен соответственно изменить свою ориентацию в зависимости от значения, возвращаемого из прикладного кода.

В поисках событий, которые можно получать через наследование, непременно проследите отношения между классами вверх по иерархии наследования. Так, если вам нужно выяснить, как получить уведомление, когда специальный подкласс, производный от класса `UILabel`, будет встроен в другое представление, вы найдете ответ в документации на класс `UILabel`. Этот класс получает соответствующее событие в силу того, что он является производным от класса `UIView`. Из документации на класс `UIView` можно узнать, что метод `didMoveToSuperview` можно заместить, чтобы получить уведомление о наступлении данного события.

Следуя дальше вверх по иерархии наследования, вы обнаружите, например, метод `initialize` класса `NSObject`. Прежде чем любой класс сможет отправить свое первое сообщение класса, включая получение экземпляра, отправляется сообщение `initialize`. Следовательно, метод `initialize` можно инициализировать, чтобы выполнить код как можно раньше в течение срока действия класса. Так, в проекте `Empty Window` экземпляр класса делегата `AppDelegate` получается очень рано во время действия приложения, и поэтому его метод `initialize` может послужить удобным местом для выполнения инициализации как можно раньше в приложении, например, для установки по умолчанию значений любых пользовательских настроек.

Реализуя метод `initialize`, необходимо попутно проверить, указывается ли ссылка `self` на интересующий класс, иначе метод `initialize` может быть вызван снова, а следовательно, прикладной код будет выполнен повторно, если используется подкласс, производный

от данного класса. Это один из тех немногих случаев, когда два класса сравниваются друг с другом непосредственно, как показано ниже. Поскольку метод `initialize` является методом класса, то ссылка `self`, по существу, означает класс, которому было отправлено сообщение `initialize`.

```
// MyClass.m:
+ (void)initialize {
    if (self == [MyClass class]) {
      // сделать что-нибудь
    }
}
```

Уведомления

Для приложения в среде Cocoa предоставляется единственный экземпляр класса `NSNotificationCenter`, неофициально называемый *центром уведомлений* и доступный в виде `[NSNotificationCenter defaultCenter]`. Этот экземпляр служит основанием для механизма отправки сообщений, называемых *уведомлениями*. В уведомление входит экземпляр класса `NSNotification` (объект уведомления). Основная идея состоит в том, что любой объект может быть зарегистрирован в центре уведомлений для получения определенных уведомлений. Другой объект может передать объект уведомления в центр уведомлений для дальнейшей отправки (или так называемой *рассылки*). После этого центр уведомлений разошлет этот объект уведомления (в самом уведомлении) всем объектам, зарегистрированным для его получения.

Механизм уведомлений нередко описывается как отправляющий или широковещательный, и на то имеются веские основания. Ведь он позволяет отправить объекту сообщение, не зная и даже не интересуясь, какие объекты принимают сообщение и сколько их вообще имеется. Благодаря этому архитектура приложения освобождается от формальной ответственности за такое подключение экземпляров, чтобы сообщение можно было передавать от одного из них к другому, что иногда может оказаться сложным и трудным делом, как поясняется в главе 13. Когда объекты принципиально удалены друг от друга, уведомления могут стать довольно простым способом обмена сообщениями между такими объектами.

Объект типа `NSNotification` содержит следующие три фрагмента данных, которые связаны с ним и могут быть извлечены с помощью методов экземпляра.

`name`

Это объект типа `NSString`, обозначающий смысловое содержание уведомления.

`object`

Это экземпляр, связанный с уведомлением (как правило, экземпляр, пославший его).

`userInfo`

Этот фрагмент данных включается далеко не во всякое уведомление. Он представляет собой словарь типа `NSDictionary` и может содержать дополнительные сведения об уведомлении. Данные, которые может содержать словарь `userInfo` типа `NSDictionary`, а также те ключи, под которыми они могут храниться в этом словаре, зависят от конкретного уведомления, поэтому обращайтесь за дополнительной справкой к документации. Например, в документации сообщается, что уведомление `UIApplicationDidChangeStatusBarFrameNotification` из класса `UIApplication` включает в себя словарь `userInfo` с ключом `UIApplicationStatusBarFrameUserInfoKey`, значение

которого обозначает фрейм строки состояния. Если вы рассылаете уведомление самостоятельно, то можете ввести в словарь `userInfo` какие угодно данные, чтобы их извлекли получатели уведомления.

Сама среда рассылает уведомления через центр уведомлений, а прикладной код может быть зарегистрирован для их получения. В документации на класс, предоставляющий уведомления, можно обнаружить отдельный раздел, посвященный уведомлениям.

Получение уведомлений

Для того чтобы зарегистрироваться на получение уведомления, следует послать в центр уведомлений одно из двух сообщений. В частности, метод `addObserver:selector:name:object:` имеет следующие параметры:

`observer:`

Это экземпляр, которому посылается уведомление. Как правило, это экземпляр, доступный по ссылке `self`. Один экземпляр обычно не регистрирует другой экземпляр в качестве получателя уведомления.

`selector:`

Это сообщение, посылаемое экземпляру наблюдателя, когда имеет место уведомление. Специально назначенный для этой цели метод должен возвращать тип `void` (фактически нечего не возвращать) и принимать один параметр, которым должен быть объект типа `NSNotification`. Следовательно, этот параметр должен быть обозначен как `NSNotification*` или `id`. Не забудьте реализовать данный метод! Если центр сообщений посылает уведомление, отправляя сообщение, указанное в качестве параметра `selector:`, а метод, принимающий это сообщение, не реализован, то приложение завершится аварийно (см. главу 3).

`name:`

Это имя уведомления, которое требуется получить и которое представлено объектом типа `NSString`. Если же этот параметр указан как `nil`, значит, запрашиваются *все* уведомления, связанные с объектом, обозначаемым параметром `object:`. Как правило, имя встроенного в среду Cocoa уведомления является константой. Как пояснялось в главе 1, это удобно, поскольку компилятор выдаст предупреждение, если допустить ошибку в имени константы. Если ввести неверное имя уведомления непосредственно в виде литерала типа `NSString`, то компилятор не выдаст предупреждение, но никаких уведомлений непостижимым образом не будет получено, поскольку ни одно из них не носит введенное имя. Такую ошибку очень трудно обнаружить.

`object:`

Это объект интересующего уведомления, который, как правило, рассылает его. Если этот параметр указан как `nil`, значит, запрашиваются *все* уведомления с именем, обозначаемым параметром `name:`. (Если же оба параметра `name:` и `object:` указаны как `nil`, значит, запрашиваются *все* уведомления!)

Например, в одном из моих приложений требуется изменять интерфейс всякий раз, когда проигрыватель музыкальных записей на мобильном устройстве начинает воспроизводить другую песню. Прикладной интерфейс API для встроенного в мобильное устройство

проигрывателя относится к классу MPMusicPlayerController, который посылает уведомление, когда в этом проигрывателе происходит смена воспроизводимой музыки. В документации на класс MPMusicPlayerController это уведомление обозначено как MPMusicPlayerControllerNowPlayingItemDidChangeNotification в разделе уведомлений.

Как следует из документации, это уведомление вообще не рассылается, если не вызвать сначала метод экземпляра beginGeneratingPlaybackNotifications из класса MPMusicPlayerController. Такая архитектура весьма характерна для среды Cocoa, где экономится время и усилия на том, чтобы не отправлять некоторые сообщения до тех пор, пока они не будут фактически активизированы. Поэтому я первым делом получаю экземпляр класса MPMusicPlayerController и вызываю его метод следующим образом:

```
MPMusicPlayerController* mp = [MPMusicPlayerController iPodMusicPlayer];
[mp beginGeneratingPlaybackNotifications];
```

Затем я регистрируюсь для получения требуемого уведомления о воспроизведении, как показано ниже.

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(nowPlayingItemChanged:)
        name:MPMusicPlayerControllerNowPlayingItemDidChangeNotification
      object:nil];
```

Теперь всякий раз, когда рассылается уведомление MPMusicPlayerControllerNowPlayingItemDidChangeNotification, в моем приложении будет вызываться приведенный ниже метод nowPlayingItemChanged:.

```
- (void) nowPlayingItemChanged: (NSNotification*) n {
   MPMusicPlayerController* mp = [MPMusicPlayerController iPodMusicPlayer];
   self->_nowPlayingItem = mp.nowPlayingItem;
   // ... и так далее ...
}
```

Зарегистрироваться для получения уведомлений можно также, вызвав метод addObserverForName:object:queue:usingBlock:. Этот метод возвращает значение, конкретное назначение которого поясняется ниже. Параметр queue: этого метода обычно имеет пустое значение nil, а непустое его значение предназначено для многопоточной обработки в фоновом режиме. Параметры name: и object: подобны описанным выше параметрам метода addObserver:selector:name:object:. Вместо наблюдателя и селектора в данном случае предоставляется блок, состоящий из конкретного кода, выполняемого при поступлении уведомления. Этот блок должен принимать в качестве единственного параметра сам объект уведомления типа NSNotification.

Такой способ регистрации для получения уведомлений имеет ряд существенных преимуществ. С одной стороны, для правильного функционирования метода addObserver:selector:name:object: необходимо получить нужный селектор и реализовать соответствующий метод. С другой стороны, все нужные операции происходят в блоке и не требуют селектора и отдельного метода, как показано ниже.

```
MPMusicPlayerController* mp = [MPMusicPlayerController iPodMusicPlayer];
[mp beginGeneratingPlaybackNotifications];
id ob = [[NSNotificationCenter defaultCenter]
 addObserverForName:MPMusicPlayerControllerNowPlayingItemDidChangeNotification
  object:nil queue:nil usingBlock:^(NSNotification *n) {
    self->_nowPlayingItem = mp.nowPlayingItem;
    // ... и так далее ...
  }
}];
```

Рассмотрим, насколько понятным и удобным для сопровождения является такой код. Интенсивное применение метода `addObserver:selector:name:object:` означает, что прикладной код в конечном итоге будет усеян методами, которые существуют только для вызова из центра уведомлений. Однако об их назначении ничто не говорит, а следовательно, требует ввода в исходный код соответствующих комментариев, напоминающих об этом. Кроме того, эти методы отделены от вызова на регистрацию, и все это вместе делает прикладной код испещренным малопонятными методами и слишком запутанным. Теперь обратите внимание на то, что в блоке не нужно переопределять переменную экземпляра `mp`, как пришлось сделать в отдельном методе `nowPlayingItemChanged:`. Она по-прежнему находится в той области действия, где была определена несколькими строками кода раньше. В этом отношении блоки очень удобны!

Снятие с регистрации

Каждый объект, зарегистрированный в качестве получателя уведомлений, вы вольны снять с регистрации, прежде чем он прекратит свое существование. Если вы не сделаете этого, а объект прекратит свое существование и будет разослано уведомление, для получения которого этот объект зарегистрирован, то центр уведомлений попытается отправить соответствующее сообщение данному, уже отсутствующему объекту. В итоге произойдет аварийный отказ в лучшем случае, а в худшем — полный хаос.

Для того чтобы снять объект с регистрации на получение уведомлений, достаточно отправить центру уведомлений сообщение `removeObserver:`. (С другой стороны, объект можно снять с регистрации на получение только отдельных уведомлений с помощью метода `removeObserver:name:object:`.) В качестве аргумента `observer:` соответствующему методу передается объект, который уже не принимает уведомления. Каким должен быть этот объект, зависит, прежде всего, от одного из следующих способов регистрации.

Вызов метода `addObserver:`.....

В этом случае наблюдатель предоставляется изначально, а следовательно, заранее известен наблюдатель, который требуется снять с регистрации.

Вызов метода `addObserverForName:`....

В результате вызова метода `addObserverForName:...` возвращается объект маркера наблюдателя, который сохраняется в переменной `id` (его класс и характер особого значения не имеют). По существу, это наблюдатель, который требуется снять с регистрации.

Самое главное — найти подходящий момент для снятия с регистрации. В качестве запасного варианта можно воспользоваться методом `dealloc` зарегистрированного экземпляра. Это последнее событие времени действия, посылаемое экземпляру перед тем, как он прекратит свое существование. Если выбрать второй способ снятия с регистрации, воспользовавшись механизмом ARC и методом `addObserverForName:...`, то следует принять во внимание некоторые дополнительные последствия для управления памятью, о которых речь пойдет в главе 12.

Если метод `addObserverForName:...` вызывается многократно из одного и того же класса, то в конечном итоге из центра уведомлений поступает множество маркеров наблюдателя, которые приходится сохранять, чтобы в дальнейшем снять их с регистрации. Если же предполагается снять с регистрации все объекты сразу, то для этой цели можно сохранить их в виде изменяемой коллекции в переменной экземпляра. Так, если имеется переменная

экземпляра _observers типа NSMutableSet, то с самого начала она инициализируется пустым множеством, как показано ниже.

```
self->_observers = [NSMutableSet set];
```

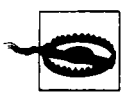
Всякий раз, когда регистрация на получение уведомлений производится с помощью блока, результат фиксируется и вводится в множество следующим образом:

```
id ob = [[NSNotificationCenter defaultCenter]
    addObserverForName:@"whatever" object:nil queue:nil
    usingBlock:^(NSNotification *note) {
       // ... все, что угодно ...
   }];
[self->_observers addObject:ob];
```

Когда же наступает время для снятия с регистрации, то организуется перечисление элементов множества, как показано ниже.

```
for (id ob in self->_observers)
    [[NSNotificationCenter defaultCenter] removeObserver:ob];
```

Неудобство организации всей этой процедуры является той ценой, которую приходится платить за преимущества, которые дают блоки при обращении с уведомлениями.



В классе NSNotificationCenter никакой интроспекции не предоставляется. В частности, к центру уведомлений типа NSNotificationCenter нельзя обратиться с запросом, чтобы выяснить, какие именно объекты зарегистрированы в нем в качестве получателей уведомлений. На мой взгляд, это серьезный пробел в функциональных возможностях среды Cocoa. Некогда мне пришлось потратить немало времени, чтобы понять, почему один из экземпляров в коде моего приложения не получал уведомление, на которое он был зарегистрирован. Простейшая самодиагностика кода не помогла. Однако в конечном итоге я понял, что в одном забытом мною фрагменте кода наблюдатель преждевременно снимался с регистрации. Из всего сказанного можно сделать следующий вывод: логика постановки и снятия с регистрации должна быть очень простой.

Рассылка уведомлений

Больше всего вас, безусловно, будет интересовать получение уведомлений от среды Cocoa, но вы можете воспользоваться механизмом уведомлений и для организации обмена данными между объектами. Как упоминалось ранее, одним из веских оснований для этого может послужить принципиальная удаленность или независимость объектов друг от друга.

Для такого применения уведомлений объекты должны играть обе роли в цепочке обмена данными. В частности, один или несколько объектов, распознаваемых по имени, по содержащемуся в них объекту или обоими способами, должны зарегистрироваться для получения уведомлений, как пояснялось ранее. Другие объекты, распознаваемые аналогичным образом, должны рассылать уведомления. Тогда центр уведомлений будет передавать сообщение от отправителя получателю рассылаемых уведомлений.

Для того чтобы организовать рассылку уведомлений, следует послать центру уведомлений сообщение postNotificationName:object:userInfo:. Например, в одном из моих приложений для простой игры в карты нужно знать, когда карта выбрана касанием экрана. В то же время самой карте ничего неизвестно об игре. Когда она выбирается подобным способом, то просто издает виртуальное восклицание, рассылая уведомление, как показано ниже.

```
- (void) singleTap: (id) g {
   [[NSNotificationCenter defaultCenter]
   postNotificationName:@"cardTapped" object:self];
}
```

Игровой объект зарегистрирован на получение уведомления `@"cardTapped"`, и поэтому он откликается на него, извлекая объект уведомления. Теперь ему известно, что карта выбрана, и он может правильно продолжить свое действие.

Здесь опускается ряд других аспектов уведомлений, о которых вам вряд ли нужно знать. Вы можете подробнее ознакомиться с ними в документации *Notification Programming Topics for Cocoa.*

Класс NSTimer

Строго говоря, таймер не является уведомлением, но ведет себя похожим образом. Это объект класса `NSTimer`, выдающий сигнал (*срабатывания*) по истечении заданного промежутка времени. Таким сигналом служит сообщение, посылаемое одному из экземпляров в прикладном коде. Следовательно, можно организовать отправку уведомления, когда истечет определенный промежуток времени. Это не идеально точное, но вполне приемлемое согласование по времени.

Управлять таймером не очень сложно, хотя и не совсем обычно. Таймер, отсчитывающий время, считается *запланированным*. Таймер может срабатывать *однократно* или *периодически*. Для того чтобы таймер прекратил свое существование, нужно сделать его *недействительным*. Таймер, установленный на однократное срабатывание, становится недействительным автоматически после своего срабатывания. Периодический таймер продолжает срабатывать до тех пор, пока не будет сделан недействительным вручную, для чего следует отправить сообщение, объявляющее таймер недействительным. Недействительный таймер следует считать запрещенным. Его нельзя больше восстановить или использовать, и вряд ли можно посылать ему сообщения.

Таймер проще всего создать с помощью метода `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` из класса `NSTimer`. Этот метод создает и планирует таймер, чтобы тот сразу же начал отсчитывать время. Он должен принимать единственный параметр в виде ссылки на таймер. Параметры `target:` и `selector:` определяют соответственно целевой объект и сообщение, которое посылается при срабатывании таймера, а параметр `userInfo:` такой же, как и при рассылке уведомлений.

Например, в одном из моих игровых приложений ведется счет. Для того чтобы оштрафовать пользователя, требуется убавить его счет, если по истечении десяти секунд после каждого своего движения пользователь не совершит следующее движение. Следовательно, всякий раз, когда он совершит движение, в моем приложении создается таймер, срабатывающий через десять секунд, а кроме того, делается недействительным любой существующий таймер. Метод, вызываемый таймером, убавляет счет.

В версии iOS 7 в классе `NSTimer` появилось свойство `tolerance`, определяющее допуск на срабатывание таймера, т.е. промежуток времени, в течение которого таймер может сработать по истечении заданного времени. Как поясняется в документации на класс `NSTimer`, допуск на срабатывание таймера, составляющий не менее `10%` от задаваемого промежутка времени (`timeInterval`), позволяет экономнее расходовать заряд батареи мобильного устройства. Применение таймеров имеет ряд последствий для управления памятью. Они обсуждаются в главе 12 наряду с применением блоков в качестве альтернативы таймерам.

Делегирование

Делегирование — это объектно-ориентированный шаблон проектирования, определяющий отношение между двумя объектами, где поведение первого объекта специально настраивается или поддерживается вторым объектом. В этом случае второй объект является *делегатом* первого. Для этого не требуется никакого наследования, и на самом деле первому объекту вообще ничего неизвестно о классе второго объекта.

Делегирование в среде Cocoa реализовано следующим образом. Встроенный класс Cocoa имеет переменную экземпляра, обычно называемую `delegate` (или хотя бы содержащую упоминание о делегате в своем имени). В переменной некоторого экземпляра этого класса Cocoa в качестве значения задается экземпляр одного из классов прикладного кода. В какие-то моменты своего действия класс Cocoa берет на себя обязательство обратиться к своему делегату за инструкциями, посылая ему некоторое сообщение. Если экземпляр класса Cocoa обнаружит, что его делегат не является пустым (`nil`) и готов к приему этого сообщения (см. описание метода `respondsToSelector:` в главе 10), то он посылает сообщение своему делегату.

Как пояснялось в главе 10, одним из основных примеров применения протоколов в среде Cocoa является делегирование. В прошлом методы делегата перечислялись в документации на класс Cocoa, а их сигнатуры становились известными компилятору через неформальный протокол (категорию в классе `NSObject`). Теперь методы делегата такого класса обычно перечисляются в документации на оригинальный протокол. В среде Cocoa насчитывается более 70 протоколов делегата, что явно свидетельствует о широком их применении в этой среде. Большинство методов делегата являются необязательными, но в некоторых случаях могут потребоваться и обязательные методы.

Делегирование в среде Cocoa

Для того чтобы специально настроить поведение экземпляра класса Cocoa посредством делегирования, следует выбрать в прикладном коде класс, в котором при необходимости объявляется соответствие протоколу, подходящему для делегирования. При выполнении приложения в свойстве `delegate` (или под другим аналогичным именем) экземпляра класса Cocoa задается экземпляр класса из прикладного кода. Это можно сделать как в самом прикладном коде, так и в nib-файле, установив связь с выходом `delegate` (или под другим аналогичным именем) соответствующего экземпляра, который должен служить в качестве делегата. Помимо обязанностей делегата для экземпляра класса Cocoa, класс делегата может выполнять и другие функции. Замечательная особенность делегирования состоит, в частности, в том, что оно оставляет место для внедрения кода делегата в архитектуру класса прикладного кода.

Рассмотрим простой пример, в котором используется предупреждающее представление типа `UIAlertView`. Если в этом представлении выбрана кнопка Cancel, то оно удаляется с экрана. Если же требуется выполнить еще что-нибудь в ответ на удаление предупреждающего представления, то ему нужно предоставить делегат, чтобы получить событие (`alertView:didDismissWithButtonIndex:`), уведомляющее об удалении данного представления. Делегат настолько часто предоставляется предупреждающему представлению типа `UIAlertView`, что специально назначаемый для него инициализатор разрешает это делать. Как правило, в качестве делегата служит экземпляр (`self`), вызывающий прежде всего предупреждающее представление, как показано ниже.

```
- (void) gameWon {
   UIAlertView* av =
      [[UIAlertView alloc] initWithTitle:@"Congratulations!"
```

```
                                    message:@"You won the game. Another game?"
                                   delegate:self
                          cancelButtonTitle:@"No, thanks."
                          otherButtonTitles:@"Sure!", nil];
    [av show];
}

- (void) alertView:(UIAlertView*) av
    didDismissWithButtonIndex: (NSInteger) ix {
      if (ix == 1) { // пользователь согласился еще поиграть
      [self newGame];
    }
}
```

У общего для приложения экземпляра ([UIApplication sharedApplication]) имеется свой делегат, выполняющий настолько важную роль в приложении, что он автоматически предоставляется в шаблонах приложений, разрабатываемых в среде Xcode. Как пояснялось в главе 6, приложение запускается на выполнение при вызове метода UIApplicationMain, как показано ниже.

```
int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
               NSStringFromClass([AppDelegate class]));
    }
}
```

В шаблоне проекта предоставляются файлы, в которых определяется класс AppDelegate. В приведенном выше фрагменте кода методу UIApplicationMain предписывается получить экземпляр класса AppDelegate и сделать его делегатом общего для приложения экземпляра, который также создан. Как пояснялось в главе 10, в классе AppDelegate формально принят протокол UIApplicationDelegate, а это означает, что он готов служить в качестве делегата. Там же упоминалось, что метод respondsToSelector: посылается затем делегату приложения, чтобы выяснить, какие именно методы протокола реализованы в нем. Так, если в нем реализован метод application:didFinishLaunchingWithOptions:, ему будет послан именно этот метод, предоставляющий прикладному коду одну из самых первых возможностей для выполнения.

Методы делегата из класса UIApplication предоставляются также в виде уведомлений. Это дает возможность экземпляру, отличающемуся от делегата приложения, быть в курсе событий, происходящих во время действия приложения, если только он зарегистрируется для их получения. Аналогичным образом в ряде других классов предоставляются дублирующие события. Например, метод делегата tableView:didSelectRowAtIndexPath: из класса UITableView соответствует уведомлению UITableViewSelectionDidChangeNotification.

В именах многих методов делегата из среды Cocoa принято употреблять модальные глаголы should (должен), will (будет) или did (сделал). Так, сообщение типа will посылается делегату перед тем, как что-нибудь произойдет; а сообщение типа did — после того, как что-нибудь произойдет. Метод типа should имеет особое назначение: он возвращает логическое значение BOOL, в ответ на которое предполагается получить логическое значение YES, разрешающее сделать что-нибудь, или логическое значение NO, запрещающее что-нибудь.

В документации поясняется, какой должна быть реакция по умолчанию. В частности, реализовывать метод типа `should` не нужно, если реакция по умолчанию всегда приемлема.

Зачастую свойство управляет некоторым общим поведением, тогда как сообщение делегата позволяет видоизменить это поведение в зависимости от обстоятельств во время выполнения. Например, свойство `scrollsToTop` управляет быстрой прокруткой представления вверх, как только пользователь коснется строки состояния. Но, даже если это свойство имеет логическое значение `YES`, подобное поведение можно запретить для конкретного вида касания экрана, если возвратить логическое значение `NO` из метода делегата `scrollViewShouldScrollToTop:`.

В поисках документации на способы уведомления о наступлении некоторых событий обращайтесь за справкой к соответствующему протоколу делегата, если таковой имеется. Так, если требуется выяснить, когда именно следует начинать редактирование в поле типа `UITextField`, которого пользователь коснулся, вы вряд ли найдете что-нибудь, связанное с этим событием, в документации на класс `UITextField`. На самом деле вам следует искать описание метода `textFieldDidBeginEditing:` в документации на протокол `UITextFieldDelegate`. Если требуется отреагировать на перестановку пользователем элементов на панели закладок, то за справкой следует обращаться к документации на протокол `UITabBarControllerDelegate`. И так далее...

Реализация делегирования

Шаблон делегата в среде Cocoa, обязанности которого описываются в протоколе, можно вполне сымитировать в прикладном коде. Несмотря на то что для настройки этого шаблона требуется некоторый практический опыт и определенное время, такой подход зачастую оказывается наиболее подходящим, поскольку это позволяет оптимально распределить знания и ответственность среди различных вовлеченных объектов. Характерным тому примером служит двухсторонняя связь, когда объект A создает и настраивает объект B, а перед тем, как объект B прекратит свое существование, он устанавливает обратную связь с объектом A.

Действие шаблона в данном примере состоит в том, что в заголовочном файле класса B определяется протокол `SomeProtocol` наряду со свойством `delegate`, обозначаемым как `id <SomeProtocol>`. Таким образом, класс A импортирует заголовочный файл из класса B и настраивает его, что вполне приемлемо и правильно, поскольку объект A собирается создать и настроить объект B. Однако классу B совсем не обязательно знать о классе A, что также приемлемо и правильно, поскольку его основная обязанность — лишь обслуживать любой другой объект. Поскольку класс A импортирует заголовочный файл из класса B, то ему известно о протоколе `SomeProtocol`, и он может принять и реализовать его обязательные методы. Когда объект A создает объект B, он также устанавливает себя в качестве делегата объекта B. Теперь объекту B известно все, что он должен знать, а именно: он может посылать сообщения по протоколу `SomeProtocol` своему делегату независимо от того, к какому именно классу принадлежит этот делегат.

Для того чтобы стало понятнее, почему в данном случае подходит именно такой шаблон, обратимся к конкретному примеру. В одном из моих приложений отображается представление, где пользователь может перемещать три ползунка для выбора цвета. Соответственно код этого представления находится в классе `ColorPickerController`. Когда пользователь нажимает кнопку Done или Cancel, представление должно быть удалено с экрана, но прежде в коде, отображающем это представление, нужно выяснить, какой именно цвет выбрал пользователь. Для этого нужно послать сообщение от экземпляра класса `ColorPickerController` обратно экземпляру класса, отображающего представление. Ниже приведено объявление этого сообщения.

```
- (void) colorPicker:(ColorPickerController*)picker
    didSetColorNamed:(NSString*)theName
             toColor:(UIColor*)theColor;
```

Невольно возникает вопрос: где должно быть размещено это объявление? В моем приложении экземпляр, который, по существу, отображает представление типа `ColorPickerController`, относится к классу `SettingsController`. Поэтому приведенный выше метод можно было бы объявить в разделе интерфейса заголовочного файла класса `SettingsController` и затем организовать импорт этого файла в классе `ColorPickerController`. Однако такой подход оказывается неверным по следующим причинам.

- В обязанности класса `SettingsController` не должно входить объявление метода, реализуемого только из уважения к классу `ColorPickerController`.
- Если рассматриваемый здесь метод объявляется в заголовочном файле класса `SettingsController`, то этот файл должен быть импортирован в классе `ColorPickerController`, чтобы послать сообщение. Однако это означает, что классу `ColorPickerController` теперь известно все о классе `SettingsController`, тогда как ему следует знать *лишь* то, что в данном классе реализуется рассматриваемый здесь метод.
- Вполне возможно, что экземпляр, посылающий данное сообщение, относится к классу `SettingsController`. Он должен быть доступен для любого класса, чтобы отобразить или удалить с экрана представление типа `ColorPickerController`. Следовательно, он уполномочен получить данное сообщение.

Следовательно, требуется, чтобы в классе `ColorPickerController` был объявлен метод, который должен вызываться из *этого же* класса, а также отправлять вслепую сообщение некоторому получателю без учета принадлежности последнего к какому-то конкретному классу. Для этого потребуется установить фактическую связь между объявлением данного метода в классе `ColorPickerController` и его реализацией в классе получателя. Именно такую связь и устанавливает протокол! Итак, в заголовочном файле класса `ColorPickerController` следует определить протокол вместе с входящим в него данным методом, а в классе, отображающем представление типа `ColorPickerController`, — установить соответствие этому протоколу. Кроме того, в классе `ColorPickerController` имеется свойство, обозначаемое как `delegate` или аналогичным образом. Этим обеспечивается канал связи для передачи сообщения, а компилятор уведомляется, что отправка данного сообщения является вполне допустимой.

Ниже приведено содержимое заголовочного файла `ColorPickerController`. Обратите внимание на применение предваряющего объявления, упоминавшегося в главе 10.

```
@protocol ColorPickerDelegate;
@interface ColorPickerController : UIViewController
@property (nonatomic, weak) id <ColorPickerDelegate> delegate;
@end
@protocol ColorPickerDelegate
// color == nil при отмене
- (void) colorPicker:(ColorPickerController *)picker
    didSetColorNamed:(NSString *)theName
             toColor:(UIColor*)theColor;
@end
```

Когда в экземпляре класса `SettingsController` создается и настраивается экземпляр класса `ColorPickerController`, то он устанавливает себя также в качестве делегата для экземпляра класса `ColorPickerController`. Если теперь пользователь выберет цвет, то

экземпляру класса ColorPickerController станет известно, куда нужно отправить сообщение colorPicker:didSetColorNamed:toColor:, а именно: его делегату. Компилятор позволит это сделать, поскольку делегат принял протокол ColorPickerDelegate, как показано ниже.

```
- (void) dismissColorPicker: (id) sender {
   // пользователь нажал кнопку Done
   [self.delegate colorPicker:self
    didSetColorNamed:self.colorName
    toColor:self.color];
}
```

Если создать в среде Xcode проект из шаблона Utility Application, то можно обнаружить, что и в нем воплощается та же самая архитектура. Этот шаблон начинается с класса MainViewController, а завершается созданием класса FlipsideViewController. Когда объект типа FlipsideViewController готов прекратить свое существование, ему требуется отправить сообщение flipsideViewControllerDidFinish: обратно тому объекту, который его создал. Таким образом, в классе FlipsideViewController определяется протокол FlipsideViewControllerDelegate, требующий реализации метода flipsideViewControllerDidFinish: наряду со свойством delegate, обозначаемым как id <FlipsideViewControllerDelegate>.

Когда экземпляр класса MainViewController создает экземпляр класса FlipsideViewController, он устанавливает себя в качестве делегата для экземпляра класса FlipsideViewController. Он действительно может это сделать, поскольку в классе MainViewController принят протокол FlipsideViewControllerDelegate! Задача решена, и дело с концом. Если же у вас возникнут сомнения по поводу настройки шаблона делегата и протокола, создайте проект по шаблону Utility Application и тщательно изучите его.

Источники данных

Источник данных подобен делегату, за исключением того, что его методы предоставляют данные для отображения другого объекта. К числу основных классов Cocoa с источниками данных относятся UITableView, UICollectionView, UIPickerView и UIPageViewController. В каждом отдельном случае источник данных должен формально соответствовать протоколу с обязательными методами.

Некоторым начинающим программировать в системе iOS непонятно, зачем вообще нужен источник данных. Почему для этого недостаточно табличных данных или хотя бы некоторой фиксированной структуры данных? Дело в том, что такая архитектура способна нарушить универсальность. В то же время применение отдельного источника данных позволяет отделить объект, отображающий данные, от объекта, управляющего данными, а также высвободить последний для хранения и получения этих данных каким угодно способом (см. описание шаблона "модель–представление–контролллер" в главе 13). Единственное требование к источнику данных состоит в том, что он должен быстро предоставлять информацию, поскольку она будет запрашиваться у него в реальном времени, когда ее потребуется отобразить.

Не совсем понятно и то, чем источник данных отличается от делегата. Однако это сделано только ради сохранения универсальности, хотя и не является обязательным требованием. На самом деле нет никаких причин для того, чтобы источник данных и делегат не должны быть одним и тем же объектом, и чаще всего они могут быть таковыми. Как правило, методы

источника данных и делегата настолько тесно взаимодействуют, что трудно даже уловить их различие.

В приведенном ниже примере реализуется представление типа UIPickerView, дающее пользователю возможность выбрать день недели по названию (на английском языке в григорианском календаре). Два первых метода являются методами источника данных из класса UIPickerView, а третий — метод делегата их того же класса UIPickerView. Все три метода служат для того, чтобы предоставить содержимое представления, в котором выбираются дни недели.

```
- (NSInteger) numberOfComponentsInPickerView: (UIPickerView*) pickerView {
   return 1;
}

- (NSInteger) pickerView: (UIPickerView*) pickerView
   numberOfRowsInComponent: (NSInteger) component {
   return 7;
}

- (NSString*) pickerView:(UIPickerView*)pickerView
            titleForRow:(NSInteger)row
           forComponent:(NSInteger)component {
   NSArray* arr = @[@"Sunday",
                    @"Monday",
                    @"Tuesday",
                    @"Wednesday",
                    @"Thursday",
                    @"Friday",
                    @"Saturday"];
   return arr[row];
}
```

Действия

Действие — это сообщение, посылаемое экземпляром подкласса, производного от класса UIControl. Оно уведомляет о важном событии, происходящем в *элементе управления* пользовательского интерфейса (см. главу 7). Все подклассы, производные от класса UIControl, представляют простые интерфейсные объекты, с которыми может непосредственно взаимодействовать пользователь. К их числу относится кнопка (UIButton), переключатель (UISwitch), сегментированный элемент управления (UISegmentedControl), ползунок (UISlider) или текстовое поле (UITextField).

Самые важные пользовательские (или *управляющие*) события перечислены в разделе, посвященном событиям и константам документации на класс UIControl. В разных элементах управления реализуются разные управляющие события. Например, событие Value Changed сегментированного элемента управления обозначает, что пользователь выбрал другой сегмент, тогда как событие Touch Up Inside пользователь выбрал нажатием кнопки. Само управляющее событие не имеет никакого действия. Элемент управления реагирует на него визуально (например, нажатая кнопка выглядит нажатой), но ни с кем не обменивается информацией о наступившем событии. Если требуется выяснить, когда наступает управляющее событие, чтобы вовремя отреагировать на него в прикладном коде, необходимо сделать так, чтобы это управляющее событие инициировало сообщение о действии.

Этот механизм действует следующим образом. В элементе управления поддерживается внутренняя таблица диспетчеризации, в которой для каждого управляющего события может

быть любое количество пар "цель-действие", где *действие* — это селектор сообщений (имя метода), а *цель* — объект, которому посылается сообщение. Когда наступает управляющее событие, элемент управления обращается к таблице диспетчеризации, находит в ней пару "цель-действие", связанную с данным управляющим событием, а затем посылает каждое сообщение о действии соответствующей цели (рис. 11.1).

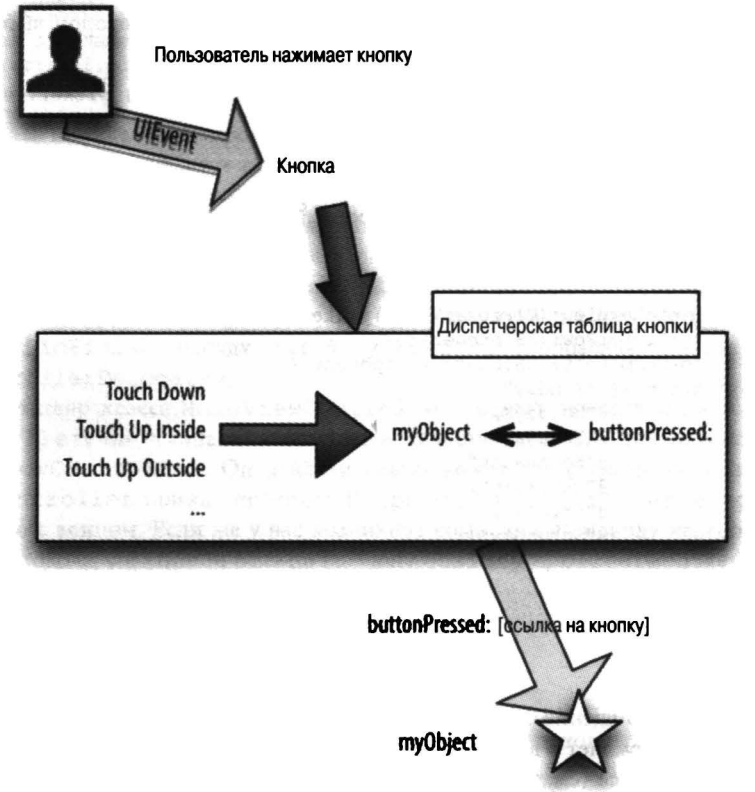


Рис. 11.1. Архитектура "цель-действие"

Манипулировать действием таблицы диспетчеризации действий в элементе управления можно двумя способами: установить связь с действием в nib-файле, как пояснялось в главе 7, или сделать это программно в прикладном коде. В последнем случае элементу управления посылается сообщение `addTarget:action:forControlEvents:`, где параметр `target:` обозначает объект, параметр `action:` — селектор, а параметр `ControlEvents:` — битовую маску (о том, как составляется битовая маска, см. в главе 1). В отличие от центра уведомлений, у элемента управления имеются также методы для интроспекции таблицы диспетчеризации.

Вернемся к примеру из главы 7. Напомним, что в этом примере имеется метод `buttonPressed:`, который должен вызываться, когда пользователь нажимает определенную кнопку в интерфейсе. Исходный код этого метода приведен ниже.

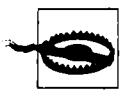
```
- (void) buttonPressed: (id) sender {
   UIAlertView* av = [[UIAlertView alloc] initWithTitle:@"Howdy!"
                                                 message:@"You tapped me."
                                                delegate:nil
```

```
                                        cancelButtonTitle:@"Cool"
                                        otherButtonTitles:nil];
    [av show];
}
```

Для того чтобы это произошло, в главе 7 была установлена связь с действием в nib-файле, а именно: событие касания кнопки `Touch Up Inside` было связано с методом `buttonPressed:` из класса `ViewController`. В действительности это означало формирование пары "цель–действие", где целью был экземпляр класса `ViewController`, владевший nib-файлом после его загрузки, тогда как действием — селектор `buttonPressed:`, а также ввод этой пары в таблицу диспетчеризации для управляющего события `Touch Up Inside`.

Вместо установления подобной связи в nib-файле то же самое можно было бы сделать непосредственно в прикладном коде. Допустим, что данная связь с действием не установлена и что вместо нее имеется связь с выходом, которая установлена от контроллера представления к кнопке и называется `button`. (На самом деле такая связь была установлена в главе 7.) В таком случае после загрузки nib-файла в контроллере представления можно настроить таблицу диспетчеризации кнопки следующим образом:

```
[self.button addTarget:self action:@selector(buttonPressed:)
                forControlEvents:UIControlEventTouchUpInside];
```



Управляющее событие может иметь несколько пар "цель–действие". Такую конфигурацию данного события можно сделать как намеренно, так и случайно. Снабдив управляющее событие парой "цель–действие" ненамеренно, но не удалив *существующую* пару "цель–действие", можно очень легко совершить ошибку и тем самым вызвать совершенно загадочное поведение. Так, если установить в nib-файле связь с действием *и* настроить таблицу диспетчеризации в прикладном коде, то нажатие кнопки приведет к тому, что метод `buttonPressed:` будет вызван *дважды*.

Сигнатура селектора действия может быть представлена в трех формах.

- Простейшая форма принимает два параметра:
  - Элемент управления, обычно обозначаемый как `id`.
  - Объект класса `UIEvent`, инициировавший управляющее событие.
- Более короткая и чаще всего применяемая форма без второго параметра. Ее примером служит метод `buttonPressed:`, принимающий единственный параметр `sender`. Когда метод `buttonPressed:` вызывается по сообщению о действии, исходящему от кнопки, параметр `sender` будет служить ссылкой на кнопку.
- Имеется и еще более короткая форма, в которой опущены оба параметра.

Что же собой представляет класс `UIEvent` и для чего он предназначен? *Событие касания* инициируется всякий раз, когда пользователь совершает какое-нибудь движение пальцем (опускает его на экран, перемещает по нему, поднимает от экрана). Объекты класса `UIEvent` относятся к числу низкоуровневых объектов, отвечающих за передачу событий касания приложению. По существу, объект типа `UIEvent` представляет собой временную метку (двойную) наряду с коллекцией (типа `NSSet`) событий касания (типа `UITouch`). Механизм действий намеренно скрывает все сложные детали формирования событий касания, но если выбрать получение объекта типа `UIEvent`, то при желании можно добраться до этих сложных деталей.

Любопытно, что ни один из параметров селектора действия не позволяет выяснить, какое именно управляющее событие привело к вызову текущего селектора действия! Например,

для того чтобы отличить управляющее событие Touch Up Inside от управляющего события Touch Up Outside, в соответствующих им парах "цель–действие" необходимо указать два разных обработчика действий. Если же передать эти события одному и тому же обработчику действий, то он не сумеет распознать, какое именно управляющее событие произошло.

Предупреждение для программистов в системе OS X

Если у вас имеется опыт разработки приложений в среде Сосоа под OS X, вы, вероятно, обратите внимание на некоторые основные отличия в реализации действий в системах OS X и iOS. В частности, у элемента управления в OS X имеется лишь одно действие, тогда как в iOS единственное событие может инициировать передачу многих сообщений о действиях нескольким целям. Кроме того, в системе OS X селектор сообщений о действиях принимает единственную форму, тогда как в системе iOS возможны три его формы.

Цепочка реагирующих элементов

Реагирующий элемент — это объект, которому известно, как получать непосредственно объекты типа UIEvent, упоминавшиеся в предыдущем разделе. Об этом ему известно потому, что он является экземпляром класса UIResponder или его подкласса. Изучая иерархию классов Сосоа, можно обнаружить, что практически любой класс, имеющий отношение к отображению информации на экране, является реагирующим элементом. Так, реагирующими элементами являются классы UIView, UIWindow, UIViewController и даже UIApplication. В iOS 5 и более поздних версиях этой операционной системы реагирующим элементом оказывается делегат.

Как следует из документации на класс UIResponder, в нем реализованы следующие четыре низкоуровневых метода для получения объектов типа UIEvent, связанных с событиями касания: touchesBegan:withEvent:, touchesMoved:withEvent:, touchesEnded:withEvent: и touchesCancelled:withEvent:. Эти методы вызываются с целью уведомить реагирующий элемент о событии касания. Реагирующий элемент первоначально уведомляется одним из этих методов о произошедшем касании, независимо от того, каким образом в прикладном коде принимаются события касания пользователем экрана, — даже если эти события *вообще* не принимаются в прикладном коде, поскольку среда Сосоа автоматически реагирует на касание без вмешательства со стороны прикладного кода.

Сначала в механизме для передачи этих событий принимается решение, какого реагирующего элемента коснулся пользователь. Методы hitTest:withEvent: и pointInside:withEvent: из класса UIView вызываются до тех пор, пока не будет найдено нужное представление, называемое *представлением проверки касания*. Затем вызывается метод sendEvent: из UIApplication, который, в свою очередь, вызывает метод sendEvent: из класса UIWindow, а тот — подходящий метод представления проверки касания (т.е. реагирующего элемента).

Реагирующие элементы приложения участвуют в так называемой *цепочке реагирующих элементов*, которая, по существу, связывает их в иерархию представлений. В частности, одно представление типа UIView может располагаться в другом, являющемся его родительским представлением, и так до тех пор, пока не будет достигнут класс UIWindow приложения (т.е. представление типа UIView, у которого отсутствует родительское представление). Цепочка реагирующих элементов выглядит снизу вверх следующим образом.

Исходное представление типа UIView (в данном случае представление проверки касания).

Контроллер типа UIViewController, управляющий этим представление типа UIView, если таковой имеется.

Родительское представление исходного представления типа UIView, а далее — его контроллер типа UIViewController, если таковой имеется. Это звено цепочки повторяется вверх по иерархии представлений до тех пор, пока не будет достигнуто следующее:

Класс UIWindow.

Класс UIApplication.

Делегат класса UIApplication.

Перекладывание ответственности

Цепочка реагирующих элементов может быть использована для того, чтобы предоставить реагирующему элементу возможность переложить ответственность за обработку события касания на другой элемент. Если реагирующий элемент принимает событие касания и *не* может его обработать, это событие может быть передано вверх по цепочке реагирующих элементов в поисках того реагирующего элемента, который *может* его обработать. Перекладывание ответственности происходит в одном из двух основных случаев.

Реагирующий элемент не реализует подходящий метод.

Реагирующий элемент реализует подходящий метод для вызова метода суперкласса.

Например, в простейшем классе UIView отсутствует собственная реализация методов обработки событий касания. Поэтому такое событие по умолчанию "проскакивает" представление типа UIView, даже если это представление проверки касания, и передается вверх по цепочке реагирующих элементов в поисках того элемента, который на него отреагирует. В некоторых случаях оказывается вполне логичным и оправданным переложить ответственность за обработку события касания на основное фоновое представление или даже на контроллер типа UIViewController, который им управляет.

Одно из моих приложений представляет собой игру в составление картинки-загадки, где прямоугольная фотография разделяется на мелкие части, которые затем перемешиваются в произвольном порядке. В качестве фонового представления в этом приложении служит подкласс Board, производный от класса UIView. Отдельные части картинки-загадки представлены типичными объектами класса UIView и являются дочерними представлениями по отношению к представлению типа Board. О том, как должна реагировать на касание каждая часть картинки-загадки, известно в представлении типа Board, где также известно общее расположение частей картинки-загадки. Следовательно, эти части не должны содержать никакой логики для обнаружения касания. Это обстоятельство позволило мне выгодно воспользоваться перекладыванием ответственности в цепочке реагирующих элементов, чтобы не реализовывать в частях картинки-загадки никаких методов обработки событий касания, а передавать эти события вверх по цепочке представлению типа Board, где обнаруживается касание и обрабатывается нажатие, а затем выбранной таким образом части картинки-загадки сообщается, что именно ей нужно делать. Пользователю, конечно, ничего об этом не известно. Он просто касается части картинки-загадки, а она реагирует на его касание.

Действия без цели

Действие без цели — это пара "цель–действие", в которой цель оказывается пустой (nil). Это означает, что целевой объект для такого действия заранее не назначен. Вместо этого соблюдается следующее правило: начиная с представления проверки касания (т.е. представления, с которым взаимодействует пользователь), в среде Cocoa осуществляется поиск вверх по цепочке реагирующих элементов объекта, способного отреагировать на сообщение о действии. Допустим, что в прикладном коде настраивается кнопка, как показано ниже.

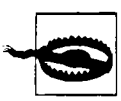
```
self.button addTarget:nil action:@selector(buttonPressed:)
     forControlEvents:UIControlEventTouchUpInside];
```

Это и есть действие без цели. Так что же произойдет, если пользователь нажмет кнопку? Прежде всего в среде Cocoa будет проверено, сможет ли сам объект типа `UIButton` отреагировать на сообщение `buttonPressed:`. Если не сможет, то поиск продолжится в родительском представлении типа `UIView` этого объекта, и так далее вверх по цепочке реагирующих элементов. Если реагирующий элемент, способный обработать событие нажатия кнопки, будет найден, сообщение о действии посылается данному объекту, а иначе оно остается необработанным (без всяких взысканий).

Таким образом, в упоминавшемся ранее приложении, где ссылка `self` обозначает контроллер, владеющий представлением, содержащим кнопку, а в классе этого контроллера фактически реализуется метод `buttonPressed:`, нажатие кнопки приведет к вызову метода `buttonPressed:`!

Для того чтобы создать действие без цели в nib-файле, следует установить соединение с прокси-объектом первого реагирующего элемента в стыковочном блоке. Именно для этой цели и предназначен данный объект! Первый реагирующий элемент не является настоящим объектом с известным классом, поэтому, прежде чем устанавливать связь между ним и действием, придется определить сообщение о действии в прокси-объекте первого реагирующего элемента следующим образом.

1. Выберите прокси-объект первого реагирующего элемента в nib-файле и перейдите к инспектору атрибутов.
2. Появится (возможно, пустая) таблица с определяемыми пользователем действиями без цели первого реагирующего элемента. Щелкните на кнопке Plus и присвойте сигнатуру новому действию. Она должна принимать единственный параметр, а следовательно, завершаться двоеточием.
3. Нажмите клавишу <Control> и перетащите курсор от элемента управления (в данном случае — кнопки типа `UIButton`) к прокси-объекту первого реагирующего элемента, чтобы указать пустонацеленное действие с указанной сигнатурой.



Применение понятия *первый реагирующий элемент* в среде Cocoa кажется не совсем понятным. Так, объекту произвольного реагирующего элемента можно присвоить состояние формально первого реагирующего элемента, послав ему сообщение `becomeFirstResponder`, при условии, что этот реагирующий элемент возвращает логическое значение `YES` из метода `canBecomeFirstResponder`. Вследствие этого объект не становится первым реагирующим элементов для обработки действий без цели! Поиски в среде Cocoa реагирующего элемента, способного обработать действие без цели, начинаются с того элемента управления, с которым пользователь взаимодействует (через представление проверки касания), а затем продолжается вверх по цепочке реагирующих элементов.

Сильная зависимость от событий

Прикладной код выполняется только потому, что среда Cocoa посылает ему событие и в нем установлен метод, готовый принять событие. Среда Cocoa обладает потенциалом посылать много событий, уведомляющих о действиях пользователя и о каждой стадии времени

действия приложения и его объектов, а также запрашивающих приложение, что следует делать дальше. Для того чтобы получать события, на которые нужно реагировать, прикладной код должен быть снабжен методами, служащими в качестве *точек входа*, т.е. методами, написанными с таким именем и в таком классе, чтобы их можно было вызывать по событиям в среде Cocoa. Действительно, нетрудно представить, что прикладной код зачастую будет состоять почти полностью из точек входа.

Данное обстоятельство представляет одно из самых главных затруднений для программирующих для операционной системы iOS. Недостаточно знать, как написать прикладной код, но нужно уметь правильно разделить на части и распределить его, учитывая те моменты, когда среда Cocoa предполагает обратиться к нему. Прежде чем вы напишете хотя бы одну строку прикладного кода, для вас уже будет составлена в основном каркасная структура класса, чтобы подготовить его к приему событий, которые собирается посылать ему среда Cocoa.

Допустим, пользовательский интерфейс приложения для мобильного телефона iPhone состоит из табличного вида, что вполне возможно. Для этого в коде приложения, скорее всего, будет использоваться подкласс, производный от класса `UITableViewController`, который в свою очередь является производным от класса `UIViewController`, а экземпляр этого подкласса, производного от класса `UITableViewController`, будет владеть и управлять табличным представлением. Этот же подкласс, вероятнее всего, будет использоваться в качестве источника данных и делегата табличного представления. Следовательно, в этом единственном классе, скорее всего, потребуется реализовать *как минимум* следующие методы.

`initWithCoder:` или `initWithNibName:bundle:`[1]

Это методы времени действия контроллера типа `UIViewController`, где выполняется специальная инициализация экземпляра.

`viewDidLoad`

Это метод времени действия контроллера типа `UIViewController`, где выполняется инициализация, связанная с представлением.

`viewDidAppear:`

Это метод времени действия контроллера типа `UIViewController`, где устанавливаются состояния, требующиеся только при отображении представления на экране. Так, если нужно зарегистрироваться для получения уведомлений или установить таймер, то все это, скорее всего, придется сделать именно в данном методе.

`viewDidDisappear:`

Это метод времени действия контроллера типа `UIViewController`, где выполняются операции, обратные тому, что делается в методе `viewDidAppear:`. Например, в этом методе, скорее всего, должно производиться снятие с регистрации или перевод в недействительное состояние таймера, установленного в методе `viewDidAppear:`.

`supportedInterfaceOrientations`

Это метод запроса контроллера типа `UIViewController`, где указываются допустимые ориентации мобильного устройства для основного представления данного контроллера.

[1] Двоеточие означает, что метод получает параметры. — *Примеч. ред.*

numberOfSectionsInTableView:

tableView:numberOfRowsInSection:

tableView:cellForRowAtIndexPath:

Это методы запроса источника данных типа `UITableView`, где указывается содержимое таблицы.

tableView:didSelectRowAtIndexPath:

Это метод действия пользовательского делегата класса `UITableView`, где организуется реагирование на касание строки таблицы.

dealloc

Это метод времени действия объекта типа `NSObject`, где выполняется очистка этого объекта из памяти по окончании срока действия объекта.

Допустим также, что метод `viewDidAppear:` был вызван для регистрации на получение конкретного уведомления и установку таймера. В таком случае у данного уведомления и таймера имеется свой селектор, если только не используется блок, а следовательно, придется реализовать методы, описываемые этими селекторами.

Таким образом, в рассматриваемом здесь примере приложения уже насчитывается около десятка методов, наличие которых считается нормой. Однако это не методы самого приложения, поскольку они вообще в нем не вызываются. Это методы среды Cocoa, введенные в приложение с тем, чтобы каждый из них вызывался в подходящий момент во время выполнения приложения.

Такую логику построения прикладной программы совсем не просто понять! Не сочтите это за критику в адрес среды Cocoa, поскольку трудно себе представить иной принцип ее действия. Если быть объективным, то следует сказать, что прикладную программу, написанную в среде Cocoa, даже если разработать ее *самостоятельно*, очень трудно читать, поскольку она состоит из многочисленных несвязанных точек входа, каждая из которых имеет свое смысловое назначение и вызывается в свой заданный момент, который не совсем очевиден из ее анализа.

Для того чтобы понять назначение и функции гипотетического класса, нужно *уже* знать, например, когда вызывается метод `viewDidAppear:` и каким образом он обычно используется. В противном случае очень трудно доискаться логики и поведения прикладной программы, не говоря уже о том, как интерпретировать даже то, что уже в ней обнаружено. Эти трудности значительно возрастают при попытке прочитать чужой прикладной код (это одна из тех причин, по которым примеры кода оказываются не всегда полезными для начинающих, как пояснялось в главе 8).

Глядя на код прикладной программы для операционной системы iOS (даже своей собственной), можно легко прийти в недоумение от одного только вида всех тех методов, которые автоматически вызываются из среды Cocoa при разных обстоятельствах. По мере накопления опыта вы, конечно, научитесь быстро распознавать в коде такие элементы, как замещаемые методы из класса `UIViewController`, делегат табличного представления и методы источника данных. С другой стороны, никакой опыт не подскажет вам, что определенный метод вызывается в виде действия кнопки или по уведомлению. Здесь немалую помощь оказывают комментарии, поэтому настоятельно рекомендуется снабжать (обильно если, требуется) комментариями каждый метод в приложении, разрабатываемом для операционной системы iOS, указывая назначение метода и при каких обстоятельствах он должен вызываться, особенно если это точка входа, в которой он вызывается из самой среды Cocoa.

При написании приложений в среде Cocoa самыми распространенными оказываются не программные ошибки в коде, а размещение кода не в том месте. Нередко прикладной код не выполняется вообще, не вовремя или не в том порядке следования его фрагментов. Подобные жалобы можно нередко можно встретить на форумах пользователей среды Cocoa в Интернете. Ниже приведены характерные тому примеры, взятые из одного из таких форумов, где они появились в течение буквально двух дней.

- "Возникает задержка между моментом появления моего представления и моментом появления нужной надписи на кнопке". Это объясняется тем, что прикладной код, устанавливающий надпись на кнопке, размещается в методе `viewDidAppear:`, а это *слишком поздно*. Выполнение этого кода должно начаться раньше, возможно, в методе `viewWillAppear:`.
- "Мои дочерние представления располагаются программно в коде, но в конечном итоге все они действуют неверно". Это объясняется тем, что прикладной код размещает дочерние представления в методе `viewDidLoad`, а это *слишком рано*. Выполнение этого кода должно начаться позже, когда уже определены размеры представления.
- "Мое представление все равно поворачивается, даже если метод `supportedInterfaceOrientations` моего контроллера представления не разрешает это делать". Дело в том, что метод `supportedInterfaceOrientations` реализован *не в том классе*. Его нужно реализовать в классе `UINavigationController`, содержащем контроллер представления.
- "Я устанавливаю связь с действием для события `Value Changed`, наступающего в текстовом поле, но мой код не вызывается, когда пользователь редактирует данные в этом поле". Дело в том, что связь была установлена *не с тем действием*. Ведь текстовое поле инициирует событие `Editing Changed`, а не `Value Changed`.

Дело усложняется еще и тем, что точно неизвестно, когда именно точка входа будет вызвана. В документации может быть дано лишь общее представление об этом, но, как правило, никак не гарантируется, когда и в каком порядке появятся события. Ваше собственное или почерпнутое из документации представление о происходящем может не совсем отвечать тому, что происходит в действительности. Ведь в прикладном коде могут быть инициированы непреднамеренные события, а в документации не разъясняется, когда именно уведомление будет отправлено.

Кроме того, в среде Cocoa возможна программная ошибка, в результате которой события вызываются совсем не так, как описано в документации. Поскольку исходный код среды Cocoa закрыт для доступа, то выяснить скрытые подробности ошибки вряд ли удастся. Поэтому при разработке приложения рекомендуется обильно снабдить его код средствами простейшей самодиагностики, воспользовавшись функцией `NSLog()` (см. главу 9). Тестируя код, внимательно следите за выводом на консоль и проверяйте логичность выводимых сообщений. Вы будете удивлены тем, что обнаружите.

Например, во время разработки одного из своих приложений я неожиданно обнаружил, что в подклассе, производном от класса `UIViewController`, метод `viewDidLoad` вызывался дважды во время запуска приложения на выполнение, а этого не должно было быть. Правда, код моего приложения был обильно снабжен вызовами функции `NSLog()`, иначе я так бы и не выявил ошибку. Введя дополнительные вызовы функции `NSLog()` в свой код, я обнаружил, что метод `viewDidLoad` вызывался в тот момент, когда еще не завершилось выполнение метода `awakeFromNib`, чего на самом деле не должно было быть. Причиной тому стала моя ошибка: я обращался к свойству `view` контроллера представления во время выполнения метода `awakeFromNib`, что и приводило к вызову метода `viewDidLoad`. Как только я исправил свою ошибку, упомянутая выше проблема исчезла.

Отложенное выполнение

В некоторых видах прикладного кода среде Сосоа может быть предписано, что нужно делать. Однако среда Сосоа — это черный ящик, и поэтому, что и особенно когда в ней будет сделано, не поддается контролю. Прикладной код может выполняться в ответ на некоторое событие, но, в свою очередь, он может инициировать новое событие или цепочку событий. Иногда это приводит к неудачному исходу: аварийному завершению или невыполнению средой Сосоа того, что ей было предписано сделать. Одна из главных причин подобных затруднений заключается в самой цепочке инициируемых событий. Иногда нужно просто выйти из этой цепочки на какое-то время и подождать до тех пор, пока все не установится, а затем продолжить.

Для этого имеется способ, называемый *отложенным выполнением*. Команда сделать что-нибудь дается среде Сосоа не сразу, а немного позже, когда все установится. Задержка, которая для этого потребуется, может быть короткой ли вообще нулевой, но она делается для того, чтобы дать среде Сосоа возможность завершить какую-нибудь внутреннюю операцию, например, расположение пользовательского интерфейса. В принципе, отложенное выполнение позволяет завершить текущий цикл исполнения, заполнить и развернуть весь текущий стек вызовов, прежде чем продолжить выполнение прикладного кода.

Пользоваться отложенным выполнением вам, скорее всего, придется намного чаще, чем вы предполагаете. С опытом у вас выработается особое чутье на то, когда отложенное выполнение может разрешить возникающие у вас затруднения. В своем коде я обычно реализую отложенное выполнение в трех следующих методах.

`performSelector:withObject:afterDelay:`

Этот метод из класса `NSObject` упоминался в конце главы 10. Он имеет ограничение на сигнатуру селектора, принимая только один параметров или вообще не принимая параметры. В связи с этим, возможно, придется немного реорганизовать свой код.

`dispatch_after`

Этот метод упоминался в главе 3. Он принимает в качестве параметра блок, а не селектор, что упрощает и повышает удобочитаемость кода.

`dispatch_async`

Нередко искомая задержка может быть не больше нуля. В данном случае предпринимается попытка пропустить следующий шаг до тех пор, пока не выяснятся последствия выполнения предыдущего шага. Поэтому достаточно подождать до тех пор, пока ничего не будет происходить. Это можно сделать с помощью метода `dispatch_async` в той же самой очереди, где все происходит в настоящий момент, а именно: в главной очереди.

Во всех трех рассмотренных выше случаях нужная операция будет выполнена позднее. Это означает, что построчное выполнение кода намеренно нарушается. Следовательно, вызов, выполнение которого отложено, будет последним в своем методе (или блоке), но возвращать значение в результате его исполнения нельзя.

В приведенном ниже примере, взятом из одного из моих приложений, код реагирует на выбор пользователем строки таблицы путем касания, создавая и отображая новое табличное представление:

```
- (void) tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
```

```
    TracksViewController *t =
        [[TracksViewController alloc]
        initWithMediaItemCollection:(self.albums)[indexPath.row]];
    [self.navigationController pushViewController:t animated:YES];
}
```

К сожалению, невинный, на первый взгляд, вызов метода `initWithMediaItemCollection:` из класса `TracksViewController` может быть выполнен не сразу. Поэтому выполнение приложения останавливается с подсвеченной строкой в таблице. Несмотря на то что это происходит недолго, все же это заметно для пользователя. Для того чтобы придать этой задержке ощущение какой-то деятельности, я наделил подкласс, производный от класса `UITableViewCell`, функцией отображения вертящегося индикатора активности при выборе строки в таблице:

```
- (void)setSelected:(BOOL)selected animated:(BOOL)animated {
    if (selected) {
        [self.activityIndicator startAnimating]; // отобразить и вращать
    } else {
        [self.activityIndicator stopAnimating];  // скрыть
    }
    [super setSelected:selected animated:animated];
}
```

Тем не менее вертящийся индикатор активности вообще не появляется и не вращается. Объясняется это тем, что события запинаются друг о друга. В частности, метод `setSelected:animated:` из класса `UITableViewCell` не вызывается до тех пор, пока не завершится метод делегата `tableView:didSelectRowAtIndexPath:` из класса `UITableView`. Однако задержка, которую мы пытаемся скрыть, происходит во время выполнения метода `tableView:didSelectRowAtIndexPath:`. Все дело в том, что этот метод не завершается достаточно быстро.

Здесь на помощь приходит отложенное выполнение! Достаточно переписать метод `tableView:didSelectRowAtIndexPath:` таким образом, чтобы он завершался немедленно. Следовательно, метод `setSelected:animated:` запускается на выполнение сразу и вертящийся индикатор активности появляется на экране. Отложенное выполнение применяется для вызова метода `initWithMediaItemCollection:` в дальнейшем, когда отображение пользовательского интерфейса стабилизируется.

```
- (void) tableView:(UITableView *)tableView
      didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    // небольшая задержка, позволяющая раскрутиться вертящемуся индикатору
    double delayInSeconds = 0.1;
    dispatch_time_t popTime =
        dispatch_time(DISPATCH_TIME_NOW, delayInSeconds * NSEC_PER_SEC);
    dispatch_after(popTime, dispatch_get_main_queue(), ^(void){
        TracksViewController *t =
            [[TracksViewController alloc]
            initWithMediaItemCollection:(self.albums)[indexPath.row]];
        [self.navigationController pushViewController:t animated:YES];
    });
}
```

ГЛАВА 12

Методы доступа и управление памятью

В этой главе мы снова рассмотрим три основных аспекта Objective-C, представленных вкратце в главе 5: методы доступа, механизм доступа к значениям по ключам и свойства. Но на этот раз мы обсудим их углубленно и досконально. Это будет сделано с учетом их особой роли в одном из самых важных и ключевых аспектов среды Cocoa: управлении памятью для хранения экземпляров классов языка Objective-C.

Методы доступа

Метод доступа служит для получения и установки значения переменной экземпляра. Метод доступа, получающий значение переменной экземпляра, называется get-*методом*, а метод доступа, устанавливающий значение переменной экземпляра, — set-*методом*.[1]

Методы доступа важны отчасти потому, что переменные экземпляра по умолчанию защищены (см. главу 5), тогда как открыто объявляемые методы — общедоступны. Без открытых методов доступа защищенная переменная экземпляра оказывается недоступной для любого объекта, в классе (или суперклассе) которого эта переменная экземпляра не объявлена.

Из сказанного выше возникает искушение сделать следующий вывод: нет никакой нужды создавать метод доступа к переменной экземпляра, не предназначенной для общего доступа, и в какой-то степени такой вывод обоснован. Но в современной версии языка Objective-C создать методы доступа так же просто, как и объявить переменную экземпляра: достаточно объявить свойство, чтобы переменная экземпляра автоматически появилась вместе с методами доступа к ней, т.е. без единой строки написанного вручную кода методов доступа. Если же вы напишете такой код, то сможете согласованно решать дополнительные задачи, связанные со значением переменной экземпляра.

Для методов доступа приняты соглашения об именовании, которые следует соблюдать. Эти соглашения просты и поясняются ниже.

[1] Двоеточие означает, что метод получает параметры. — *Примеч. ред.*

Get-метод

Имя метода установки должно начинаться с префикса set, после которого следует начинающийся с прописной буквы вариант имени переменной экземпляра (без начального знака подчеркивания, если таковой присутствует в имени переменной экземпляра). Метод доступа должен принимать единственный параметр: новое значение, присваиваемое переменной экземпляра. Так, если переменная экземпляра называется myVar (или _myVar), то метод установки ее значения должен называться setMyVar:.

Set-метод

Этот метод должен называться по имени переменной экземпляра (без начального знака подчеркивания, если таковой присутствует в имени переменной экземпляра). Такое обозначение не вызовет никаких недоразумений ни у вас, ни у компилятора, поскольку имена переменной экземпляра и get-метода для получения ее значения используются в совершенно разных контекстах. Так, если переменная экземпляра называется myVar (или _myVar), то метод установки ее значения должен называться MyVar.

Если переменная экземпляра принимает логическое значение `BOOL`, то в начале имени get-метода можно дополнительно указать префикс is (например, get-метод для получения значения переменной экземпляра `showing` или `_showing` может называться `isShowing`), хотя делать это совсем не обязательно.

Соглашения об именовании методов доступа позволяют языковым средствам Objective-C вызывать эти методы только по одному имени. Доступ к значениям по ключам начинается с символьной строки и продолжается вызовом методов доступа. Кроме того, для вызова метода доступа можно использовать имя свойства. Все это становится возможным благодаря принятым соглашениям об именовании методов доступа.

Более того, соглашения об именовании, по существу, обеспечивают методам доступа самостоятельное существование независимо от любой переменной экземпляра. Если методы доступа уже имеются, то нет таких правил, которые обязывали бы называть их по имени настоящих переменных экземпляра! Напротив, можно намеренно создать методы `myVar` и `setMyVar:` в отсутствие переменной экземпляра `myVar` (или `_myVar`).

Это может быть сделано, например, для того, чтобы методы доступа подспудно выполняли совсем другие функции, маскируя настоящее имя переменной экземпляра, если она существует. Тем не менее все зависящие от методов доступа языковые средства Objective-C продолжают действовать благодаря принятым соглашениям об именовании. По существу, методы доступа служат фасадом, за которым от вызывающего кода скрываются любые внутренние подробности о возможном существовании определенных переменных экземпляра.

Доступ к значениям по ключам

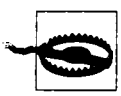
В среде Cocoa имя метода доступа получается из символьной строки во время выполнения с помощью механизма, называемого *доступом к значениям по ключам*, или сокращенно KVC, где *ключ* — это символьная строка типа `NSString` с именем значения, к которому осуществляется доступ. В основу механизма доступа к значениям по ключам положен неформальный протокол (по существу, категория) `NSKeyValueCoding`, которому соответствует класс `NSObject`, а следовательно, каждый объект. Тема доступа к значениям по ключам довольно обширна, и поэтому за дополнительными сведениями обращайтесь к документации *Key-Value Coding Programming Guide*, предоставляемой компанией Apple, где эта тема освещена наиболее полно.

К числу основных методов доступа к значениям по ключам относятся `valueForKey:` и `setValue:forKey:`. При вызове одного из этих методов для объекта осуществляется интроспекция этого объекта. Проще говоря, сначала находится подходящий селектор. Если он отсутствует, то доступ к переменной экземпляра осуществляется непосредственно.

Класс считается *совместимым* с механизмом доступа к значениям по заданному ключу, если он реализует методы или обладает переменной экземпляра для доступа по этому ключу. Всякая попытка получить доступ к ключу, для которого класс несовместим с доступом к значениям по ключам, приведет во время выполнения к исключению со следующим сообщением: "Этот класс несовместим с доступом к значениям по ключу имяКлюча". (Последнее слово в этом сообщении об ошибке является символьной строкой, обозначающей, хотя и без кавычек, ключ, ставший причиной неисправности.) Допустим, что имеется следующий вызов метода доступа:

```
[myObject setValue:@"Hello" forKey:@"greeting"];
```

Сначала в этом вызове осуществляется поиск метода `setGreeting:` в классе `myObject`. Если этот метод существует, то он вызывается, и в качестве аргумента ему передается строковое значение `@"Hello"`. При неудачном исходе вызова этого метода строковое значение `@"Hello"` присваивается непосредственно переменной экземпляра `greeting` (или `_greeting`), если она присутствует в классе `myObject`. Если же и эта операция присваивания завершится неудачно, то генерируется исключение и выполнение приложения завершается аварийно.



Механизм доступа к значениям по ключам позволяет полностью обойти защищенность переменной экземпляра! Среде Cocoa известно, что вам как программисту это может и не понадобиться, поэтому в ней предоставляется метод класса `accessInstanceVariablesDirectly`, который можно переопределить таким образом, чтобы он возвращал логическое значение `NO` вместо логического значения `YES` по умолчанию.

В качестве значения обоим методам, `valueForKey:` и `setValue:forKey:`, требуется объект. В сигнатуре метода доступа, а в его отсутствие — в самой переменной экземпляра, возможно, и нельзя использовать объект в качестве значения, и поэтому механизм доступа к значениям по ключам автоматически выполняет соответствующее преобразование. В частности, числовые типы данных, включая и `BOOL`, выражаются в виде объекта типа `NSNumber`, а остальные типы данных (например, `CGRect` и `CGPoint`) — в виде объекта типа `NSValue`. Еще одной парой полезных методов являются `dictionaryWithValuesForKeys:` и `setValuesForKeysWithDictionary:`, которые позволяют получать и устанавливать многие пары "ключ-значение" средствами класса `NSDictionary` в одной команде.

По существу, механизм доступа к значениям по ключам позволяет во время выполнения решить, исходя из объекта типа `NSString`, какой именно метод доступа следует вызывать. Используя объект типа `NSString` вместо вызова метода доступа, можно обойтись без проверки на стадии компиляции на предмет отправляемого сообщения. Более того, механизму KVC ничего неизвестно о конкретном классе объекта, к которому происходит обращение. Так, можно послать сообщение `valueForKey:` любому объекту и успешно получить результат, при условии, что класс этого объекта совместим с механизмом KVC по заданному ключу. Это дает возможность обойтись без проверки во время компиляции на предмет объекта, которому посылается сообщение. Это довольно весомые преимущества механизма доступа к значениям по ключам, благодаря которым я сам часто пользуюсь этим механизмом.

Рассмотрим практический пример применения механизма доступа к значениям по ключам в прикладном коде. В моем приложении flashcard имеется класс Term, представляющий латинский термин, и в нем определено немало переменных экземпляра. На каждой карточке отображается один латинский термин, а значения переменных его экземпляра — в разных текстовых полях. Если пользователь коснется любого из трех текстовых полей, ему должен быть предоставлен интерфейс для перехода от текущего латинского термина к следующему термину, значение которого отличается от присутствующего в данном текстовом поле. Следовательно, прикладной код остается одним и тем же для всех трех текстовых полей. Единственное отличие заключается в выборе подходящей переменной экземпляра в поисках следующего термина для отображения. Безусловно, выразить такой параллелизм проще всего с помощью доступа к значениям по ключам, как показано ниже.

```
NSInteger tag = g.view.tag;
   // переменная tag служит признаком выбранного текстового поля
NSString* key = nil;
switch (tag) {
    case 1: key = @"lesson"; break;
    case 2: key = @"lessonSection"; break;
    case 3: key = @"lessonSectionPartFirstWord"; break;
}
// получить текущее значение из соответствующей переменной экземпляра
NSString* curValue = [[self currentCardController].term valueForKey: key];
// ...
```

Целый ряд встроенных классов Cocoa позволяет особым образом применять механизм доступа к значениям по ключам. Ниже приведены характерные тому примеры.

- Если объекту класса NSArray посылается сообщение valueForKey:, он пересылает его каждому из своих элементов и возвращает новый массив, состоящий из полученных результатов, изящным и кратким (подобно дешевой карте) способом. Аналогичным образом ведет себя и класс NSSet.
- В классе NSDictionary реализован метод valueForKey: в качестве альтернативы методу objectForKey: (он особенно полезен при наличии массива словарей типа NSArray). Аналогичным образом метод setValue:forKey: служит в классе NSMutableDictionary аналогом метода setObject:forKey:, за исключением того, что его параметр value: может принимать пустое значение nil. В этом случае вызывается метод removeObject:forKey:.
- В классе NSSortDescriptor для сортировки массива типа NSArray каждому элементу этого массива посылается сообщение valueForKey:. Благодаря этому упрощается сортировка массива словарей по значению конкретного словарного ключа или массива объектов по значению конкретной переменной экземпляра.
- Классы CALayer и CAAnimation позволяют применять механизм доступа к значениям по ключам для определения и извлечения значений произвольных ключей, как из словаря. Это удобно для присоединения данных идентификации и конфигурации к одному из экземпляров этих классов.
- Класс NSManagedObject, применяемый вместе с базой данных Core Data, гарантированно совместим с механизмом доступа к значениям по ключам для атрибутов, настраиваемых в модели сущностей. Следовательно, доступ к этим атрибутам обычно осуществляется с помощью методов valueForKey: и setValue:forKey:.

Механизм KVC и выходы

Механизм доступа к значениям по ключам положен в основу принципа действия связей с выходами (см. главу 7). Имя выхода представлено символьной строкой в nib-файле. Механизм доступа к значениям по ключам превращает символьную строку в критерий поиска подходящих методов доступа.

Допустим, имеется класс `MyClass` с переменной экземпляра `myVar` и в nib-файле установлена связь из выхода переменной `myVar` экземпляра, представляющего этот класс в nib-файле, к nib-объекту класса `MyOtherClass`. При загрузке nib-файла имя выхода `myVar` преобразуется в имя метода `setMyVar:`, а затем метод `setMyVar:` экземпляра класса `MyClass`, если он существует, вызывается с экземпляром класса `MyOtherClass` в качестве параметра. Таким образом, в качестве значения переменной `myVar` экземпляра класса `MyClass` устанавливается экземпляр класса `MyOtherClass` (см. рис. 7.7).

Если что-нибудь пойдет не так при проверке на совпадение имен выхода в nib-файле и переменной экземпляра или метода доступа в классе, то при загрузке nib-файла во время выполнения любая попытка воспользоваться в среде Cocoa механизмом доступа к значениям по ключам, чтобы установить значение в объекте на основании имени выхода, потерпит неудачу. В конечном в итоге возникнет исключение, уведомляющее о том, что класс несовместим с механизмом доступа к значениям по заданному ключу (в данном случае — по имени выхода). Это означает, что во время загрузки nib-файла в приложении произойдет аварийный сбой. Нечто подобное может, вероятнее всего, произойти потому, что сначала выход формируется правильно, а затем в классе происходит изменение имени (или вообще удаление) переменной экземпляра или метода доступа (см. раздел "Неправильная конфигурация выхода" в главе 7).

С другой стороны, имена методов доступа не следует употреблять в тех методах, которые не предназначены для доступа. Если вернуться к приведенному выше примеру класса `MyClass` и переменной экземпляра `myVar`, то вряд ли в этом классе понадобился бы метод `setMyVar:`, если он не предназначен для доступа к переменной экземпляра `myVar`. Если бы в этом классе действительно имелся такой метод, он вызывался бы при загрузке nib-файла, а механизм доступа к значениям по ключам попытался бы разрешить имя выхода `myVar` в nib-файле. Следовательно, данному методу был бы передан экземпляр класса `MyOtherClass`. В этом нет никакой ошибки, но экземпляр класса `MyOtherClass` не был бы присвоен переменной экземпляра `myVar`, поскольку `setMyVar:` не является методом доступа. В итоге ссылки в прикладном коде на переменную экземпляра `myVar` оказались бы пустыми (`nil`). Метод `setMyVar:` выполнял бы роль ложного фасада, препятствуя установке значения переменной экземпляра `myVar` во время загрузки nib-файла. Как ни странно, подобные ошибки совершаются довольно часто.

Пути к ключам

Путь к ключу позволяет выстраивать ключи цепочкой в одном выражении. Если объект относится к классу, совместимому с механизмом доступа к значениям по одному ключу, а само значение этого ключа является объектом класса, совместимого с механизмом доступа к значениям по другому ключу, то эти ключи можно связать в цепочку, вызвав методы `valueForKeyPath:` и `setValue:forKeyPath:`. Символьная строка с путями к ключам похожа на последовательный ряд имен ключей, соединенных записью через точку. Например, выражение `valueForKeyPath:@"key1.key2"`, по существу, означает, что сначала метод `valueForKey:` вызывается с ключом `@"key1"` для получателя сообщений, а затем тот же самый метод `valueForKey:` вызывается для объекта, возвращаемого в результате первого вызова, но на этот раз с ключом `@"key2"`.

В качестве примера, иллюстрирующего такую короткую запись, допустим, что у объекта myObject имеется переменная экземпляра theData, содержащая массив словарей, в котором у каждого словаря имеются ключи name и description. Ниже показано, как конкретное значение переменной экземпляра theData выводится с помощью функции NSLog().

```
(
  {
      description = "The one with glasses.";
      name = Manny;
  },
  {
      description = "Looks a little like Governor Dewey.";
      name = Moe;
  },
  {
      description = "The one without a mustache.";
      name = Jack;
  }
)
```

В таком случае в результате вызова [myObject valueForKeyPath: @"theData.name"] возвращается массив, состоящий из символьных строк @"Manny", @"Moe" и @"Jack". Если вам непонятен такой результат, вернитесь на несколько абзацев назад к перечню примеров применения механизма KVC, среди которых рассматривается реализация метода valueForKey: в классе NSDictionary. Вспомните также обсуждение в главе 7 определяемых пользователем атрибутов времени выполнения. Ключ, вводимый в nib-файл при определении атрибута времени выполнения в инспекторе идентичности, на самом деле является путем к ключу.

Методы доступа к массиву

Механизм доступа к значениям по ключам позволяет реализовать в объекте ключ таким образом, как будто его значением является массив (или множество), даже если это на самом деле не так. Это аналогично сказанному ранее о выполнении методами доступа роли фасада, выдвигающего на передний план имя переменной экземпляра и скрывающего все сложные подробности работы такого механизма. В качестве примера в приведенном ниже фрагменте кода методы доступа вводятся в класс упоминавшегося ранее объекта myObject.

```
- (NSUInteger) countOfPepBoys {
   return [self.theData count];
}

- (id) objectInPepBoysAtIndex: (NSUInteger) ix {
   return (self.theData)[ix];
}
```

Благодаря реализации методов типа countOf... и objectIn...AtIndex: механизму доступа к значениям по ключам предписывается действовать таким образом, как будто заданный ключ (в данном случае — @"pepBoys") существует и является массивом. Попытка извлечь значение ключа @"pepBoys" с помощью механизма доступа к значениям по ключам будет успешной, а в итоге возвратится объект, который можно рассматривать как массив, хотя на самом деле это объект-заместитель (класса NSKeyValueArray). Итак, теперь можно сделать вызов [myObject valueForKey: @"pepBoys"], чтобы получить этот объект-заместитель массива, а также вызов [myObject valueForKeyPath: @"pepBoys.name"], чтобы получить такой же массив символьных строк, как и прежде.

Приведенный выше конкретный пример может показаться несколько простоватым, поскольку в его основу уже положена реализация переменной экземпляра массива. Но нетрудно представить и такую реализацию, где результат вызова метода `objectInPepBoysAtIndex:` получается с помощью операции совершенно другого рода.

Объект-заместитель, возвращаемый через подобный фасад, ведет себя как объект типа `NSArray`, а не типа `NSMutableArray`. Если требуется предоставить вызывающему коду возможность манипулировать объектом-заметителем, предоставляемым фасадом механизма KVC, как будто это изменяемый массив, то придется реализовать еще два метода, а вызывающий код должен получить другой объект-заместитель, вызвав метод `mutableArrayValueForKey:`. Ниже приведен пример реализации таких методов; при этом предполагается, что `theData` — это изменяемый массив.

```
- (void) insertObject: (id) val inPepBoysAtIndex: (NSUInteger) ix {
   [self.theData insertObject:val atIndex:ix];
}

- (void) removeObjectFromPepBoysAtIndex: (NSUInteger) ix {
   [self.theData removeObjectAtIndex: ix];
}
```

Теперь можно сделать вызов `[myObject mutableArrayValueForKey: @"pepBoys"]`, чтобы получить нечто похожее на изменяемый массив. (Истинная польза от метода `mutableArrayValueForKey:` станет яснее, когда речь пойдет о механизме наблюдения за значениями по ключам в главе 13.)

Трудности для программирования состоят в том, что ни один из этих методов невозможно найти непосредственно в документации, поскольку в их именах указываются ключи, характерные для конкретного объекта. В частности, из документации нельзя выяснить, для чего предназначен метод `removeObjectFromPepBoysAtIndex:`. Для этого нужно узнать каким-то другим способом, что данный метод является частью реализации соответствия механизму доступа к значениям по ключу `@"pepBoys"`, который может быть получен в качестве изменяемого массива. В связи с этим рекомендуется непременно снабжать прикладной код комментариями, чтобы его можно было понять в дальнейшем.

Еще одна трудность состоит, конечно, в том, что неправильная интерпретация имени метода может стать причиной того, что объект окажется *несовместимым* с механизмом доступа к значениям по ключам. В этом случае выяснить причину, по которой прикладной код выполняется не так, как нужно, будет очень трудно.

Управление памятью

Многих начинающих программировать в среде Cocoa удивляет, что объектам Cocoa требуется управление памятью и что ошибки в управлении памятью могут привести к стремительному увеличению чрезмерно расходуемой оперативной памяти, аварийным сбоям и загадочно неверному поведению приложения. Правда, если действует механизм ARC, ваши личные обязанности как программиста в отношении непосредственного управления памятью значительно сокращаются, что, конечно, является огромным облегчением, поскольку вы, вероятнее всего, совершите меньше ошибок и сможете уделить больше времени функционированию самого приложения вместо того, чтобы заниматься управлением памятью. Но, как показывает мой личный опыт, даже механизм ARC не избавляет от ошибок управления памятью, а следовательно, вам все же нужно понимать основные принципы управления памятью в среде Cocoa, чтобы знать, что именно механизм ARC может сделать для вас и как обращаться к нему в тех случаях, когда от вас потребуется помощь. Поэтому не сочтите за труд прочитать этот раздел, даже если вы собираетесь полностью положиться на механизм ARC.

Принципы управления памятью в среде Cocoa

Необходимость управлять памятью объясняется тем, что ссылки на объекты являются указателями. Как пояснялось в главе 1, сами указатели являются простыми (по существу, целыми) значениями в языке C и управляются автоматически, тогда как указатель на объект на самом деле указывает на область памяти, которая должна быть явно выделена, когда объект начинает свое существование, и также явно освобождена, когда объект прекращает свое существование. Как вам должно быть уже известно, память выделяется с помощью метода `alloc`. Как освободить память и когда это сделать?

Объект должен хотя бы прекратить свое существование, когда больше нет объектов, имеющих указатель на него. Объект без указателя на него бесполезен и занимает место в оперативной памяти, но ни один из других объектов не имеет и даже не может получить ссылку на него. Эта проблема называется *утечкой памяти*. Во многих языках программирования эта проблема решается по принципу, называемому *"сборкой мусора"*. Проще говоря, утечка памяти предотвращается в языке программирования периодическим просмотром центрального списка всех объектов и удалением тех из них, на которые отсутствуют указатели. Но внедрение "сборки мусора" в язык Objective-C было бы неоправданно дорогой мерой для работающих под управлением операционной системы iOS мобильных устройств, где объем оперативной памяти строго ограничен, а процессор обладает относительно малым быстродействием (а возможно, и единственным ядром). Следовательно, управлять памятью в операционной системе iOS приходится в большей или меньшей степени вручную.

Но управлять памятью вручную не так-то просто, поскольку объект должен прекратить свое существование вовремя, а не раньше или позже. Допустим, что язык программирования наделяется средствами, позволяющими одному объекту давать команду другому объекту прекратить свое существование в настоящий момент. Но у нескольких объектов может быть указатель (т.е. ссылка) на один тот же объект. Так, если у объектов `Manny` и `Moe` имеется указатель на объект `Jack` и объект `Manny` дает объекту `Jack` команду прекратить свое существование, то несчастный объект `Moe` останется с указателем на несуществующий объект, а еще хуже — на "мусор". Если объект, на который ссылается указатель, удален без его ведома, то такой указатель называется *висячим*. Следовательно, если объект `Moe` воспользуется висячим указателем для отправки сообщения объекту, который он все еще считает существующим, то приложение завершится аварийно.

Для того чтобы предотвратить появление висячих указателей и утечку памяти, в языке Objective-C и среде Cocoa реализована стратегия ручного управления памятью, исходя из так называемого *подсчета сохраняемых ссылок*, который ведется для каждого находящегося в памяти объекта. Другие объекты могут увеличить или уменьшить подсчет сохраняемых ссылок на данный объект. До тех пор, пока подсчет сохраняемых ссылок остается положительным, объект существует. Ни один из объектов не обладает полномочиями дать другому объекту команду на уничтожение. Напротив, как только ведущийся подсчет сохраняемых ссылок на объект достигнет нуля, он уничтожается автоматически.

В соответствии с описанной выше стратегией каждый объект, которому требуется существующий объект `Jack`, должен увеличить его подсчет сохраняемых ссылок, а когда этот объект больше не нужен, его подсчет сохраняемых ссылок должен быть уменьшен. Таким образом, задача эффективного управления памятью вручную решается при условии, что поведение всех объектов согласуется в данной стратегией, которая состоит в следующем.

- Висячие указатели не могут появиться потому, что любой объект, у которого имеется указатель на объект `Jack`, своевременно увеличил подсчет сохраняемых ссылок на объект `Jack`, гарантировав тем самым его существование.

- Утечки памяти невозможны потому, что любой объект, которому больше не требуется объект `Jack`, уменьшает подсчет сохраняемых ссылок на него, тем самым гарантируя, что объект `Jack` в конечном итоге прекратит свое существование, когда подсчет сохраняемых ссылок на него достигнет нуля, указывая на то, что объект `Jack` больше не нужен ни одному из других объектов.

Очевидно, что все это зависит от того, будут ли все объекты взаимодействовать в полном согласии с данной стратегией управления памятью. Объекты Cocoa (т.е. такие объекты, которые являются экземплярами классов Cocoa) ведут себя в этом отношении вполне корректно, но именно *вы* как программист должны обеспечить их корректное поведение. До появления механизма ARC корректное поведение объектов обеспечивалось полностью программистом и его прикладным кодом. Если же действует механизм ARC, то корректное поведение объектов в большей или меньшей степени обеспечивается автоматически, при условии, что вы как программист знаете, как обращаться с поведением объектов, поведение которых обеспечено механизмом ARC.

Правила ручного управления памятью в среде Cocoa

Поведение объекта оказывается корректным по отношению к управлению памятью при условии, что он соблюдает определенные очень простые правила соответствия основным принципам управления памятью, изложенным в общих чертах в предыдущем разделе.

Прежде чем рассматривать эти правила, следует напомнить, что имя переменной, в том числе и экземпляра, представляет собой обычный указатель, что нередко смущает начинающих программировать в операционной системе iOS. Имя переменной, указывающее на объект, зачастую трактуется как объект, хотя оно таковым не является. Эти два понятия легко спутать, поэтому старайтесь не попасть в эту западню, неверно трактуя их. Когда сообщение посылается указателю, оно на самом деле посылается *через* этот указатель объекту, на который он указывает. Правила управления памятью относятся к объектам, а не к именам, ссылкам или указателям. Подсчет сохраняемых ссылок на указатель нельзя увеличить или уменьшить, поскольку такого объекта не существует. Память, занимаемая указателем, управляется автоматически и имеет крошеный объем. Управление памятью имеет отношение к объекту, на который ссылается указатель. Именно поэтому в примерах, рассматриваемых на страницах этой книги, объекты называются `Manny`, `Moe`, `Jack` или `A` и `B`, а не по именам переменных. Вопрос сохранения объекта `Jack` не имеет никакого отношения к *имени*, по которому любой другой объект *обращается* к объекту `Jack`.

Теперь вернемся к главному предмету этого раздела. Ниже приведены правила ручного управления памятью в среде Cocoa.

- Для того чтобы увеличить подсчет сохраняемых ссылок на любой объект, достаточно послать ему сообщение `retain`. Эта операция называется *сохранением* объекта. Она гарантирует, что объект сохранится, по крайней мере, до тех пор, пока подсчет сохраняемых ссылок на него не уменьшится снова. Для большего удобства вызываемый метод `retain` возвращает в качестве своего значения сохраненный объект. Это означает, что в результате вызова `[myObject retain]` возвращается объект, на который указывает имя `myObject`, но при этом подсчет сохраняемых ссылок этого объекта увеличивается.

- Когда классу дается команда `alloc` (или `new`), начинает свое существование получающийся в итоге экземпляр этого класса, причем подсчет сохраняемых ссылок на него уже увеличен. Поэтому *не* нужно (да и не следует) сохранять объект, экземпляр которого только что получен по команде `alloc` или `new`. Аналогично, когда экземпляру дается команда `copy`, получающийся в итоге новый объект (т.е. копия) начинает свое

существование, причем подсчет сохраняемых ссылок на него уже увеличен. Следовательно, *не* нужно (да и не следует) сохранять объект, экземпляр которого только что получен по команде `copy`.

- Для того чтобы уменьшить подсчет сохраняемых ссылок на любой объект, достаточно отправить ему сообщение `release`. Эта операция называется *освобождением* объекта.
- Если объект `A` получил объект `B` по команде `alloc` (либо `new`) или `copy` или же если объект `A` дал команду `retain` объекту `B`, то в конечном итоге объект `A` должен уравновесить эту операцию, выдав *один раз* команду `release` объекту `B`. После этого объект `B` должен принять во внимание, что объект `B` уже не существует. Это *золотое правило управления памятью*, позволяющее согласованно и правильно управлять ею.

Для того чтобы лучше понять золотое правило ручного управления памятью в среде Cocoa, следует мыслить категориями *владения*. Так, если объект `Manny` дал команду `alloc`, `retain` или `copy` объекту `Jack`, он тем самым подтвердил право на владение этим объектом. Одновременно объектом `Jack` могут владеть несколько объектов, но каждый такой объект отвечает только за правильное управление собственным владением объектом `Jack`. Объект, владеющий объектом `Jack`, отвечает в конечном счете за освобождение этого объекта, тогда как объект, не владеющий им, вообще не должен освобождать его. До тех пор, пока объекты, владеющие объектом `Jack`, ведут себя подобным образом, утечки памяти или висячие указатели на этот объект не возникнут.

Если действует механизм ARC, рассматриваемый в следующем разделе, эти правила остаются в силе, но они соблюдаются компилятором автоматически. В приложении на основе механизма ARC *вообще* не нужно, а на самом деле даже запрещено, давать команду `retain` или `release` вручную. Эти команды компилятор выдает автоматически, придерживаясь тех же самых принципов ручного управления памятью. Поскольку компилятор точнее и намного надежнее соблюдает правила управления памятью, чем вы как программист, то он вряд ли совершит ошибки, которые вы могли бы допустить по небрежности или недоразумению.

В тот момент, когда объект освобождается из памяти, появляется возможность уничтожить его. До появления механизма ARC это обстоятельство вызывало немалое беспокойство у программистов. В прикладной программе, не опирающейся на механизм ARC, приходится следить за тем, чтобы любые сообщения не посылались последовательно по любому указателю на объект, который уничтожен, включая и указатель, использованный для освобождения объекта. В противном случае такой указатель может стать висячим! Если существует какая-нибудь опасность попытаться случайно воспользоваться этим висячим указателем, то рекомендуется сделать его *пустым*, чтобы он указывал на пустое значение (`nil`). Сообщение, отправляемое по пустому указателю, не возымеет никакого действия. Оно не принесет никакой пользы, но и не нанесет большого вреда, как куриный бульон.

Данная стратегия строго соблюдается в прикладной программе, опирающейся на механизм ARC. В частности, механизм ARC делает пустым любой указатель на объект, которому было послано последнее уравновешивающее сообщение `release`, уведомляющее о том, что он может теперь прекратить свое существование. Как упоминалось в главе 3, механизм ARC делает также пустым указатель на экземпляр, когда он объявляется, если только не инициализировать его вручную, чтобы он указывал на конкретный экземпляр. Из этого неизбежно следует такой замечательный вывод: *если действует механизм ARC, то всякий указатель на экземпляр указывает на конкретный экземпляр или на пустое значение*. Но, к сожалению, это не избавляет полностью от появления висячих указателей, поскольку в самой среде Cocoa механизм ARC не применяется, как поясняется в следующем разделе.

Отладка ошибок управления памятью

Ошибки управления памятью относятся к числу наиболее распространенных ловушек, в которые попадают не только начинающие, но и опытные программирующие в среде Cocoa. Хотя такие ошибки совершаются намного реже, если действует механизм ARC, они все же вполне возможны, особенно потому, что программисты, пользующиеся механизмом ARC, склонны неверно думать, что они застрахованы от подобных ошибок. Практический опыт учит пользоваться всеми доступными средствами в поисках возможных ошибок. Ниже перечислены некоторые из этих средств (см. также главу 9).

- Измеритель памяти в навигаторе отладки (Debug) предоставляет диаграмму использования памяти при выполнении приложения, позволяя следить за возможными утечками памяти или другим неоправданно чрезмерным ее использованием.
- Статический анализатор, вызываемый по команде Product⇨Analyze, предоставляет немало полезных сведений об управлении памятью и может помочь привлечь ваше внимание к потенциальным ошибкам управления памятью.
- На панели Instruments (Инструменты), доступной по команде Product⇨Profile, имеются отличные средства для выявления утечек памяти и слежения за управлением памятью, выделяемой для отдельных объектов.
- Проверенная временем простейшая самодиагностика помогает убедиться в том, что объекты ведут себя должным образом. В частности, реализуйте метод `dealloc` с вызовом функции `NSLog()`. Если она не вызывается, значит, объект не собирается прекращать свое существование. Такой прием позволяет выявить ошибки, которые не способен обнаружить ни статический анализатор, ни инструментальные средства на панели Instruments.
- Особенно трудно выявить висячие указатели, но их зачастую можно обнаружить, активизировав объекты-зомби. Это нетрудно сделать на панели Instruments с помощью шаблона `Zombies`. С другой стороны, можно отредактировать действие `Run` в своей схеме, перейти на вкладку Diagnostics и установить флажок Enable Zombie Objects. В итоге ни один из объектов не прекратит свое существование, но вместо этого будет заменен объектом-зомби, выводящим на консоль уведомление вроде "Сообщение послано освобожденному из памяти экземпляру", если отправить ему сообщение. Не забудьте дезактивизировать объекты-зомби по завершении отслеживания висячих указателей. Не пользуйтесь объектами-зомби вместе с инструментом Leaks, поскольку эти объекты *сами* являются утечками.

Но даже помощи всех перечисленных выше инструментальных средств может оказаться недостаточно в разрешении всех затруднений, связанных с управлением памятью. Например, такие объекты, как представление типа `UIView`, содержащее крупное изображение, сами по себе невелики, а следовательно, в измерителе памяти или на панели Instruments может быть и не зарегистрировано чрезмерное использование памяти. Тем не менее они требуют значительного объема дополнительной памяти. Необходимость поддерживать ссылки на слишком большое количество подобных объектов может в конечном итоге привести к тому, что приложение будет просто уничтожено системой. Отслеживать подобные ошибки непросто.

Назначение и функции механизма ARC

При создании нового проекта и выборе шаблона приложения в среде Xcode 5 для этого проекта по умолчанию применяется механизм ARC. Это, среди прочего, означает следующее.

- Параметр Objective-C Automatic Reference Counting (`CLANG_ENABLE_OBJC_ARC`) настройки режима построения в компиляторе LLVM принимает логическое значение YES.
- Операторы `retain` или `release` отсутствуют в файлах шаблона проекта с расширением .m.
- Любой код, автоматически вводимый впоследствии в среде Xcode, например, при формировании свойства путем перетаскивания при нажатой клавише <Control> из nib-файла в прикладной код, соответствует соглашениям, принятым в механизме ARC.

Существующий проект, не поддерживающий механизм ARC, можно также привести в соответствие с этим механизмом. С этой целью выберите в среде Xcode команду меню Edit⇨Refactor⇨Convert to Objective-C ARC, которая поможет внести необходимые изменения в код. Принимать механизм ARC для всего проекта совсем не обязательно. Так, если имеется старый код, несовместимый с механизмом ARC и, возможно, написанный кем-то другим, его, возможно, потребуется внедрить в проект, поддерживающий механизм ARC, не внося существенных изменений в код, несовместимый с этим механизмом. Для этого сосредоточьте весь код, несовместимый с механизмом ARC, в отдельных файлах и для каждого из этих файлов отредактируйте цель, перейдите на вкладку Build Phases (Стадии построения), дважды щелкните на листинге из файла с кодом, несовместимым с механизмом ARC, в разделе Compile Sources и наберите `-fno-objc-arc` в текстовом поле, чтобы ввести эту директиву в столбец Compiler Flags.

На самом деле механизм ARC является средством LLVM 3.0 или более поздней версии этого компилятора и одной из главных целей его разработки. За более полной информацией по данному вопросу обращайтесь по адресу `http://clang.llvm.org/docs/AutomaticReferenceCounting.html`.

В процессе компиляции файла с расширением .m и с учетом механизма ARC компилятор будет интерпретировать явные команды `retain` и `release` как ошибку, а вместо них вставит подспудно свои команды, выполняющие те же самые функции, что и команды `retain` и `release`. В итоге прикладной код переводится в режим ручного управления памятью в соответствии с описанными ранее принципами и золотым правилом. Но автором кода ручного управления памятью является компилятор, а сам этот код невидим, хотя его и можно при желании прочитать на уровне языка ассемблера.

Механизм ARC вставляет команды `retain` и `release` на двух стадиях.

1. Механизм ARC ведет себя очень консервативно и при малейшем сомнении сохраняет, а затем освобождает объекты из памяти. По существу, этот механизм сохраняет объекты при всяком стечении обстоятельств, которое может иметь малейшие последствия для управления памятью. В частности, он сохраняет объект, когда последний передается в качестве аргумента, присваивается переменной и т.д. Он может даже вставлять временные переменные, чтобы достаточно рано обратиться к объекту и иметь возможность сохранить его. Но, конечно, он и освобождает объект согласованно. Это означает, что по завершении каждой стадии управление памятью совершается технически корректно. Один объект может сохраняться и освобождаться намного чаще, чем при

использовании команд retain и release вручную, но, по крайней мере, можно быть уверенным в том, что висячие указатели и утечки объектов из памяти не возникнут.

2. Механизм ARC выполняет оптимизацию, удаляя столько пар команд retain и release из каждого объекта, сколько можно, обеспечивая в то же время надежность с точки зрения фактического поведения программы. Это означает, что по завершении второй стадии управление памятью по-прежнему совершается не только технически корректно, но и эффективно.

Рассмотрим в качестве примера следующий код:

```
- (void) myMethod {
   NSArray* myArray = [NSArray array];
   NSArray* myOtherArray = myArray;
}
```

Для выполнения этого фрагмента кода дополнительный код управления памятью на самом деле не требуется по причинам, рассматриваемым в следующем разделе. Даже если не применять механизм ARC, то вводить команды retain и release в этот код не нужно. Но нетрудно представить, что на первой стадии выполнения этого кода механизм ARC будет действовать весьма консервативно, гарантируя, что переменная будет иметь пустое значение nil или же указывать на объект, сохраняя каждое присваиваемое ей значение и в то же время освобождая каждое присвоенное ей прежде значение, на которое она указывала, при условии, что это значение было ранее сохранено при его присваивании данной переменной. Таким образом, нетрудно представить, хотя это вряд ли будет совершенно правильно, что рассматриваемый здесь код будет сначала скомпилирован с учетом действия механизма ARC в эквивалентный код, приведенный в примере 12.1.

Пример 12.1. Возможный сценарий консервативного управления памятью с помощью механизма ARC

```
- (void) myMethod {
   // сделать пустыми все указатели на новые объекты
   NSArray* myArray = nil;
   // сохранить при присваивании, освободить предыдущее значение
   id temp1 = myArray;
   myArray = [NSArray array];
   [myArray retain];
   [temp1 release]; // (no effect, it's nil)
   // сделать пустыми все указатели на новые объекты
   NSArray* myOtherArray = nil;
   // сохранить при присваивании, освободить предыдущее значение
   id temp2 = myOtherArray;
   myOtherArray = myArray;
   [myOtherArray retain];
   [temp2 release]; // (no effect, it's nil)
   // завершить метод, сохранить равновесие в локальных переменных
   // сделать пустыми все указатели путем освобождения из памяти, когда
   // все остальное останется уравновешенным
   [myArray release];
   myArray = nil;
   [myOtherArray release];
   myOtherArray = nil;
}
```

Далее вступает в действие оптимизатор ARC, уменьшая объем выполняемой работы. Например, он может обнаружить, что переменные myArray и myOtherArray превращаются

в указатели на тот же самый объект, и поэтому он может удалить некоторые промежуточные команды сохранения и освобождения из памяти. Кроме того, оптимизатор ARC может обнаружить, что посылать команду `release` пустому значению `nil` не нужно. Но команды сохранения и освобождения из памяти настолько эффективны, когда действует механизм ARC, что вряд ли будет иметь большое значение, если оптимизатор не удалит любые промежуточные команды сохранения и освобождения из памяти.

Намного большее значение имеет уравновешивание ручного управления памятью, чем согласованное выполнение команд `retain` и `release`. В частности, как упоминалось ранее, команды `alloc` и `copy` порождают объекты, подсчет сохраняемых ссылок которых уже увеличен, и поэтому они также должны быть уравновешены командой `release`. Для того чтобы подчиняться этой части правил управления памятью в среде Cocoa, механизм ARC прибегает к *предположениям относительно именования методов*.

В частности, когда в прикладном коде получается объект в качестве значения, возвращаемого в результате вызова метода, механизм ARC ищет начальное слово (или несколько слов) в имени метода, имеющем смешанное написание. (Термином *смешанное написание* описывается составное слово, отдельные слова которого начинаются с прописной буквы, например: `СмешанноеНаписание`.) Если начальным словом в имени метода оказывается `alloc`, `init`, `new`, `copy` или `mutableCopy`, то в механизме ARC предполагается, что объект возвращается методом вместе с увеличенным подсчетом сохраняемых ссылок, который должен быть уравновешен соответствующей командой `release`.

Так, если бы в приведенном выше примере в результате вызова `[NSArray new]` вместо вызова `[NSArray array]` был получен массив, то механизму ARC стало бы известно, что потребуется дополнительное освобождение из памяти, чтобы уравновесить увеличенный подсчет сохраняемых ссылок на объект, возвращаемый из метода, имя которого начинается на new.

В связи с этим на вас возлагается ответственность *не* именовать свои методы таким образом, чтобы вызывать своего рода сигнал тревоги у механизма ARC. Поэтому проще всего *не* начинать имена любых методов с alloc, init (если только речь не идет о написании инициализатора), new, copy или mutableCopy. Употребление этих слов в именах методов вряд ли нанесет много вреда, но все же лучше не рисковать, но подчинить, насколько это возможно, механизм ARC соглашениям об именовании. (Из этого затруднительного положения имеются и другие пути выхода, если неверно именованные методы нельзя никак изменить, но здесь они не рассматриваются.)

Управление памятью для объектов Cocoa

Встроенные объекты Cocoa принимают владение передаваемых им объектов, сохраняя их, если считают это целесообразным. Такое сохранение, разумеется, уравновешивается последующим освобождением из памяти. На самом деле это настолько общее явление, что если объект Cocoa *не* собирается сохранять передаваемый ему объект, то об этом специально упоминается в документации. Особенно это касается коллекций типа `NSArray` или `NSDictionary` (подробнее о классах коллекций см. в главе 10).

Объект вряд ли может стать элементом коллекции, если он может в любой момент прекратить свое существование. Следовательно, когда элемент вводится в коллекцию, владение этим объектом подтверждается в ней его сохранением, а после этого коллекция действует как вполне корректный владелец объекта. Так, если это изменяемая коллекция, то при удалении из нее элемента последний освобождается из памяти. Если же объект коллекции прекращает свое существование, то из памяти освобождаются все ее элементы. (Если прежний элемент,

освобожденный коллекцией из памяти, нигде больше не хранится, то он прекращает свое существование.)

До появления механизма ARC удаление объекта из измененяемой коллекции таило в себе потенциальную западню. Рассмотрим следующий фрагмент кода:

```
NSString* s = myMutableArray[0];
[myMutableArray removeObjectAtIndex: 0];
// неудачная идея в коде, не поддерживающем механизм ARC!
```

Как только что пояснялось, когда объект удаляется из изменяемой коллекции, он освобождается этой коллекцией из памяти. Поэтому во второй строке предыдущего фрагмента кода происходит неявное освобождение объекта из памяти, который прежде хранился в нулевом элементе коллекции типа `myMutableArray`. Если это приводит к уменьшению подсчета сохраняемых ссылок на объект до нуля, то такой объект уничтожается. Тогда указатель s станет висячим, а впоследствии в прикладном коде может произойти сбой при попытке воспользоваться этим указателем как символьной строкой. Следовательно, до появления механизма ARC приходилось запоминать факт сохранения любого объекта, извлеченного из коллекции, прежде чем выполнить над ним любую операции, способную привести к его разрушению, как показано в примере 12.2.

Пример 12.2. Гарантирование сохранности элементов коллекции в коде, не поддерживающем механизм ARC

```
NSString* s = myMutableArray[0];
[s retain]; // в этом коде механизм ARC не поддерживается
[myMutableArray removeObjectAtIndex: 0];
```

Разумеется, при таком управлении памятью, выделяемой под интересующий объект, после подтверждения своих прав на владение им необходимо также принять меры к его освобождению впоследствии, иначе произойдет утечка объекта из памяти. Тем не менее тот же самый код идеально работает и в том случае, если действует механизм ARC, как показано ниже.

```
NSString* s = myMutableArray[0];
[myMutableArray removeObjectAtIndex: 0];
// этот код вполне работоспособен и тогда, когда действует механизм ARC
```

Дело в том, что, как упоминалось выше, механизм ARC изначально действует чрезвычайно консервативно. Как и в примере 12.1, механизм ARC сохраняет объект при его присваивании, и поэтому нетрудно представить, что этот механизм будет действовать по сценарию, приведенному в примере 12.3.

Пример 12.3. Возможный сценарий гарантируемой механизмом ARC сохранности элемента коллекции

```
NSString* s = nil;
// сохранить значение при его присваивании, освободить предыдущее значение
id temp = s;
s = myMutableArray[0];
[s retain];
[temp release]; // не возымеет никакого действия, т.к. это пустое значение,
// а следующий шаг можно уже сделать без опаски
[myMutableArray removeObjectAtIndex: 0];
// ... и в дальнейшем ...
[s release];
s = nil;
```

Оказывается, это именно то, что и нужно! К тому моменту, когда дело дойдет до вызова метода `removeObjectAtIndex:`, подсчет сохраняемых ссылок на объект, полученный из массива, окажется увеличенным, поддерживая существование объекта таким же образом, как и в коде без механизма ARC из примера 12.2.

Автоматическое освобождение из памяти

Каким же образом происходит управление памятью при вызове метода и получении в качестве результата экземпляра, названного *готовым* в главе 5? Рассмотрим, например, следующий фрагмент кода:

```
NSArray* myArray = [NSArray array];
```

Согласно золотому правилу управления памятью объект, на который теперь указывает переменная `myArray`, не требует управления памятью. Для его получения не была выдана команда `alloc`, а следовательно, не были затребованы права на его владение, и поэтому нет никакой нужды его освобождать. Но как такое возможно? Каким образом класс `NSArray` способен предоставить массив, который не нужно освобождать без опасения утечки этого объекта?

Для того чтобы развеять таинственность происходящего, воспользуйтесь явным управлением памятью, т.е. кодом без поддержки механизма ARC, чтобы все команды сохранения и освобождения из памяти стали видимыми, а также попробуйте поставить себя на место класса `NSArray`. Как реализовать метод `array`, чтобы сформировать массив, управление памятью для которого недоступно в вызывающем коде? Не думайте, что для этого достаточно вызвать какой-нибудь *другой* метод из класса `NSArray`, поставляющий готовый экземпляр. Это лишь приведет к отсрочке решения вопроса на один шаг назад. Выполняя роль класса `NSArray`, вы рано или поздно вынуждены будете сформировать экземпляр заново и возвратить его, как показано ниже.

```
- (NSArray*) array {
   NSArray* arr = [[NSArray alloc] init];
   return arr; // пожалуй, не так быстро...
}
```

По-видимому, такой подход не годится. Значение переменной `arr` было сформировано по команде `alloc`. В соответствии с золотым правилом управления памятью это означает, что из памяти нужно также освободить объект, на который указывает переменная `arr`. Но когда это можно сделать? Если сделать это *до* возврата переменной `arr`, она будет указывать на "мусор", который и будет в конечном итоге поставлен вызывающему коду. Но этого нельзя сделать и *после* возврата переменной `arr`, поскольку при выдаче команды `return` метод все еще существует!

Очевидно, что требуется найти какой-нибудь другой способ поставки этого объекта, не уменьшая в *настоящий момент* подсчет сохраняемых ссылок на него, чтобы он просуществовал настолько долго, насколько потребуется для его получения и обработки в вызывающем коде. Но в то же время нужно обеспечить уменьшение подсчета сохраняемых ссылок на этот объект, чтобы уравновесить вызов метода `alloc` и самостоятельно выполнить управление памятью данного объекта. Решение, которое явно напрашивается в коде без поддержки механизма ARC, состоит в автоматическом освобождении из памяти с помощью метода `autorelease`, как показано ниже.

```
- (NSArray*) array {
   NSArray* arr = [[NSArray alloc] init];
   [arr autorelease];
```

```
    return arr;
}
```

Поскольку метод `autorelease` возвращает объект, которому он посылается, то код автоматического освобождения из памяти можно сделать еще короче следующим образом:

```
- (NSArray*) array {
    NSArray* arr = [[NSArray alloc] init];
    return [arr autorelease];
}
```

Метод `autorelease` действует следующим образом. Прикладной код выполняется в присутствии так называемого *автоматически освобождаемого пула*. (Заглянув в файл `main.m`, можно обнаружить в нем фактически созданный автоматически освобождаемый пул.) Когда метод `autorelease` посылается объекту, этот объект размещается в автоматически освобождаемом пуле, при этом ведется автоматический подсчет количества размещений объекта в данном пуле. Время от времени, когда ничего другого не происходит, этот пул автоматически освобождается. Это означает, что автоматически освобождаемый пул посылает команду `release` каждому хранящемуся в нем объекту и тем самым освобождается от всех этих объектов. Если это приводит к обнулению подсчета сохраняемых ссылок на объект, то объект уничтожается обычным способом. Таким образом, автоматическое освобождение методом `autorelease` похоже на обычное освобождение из памяти методом `release` и, по существу, является формой последнего, но при условии, что это делается не сию секунду, а позднее.

Знать, когда именно пул освободится автоматически, не нужно, да и невозможно, если только не заставить освободиться его принудительно, как будет показано ниже. Следует, однако, иметь в виду, что при вызове такого метода, как `array`, вызывающему коду придется приостановить на значительное врем свое выполнение в ожидании поставляемого объекта, а при желании — сохранить этот объект.

Объект, поставляемый такими методами, как `array`, называется *автоматически освобождаемым*. Объект, выполняющий поставку, на самом деле завершает свое управление памятью для поставляемого объекта. Следовательно, поставляемый объект имеет потенциально, но не окончательно нулевой подсчет сохраняемых ссылок. Поставляемый объект не исчезнет сразу же после вызова `[NSArray array]`, поскольку код все еще выполняется, а следовательно, и пул *не* освободится автоматически *сию же секунду*. Поэтому получателю такого объекта следует иметь в виду, что он может быть автоматически освобождаемым. Такой объект не исчезнет до тех пор, пока выполняется код, из которого был вызван метод, поставляющий этот объект. Но если получающему объекту требуется сохранить поставляемый объект для последующего применения, то он должен это сделать непременно.

Именно поэтому после приема готового экземпляра вы как программист не несете никакой ответственности за управление памятью, как, например, при вызове `[NSArray array]`. Экземпляр, получаемый вами *другим* способом, а не согласно золотому правилу управления памятью, вам уже не принадлежит. Им владеет какой-нибудь другой объект, или же он освобождается автоматически. Если же этот экземпляр принадлежит какому-нибудь другому объекту, то он не будет уничтожен до тех пор, пока не перестанет принадлежать данному объекту, как было показано на примере массива типа `NSMutableArray` в предыдущем разделе. Если вас беспокоит, что нечто подобное может произойти, то вы должны завладеть готовым экземпляром, сохранив его, а затем освободив по своему усмотрению. И наконец, если экземпляр освобождается автоматически, то он, безусловно, будет храниться достаточно долго, чтобы прикладной код завершил свое выполнение, поскольку пул не освободится автоматически до тех пор, пока выполнение прикладного кода не завершится. Опять же, если экземпляр требуется сохранить еще дольше, для этого придется завладеть им.

Как и следовало ожидать, если действует механизм ARC, то все операции управления памятью происходят согласованно. Для этого совсем не нужно, да и нельзя на самом деле выдавать команду `autorelease`. Напротив, механизм ARC сделает это автоматически по описанным ранее правилам именования методов. Например, метод, вызывающий метод `array`, не должен начинаться со слова `new`, `init`, `alloc`, `copy` или `mutableCopy`, если он именуется в смешанном написании. Следовательно, он должен возвращать объект, управление памятью для которого уравновешивается, используя метод `autorelease` для окончательного освобождения из памяти. Механизм ARC проверит, что это именно так и есть. С другой стороны, в методе, вызывающем метод `array` и принимающем возвращаемый из него объект, должно предполагаться, что этот объект автоматически освобождается и может прекратить свое существование, если только не сохранить его. Именно это и предполагается в механизме ARC.

Иногда пул требуется немедленно освободить автоматически. Рассмотрим следующий фрагмент кода:

```
for (NSString* aWord in myArray) {
  NSString* lowerAndShorter = [[aWord lowercaseString] substringFromIndex:1];
  [myMutableArray addObject: lowerAndShorter];
}
```

На каждом шаге цикла в приведенном выше фрагменте кода в автоматически освобождаемый пул вводятся два объекта: вариант исходной символьной строки строчными буквами, а также укороченный ее вариант. Первый объект (т.е. вариант символьной строки строчными буквами) является исключительно *промежуточным объектом*, ведь по завершении текущего шага цикла указатель на этот объект имеется только в автоматически освобождаемом пуле. Если бы этот цикл повторялся очень много раз или же если бы промежуточные объекты были очень крупными по размерам, для их хранения потребовалось бы немало оперативной памяти. Все эти промежуточные объекты будут освобождены из памяти при автоматическом освобождении пула, чтобы не произошла их утечка из памяти. Тем не менее они накапливаются в оперативной памяти, и в некоторых случаях это может привести к ее исчерпанию еще до автоматического освобождения пула. Подобное затруднение может стать даже более серьезным, чем кажется на первый взгляд, поскольку вы можете повторно вызывать встроенный метод Cocoa, который будет без вашего ведома формировать немалое число промежуточных автоматически освобождаемых объектов.

В качестве выхода из подобного затруднения можно вмешаться в механизм действия автоматически освобождаемого пула, предоставив собственный аналогичный пул. Это вполне допустимо, поскольку автоматически освобождаемый пул служит для хранения автоматически освобождаемого объекта в самом последнем из созданных пулов. Для этого достаточно создать автоматически освобождаемый пул в начале цикла и освободить его в конце цикла на каждом его шаге. В современной версии языка Objective-C для этой цели код, который выполняется под действием собственного автоматически освобождаемого пула, заключается в фигурные скобки директивы `@autoreleasepool{}`, как показано ниже.

```
for (NSString* aWord in myArray) {
   @autoreleasepool {
      NSString* lowerAndShorter =
         [[aWord lowercaseString] substringFromIndex:1];
      [myMutableArray addObject: lowerAndShorter];
   }
}
```

Многие классы представляют программистам два равнозначных способа получения объекта: в виде автоматически сохраняемого объекта (готового экземпляра) или же объекта, самостоятельно создаваемого методом `alloc` и определенной формой метода `init` (т.е. инициализации с самого начала). Например, в классе `NSMutableArray` для этой цели предоставляется метод класса `array` и метод экземпляра `init`. Какой же из этих методов следует использовать? В общем, метод `alloc` или определенную форму метода `init` следует применять там, где лучше всего сформировать с их помощью объект. Такая стратегия препятствует появлению задержек, возникающих в автоматически освобождаемом пуле, а также как можно более экономному расходованию памяти.

Управление памятью для переменных экземпляра (без механизма ARC)

До появления механизма ARC основное место, где программисты могли совершить ошибку при управлении памятью, было связано с переменными экземпляра. Управление памятью, выделяемой для временных переменных экземпляра в одном методе, осуществляется довольно просто, поскольку все тело метода видно сразу, и остается лишь следовать золотому правилу управления памятью, уравновешивая каждую команду `retain`, `alloc` или `copy` командой `release`. Если из этого метода возвращается объект с увеличенным подсчетом сохраненных ссылок, то эти команды следует уравновешивать командой `autorelease`. Однако переменные экземпляра усложняют управление памятью по многим причинам, включая следующие:

Переменные экземпляра являются сохраняемыми

Создаваемые переменные экземпляра сохраняются по завершении метода, окончании выполнения прикладного кода и автоматическом освобождении пула. Так, чтобы значение объекта, на который указывает переменная экземпляра, не исчезло как дым, оставив после себя висячий указатель, рекомендуется сохранить его во время присваивания переменной экземпляра.

Управление памятью для переменных экземпляра осуществляется из разных мест в прикладном коде

Управление памятью, выделяемой для переменных экземпляра, может быть распределено среди нескольких различных методов, что затрудняет достижение правильной работы прикладного кода и отладку, если он работает неправильно. Так, если сохранить значение, присвоенное переменной экземпляра, то в дальнейшем его придется освободить, согласуясь с золотым правилом управления памятью, чтобы исключить ее утечку, но сделать это нужно в каком-нибудь другом методе.

Переменные экземпляра могут вам не принадлежать

Вам нередко придется присваивать или получать значение из переменной экземпляра, принадлежащей другому объекту. Допустим, вы разделяете общий доступ к значению вместе с каким-нибудь другим сохраняемым объектом. Если этот объект прекратит свое существование и освободит свои переменные экземпляра, а у вас имеется указатель на значение в переменной экземпляра этого объекта и вы не подтвердили свои права на владение данным значением, сохранив его, то в конечном итоге в вашем распоряжении может оказаться висячий указатель.

Следовательно, до появления механизма ARC простая, на первый взгляд, операция присваивания объекта переменной экземпляра могла быть сопряжена с большим риском. Рассмотрим в качестве примера следующий фрагмент кода:

```
NSMutableDictionary* d = [NSMutableDictionary dictionary];
// ... здесь следует код заполнения словаря по переменной d ...
self->_theData = d;
// в коде без поддержки механизма ARC это была бы неудачная идея!
```

До появления механизма ARC такой код мог быть чреват серьезной ошибкой. По завершении этого кода остается сохраняемый указатель на автоматически освобождаемый объект, права на владение которым вообще отсутствуют. Он может исчезнуть, оставив после себя висячий указатель. Очевидно, что в качестве выхода из положения следует сохранить данный объект во время его присваивания переменной экземпляра. Этого можно добиться следующим образом:

```
[d retain];
self->_theData = d;
```

С другой стороны, то же самое можно сделать так:

```
self->_theData = d;
[self->_theData retain];
```

Поскольку метод `retain` возвращает объект, которому он посылается, то сохранить этот объект можно и так, как показано ниже.

```
self->_theData = [d retain];
```

Однако ни один из этих способов на самом деле нельзя считать удовлетворительным. Рассмотрим, в частности, те хлопоты, которые может доставить присваивание *другого* значения переменной экземпляра `self->_theData`. В этом случае нужно будет не забыть освободить объект, на который уже имеется указатель, чтобы уравновесить сохранение этого объекта, а также сохранить следующее значение. Поэтому было бы намного лучше инкапсулировать управление памятью, выделяемой для данной переменной экземпляра, в *-методе доступа*, а точнее — в set-методе. Благодаря этому управление памятью окажется корректным, поскольку оно будет всегда осуществляться через метод доступа. В примере 12.4 показано, каким образом может выглядеть стандартный образец такого метода.

Пример 12.4. Простой сохраняющий set-метод

```
- (void) setTheData: (NSMutableArray*) value {
   if (self->_theData != value) {
   [self->_theData release];
   self->_theData = [value retain];
  }
}
```

В примере 12.4 освобождается объект, на который в настоящий момент указывает переменная экземпляра, и если этот объект оказывается пустым (`nil`), то никакого вреда такая операция не приносит. Кроме того, входящее значение сохраняется перед его присваиванием переменной экземпляра, и если это значение оказывается пустым (`nil`), то и такая операция никакого вреда не приносит. Проверка на соответствие входящего значения тому же самому объекту, на который указывает переменная экземпляра, делается не только из соображений экономии, но и потому, что если освободить данный объект, то он может исчезнуть, а значение превратится в висячий указатель, который (страшно подумать) может быть затем присвоен переменной экземпляра `self->_theData`.

Устанавливающий метод доступа теперь автоматически управляет памятью корректно, при условии, что он всегда применяется для установки переменной экземпляра. В этом заключается одна из основных причин, по которым методы доступа так важны! Следовательно,

присваивание переменной экземпляра в первоначальном варианте кода должно выглядеть следующим образом:

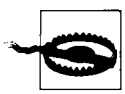
```
[self setTheData: d];
```

Обратите внимание на то, что этот метод установки можно также использовать для освобождения значения переменной экземпляра и установки пустого значения в этой переменной. Благодаря этому исключается появление висячего указателя, и все это делается одновременно, как показано ниже.

```
[self setTheData: nil];
```

Применение метода доступа для управления памятью дает еще одно преимущество. Впрочем, управление памятью для данной переменной экземпляра еще не завершено. Нужно еще не забыть освободить объект, на который указывала данная переменная экземпляра в последний момент перед тем, как прекратит свое существование объект этого экземпляра. В противном случае, если данная переменная экземпляра указывает на сохранившийся объект, произойдет утечка памяти. В качестве принимаемой в последний момент меры обычно служит метод `dealloc` из класса `NSObject` (см. главу 10), который вызывается, когда объект прекращает свое существование.

```
- (void) dealloc {
   [self->_theData release];
   [super dealloc];
}
```



Ни в коем случае не вызывайте метод `dealloc` в своем коде, кроме последнего вызова по ссылке `super` в переопределяемом варианте метода `dealloc`. Если действует механизм ARC, то вызывать метод `dealloc` *нельзя*, и это еще один пример того, как механизм ARC уберегает вас от собственных ошибок.

Как видите, до появления механизма ARC управление памятью, выделяемой для переменных экземпляров объектов, требовало немалых трудов! Для этого нужно было не только правильно написать устанавливающие методы доступа к переменным экземпляра, но и обеспечить вызов метода `dealloc` для *каждого* объекта, чтобы освободить *каждую* переменную экземпляра, значение которой было сохранено. Это, очевидно, была еще одна весьма вероятная возможность совершить ошибку.

Однако это еще не все! Как насчет инициализатора, устанавливающего значение переменной экземпляра? В этом случае требуется управление памятью. Здесь, к сожалению, на помощью не придет метод доступа, как, впрочем, и для обращения к собственным переменным экземпляра в методе `dealloc`. Следовательно, собственные методы доступа нельзя применять для обращения к собственным переменным экземпляра в инициализаторе (см. главу 5). Дело в том, что, с одной стороны, объект еще не полностью сформирован, а с другой — метод доступа может иметь другие побочные эффекты.

За иллюстрацией обратимся к примеру инициализатора из главы 5 (см. пример 5.3). Перепишем его исходный код таким образом, чтобы инициализировать объект `Dog` по имени. Такая возможность не рассматривалась в главе 5 потому, что символьная строка является объектом, а следовательно, выделяемая для нее память подлежит управлению! Теперь представьте, что имеется переменная экземпляра `_name`, значением которой является объект типа `NSString`, и что требуется инициализатор, позволяющий вызывающему коду передать значение этой переменной. В примере 12.5 показано, как это можно реализовать непосредственно в коде.

Пример 12.5. Простой пример инициализатора, сохраняющего переменную экземпляра

```
- (id) initWithName: (NSString*) s {
   self = [super init];
   if (self) {
     self->_name = [s retain];
   }
   return self;
}
```

На самом деле более вероятным для объекта типа `NSString` было бы копирование, а не только сохранение. Дело в том, что у класса `NSString` имеется изменяемый подкласс `NSMutableString`, а следовательно, какой-нибудь другой объект может вызвать метод `initWithName:` и передать изменяемую символьную строку, ссылку на которую он все еще хранит, а затем видоизменить ее таким образом, чтобы кличка собаки (объект `Dog`) незаметно изменилась. В примере 12.6 показано, каким образом исходный код инициализатора будет выглядеть на этот раз.

Пример 12.6. Простой пример инициализатора, копирующего переменную экземпляра

```
- (id) initWithName: (NSString*) s {
   self = [super init];
   if (self) {
      self->_name = [s copy];
   }
   return self;
}
```

В примере 12.6 существующее значение переменной экземпляра `_name` не освобождается. Безусловно, она не указывает на какое-то конкретное предыдущее значение, поскольку такое значение отсутствует. Следовательно, освобождать значение этой переменной экземпляра не имеет смысла.

Таким образом, управление памятью, выделяемой для переменной экземпляра без применения механизма ARC, может быть осуществлено в трех местах: в инициализаторе, методе установки и методе `dealloc`. Это довольно распространенная архитектура! Для того чтобы заглянуть во многие места и проверить в них правильность и согласованность управления памятью, требуется немало труда, а это зачастую чревато ошибками. Однако именно так приходится поступать, если механизм ARC не применяется (хотя в языке Objective-C имеется, по крайней мере, возможность для автоматического формирования методов доступа, как будет показано далее в этой главе).

Управление памятью для переменных экземпляра (с помощью механизма ARC)

Если применяется механизм ARC, то он будет автоматически управлять памятью, выделяемой для переменной экземпляра. Это избавляет вас от необходимости (а в общем, и возможности) управлять памятью самостоятельно. Все это означает, что механизм ARC будет выполнять буквально, но незаметно для вас те же самые операции управления памятью, которые были описаны в предыдущем разделе.

Рассмотрим сначала непосредственное присваивание значения переменной экземпляра. По умолчанию механизм ARC интерпретирует переменную экземпляра таким же образом, как и любую другую переменную. Это означает, что после присваивания значения этой переменной экземпляра он создает временную переменную, сохраняет в ней присвоенное значение,

освобождает текущее значение переменной экземпляра и выполняет присваивание. Для этого достаточно написать строку кода, аналогичную приведенной ниже.

```
self->_theData = d;
```

По существу, механизм ARC, соблюдая правило сохранения нового присваиваемого значения и освобождения старого, заменяет код по следующему вымышленному сценарию:

```
// вымышленный сценарий сохранения нового присваиваемого значения
// и освобождения предыдущего значения
id temp = self->_theData;
self->_theData = d;
[self->_theData retain];
[temp release];
```

Именно так и должно было бы происходить присваивание значения переменной экземпляра без применения механизма ARC. От вашего внимания не должно ускользнуть, что именно такой код вам пришлось бы написать для создания формального метода установки, как в примере 12.4. Как видите, даже такая простая операция, как присваивание значения переменной экземпляра, требует немалых хлопот, связанных с управлением памятью!

Это же относится и к написанию формального метода установки. Однако теперь метод установки оказывается намного проще, чем без применения механизма ARC. На самом деле он может состоять только из одного оператора присваивания, как показано ниже, поскольку механизм ARC довольно корректно выполняет все служебные операции управления памятью по описанному выше сценарию.

```
- (void) setTheData: (NSMutableArray*) value {
   self->_theData = value;
}
```

Более того, когда объект прекращает свое существование, механизм ARC освобождает сохраняемые значения переменных экземпляра, а следовательно, избавляет от необходимости вызывать для этой цели метод `dealloc`! Тем не менее, если действует механизм ARC, метод `dealloc` приходится по-прежнему реализовывать по ряду других причин. Например, в этом методе уместно снимать объекты с регистрации на получение уведомлений (см. главу 11), но в то же время в нем не требуется вызывать метод `release` для любых переменных экземпляра, а также метод из суперкласса по ссылке `super`. (Когда действует механизм ARC и вызывается метод `dealloc`, значения переменных экземпляра еще *не* освобождены, и поэтому к ним можно обращаться в этом методе.)

Что же делать, если в отсутствие вызова метода `release`, а при действующем механизме ARC его вызывать запрещено, требуется освободить значение переменной экземпляра? Ответ на этот вопрос довольно прост: установить пустое (`nil`) значение переменной экземпляра (возможно, с помощью метода установки). Если установить пустое значение этой переменной, механизм ARC по умолчанию освободит существующее ее значение автоматически.

В заключение, обсудим последствия применения механизма ARC для написания инициализатора, в котором устанавливаются значения переменных экземпляров объектов, как показано в примерах 12.5 и 12.6. Код для этих инициализаторов останется при применении механизма ARC таким же, как и без него, за исключением того, что не нужно да и нельзя выдавать команду `retain`. Следовательно, при действующем механизме ARC исходный код из примера 12.5 станет таким, как показано в примере 12.7.

Пример 12.7. Простой пример инициализатора, сохраняющего переменную экземпляра при действующем механизме ARC

```
- (id) initWithName: (NSString*) s {
   self = [super init];
   if (self) {
     self->_name = s;
   }
   return self;
}
```

Исходный код инициализатора из примера 12.6 останется без изменения при действующем механизме ARC, как показано в примере 12.8. Тем не менее можно по-прежнему выдать команду `copy`, а механизму ARC известно, как управлять памятью объекта, возвращаемого из метода, смешанное написание имени которого начинается со слова `copy` или просто ограничивается им.

Пример 12.8. Простой пример инициализатора, копирующего переменную экземпляра при действующем механизме ARC

```
- (id) initWithName: (NSString*) s {
   self = [super init];
   if (self) {
     self->_name = [s copy];
   }
   return self;
}
```

Циклы сохранения и слабые ссылки

Механизм ARC действует автоматически и бездумно. Ему ничего не известно о логике отношений между объектами в приложении. Однако иногда механизму ARC приходится давать дополнительные инструкции, чтобы он не нанес какой-нибудь ущерб. Таким ущербом может, в частности, стать цикл сохранения.

Цикл сохранения — это ситуация, в которой каждый из объектов `A` и `B` сохраняют друг друга. Если в подобной ситуации объектам разрешено быть сохраненными, это в конечном итоге приведет к утечке обоих объектов из памяти, поскольку подсчет сохраняемых ссылок ни на один из них нельзя уменьшить до нуля. Эту ситуацию можно рассматривать и следующим образом: сохраняя объект `B`, объект `A` сохраняет также самого себя, а это препятствует его уничтожению.

Цикл сохранения может возникнуть совершенно невинно, поскольку отношения в графе объекта могут оказаться двухсторонними. Например, в системе заказов и товаров должны быть известны как заказываемые товары, так и заказы, в которых перечисляются товары. Поэтому может возникнуть ситуация, когда заказ сохраняет свои товары, а отдельный товар сохраняет свои заказы. В таком случае образуется цикл сохранения. В качестве примера, иллюстрирующего подобное затруднение, рассмотрим простой класс `MyClass` с единственной переменной экземпляра `_thing` и открытым методом установки `setThing:`, а также регистрацией в методе `dealloc`, как показано ниже.

```
@implementation MyClass {
   id _thing;
}

- (void) setThing: (id) what {
```

```
    self->_thing = what;
}

- (void)dealloc {
    NSLog(@"%@", @"dealloc");
}
@end
```

Теперь выполните следующий фрагмент кода:

```
MyClass* m1 = [MyClass new];
MyClass* m2 = [MyClass new];
m1.thing = m2;
m2.thing = m1;
```

Теперь объекты, на которые указывают переменные m1 и m2, сохраняют друг друга, поскольку механизм ARC по умолчанию сохраняет их во время присваивания. При выполнении приведенного выше фрагмента кода метод dealloc вообще не вызывается ни для одного из экземпляров класса MyClass — даже после того, как переменные m1 и m2, автоматически указывающие на их объекты, выйдут из области своего действия и будут уничтожены. В конечном итоге произойдет утечка памяти самих объектов типа MyClass.

Для того чтобы воспрепятствовать сохранению объекта, присваиваемого переменной экземпляра, достаточно указать ее как *слабую ссылку*. Это делается с помощью описателя __weak в объявлении переменной экземпляра, как показано ниже.

```
@implementation MyClass {
    __weak id _thing;
}
```

Теперь цикл сохранения не возникнет. В рассматриваемом здесь примере утечка памяти не возникает, а оба объекта типа MyClass прекращают свое существование по завершении выполнения кода из данного примера. Механизм ARC отправит каждому из них сообщение release, чтобы уравновесить новые вызовы для их создания, как только переменные m1 и m2, автоматически указывающие на эти объекты, выйдут из области своего действия, и никто больше не попытается их сохранить.

В механизме ARC ссылка, не объявленная явно слабой, считается *строгой ссылкой*. Следовательно, строгой оказывается такая ссылка, по которой механизм ARC сохраняет объект во время присваивания. На самом деле для обозначения строгих ссылок имеется специальный описатель __strong, но на практике вам вообще не придется его применять, поскольку это делается по умолчанию. Имеются еще два дополнительных, но редко применяемых описателя: __unsafe_unretained и __autoreleasing.

На практике слабая ссылка чаще всего применяется для связывания объекта с его делегатом (см. главу 11). Делегат является независимой сущностью, и поэтому у объекта нет никаких причин завладеть своим делегатом. На самом деле объект, как правило, обслуживает делегат, а не владеет им. Владение как таковое зачастую следует другим путем: объект A может создать *и сохранить* объект B, а также сделаться делегатом объекта B. Это может привести к возникновению цикла сохранения. Следовательно, большинство делегатов должны быть объявлены как слабые ссылки. Например, в исходном коде проекта, созданном в среде Xcode по шаблону Utility Application, можно обнаружить следующую строку:

```
@property (weak, nonatomic) id <FlipsideViewControllerDelegate> delegate;
```

где ключевое слово weak служит для объявления свойства, как подробнее поясняется далее в этой главе. Это равнозначно объявлению переменной экземпляра _delegate как слабой ссылки __weak.

Для того чтобы воспрепятствовать в коде без поддержки механизма ARC возникновению цикла сохранения по ссылке, достаточно не сохранять объект, когда он, прежде всего, присваивается этой ссылке. Ведь управление памятью вообще не распространяется на ссылку. Это можно видеть на примере слабой ссылки, хотя она и не подобна слабой ссылке при поддержке механизма ARC. Слабая ссылка без поддержки механизма ARC рискует превратиться в висячий указатель, когда экземпляр, на который она указывает, освобождается (каким-нибудь *другим*, сохранившим его объектом) и затем уничтожается. Следовательно, ссылка может быть непустой, но указывать на "мусор", и поэтому посылка ей сообщения может иметь таинственно пагубные последствия.

Как ни странно, этого не может произойти со слабой ссылкой при действующем механизме ARC. Когда подсчет сохраняемых ссылок на экземпляр достигает нуля, а сам экземпляр близок к исчезновению, при действующем механизме ARC любая слабая ссылка, указывавшая на него, автоматически становится пустой! (Это поразительное действие происходит подспудно в ходе служебных операций, когда объект присваивается слабой ссылке. По существу, механизм ARC отмечает этот факт в блокнотном списке.) Это *еще* одна причина, по которой следует, по возможности, отдавать предпочтение механизму ARC. Ведь этот механизм зачастую пренебрежительно, хотя и аккуратно относится к слабым ссылкам в коде, где он не поддерживается, считая их "небезопасными". (Такие ссылки на самом деле относятся к типу __unsafe_unretained, как упоминалось ранее.)

К сожалению, в *большей части* среды Cocoa механизм ARC *не* применяется. Управление памятью в среде Cocoa тщательно выписано, и поэтому операции сохранения и освобождения объектов из памяти в общем уравновешены, чтобы не вызывать утечек памяти. Тем не менее свойства встроенных классов Cocoa, поддерживающие слабые ссылки, являются слабыми ссылками без поддержки механизма ARC, поскольку они стары и обратно совместимы, тогда как механизм ARC является новым. Такие свойства объявляются с помощью ключевого слова assign. Например, свойство delegate из класса UINavigationController объявляется следующим образом:

```
@property(nonatomic, assign) id<UINavigationControllerDelegate> delegate
```

Следовательно, даже если в прикладном коде применяется механизм ARC, отсутствие его поддержки в среде Cocoa означает, что ошибки управления памятью по-прежнему возможны. Так, ссылка на делегат класса UINavigationController может в конечном итоге превратиться в висячий указатель на "мусор", если объект, на который она делается, прекратит свое существование. Если же кто-нибудь (вы или среда Cocoa) попытается послать сообщение по такой ссылке, приложение завершится аварийным сбоем, а поскольку нечто подобное, как правило, происходит намного позже момента совершения настоящей ошибки, то выяснить истинную причину аварийного сбоя будет очень трудно. Типичные признаки такого сбоя состоят в том, что он происходит в методе objc_retain от компании Apple и упоминается в исключении EXC_BAD_ACCESS (рис. 12.1). В таком случае, возможно, придется активизировать объекты-зомби в целях отладки прикладного кода, как пояснялось ранее в этой главе.

Предотвращение подобной ситуации входит в ваши обязанности. Если вы присвоите какой-нибудь объект (например, делегат класса UINavigationController) слабой ссылке без поддержки механизма ARC и если данный объект собирается прекратить свое существование в тот момент, когда эта ссылка по-прежнему существует, то вы обязаны присвоить ей пустое значение (nil) или какой-нибудь другой объект, чтобы сделать ее безвредной.

```
Thread 1
Queue: com.apple.main-thread
0 objc_retain
1 __57-[AppDelegate application:...
2 _dispatch_call_block_and_release
11 UIApplicationMain
12 main
13 start

8  0x1474ddf:  testl  %eax, %eax
9  0x1474de1:  je     0x1474dfa                    ; objc_retain + 42
10 0x1474de3:  movl   (%eax), %edx
11 0x1474de5:  movl   16(%edx), %edx   Thread 1: EXC_BAD_ACCESS (code=1, address=0xe08e6e87)
12 0x1474de8:  andl   $-4, %edx
13 0x1474deb:  testb  $64, 1(%edx)
14 0x1474def:  je     0x1474dfe                    ; objc_retain + 46
15 0x1474df1:  addl   $8, %esp
16 0x1474df4:  popl   %ebp
```

Рис. 12.1. Аварийный сбой в результате отправки сообщения по висячему указателю

Необычные случаи управления памятью

Класс `NSNotificationCenter` предоставляет ряд любопытных средств для управления памятью. Вам полезно будет знать о них, поскольку вы, скорее всего, будете пользоваться уведомлениями в своем прикладном коде (см. главу 11).

Если вы зарегистрировались в центре уведомлений, используя метод `addObserver:selector:name:object:`, значит, вы передали этому центру ссылку на некоторый объект (обычно ссылку `self`) в качестве первого аргумента данного метода. Это слабая ссылка без поддержки механизма ARC, и поэтому существует опасность, что после того, как данный объект прекратит свое существование, центр уведомлений попытается отправить уведомление по ссылке, которая, по сути, делается на "мусор". Именно поэтому вам нужно снять объект с регистрации на уведомления, прежде чем произойдет нечто подобное. Аналогичная ситуация возникает и с делегатами, как пояснялось ранее.

Если же вы зарегистрировались в центре уведомлений, используя метод `addObserverForName:object:queue:usingBlock:`, то управлять памятью, особенно, если действует механизм ARC, будет намного труднее по следующим причинам.

- Маркер наблюдателя, возвращаемый в результате вызова метода `addObserverForName:object:queue:usingBlock:`, сохраняется центром уведомлений до его снятия с регистрации.
- Маркер наблюдателя может также сохранять ваш объект (по ссылке `self`) через блок. В таком случае до тех пор, пока вы не снимите маркер наблюдателя с регистрации в центре уведомлений, он будет сохраняться в этом центре. Это означает утечку вашего объекта из памяти до тех пор, пока вы не снимите его с регистрации на получение уведомлений. Однако вы не сможете этого сделать в методе `dealloc`, поскольку его нельзя будет вызвать до тех пор, пока ваш объект зарегистрирован.
- Кроме того, если ваш объект сохраняет маркер наблюдателя, а тот — ваш объект, то невольно возникает цикл сохранения.

Рассмотрим следующий пример кода, в котором происходит регистрация на получение уведомлений и присваивание маркера наблюдателя переменной экземпляра:

```
self->_observer = [[NSNotificationCenter defaultCenter]
   addObserverForName:@"heyho"
     object:nil queue:nil usingBlock:^(NSNotification *n) {
       NSLog(@"%@", self);
   }];
```

В данном примере преследуется конечная цель — снять наблюдателя с регистрации. Именно поэтому сохраняется ссылка на него. Это сделать вполне естественно в методе `dealloc` следующим образом:

```
- (void) dealloc {
   [[NSNotificationCenter defaultCenter] removeObserver:self->_observer];
}
```

И все же такой прием оказывается неработоспособным. Ведь метод dealloc вообще не вызывается из-за появления цикла сохранения. Объект, доступный по ссылке self, сохраняет наблюдателя, но из-за блока наблюдатель также сохраняет объект self. Следовательно, происходит утечка этого объекта из памяти, и он вообще не регистрируется.

Нарушить цикл сохранения можно двумя способами. Один из них состоит в освобождении объекта _observer после его снятия с регистрации. До тех пор, пока этого не будет сделано, метод dealloc по-прежнему не будет вызываться. Следовательно, нужно найти другое место, кроме метода dealloc, где можно было бы снять с регистрации объект _observer и освободить его.

Так, если речь идет о контроллере типа UIViewController, то одним из таких мест может стать метод viewDidDisappear: из представления этого контроллера. Он вызывается при удалении представления контроллера из пользовательского интерфейса, как показано ниже.

```
- (void) viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:animated];
    [[NSNotificationCenter defaultCenter] removeObserver:self.observer];
    self->_observer = nil; // освободить наблюдатель
}
```

Когда наблюдатель снимается с регистрации, он освобождается центром уведомлений. Когда он освобождается еще и вручную, то уже никем больше не сохраняется. В итоге наблюдатель прекращает свое существование, попутно освобождая объект self, который может затем прекратить свое существование в должном порядке, когда вызывается метод dealloc и утечки этого объекта не происходит. С другой стороны, если переменная экземпляра _observer отмечена как слабая ссылка __weak, можно пропустить последнюю строку в приведенном выше фрагменте кода. Когда наблюдатель снимается с регистрации, он будет освобожден центром уведомлений, а поскольку наблюдатель сохранялся только этим центром, то он разрушается, попутно освобождая объект self.

Такой подход требует аккуратного управления памятью, потому что метод viewDidDisappear: может быть вызван не один раз в течение срока действия контроллера представления. Поэтому придется зарегистрироваться на получение уведомлений в каком-нибудь другом симметричном месте, например, в методе viewWillAppear:. Если объект self *не* является контроллером представления, то найти подходящее место, кроме метода dealloc, чтобы снять этот объект с регистрации, будет не так-то просто.

На мой взгляд, лучше принять следующее решение: не позволить наблюдателю сохранить объект self. Благодаря этому в первую очередь исключается появление цикла сохранения. Для того чтобы объект self не сохранился в блоке, достаточно не упоминать его (или любую переменную его экземпляра) в блоке. Раз уж не возникает цикл сохранения, то будет вызван метод dealloc, где появится возможность снять с регистрации наблюдатель, что и требовалось изначально сделать.

Так как же избежать упоминания объекта self в блоке, если этому объекту требуется послать сообщение в блоке? Для этого можно воспользоваться несложным, но изящным приемом, называемым "пляской слабых и сильных ссылок" (см. пример 12.9). Эффективность такого приема объясняется тем, что ссылка вообще не упоминается в блоке непосредственно. На самом деле ссылка на объект self не передается в блоке, но в этот момент она становится слабой, чего оказывается достаточно, чтобы предотвратить сохранение в блоке объекта self, а возможно, и возникновение цикла сохранения. В самом же блоке эта ссылка преобразуется в строгую ссылку, а далее все продолжается, как обычно.

Пример 12.9. Пляска слабых и строгих ссылок, препятствующая сохранению в блоке объекта self

```
__weak MyClass* wself = self; ❶
self->_observer = [[NSNotificationCenter defaultCenter]
   addObserverForName:@"heyho"
   object:nil queue:nil usingBlock:^(NSNotification *n) {
      MyClass* sself = wself; ❷
      if (sself) {
         // свободно обратиться к объекту по ссылке sself,
         // но ни в коем случае не по ссылке self ❸
      }
}];
```

Ниже перечислены стадии "пляски слабых и строгих ссылок", схематически обозначенные в примере 12.9.

❶ Локальная слабая ссылка на объект `self` формируется за пределами блока, но там, где она может быть доступна из блока. Именно такая слабая ссылка и будет передана в блок.

❷ В блоке эта слабая ссылка присваивается обычной строгой ссылке. Слабые ссылки носят временный характер. Это означает, что объект, доступный по слабой ссылке (даже по ссылке `self`), может исчезнуть из виду при переходе от одной строки кода к другой. В данном случае слабая ссылка окажется пустой, но отправка сообщения объекту обойдется недешево, а прямое обращение к переменной его экземпляра было бы пагубным. Более того, для проверки, является ли ссылка пустой, отсутствует подходящий потокобезопасный способ. Поэтому в качестве выхода из этого положения слабая ссылка присваивается строгой ссылке.

❸ Строгая ссылка используется вместо любых ссылок на объект `self` в блоке. Вся операция, связанная со строгой ссылкой, заключена в проверке на пустую ссылку. Ведь если объект, доступный по слабой ссылке, исчезнет из виду, то пропадет всякий смысл продолжать дальше.

Еще один необычный для управления памятью случай представляет класс `NSTimer` (см. главу 10). В документации на класс `NSTimer` говорится, что в циклах исполнения сохраняются их таймеры, а в отношении метода `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` — целевой объект сохраняется таймером и освобождается, когда таймер становится недействительным. Это означает, что если не сделать периодический таймер недействительным, то целевой объект будет сохранен в цикле исполнения. Единственный способ воспрепятствовать этому — послать таймеру сообщение `invalidate`. (Подобное затруднение не возникает с непериодическим таймером, поскольку такой таймер становится недействительным сразу же после своего срабатывания.)

При вызове метода `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` в качестве его аргумента `target:`, вероятнее всего, предоставляется объект `self`. Это означает, что объект `self` сохраняется и не может прекратить свое существование до тех пор, пока таймер не будет сделан недействительным. Однако этого нельзя сделать в собственной реализации метода `dealloc`, поскольку данный метод невозможно вызвать до тех пор, пока периодически срабатывающему таймеру не будет послано сообщение `invalidate`. Таким образом, нужно найти какой-нибудь другой подходящий момент для отправки сообщения `invalidate` таймеру. Другого удобного выхода из этого положения не существует, поэтому остается только подыскать такой момент.

Блоковую альтернативу периодическому таймеру предоставляет технология GCD. "Объект" таймера `dispatch_source_t` должен быть сохранен, как правило, в качестве переменной

экземпляра, автоматическое управление памятью для которого берет на себя механизм ARC, хотя это и псевдообъект. Такой таймер будет срабатывать периодически после его первоначального "возобновления", а его срабатывание прекратится, как только он будет освобожден из памяти, для чего, как правило, переменной его экземпляра присваивается пустое значение (nil). Для того чтобы таймер не заблокировался при освобождении и не возник цикл сохранения, как это происходит с наблюдателями уведомлений, необходимо принять соответствующие меры предосторожности. Ниже приведена типичная заготовка кода, реализующего такой таймер.

```
@implementation MyClass {
   dispatch_source_t _timer; // этим псевдообъектом будет управлять ARC
}

- (void)doStart:(id)sender {
   self->_timer = dispatch_source_create(
       DISPATCH_SOURCE_TYPE_TIMER,0,0,dispatch_get_main_queue());
   dispatch_source_set_timer(
       self->_timer, dispatch_walltime(nil, 0),
       1 * NSEC_PER_SEC, 0.1 * NSEC_PER_SEC
       );
   __weak id wself = self;
   dispatch_source_set_event_handler(self->_timer, ^{
       MyClass* sself = wself;
       if (sself) {
           [sself dummy:nil]; // предотвращает цикл сохранения
       }
   });
   dispatch_resume(self->_timer);
}

- (void)doStop:(id)sender {
   self->_timer = nil;
}

- (void) dummy: (id) dummy {
    NSLog(@"timer fired");
}

- (void) dealloc {
    [self doStop:nil];
}
@end
```

Другие классы Cocoa с необычным управлением памятью, как правило, ясно разъясняются в документации. Например, в документации на класс UIWebView предупреждается, что перед освобождением экземпляра класса UIWebView, для которого установлен делегат, нужно сначала установить пустое значение (nil) его свойства delegate. Объект класса CAAnimation *сохраняет* свой делегат. Это исключительный случай, способный доставить немало хлопот, если не знать о нем.

Имеются также случаи, когда в документации отсутствуют особые соображения по поводу управления памятью, но сам механизм ARC может предупредить о возможном возникновении цикла сохранения из-за применения самоссылки в блоке. Примером тому служит обработчик завершения кода в методе экземпляра setViewControllers:direction:animated:completion: класса UIPageViewController. Если в блоке completion: содержится ссылка на тот же самый экземпляр класса UIPageViewController, которому этот метод посылается, то компилятор выдаст предупреждение о том, что строгая фиксация контроллера

представления страницы (pvc или под другим именем), скорее всего, приведет к возникновению цикла сохранения. Вместо этого можно организовать слабую фиксацию контроллера представления страницы pvc, применяя описанный ранее прием "пляски слабых и строгих ссылок".

В отношении управления памятью основные классы коллекций, внедренные в версии iOS 6, в том числе NSPointerArray, NSHashTable и NSMapTable, аналогичны соответствующим классам NSMutableArray, NSMutableSet и NSMutableDictionary, но выбор конкретной стратегии управления памятью, выделяемой для объектов этих классов, зависит от вас. Например, объект типа NSHashTable, создаваемый методом класса weakObjectsHashTable, поддерживает слабые ссылки на свои элементы. При действующем механизме ARC эти слабые ссылки заменяются пустыми, если подсчет сохраняемых ссылок на указываемый ими объект уменьшается до нуля. Можно найти немало примеров применения этих классов для исключения циклов сохранения.

Загрузка nib-файлов и управление памятью

При загрузке nib-файла в системе iOS nib-объекты верхнего уровня, экземпляры которых при этом получаются, становятся автоматически освобождаемыми. Если не сохранить их так или иначе, то они вскоре исчезнут как дым. Предотвратить это можно двумя основными способами.

Сохранение объектов верхнего уровня

Если при загрузке nib-файла вызвать метод loadNibNamed:owner:options: или instantiateWithOwner:options: из класса UINib (см. главу 7), то в итоге возвратится массив типа NSArray, состоящий из объектов верхнего уровня, экземпляры которых получаются с помощью механизма загрузки nib-файлов. Следовательно, достаточно сохранить этот массив типа NSArray или же содержащиеся в нем объекты.

Например, когда экземпляр контроллера представления автоматически получается из раскадровки, он фактически загружается из nib-файла лишь с одним объектом верхнего уровня — контроллером представления. В конечном итоге из метода instantiateWithOwner:options: возвращается контроллер представления как единственный элемент массива. Этот контроллер представления затем сохраняется динамически в назначенном для него месте в иерархии контроллеров представления. Например, в приложении с основной раскадровкой функция UIApplicationMain() служит для получения экземпляра первоначального контроллера представления из своего nib-файла и его присваивания свойству rootViewController объекта окна, который и сохраняет его.

Граф объекта

Если экземпляр объекта получается из nib-файла и является местом назначения для выхода, объект в источнике этого выхода может сохранить его (как правило, методом доступа). Таким образом, в цепочке выходов могут сохраняться многие объекты (рис. 12.2). Например, когда контроллер представления автоматически загружает свое основное представление из nib-файла, у его заместителя из nib-файла имеется выход представления к объекту верхнего уровня типа UIView в этом же nib-файле, а в методе setView: из класса UIViewController сохраняется его параметр.

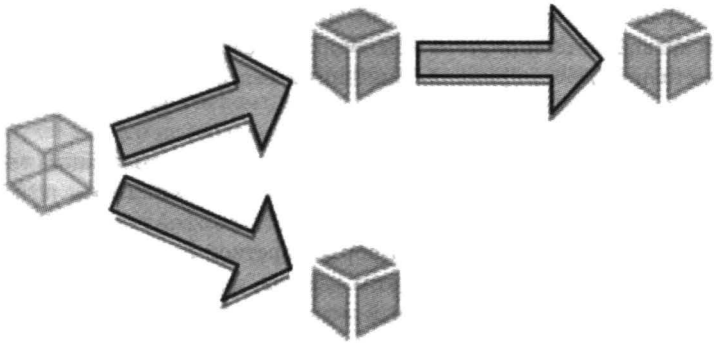
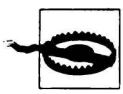


Рис. 12.2. Граф объекта с сохранением

Более того, основное представление сохраняет свои подчиненные представления, чтобы сохранились все они, их подчиненные представления и так далее по иерархии. (Именно поэтому экземпляр класса `IBOutlet` или свойство, создаваемое вами в ваших собственных контроллерах, как правило, обозначается как слабая ссылка __weak. Несмотря на то что это необязательно, такая ссылка, как правило, не должна быть строгой __strong, поскольку интерфейсный объект назначения в nib-файле уже подлежит сохранению благодаря своему месту в графе объекта.)



Предупреждение для программистов в системе OS X

Управление памятью, выделяемой для экземпляров объектов из загружаемых nib-файлов, осуществляется в системе OS X иначе, чем в системе iOS. Так, в системе OS X экземпляры объектов из загружаемых nib-файлов не являются автоматически освобождаемыми, и поэтому они не должны сохраняться. Управление памятью в любом случае осуществляется автоматически, поскольку владельцем файла обычно является контроллер типа `NSWindowController`, который берет на себя все эти обязанности. В системе iOS память, выделяемая для nib-объектов верхнего уровня, обязательно подлежит управлению.

Управление памятью для глобальных переменных

В языке C, а следовательно, и в Objective-C, переменную разрешается объявлять за пределами любого метода. В книге *K&R* (см. главу 1) такая переменная называется *внешней* (см. *K&R*, раздел 4.3), а я называю ее *глобальной переменной*. Обычно принято, хотя и не строго обязательно, уточнять объявление такой переменной как `static`. Такое уточнение имеет отношение к области действия переменной и не оказывает влияния на ее сохраняемость или глобальный характер.

Глобальные переменные относятся к типу переменных языка C, определяемому на уровне файла. Им ничего неизвестно об экземплярах. В той степени, в какой глобальные переменные имеют отношение к объектно-ориентированному программированию, они могут быть определены в файле класса, а следовательно, они, по существу, являются переменными уровня класса. В коде Objective-C глобальная переменная зачастую используется как константа. Иногда глобальную переменную можно инициализировать при ее объявлении, как показано ниже.

```
NSString* const MYSTRING = @"my string";
```

Если само значение, которое требуется присвоить глобальной переменной, не является константой, то его придется присваивать в конкретном коде, и тогда возникает вопрос: где разметить такой код? Глобальная переменная, по существу, является переменной уровня класса, и поэтому ее целесообразно инициализировать на ранней стадии существования класса, т.е. в методе `initialize` (см. главу 11).

Если значение глобальной переменной определяется на уровне класса, то оно сохраняется в течение всего времени работы программы. Глобальные переменные не имеют никакого отношения к управлению памятью, выделяемой для экземпляров, поскольку они никак не связаны с экземплярами. В каком-то смысле можно говорить об утечке глобальных переменных из памяти, но следует подчеркнуть, что это происходит так же безвредно, как и утечка объектов классов.

Если не пользоваться механизмом ARC, то управлять памятью, выделяемой для глобальных переменных, придется таким же образом, как и для переменных экземпляра. В частности, объект, присваиваемый глобальной переменной, должен быть сохранен, и если в дальнейшем ей будет присвоен другой объект, то первый объект должен быть освобожден. При действующем механизме ARC первоначальное присваивание объекта глобальной переменной приводит к его сохранению, а при последующем присваивании управление памятью осуществляется автоматически и корректно. Как правило, объект, присваиваемый глобальной переменной, не освобождается, а остается до тех пор, пока существует класс, т.е. в течение всего времени работы приложения, что совсем неплохо.

Управление памятью для ссылок типа `CFTypeRef`

Ссылка типа `CFTypeRef` является указателем на структуру, а имя ее типа обычно оканчивается на Ref (см. главу 3). Она обычно получается с помощью функции языка C, где предусмотрены функции для обращения с ней. Эта ссылка похожа на объект, хотя и не является полноценным объектом Cocoa в языке Objective-C, и поэтому выделяемой для нее памятью следует управлять таким же образом, как и для объекта Cocoa. Тем не менее в механизме ARC отсутствует поддержка для такого управления памятью. Этот механизм управляет памятью, выделяемой для объектов Objective-C, но не для ссылок типа `CFTypeRef`. Поэтому управлять памятью, выделяемой для ссылок типа `CFTypeRef`, приходится вручную, даже если применяется механизм ARC. Пересекая своего рода мост между сторонами ссылок типа `CFTypeRef` и полноценных объектов Objective-C при обмене объектами, вы должны помочь механизму ARC уяснить его обязанности в отношении управления памятью.

Об ответственности за управление памятью, выделяемой для объектов Objective-C, напоминают имена некоторых методов (`alloc`, `copy` и `retain`), и это же относится к ссылкам типа `CFTypeRef`. Золотое правило в данном случае состоит в следующем: если вы получаете объект типа `CFTypeRef` с помощью функции, в имени которой содержится слово `Create` или `Copy`, то на вас возлагается ответственность за его освобождение из памяти. По умолчанию такой объект освобождается из памяти с помощью функции `CFRelease()`, но некоторые функции создания объектов применяются в паре со своими функциями освобождения из памяти.

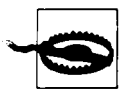
В качестве примера ниже приведен фрагмент кода, взятый из одного из моих приложений. Этот код строго смоделирован по примеру кода от компании Apple, и в нем устанавливается цветовое пространство для базового образца, предназначенного для рисования.

```
- (void) addPattern: (CGContextRef) context color: (CGColorRef) incolor {
  CGColorSpaceRef baseSpace = CGColorSpaceCreateDeviceRGB();
  CGColorSpaceRef patternSpace = CGColorSpaceCreatePattern(baseSpace);
  CGContextSetFillColorSpace(context, patternSpace);
  CGColorSpaceRelease(patternSpace);
```

```
    CGColorSpaceRelease(baseSpace);
    // ...
}
```

В данном примере не так важно назначение приведенного выше фрагмента кода, как то обстоятельство, что значения ссылок `baseSpace` и `patternSpace` относятся к типу `CFTypeRef`, а точнее — к типу `CGColorSpaceRef`. Эти ссылки получаются с помощью функций, в именах которых содержится слово `Create`, а следовательно, они освобождаются из памяти с помощью функции `CGColorSpaceRelease()`, эквивалентной методу `release`, после их применения.

Аналогично с помощью функции `CFRetain()` можно сохранить объект типа `CFTypeRef`, если есть опасение, что он может прекратить свое существование в то время, когда он все еще требуется. Однако после этого следует обязательно уравновесить данную операцию вызовом функции `CFRelease()`.



Сообщения можно посылать объекту Objective-C даже в том случае, если он пустой. В то же время функция `CFRelease()` не может принимать пустое значение (`nil`) в качестве своего аргумента. Поэтому убедитесь в том, что переменная типа `CFTypeRef` не содержит пустое значение, прежде чем освобождать ее.

Теперь обсудим, как же пересечь упомянутый выше мост между ссылками типа `CFTypeRef` и полноценными объектами Objective-C. Как пояснялось в главе 3, многие типы объектов из библиотеки Core Foundation свободно состыкованы подобными мостами с соответствующими типами объектов Cocoa. (Их перечень приведен в главе "Toll-Free Bridged Types" документации *Core Foundation Design Concepts*, предоставляемой компанией Apple.) Теоретически управление памятью остается таковым независимо от того, применяется ли оно в каркасе Core Foundation или среде Cocoa. Так, если получить ссылку типа `CFStringRef` с помощью функции `Create()` или `Copy()` и затем присвоить ее переменной экземпляра типа `NSString`, то отправка ей сообщения `release` через переменную экземпляра типа `NSString` окажется столь же полезной, как и вызов для нее функции `CFRelease()`. До появления механизма ARC управление памятью, выделяемой для данной ссылки, на этом завершалось.

Механизм ARC не позволяет поручить или изъять из его компетенции управление памятью, выделяемой для объекта, без явно указанной информации о том, как это управление должно осуществляться. Когда действует механизм ARC и объект начинает свое существование в результате получения его экземпляра, управление выделяемой для него памятью осуществляется механизмом ARC от начала и до конца существования этого объекта. Когда выполняется приведение типа объекта Objective к типу `CFTypeRef`, управление выделяемой для него памятью поручается компетенции механизма ARC. Однако механизм ARC не станет этого делать без дополнительной информации, поскольку неясны его обязанности по управлению памятью в данный момент. Аналогично, когда тип `CFTypeRef` приводится к типу объекта, управление памятью, выделяемой для готового экземпляра этого объекта, поручается компетенции механизма ARC, но опять же он не станет этого делать без дополнительной информации. Если же попытаться пересечь мост между ссылкой типа `CFTypeRef` и полноценным объектом, не предоставив механизму требующуюся ему информацию, компилятор выдаст сообщение об ошибке `"Implicit conversion ... requires a bridged cast"` (Неявное преобразование ... требуется стыковочное приведение типов).

Для правильного предоставления такой информации от вас потребуется чуть больше умственных усилий, но это даже хорошо, потому что вы сможете сообщить механизму ARC, что он должен сделать автоматически из того, вам иначе пришлось бы делать вручную. Например,

до появления механизма ARC вы могли получить ссылку типа CFStringRef с помощью функции Create() или Copy(), привести ее к типу NSString, а в дальнейшем послать ей сообщение release как объекту типа NSString. Если же действует механизм ARC, вы уже не сможете послать сообщение release, но в то же время можете настроить механизм ARC на те же самые действия: пересекая мост между ссылкой типа CFStringRef и полноценным объектом, передать эту ссылку с помощью функции CFBridgingRelease(). В результате будет получен идентификатор id, который может быть присвоен переменной экземпляра типа NSString. В конечном итоге механизм освободит этот идентификатор из памяти, чтобы уравновесить подсчет сохраняемых ссылок, первоначально установленный функцией Create() или Copy().

Свободное стыкование типов может быть достигнуто тремя способами.

Приведение к типу __bridge

Приведение выполняется к явно определяемому стыковочному типу __bridge, наводящему мост между исходным и целевым типами данных. Это означает, что обязанности по управлению памятью не зависят ни от одной из сторон стыкования приводимых типов.

Ниже приведен пример (из главы 9), в котором осуществляется стыковочное приведение типа объекта Objective-C к типу ссылки CFTypeRef. Объект imageSource имеет тип NSURL и поэтому свободно стыкуется со ссылкой типа CFURL.

```
CGImageSourceRef src =
    CGImageSourceCreateWithURL((__bridge CFURLRef)imageSource, nil);
```

Механизму ARC нужно сообщить его обязанности по управлению памятью. В данном случае они отсутствуют, а следовательно, объект imageSource типа NSURL будет и дальше существовать на стороне объектов Objective-C, и управление выделяемой для него памятью должно осуществляться механизмом ARC, как обычно. Объект imageSource предоставляется функции CGImageSourceCreateWithURL() лишь на мгновение, чтобы из него можно было сформировать ссылку типа CGImageSourceRef. С другой стороны, объект src типа CGImageSourceRef будет полностью существовать на стороне ссылок типа CFTypeRef, и поэтому управлять выделяемой для него памятью придется вручную, вызывая для него функцию CFRelease() после того, как он больше не нужен.

Функция CFBridgingRelease()

В этом случае осуществляется стыкование ссылочного типа CFTypeRef с типом объекта Objective-C, а механизм ARC уведомляется, что управление памятью, выделяемой для этого объекта, не завершено, т.е. подсчет сохраняемых ссылок на стороне ссылочного типа CFTypeRef был увеличен (возможно, в результате формирования ссылки с помощью функции Create() или Copy()). Поэтому на механизм ARC возлагается обязанность выполнить в конечном итоге соответствующее освобождение из памяти на стороне типа объекта. С другой стороны, можно выполнить приведение к стыковочному типу __bridge_transfer. Ниже представлен пример фрагмента кода из настоящего приложения, где он следует сразу же после фрагмента кода из предыдущего примера.

```
CFDictionaryRef res1 = CGImageSourceCopyPropertiesAtIndex(src, 0, nil);
NSDictionary* res = CFBridgingRelease(res1);
```

В данном примере функция CFRelease() *вообще* не будет вызвана для объекта res1 типа CFDictionaryRef. Он был создан на стороне ссылочного типа CFTypeRef, поскольку для этого потребовалась функция CGImageSourceCopyPropertiesAtIndex(), что в конечном итоге привело к увеличению подсчета сохраняемых ссылок (вследствие

копирования объекта). Стыкование типов произошло полностью и бесповоротно. В итоге был получен объект res языка Objective-C, который механизму ARC придется освободить из памяти, когда для этого настанет подходящий момент.

Функция CFBridgingRetain()

Это совершенно противоположный случай по сравнению с применением функции CFBridgingRelease(). В данном случае происходит стыкование типа объекта Objective-C с ссылочным типом CFTypeRef. Механизм ARC уведомляется о необходимости оставить незавершенным управление памятью, выделяемой для данного объекта. Принимая во внимание увеличение подсчета сохраняемых ссылок на стороне типа объекта Objective-C, функцию CFRelease() придется вызвать вручную на стороне ссылочного типа CFTypeRef. С другой стороны, можно выполнить приведение к стыковочному типу __bridge_retained.

Как правило, объект Objective-C необязательно рассматривать как свободно состыкованный эквивалент типа CFTypeRef, чтобы обращаться с ним, используя функции C, позволяющие сделать то, чего нельзя сделать в противном случае. Теоретически на стороне ссылок типа CFTypeRef управление памятью не требуется, поскольку объект будет существовать на стороне объектов Objective-C. Однако в действительности он может и *не* существовать на стороне объектов Objective-C, поскольку он мог быть освобожден из памяти, и тогда ссылка типа CFTypeRef будет делаться на "мусор". Во избежание этого следует вызвать функцию CFBridgingRetain() при стыковании типов, а по завершении работы на стороне ссылок типа CFTypeRef — вызвать функцию CFRelease(). С другой стороны, вполне возможно передать объект со стороны объектов Objective-C, вызвав функцию CFBridgingRetain(), а в дальнейшем возвратить его обратно с помощью функции CFBridgingRelease().

Иногда ссылку типа CFTypeRef требуется присвоить переменной id или параметру метода. Например, метод setContents: из класса CALayer ожидает получить параметр id, но его конкретным значением должна быть ссылка типа CGImageRef. Это вполне допустимо, поскольку, как упоминалось в главе 3, любая ссылка типа CFTypeRef свободно состыкована с параметром id. Тем не менее компилятор выдаст предупреждение, если не выполнить приведение к типу параметра id, и для этой цели может также потребоваться описатель стыковочного типа __bridge, как показано ниже.

```
CGImageRef moi = // что-нибудь одно или другое
self.v.layer.contents = (__bridge id)moi;
```

Если же ссылка типа CFTypeRef поступает из встроенного метода без промежуточного присваивания переменной, то описатель стыковочного типа __bridge может и не потребоваться, как следует из приведенного ниже примера.

```
self.v.layer.contents = (id)[UIImage imageNamed:@"moi"].CGImage;
```

Дело в том, что встроенный метод (например, метод экземпляра CGImage из класса UIImage) сам предоставляет сведения о стыковании типов, вполне удовлетворяющие компилятор. Это видно из следующего заголовка метода CGImage:

```
- (CGImageRef)CGImage NS_RETURNS_INNER_POINTER;
```

Управление памятью для данных контекста пустых указателей

Некоторые методы из классов Cocoa принимают необязательный параметр, обозначаемый как void* и нередко называемый context:. На первый взгляд, параметр void*, обозначающий тип универсального указателя, подобен параметру id, обозначающему универсальный тип объекта, поскольку ссылка на объект является указателем. На самом деле параметр id обозначает универсальный тип объекта, тогда как параметр void* — просто указатель языка C. Это означает, что в среде Cocoa значение типа void* не интерпретируется как объект, и поэтому выделяемая для него память не требует автоматического управления. Таким образом, в ваши обязанности входит обеспечить сохранение этого значения в памяти настолько долго, насколько это будет для вас полезно.

Насколько долго этого будет достаточно? Назначение аргумента context:, передаваемого методу, состоит в том, чтобы быть возвращенным в качестве параметра context: при обратном вызове и помочь распознать что-нибудь или предоставить дополнительные сведения. Так, если предоставить значение параметра context: при вызове метода beginAnimations:context:, то от среды Cocoa можно ожидать вызова реализации метода animationDidStop:finished:context: с тем же самым значением параметра context:. Следовательно, значение параметра context: необходимо сохранить до тех пор, пока не будет сделан обратный вызов метода animationDidStop:finished:context:.

Иногда параметр context: обозначает только идентификатор с намерением лишь сравнить параметр context: в обратном вызове с исходно переданным аргументом context:. Следовательно, он может быть сохранен как внешняя переменная в том классе, где он используется. Ниже приведен обычно рекомендуемый образец такого применения параметра context: в коде.

```
static void* const MYCONTEXT = (void*)&MYCONTEXT;
```

В данном случае имя MYCONTEXT имеет значение, указывающее на собственную область памяти, а параметр context: в обратном вызове может быть сравнен с ним непосредственно. Еще одним аргументом context:, часто рекомендуемым в качестве значения идентификатора, служит ссылка на объект self. Очевидно, что такая ссылка будет существовать при условии, что ей может быть отправлено сообщение обратного вызова, а следовательно, она имеет прямую сохраняемость.

С другой стороны, параметр context: может обозначать отдельный полноценный объект. В этом случае его нужно где-то хранить. Для этой цели лучше всего подходит переменная экземпляра, поскольку внешняя переменная действует на уровне класса, а для работы с другими значениями параметра context: могут потребоваться иные экземпляры данного класса.

Если действует механизм ARC, то объект Objective-C нельзя передать вместо ожидаемого значения типа void*, и наоборот, не указав дополнительные сведения для управления памятью. Это еще один случай "пересечения моста" (т.е. стыкования типов), рассматривавшегося в предыдущем разделе. Любое управление памятью уже произведено на стороне объектов Objective-C, и поэтому достаточно простого приведения к стыковочному типу __bridge. Если же в качестве аргумента context: передается значение, то требуется приведение его типа (__bridge void*), а когда оно возвращается из обратного вызова, — обратное приведение его типа (__bridge id).

Приведенные выше соображения не распространяются на параметры, обозначаемые как объекты. Например, при вызове метода postNotificationName:object:userInfo: параметр userInfo: обозначается как объект типа NSDictionary и сохраняется автоматически центром уведомлений, а освобождается после рассылки уведомления. Управление памятью, выделяемой для этого объекта, происходит подспудно и поэтому не является вашей заботой.

Свойства

Свойство (property) — это удобное синтаксическое средство для вызова метода доступа с помощью записи через точку (см. главу 5). В приведенных ниже строках кода представлены равнозначные способы вызова метода для установки значения свойства.

```
[self setTheData: d];
self.theData = d;
```

В следующих строках кода представлены равнозначные способы вызова метода для получения значения свойства.

```
d = [self theData];
d = self.theData;
```

Именно объявление метода доступа позволяет использовать соответствующую запись свойства. В частности, объявление метода установки дает возможность использовать запись свойства в левом значении выражения для его присваивания свойству, а объявление get-метода — использовать запись свойства в правом значении выражения для его извлечения из свойства.

Свойство можно объявить и явно вместо того, чтобы объявлять get- и set-методы. Следовательно, объявление свойства служит в качестве сокращенного способа объявления методов доступа, как, впрочем, и для сокращенного вызова этих методов. Объявление свойства позволяет добиться много большего в прикладном коде, а насколько больше — это зависит от конкретной системы и применяемой версии Xcode. Возможности объявления свойства перечислены ниже по порядку увеличения их потенциала, что отражает историю их внедрения.

- Объявление свойства избавляет от необходимости объявлять методы доступа. Ведь намного проще объявить одно свойство, чем два метода доступа.
- В объявление свойства включается оператор, определяющий *стратегию управления памятью* для метода установки. Это дает вам как программисту возможность легко выяснить, взглянув только на объявление свойства, каким образом будет интерпретироваться входящее значение. В противном случае для этого придется заглядывать в исходный код метода установки. Если метод установки принадлежит встроенному классу Cocoa, то вам не удастся этого сделать. Даже в своем коде не так-то просто обратиться за непосредственной справкой к нужному методу установки.
- С помощью объявления свойства можно дополнительно *опустить* написание одного или обоих методов доступа. Компилятор сделает это автоматически! Для этого достаточно ввести директиву `@synthesize` в раздел реализации класса. Вполне естественно, что такой автоматически конструируемый метод доступа называется *синтезированным*.
- Написание методов доступа — это непростое и чреватое ошибками занятие, поэтому следует выгодно пользоваться любой возможностью для автоматического написания правильного кода. Более того, это не потребует от вас никаких усилий, если созданный вами класс совместим с механизмом доступа к значениям по ключам для имени метода доступа. Кроме того, стратегия управления памятью для метода установки, указанная в объявлении свойства, соблюдается синтезированным методом установки. Любое ваше желание для среды Cocoa закон!
- С помощью синтезированного метода доступа можно дополнительно *опустить* объявление переменной экземпляра, получаемой или устанавливаемой методом доступа,

поскольку такое объявление делается неявно и автоматически! Неявное автоматическое объявление переменных экземпляра было внедрено как часть главного усовершенствования в языке Objective-C. В документации прежний период развития этого языка, когда переменные экземпляра приходилось объявлять самостоятельно, даже с помощью объявленного свойства и синтезированного метода, называется "устаревшей средой исполнения", а позднейший период, когда стало возможным автоматическое неявное объявление переменных экземпляра, — "современной средой исполнения". Как часть директивы `@synthesize` можно указать имя переменной экземпляра, которая объявляется неявно и автоматически.

- Максимальные удобства предоставляет современный компилятор LLVM, начиная с версии 4.0 (Xcode 4.4). Теперь можно вообще опустить директиву `@synthesize`, поскольку компилятор вводит ее неявно и автоматически! Это так называемый *автоматический синтез*.
- Единственный недостаток автоматического синтеза заключается в том, что из-за пропуска директивы `@synthesize` просто негде указать имя автоматически объявляемой переменной экземпляра. Впрочем, это не такой уж и крупный недостаток, поскольку имя переменной экземпляра предоставляется согласно простому правилу, чего оказывается достаточно в большинстве случаев.

Благодаря автоматическому синтезу одного лишь присутствия объявления свойства (в единственной строке кода) оказывается достаточно для запуска всех видов автоматического поведения. Это равнозначно объявлению методов доступа как вручную, так и автоматически (в соответствии объявленной стратегией управления памятью), а также неявному автоматическому объявлению переменных экземпляра.

Стратегии управления памятью для свойств

Возможные стратегии управления памятью, выделяемой для свойств, имеют непосредственное отношение к тому, что уже было сказано ранее в этой главе о ссылочных типах механизма ARC и возможном поведении методов установки. Ниже приводится краткое описание этих стратегий.

`strong, retain`

Эта стратегия выбирается по умолчанию. Оба термина являются полными синонимами друг друга и могут употребляться в прикладном коде как с поддержкой механизма ARC, так и без его поддержки. Термин `retain` унаследован из периода, предшествовавшего появлению механизма ARC. При действующем механизме ARC сама переменная экземпляра становится обычной (строгой) ссылкой, и когда ей, так или иначе, будет присвоено значение, механизм ARC сохранит входящее значение и освободит из памяти существующее значение этой переменной экземпляра. Если механизм ARC не действует, то метод установки сохранит входящее значение и освободит из памяти существующее значение переменной экземпляра.

`copy`

Эта стратегия подобна стратегии `strong` или `retain`, за исключением того, что метод доступа копирует входящее значение переменной экземпляра, посылая ей сообщение `copy`, а копия с уже увеличенным подсчетом сохраняемых ссылок присваивается переменной экземпляра. Такая стратегия особенно пригодна в том случае, если у неизменяемого класса

имеется изменяемый подкласс (например, NSString и NSMutableString или NSArray и NSMutableArray) и требуется предотвратить передачу из кода, вызывающего метод установки, объект изменяемого подкласса. В соответствии с принципом полиморфизма (см. главу 5) это вполне допустимо в коде, вызывающем метод установки, поскольку там, где ожидается экземпляр класса, может быть также передан экземпляр его подкласса. Однако при вызове метода copy создается экземпляр неизменяемого класса (см. главу 10), что препятствует вызывающему коду сохранить ссылку на входящее значение и в дальнейшем видоизменить его без вашего ведома.

weak

При действующем механизме ARC переменная экземпляра становится слабой ссылкой. Механизм ARC присвоит ей входящее значение, не сохраняя его, а также незаметно установит в ней пустое значение, если экземпляр, на который она ссылается, прекратит свое существование. Как пояснялось ранее в этой главе, такая стратегия удобна для защиты от возможного возникновения цикла сохранения или для сокращения издержек, когда заранее известно, что управление памятью не требуется, как это происходит с интерфейсным объектом, который уже сохранен своим родительским представлением. Метод установки может быть синтезирован только при действующем механизме ARC. Несмотря на то что применять стратегию weak в коде без поддержки механизма ARC теоретически возможно, на практике это вряд ли стоит делать.

assign

Эта стратегия унаследована из периода, предшествовавшего появлению механизма ARC, и применяется таким же образом, как и стратегия weak. Метод установки не управляет памятью, а входящее значение непосредственно присваивается переменной экземпляра. В этом случае переменная экземпляра *не* становится слабой ссылкой с поддержкой механизма ARC, и в ней *не* устанавливается автоматически пустое значение, если экземпляр, на который она ссылается, прекращает свое существование. Она оказывается слабой ссылкой без поддержки механизма ARC (__unsafe_unretained) и в конечном итоге может стать висячим указателем.

Как упоминалось ранее, объявленная стратегия управления памятью, выделяемой для свойства, является инструкцией компилятору, если метод установки синтезируется. Если метод установки *не* синтезируется, то объявленная стратегия управления памятью оказывается "чисто условной", как отмечается в документации. Это означает, что если написать свой метод доступа, то его поведение лучше сделать таким, как объявлено, хотя ничто к этому не принуждает.

Синтаксис объявления свойств

Свойства могут быть объявлены везде, где объявляются методы, например, в разделе интерфейса заголовочного файла с расширением .h, в расширении класса или объявлении протокола. Ниже схематически показан синтаксис объявления свойства.

```
@property (атрибут, атрибут, ...) тип имя;
```

Вот пример объявления свойства:

```
@property (nonatomic, strong) NSMutableArray* theData;
```

Обычно тип и имя соответствуют типу и имени переменной экземпляра, но на самом деле они обозначают имя свойства (в записи через точку), а также имена используемых по

умолчанию set-метода (`setTheData:`) и get-метода (`theData`) наряду с типом значения, передаваемого set-методу установки и возвращаемого get-методом. Если данное свойство может быть представлено выходом в nib-файле, то перед типом можно указать `IBOutlet`. Это всего лишь указание для среды Xcode, не имеющее формального значения.

В качестве типа совсем не обязательно указывать тип объекта. Это может быть простой тип, в том числе `BOOL`, `CGFloat` или `CGSize`. Разумеется, в этом случае никакого управления памятью не происходит, поскольку оно и не требуется, и поэтому никакой стратегии управления памятью не должно быть объявлено, но преимущества применения свойства остаются. В частности, методы доступа могут быть синтезированы, а переменная экземпляра объявлена автоматически.

Ниже перечислены значения, которые может принимать атрибут.

Стратегия управления памятью

Имена этих стратегий перечислены в предыдущем разделе. В качестве атрибута указывается конкретная стратегия управления памятью. Если же действует механизм ARC, то обычно указывается стратегия `strong`; она же выбирается по умолчанию, если при действии механизма ARC стратегия управления памятью вообще опускается в объявлении свойства.

`nonatomic`

Если опустить это значение атрибута, в методах доступа будет применяться блокировка для обеспечения правильной работы приложения, при условии, что оно является многопоточным. Установка этого значения атрибута редко вызывает осложнения, тогда как блокировка замедляет работу методов доступа. Поэтому значение атрибута `nonatomic` чаще всего указывается в объявлении свойства. К сожалению, это значение не выбирается по умолчанию, но тут уж ничего не поделаешь. Для того чтобы компилятор выдавал предупреждение, если значение атрибута `nonatomic` случайно опущено, можно также установить режим построения Implicit Atomic Objective-C Properties.

`readwrite` или `readonly`

Если эти значения атрибута опущены, то по умолчанию выбирается значение `readwrite`. Если указано значение атрибута `readonly`, то любая попытка вызвать метод установки или воспользоваться свойством в качестве метода установки приведет к ошибке компилятора. Если должны быть синтезированы методы установки, то ни один из них не будет синтезирован.

`getter=gname, setter=sname:`

По умолчанию из имени свойства выводятся имена get- и set-методов, вызываемых при использовании этого свойства. Так, если свойство называется `myProp`, то по умолчанию для get-метода выбирается имя `myProp`, а для set-метода — имя `setMyProp:`. Для того чтобы изменить это положение, можно использовать любое или оба эти значения атрибутов. Так, если указать `getter=beeblebrox` при объявлении свойства, то get-метод для получения этого свойства будет называться `beeblebrox`. Если методы доступа синтезируются, то get-методу получения будет присвоено такое же самое имя. Это никак не отразится на пользователях данного свойства, но явный вызов метода доступа по несуществующему имени приведет к ошибке компилятора.

Для того чтобы объявить свойство *закрытым*, достаточно разместить его в расширении класса (см. главу 10). Чаще всего расширение класса размещается в самом начале файла

реализации с расширением .m, т.е. до раздела реализации. В итоге имя свойства может использоваться как в самом классе, так и при вызове методов доступа, но не в других классах, как показано в примере 12.10.

Пример 12.10. Объявление закрытого свойства

```
// MyClass.m:
@interface MyClass ()
@property (nonatomic, strong) NSMutableArray* theData; // закрытое свойство
@end

@implementation MyClass
// здесь следует другой код
@end
```

Сведения о закрытых свойствах классов не наследуются их подклассами. В качестве выхода из этого положения можно перенести раздел интерфейса с расширением класса в отдельный заголовочный файл с расширением .h и затем организовать его импорт в файлах реализации как суперкласса, так и подкласса.

Размещать объявление свойства в расширении класса целесообразно еще и потому, что это позволяет *переопределить* свойство. Например, свойство можно объявить как доступное только для чтения (`readonly`) вне класса, но доступное для чтения и записи (`readwrite`) внутри класса. Для того чтобы реализовать такую возможность, достаточно объявить свойство как `readonly` в разделе инструмента заголовочного файла, где оно будет доступно извне, а затем переопределить его без атрибута `readonly` в расширении класса, определяемом в файле реализации, где оно будет доступно только в самом этом классе. Все остальные атрибуты должны совпадать в обоих объявлениях свойства. По умолчанию в отсутствие атрибута `readonly` выбирается атрибут `readwrite`, а следовательно, в этом классе можно вызвать метод установки или указать свойство в качестве левого значения выражения, тогда как в других классах оно будет недоступно.

Синтез методов доступа к свойствам

Для запроса на автоматический синтез методов доступа служит директива `@synthesize`. Она указывается в любом месте раздела реализации класса сколько угодно раз и принимает разделяемый запятыми список имен свойств. Поведение и имена синтезируемых методов доступа будут соответствовать описанным ранее атрибутам в объявлении свойств. В частности, используя синтаксис `имяСвойства=имяПеременнойЭкземпляра` в директиве `@synthesize`, можно объявить, что синтезируемые методы должны иметь доступ к переменной экземпляра, имя которой отличается от имени свойства. В противном случае имя переменной экземпляра будет *таким же*, как и у свойства. Как упоминалось ранее, объявлять переменную экземпляра совсем не обязательно, поскольку она будет объявлена автоматически в процессе синтеза методов доступа.

Переменная экземпляра, автоматически объявляемая в процессе синтеза методов доступа, является строго закрытой. Это означает, что она не наследуется подклассами того класса, где объявляется. Данное обстоятельство редко вызывает осложнения, но если они все же возникают, то переменную экземпляра следует просто объявить явным образом.

Таким образом, объявив свойство theData, можно сделать явный запрос на синтез методов доступа в разделе реализации класса следующим образом:

```
@synthesize theData;
```

В итоге любые методы доступа (theData и setTheData:, если только не изменить их имена в объявлении свойства) будут подспудно синтезированы автоматически. Если переменная экземпляра theData не была объявлена, то это будет также сделано автоматически. Имя автоматически объявляемой переменной экземпляра может, вероятнее всего, понадобиться, поскольку не исключено, что к ней потребуется прямой доступ, особенно в инициализаторе (и в методе dealloc, если механизм ARC не применяется), а также в любых создаваемых самостоятельно методах доступа.

Начиная с версии Xcode 4.2 компания Apple стала придерживаться в шаблонах приложений соглашения о том, что переменная экземпляра получает имя, отличающееся от имени свойства предшествующим знаком подчеркивания. Например, в разделе реализации класса AppDelegate можно обнаружить следующую строку кода:

```
@synthesize window = _window;
```

Очевидная польза от соблюдения этого соглашения об именовании состоит в том, что в прикладном коде можно обращаться к свойству явным образом: self.window. Если случайно обратиться к переменной экземпляра непосредственно по имени window, то будет получена ошибка компилятора, поскольку переменная экземпляра под таким именем отсутствует, но есть переменная, называемая _window. Следовательно, данное соглашение об именовании предотвращает случайный прямой доступ к переменной экземпляра без передачи через методы доступа. Оно также позволяет ясно отличать код, в котором имена обозначают переменные экземпляра, поскольку они начинаются со знака подчеркивания. Кроме того, подобная стратегия освобождает от необходимости использовать имя свойства (в данном случае — window) для обозначения локальной переменной в методе, не получая при этом предупреждение от компилятора о затенении имени переменной экземпляра.

Этой же стратегии именования следует и автоматический синтез. Так, если *опустить* директиву @synthesize, автоматически объявляемая переменная экземпляра получит (также автоматически) имя свойства с *предшествующим* знаком подчеркивания. Например, объявление свойства theData автоматически приводит к объявлению переменной экземпляра _theData. Если же это не устроит вас по какой-нибудь причине, воспользуйтесь непосредственно директивой @synthesize. (Напомним, что если не указать в этом случае имя переменной экземпляра явным образом, то ее выбираемое по умолчанию имя будет *таким же*, как и у свойства, т.е. *без* знака подчеркивания.)

В примере 12.11 наглядно иллюстрируется полная реализация класса Dog с открытым свойством name, как утверждалось ранее в этой главе. Это свойство подлежит автоматическому синтезу, а следовательно, автоматически объявляется и соответствующая переменная экземпляра _name. На нее приходится ссылаться непосредственно в инициализаторе.

Пример 12.11. Полный пример автоматического синтеза свойства

```
// Dog.h:
@interface Dog : NSObject
- (id) initWithName: (NSString*) s;
   @property (nonatomic, copy) NSString* name;
@end

// Dog.m:
@implementation Dog
```

```
- (id) initWithName: (NSString*) s {
   self = [super init];
   if (self) {
     self->_name = [s copy];
   }
   return self;
}

- (id) init {
   NSAssert(NO, @"Making a nameless dog is forbidden.");
   return nil;
}
@end
```

Независимо от того, включаете ли вы директиву `@synthesize` в прикладной код явным образом или выгодно используете в нем автоматический синтез, в любом случае вы не лишаетесь возможности написать один или оба метода доступа самостоятельно. Синтез означает, что любые *не* предоставляемые вами методы доступа будут предоставлены автоматически. Если же вы воспользуетесь автоматическим синтезом (но не директивой `@synthesize`) и предоставите *оба* метода доступа, то *не* получите никакой *автоматически* объявляемой переменной. Это вполне благоразумная стратегия: вы берете на себя полный контроль над методами доступа, а значит, получаете ручной контроль над переменной экземпляра.

В качестве полезного приема можно выгодно воспользоваться синтаксисом `имяСвойства==имяПеременнойЭкземпляра` в директиве `@synthesize`, чтобы переопределить синтезируемый метод доступа, не теряя его функциональные возможности. Допустим, требуется, чтобы метод установки не только выполнял установку значения переменной экземпляра `_myIvar`, но и другие функции. Для этого можно, в частности, написать собственный метод установки, но создавать его сызнова — занятие трудоемкое и чреватое ошибками, тогда как синтезированный метод установки выполняет свои функции исправно и не требует написания ни единой строки кода.

В качестве выхода из этого положения можно объявить свойство `myIvar` наряду с соответствующим закрытым свойством (см. пример 12.10), назвав последнее, скажем, `myIvarAlias`, а затем синтезировать свойство `myIvarAlias` для доступа к переменной экземпляра `_myIvar`. Далее нужно написать вручную методы доступа к свойству `myIvar`, которые должны как минимум воспользоваться свойством `myIvarAlias` соответственно для установки и получения значения переменной `_myIvar`. Однако самое главное, что эти методы можно наделить и *другими* функциями (см. пример 12.12), которые будет выполнять тот метод, который получает или устанавливает значение свойства `myIvar`.

Пример 12.12. Переопределение синтезированных методов доступа

```
// в заголовочном файле:
@interface MyClass : NSObject
@property (nonatomic, strong) NSNumber* myIvar;
@end

// в файле реализации:
@interface MyClass ()
@property (nonatomic, strong) NSNumber* myIvarAlias;
@end

@implementation MyClass
@synthesize myIvarAlias=_myIvar;
- (void) setMyIvar: (NSNumber*) num {
```

```
   // выполнить здесь другие операции
   self.myIvarAlias = num;
}

- (NSNumber*) myIvar {
   // выполнить здесь другие операции
   return self.myIvarAlias;
}
@end
```

Динамические методы доступа

Вместо написания собственных методов доступа, включения в прикладной код директивы `@synthesize` или применения автоматического синтеза можно снабдить объявление свойства директивой `@dynamic` (в разделе реализации). Эта директива предписывает компилятору не предоставлять методы доступа автоматически, даже если он не обнаружит реализации ни одного из методов доступа к данному свойству. В любом случае компилятор должен разрешить объявление свойства на том основании, что во время выполнения, когда дело дойдет до вызова одного из методов доступа, этот вызов будет так или иначе обработан в прикладном коде, хотя компилятору может быть и невдомек, как это будет сделано. По существу, упомянутой выше директивой подавляется система предупреждения компилятора. Она просто отступает в сторону, предоставляя вам возможность действовать на свой страх и риск во время выполнения.

Это редкое, но не такое уж и неслыханное явление. Оно возникает, главным образом, в следующих двух контекстах: при определении собственного свойства анимационного представления, а также при использовании управляемых свойств в базе данных Core Data. В обоих случаях вы берете на себя бразды правления средой Cocoa, чтобы каким-нибудь чудесным образом обработать вызовы методов доступа. Несмотря на то что вам точно неизвестно, *каким* образом такая обработка выполняется в среде Cocoa, вас это не должно особенно беспокоить.

Если эту чудесную обработку требуется выполнить самостоятельно, то интересно, прежде всего, знать, как в подобных случаях это делается в самой средой Cocoa? Ответ на этот вопрос кроется в возможностях языка Objective-C организовывать динамический обмен сообщениями. Хотя это расширенные возможности, тем не менее они настолько привлекательны, что так и напрашиваются на демонстрацию на конкретном примере.

Итак, предлагается написать класс, в котором объявляются свойства `name` (типа `NSString`) и `number` (типа `NSNumber`), но отсутствуют методы доступа к этим свойствам и не предполагается их синтез. Вместо этого свойства `name` и `number` объявляются в разделе реализации данного класса как динамические. Поскольку мы отказываемся от любой помощи, которую может предоставить автоматический синтез, то нам придется самостоятельно объявить переменные экземпляра, как показано ниже.

```
// раздел интерфейса, в котором объявляются name и number
@implementation MyClass {
   NSString* _name;
   NSNumber* _number;
}

@dynamic name, number;
// ...ввести здесь код чудесной обработки...
@end
```

Теперь перейдем непосредственно к изобретению способов чудесной обработки. В данном примере для этой цели используется малоизвестный метод `resolveInstanceMethod:` из класса `NSObject`.

Отправка сообщения объекту отличается от вызова того же самого метода для этого объекта. Если этот метод отсутствует у данного объекта, то предпринимаются дополнительные шаги, чтобы разрешить такой метод. Один из этих первых шагов заключается в следующем: когда к объекту *в первый раз* поступает заданное сообщение, у которого нет соответствующего метода в классе этого объекта, среда выполнения пытается найти реализацию метода `resolveInstanceMethod:` в этом классе. Если она найдет такой метод, то вызовет его, передав ему селектор данного сообщения. Метод `resolveInstanceMethod:` возвратит логическое значение типа `BOOL`. Если это логическое значение `YES`, то оно означает, что все в порядке и можно вызывать данный метод.

Как метод `resolveInstanceMethod:` вообще мог возвратить такой ответ? Что он мог сделать такого, чтобы стал возможным вызов несуществующего метода? Он мог просто создать этот метод. Динамические возможности языка Objective-C позволяют это сделать. Не следует забывать, что метод `resolveInstanceMethod:` вызывается лишь один раз для каждого разрешаемого метода. Следовательно, если он создал такой метод и возвратил логическое значение `YES`, то проблему существования этого метода можно в дальнейшем считать разрешенной раз и навсегда.

Для того чтобы создать метод в реальном времени, т.е. динамически, следует вызвать функцию `class_addMethod()`. Для этого придется организовать импорт файла, введя `<objc/runtime.h>` или соответствующую директиву `@import` языка Objective-C, если применяются модули. Данная функция принимает следующие параметры.

- Класс, в который предполагается ввести метод.
- Селектор для вводимого метода (по существу, это имя данного метода).
- Указатель на метод экземпляра (IMP) для данного метода. Что такое IMP? Это *функция, возвращающая данный метод*. За каждым методом Objective-C стоит функция C. Эта функция принимает те же самые параметры, что и метод Objective-C, а кроме них — еще два параметра, которые указываются первыми. Это объект, действующий как `self` в данной функции, а также селектор для метода, который данная функция возвращает.
- Символьная строка C, описывающая в специальном коде тип значения, возвращаемого функцией. Этот тип является также типом значения, возвращаемого методом. Под *типом* здесь подразумевается нечто большее, чем тип C. В частности, тип каждого объекта считается одним и тем же.

В рассматриваемом здесь примере необходимо охватить два динамических свойства (`name` и `number`) и два метода доступа к ним (`setName:` и `setNumber:`). Итак, чтобы вызвать функцию `class_addMethod()` в методе `resolveInstanceMethod:`, нужно также написать функции C, которые будут действовать как IMP для каждого из упомянутых методов доступа. Для этого можно было бы написать четыре функции C, но это было бы бесполезно! Если сделать это, то зачем вообще утруждать себя применением динамического метода доступа? Вместо этого предлагается написать только две функции C: одну — для обращения к *любому* get-методу, который, возможно, потребуется направить ей, а другую — для обращения к *любому* set-методу, который также может быть направлен ей.

Рассмотрим сначала get-метод, поскольку он намного проще. Что же вообще должен делать обобщенный get-метод? Он должен получать доступ к соответствующей переменной

экземпляра. Как должна называться такая переменная? Получается так, что эта переменная должна носить имя метода с предшествующим знаком подчеркивания. Поэтому сначала нужно извлечь имя метода, что вполне возможно, поскольку оно передано данной функции в качестве ее второго параметра, т.е. селектора, а затем присоединить его к знаку подчеркивания и возвратить в качестве значения той переменной экземпляра, имя которой выводится. Ради простоты примера значение данной переменной экземпляра получается с помощью механизма доступа к значениям по ключам, как показано ниже. Наличие знака подчеркивания означает, что это не приведет к зацикливанию, т.е. функция не завершится вызовом самой себя в бесконечной рекурсии.

```
id callValueForKey(id self, SEL _cmd) {
   NSString* key = NSStringFromSelector(_cmd);
   key = [@"_" stringByAppendingString:key];
   return [self valueForKey:key];
}
```

Теперь, когда имеется get-метод, становится понятнее, как написать set-метод. В частности, чтобы получить имя переменной экземпляра, придется выполнить несколько более сложное манипулирование именем селектора, а именно: отбросить слово `set` в начале этого имени и двоеточие в его конце, сделать строчной первую букву и затем присоединить полученное имя к знаку подчеркивания, как и прежде. Ниже показано, как все это реализуется непосредственно в коде.

```
void callSetValueForKey(id self, SEL _cmd, id value) {
   NSString* key = NSStringFromSelector(_cmd);
   key = [key substringWithRange:NSMakeRange(3, [key length]-4)];
   NSString* firstCharLower =
      [[key substringWithRange:NSMakeRange(0,1)] lowercaseString];
   key = [key stringByReplacingCharactersInRange:NSMakeRange(0,1)
         withString:firstCharLower];
   key = [@"_" stringByAppendingString:key];
   [self setValue:value forKey:key];
}
```

В заключение, можно приступить к написанию метода `resolveInstanceMethod:`. В данном примере этот метод выполняет роль контролера, проверяющего, является ли метод, который должен быть вызван, методом доступа к одному из динамических свойств, как показано ниже. (Если вам не понятно, что собой представляет закодированная символьная строка, указанная в приведенном ниже коде в качестве четвертого аргумента функции `class_addMethod()` языка C, обращайтесь за справкой к документации, где поясняется ее назначение.)

```
+ (BOOL) resolveInstanceMethod: (SEL) sel {
   // этот метод будет вызван
   if (sel == @selector(setName:) || sel == @selector(setNumber:)) {
       class_addMethod([self class], sel, (IMP) callSetValueForKey, "v@:@");
       return YES;
   }
   if (sel == @selector(name) || sel == @selector(number)) {
       class_addMethod([self class], sel, (IMP) callValueForKey, "@@:");
       return YES;
   }
   return [super resolveInstanceMethod:sel];
}
```

Рассмотренная выше реализация динамических методов доступа довольно проста. В частности, ради простоты в ней применяется механизм доступа к значениям по ключам. Здесь также пришлось отказаться от копирования семантики в методе установки из класса `NSString`. Это довольно распространенный прием, который все же позволяет раскрыть внутренний механизм языка Objective-C.

ГЛАВА 13

Связь между объектами

По мере роста числа объектов в приложении могут возникнуть вопросы, касающиеся отправки сообщений и обмена данными между объектами. Эти вопросы, по существу, имеют отношение к архитектуре приложения. Построение прикладного кода может потребовать некоторого планирования, чтобы свести вместе все составные части приложения и обеспечить обмен информацией между ними в нужный момент. В этой главе рассматриваются некоторые организационные вопросы, помогающие установить связь между объектами.

Задача связи сводится к способности одного объекта *видеть* другой. Например, объект `Manny` должен быть в состоянии неоднократно и надежно находить объект `Jack` в течение длительного периода времени, чтобы иметь возможность посылать ему сообщения. Одно из очевидных решений этой задачи состоит в том, чтобы объявить переменную экземпляра объекта `Manny` и присвоить ей в качестве значения объект `Jack`. Такое решение оказывается пригодным особенно в тех случаях, когда объекты `Manny` и `Jack` разделяют некоторые общие обязанности или снабжают друг друга определенными функциональными возможностями. Объект приложения и его делегат, табличное представление и его источник данных, контроллер и представление, которым он управляет, — все это примеры, в которых один объект должен иметь переменную своего экземпляра, указывающую на другой объект.

Это совсем не означает, что объект `Manny` должен утверждать право на владение объектом `Jack`, как того требует стратегия управления памятью (см. главу 12). Как правило, объект не сохраняет свой делегат или источник данных. Аналогично объект, реализующий шаблон "цель-действие", например, объект класса `UIControl`, не сохраняет свою цель. Однако тогда объект `Manny` должен быть готовым к тому, что предполагаемая им ссылка на объект `Jack` окажется пустой, а следовательно, необходимо принять меры к тому, чтобы она не превратилась в висячий указатель (правда, это маловероятно, если действует механизм ARC). С другой стороны, контроллер представления оказывается бесполезным в отсутствие представления, которым он должен управлять. Получив такое представление, он сохранит его, освободив лишь тогда, как сам прекратит свое существование. Возможны и другие подобные ситуации, в которых требуется, чтобы один объект владел другим объектом.

Объекты могут также устанавливать двухстороннюю связь, *не* сохраняя постоянные ссылки друг на друга в переменных экземпляра. Например, объект `Manny` может послать объекту `Jack` сообщение, где одним из параметров является ссылка на объект `Manny`. Это может быть лишь форма идентификации или приглашения объекта `Jack` послать сообщение обратно объекту `Manny`, если объекту `Jack` требуются дополнительные сведения на время выполнения полученного им метода. Опять же это общий шаблон. Параметром делегата сообщения

textFieldShouldBeginEditing: служит ссылка на объект типа UITextField, пославший это сообщение. Первым параметром сообщения "цель-действие" является ссылка на отправителя этого сообщения. Таким образом, объект Manny делает себя сразу же видимым для объекта Jack, которому совсем не обязательно сохранять объект Manny, чтобы избежать возникновения цикла сохранения.

Как же объекту Manny получить сначала ссылку на объект Jack? Это непростой вопрос. Искусство программирования для операционной системы iOS в частности и объектно-ориентированного программирования вообще состоит главным образом в том, чтобы организовать *получение* одним объектом *ссылки* на другой объект. (Этот вопрос обсуждался также в главе 5.) В каждом отдельном случае решать данную задачу приходится иначе. Тем не менее появился ряд общих шаблонов, упрощающих ее решение и вкратце рассматриваемых в этой главе.

Имеются также способы, позволяющие объекту Manny послать сообщение, которое объект Jack *получит*, даже если оно не адресуется этому объекту непосредственно, или ничего неизвестно, или вообще безразлично, что собой представляет объект Jack. Примерами тому служат уведомления и механизм наблюдения за значениями по ключам, которые также обсуждаются в этой главе. В завершение главы рассматривается следующий важный вопрос: какие виды объектов должны видеть друг друга в общей области действия типичной прикладной программы для операционной системы iOS.

Видимость, достигаемая получением экземпляра

Каждый экземпляр получается откуда-то и по чьему-то требованию. В частности, один объект посылает сообщение другому объекту с требованием получить, прежде всего, такой экземпляр. Следовательно, у запрашивающего объекта в данный момент имеется ссылка на требуемый экземпляр. Когда объект Manny формирует объект Jack, у него имеется ссылка на объект Jack.

Этот простой факт может послужить в качестве отправной точки для установления будущей связи между объектами. Если объект Manny формирует объект Jack и ему заранее известно, что в дальнейшем ему потребуется ссылка на объект Jack, то объект Manny может сохранить ссылку, полученную, прежде всего, при формировании объекта Jack. С другой стороны, объекту Manny может быть известно, что объекту Jack в дальнейшем потребуется ссылка на объект Manny. В таком случае объект Manny может предоставить эту ссылку сразу же после формирования объекта Jack, а затем объект Jack сохранит ее.

Характерным тому примером служит делегирование. Объект Manny может создать объект Jack и сразу же сделать себя его делегатом. На самом деле объект Jack можно наделить, если это очень важно, инициализатором, чтобы объект Manny мог создать объект Jack и *одновременно* передать ему ссылку на себя во избежание любых оплошностей. Сравните это с подходом, принятым в классах UIActionSheet и UIAlertView, где делегат является одним из параметров инициализатора, или же в классе UIBarButtonItem, где цель является одним из параметров инициализатора.

Делегирование — это лишь один пример. Когда объект Manny создает объект Jack, последнему может понадобиться ссылка не на объект Manny, а на нечто известное или имеющееся у этого объекта. По-видимому, объект Jack можно наделить методом, чтобы объект Manny мог передать информацию, требующуюся объекту Jack. Опять же этот метод, возможно, стоит сделать инициализатором объекта Jack, если подобная информация жизненно важна для данного объекта.

Приведенный ниже пример кода взят из контроллера табличного представления одного из моих приложений. Когда пользователь выбирает строку таблицы постукиванием

пальцем по ней, создается контроллер вторичного табличного представления (экземпляр класса TrackViewController), которому передаются требующиеся ему данные, а затем отображается табличное представление. Класс TrackViewController был намеренно разработан вместе со специальным инициализатором initWithMediaItemCollection:, чтобы сделать его буквально обязанным получить доступ к данным, требующимся контроллеру типа TrackViewController с момента начала его существования.

```
- (void)showItemsForRow: (NSIndexPath*) indexPath {
    // создать подтаблицу дорожек звукозаписи и перейти к ней
    TrackViewController *t =
       [[TrackViewController alloc] initWithMediaItemCollection:
          (self.albums)[indexPath.row]];
    [self.navigationController pushViewController:t animated:YES];
}
```

В данном примере объект self не хранит ссылку на новый экземпляр класса TrackViewController, а объект типа TrackViewController не приобретает ссылку на объект self. Тем не менее объект self создает экземпляр класса TrackViewController, и поэтому на какое-то мгновение получает ссылку на него. Следовательно, объект self выгодно использует данный момент, чтобы передать экземпляру класса TrackViewController нужную ему информацию, поскольку трудно найти более подходящий для этого момент. Искусство передачи данных отчасти состоит в том, чтобы знать, когда наступит подходящий момент, и не пропустить его.

При загрузке nib-файла его владелец, по существу, получает экземпляры объектов в nib-файле. (Владельцем файла с расширением .xib является объект, представленный заместителем владельца файла в nib-файле, а владельцем раскадровки — контроллер представления верхнего уровня; см. главу 7). Подготавливая в nib-файле выходы от владельца этого файла, вы тем самым устраиваете получение владельцем nib-файла ссылок на экземпляры nib-объектов в тот момент, когда они получаются посредством механизма загрузки nib-файлов.

Несколько другая архитектура возникает, когда объект Manny получает экземпляр объекта Jack, а объект Moe его не получает. В то же время у объекта Moe есть ссылка на объект Manny, и объекту Moe известно, что у объекта Manny есть информация, которой он может поделиться с объектом Jack. Следовательно, объект Moe связывает вместе объекты Manny и Jack. Это подходящий момент для объекта Manny.

Например, начинающих программировать в операционной системе iOS нередко озадачивает, что объекты могут получить ссылки друг на друга, если их экземпляры будут получены из разных nib-файлов, например, разных файлов с расширением .xib или сцен в раскадровке. Жаль, что нельзя установить связь между объектом в nib-файле A и объектом в nib-файле B. Особенно жаль, когда оба объекта оказываются в одной раскадровке. Как пояснялось во врезке "Связи между nib-файлами" в главе 7, такая связь может быть бесполезной, именно поэтому она и невозможна. Тем не менее один объект (Manny) может стать владельцем загружаемого nib-файла A, а другой объект (Jack) — владельцем загружаемого nib-файла B. Тогда им, возможно, удастся увидеть друг друга. В этом случае затруднение разрешается при наличии всех необходимых выходов. Возможно, какому-нибудь третьему объекту (Moe) удастся увидеть оба эти объекта и предоставить им путь для связи.

Именно это и происходит в раскадровке. Когда в раскадровке начинается переход, получается экземпляр контроллера целевого представления этого перехода, а сам переход получает ссылку на него. В то же время контроллер исходного представления уже существует, а у перехода имеется также ссылка на него. Таким образом, переход посылает контроллеру исходного представления сообщение prepareForSegue:sender:, содержащее ссылку на самого себя (т.е. на этот переход).

В данном случае переход выступает в роли объекта Moe, связывая вместе объекты Manny (контроллер исходного представления) и Jack (контроллер целевого представления). При этом у контроллера исходного представления появляется удобный момент (как и у объекта Manny) получить ссылку на новый экземпляр контроллера целевого представления, стать делегатом этого контроллера, передать ему любую требующуюся информацию и т.д. (Если создать проект по шаблону Utility Application, то можно обнаружить, что именно это и происходит в исходном коде класса MainViewController.)

Видимость, достигаемая отношением

Возможно, объектам потребуется видеть друг друга автоматически благодаря их взаимному расположению в содержащей их структуре. Прежде чем решать вопрос предоставления одного объекта по ссылке на другой объект, следует выяснить, имеется ли *уже* цепочка ссылок, ведущая от одного из них к другому.

Например, дочернее представление может видеть родительское представление через свое свойство superview. Родительское представление может видеть все свои дочерние представления через свое свойство subviews и выбрать конкретное дочернее представление через свойство tag этого дочернего представления, вызвав метод viewWithTag:. Дочернее представление в окне может видеть это окно через свое свойство window. Следовательно, перемещаясь вверх и вниз по иерархии представлений с помощью свойств, можно получить требуемую ссылку.

Ниже приведен пример кода, взятый из одного из моих приложений, где многие кнопки и текстовые поля связаны в пары. Так, если кнопка выбирается постукиванием пальцем по ней, то в соответствующем текстовом поле нужно предоставить текст. Как же получить ссылку на текстовое поле, связанное с выбранной кнопкой? Для этого в nib-файле назначаются дескрипторы кнопок (101, 102, 103 и т.д.), а также дескрипторы соответствующих им текстовых полей (1, 2, 3 и т.д.).

```
UIView* v = sender; // кнопка
UIView* v2 = [v.superview viewWithTag:(v.tag - 100)];
```

Аналогично один реагирующий элемент (см. главу 11) может видеть следующий реагирующий элемент в цепочке посредством метода nextResponder. Это также означает, что благодаря структуре цепочки реагирующих элементов основное преставление контроллера может видеть контроллер преставления. В приведенном ниже фрагменте кода перемещением вверх по иерархии получается ссылка на контроллер преставления, отвечающий за отображение всей сцены в целом.

```
UIResponder* r = self; // представление типа UIView в интерфейсе
while (![r isKindOfClass:[UIViewController class]])
   r = r.nextResponder;
```

Аналогично контроллеры представлений сами являются частью иерархии вложенности и поэтому могут видеть друг друга. Если один контроллер представления предоставляет представление через второй контроллер представления, то последний является для первого контроллером типа presentedViewController, а первый для последнего — контроллером типа presentingViewController. Если же контроллер представления вложен в контроллер типа UINavigationController, то последний является для первого контроллером типа navigationController. Управление видимым представлением контроллера типа UINavigationController осуществляется его контроллером типа visibleViewController. Кроме того, из любого из этих контроллеров представлений можно достигнуть нужного представления через его свойство view.

Все эти отношения являются открытыми. Так, если можно получить ссылку только на один объект в любой из этих структур или аналогичной структуре, то, по существу, появляется возможность перемещаться по всей структуре, следуя по цепочке отношений и добираясь в ней до любого другого объекта.

Глобальная видимость

Некоторые объекты оказываются глобально видимыми, т.е. они доступны для всех остальных объектов. Характерным тому примером служат объекты классов. Классы нередко содержат методы, поставляющие экземпляры одиночных объектов. Некоторые из этих одиночных объектов, в свою очередь, обладают свойствами, делающими другие объекты также глобально видимыми.

Например, любой объект может видеть экземпляр одиночного объекта типа `UIApplication`, делая вызов `[UIApplication sharedApplication]`. Поэтому любой объект может также видеть основное окно приложения, поскольку это свойство `keyWindow` экземпляра одиночного объекта типа `UIApplication`. Кроме того, любой объект может видеть делегат приложения, поскольку это его свойство `delegate`. Эта цепочка продолжается далее: любой объект может видеть контроллер корневого представления приложения, поскольку это контроллер типа `rootViewController` основного окна, а оттуда можно перейти к иерархии контроллеров представлений и иерархии самих представлений, как пояснялось в предыдущем разделе.

Вы можете сделать свои объекты глобально видимыми, присоединив их к глобально видимому объекту. Открытое свойство делегата приложения, которое вы вольны создать, становится глобально видимым благодаря глобальной видимости делегата приложения, которое само оказывается общедоступным.

Глобально видимым является также общедоступный объект пользовательских настроек по умолчанию, получаемый в результате вызова `[NSUserDefaults standardUserDefaults]`. Этот объект служит шлюзом для хранения и извлечения пользовательских настроек по умолчанию, аналогичных словарю (коллекции значений, именуемых по ключам). Пользовательские настройки по умолчанию автоматически сохраняются при выходе из приложения и автоматически доступны, когда приложение запускается на выполнение снова. Таким образом, эти настройки служат одним из способов сохранения информации между последовательными запусками приложения. Поскольку они глобально видимы, то служат также для передачи значений в самом приложении.

Например, в одном из моих приложений имеется настройка, называемая `@"hazyStripy"`. Она определяет, следует ли отображать некоторый видимый интерфейсный объект с дымчатым или полосатый заполнением. Эту настройку пользователь может изменить, и для этой цели предоставляется интерфейс глобальных параметров настройки. Когда пользователь отображает этот интерфейс, то среди пользовательских настроек по умолчанию анализируется настройка `@"hazyStripy"`, чтобы настроить интерфейс в соответствии с ее состоянием. Если пользователь взаимодействует с интерфейсом глобальных параметров настройки с целью изменить настройку `@"hazyStripy"`, то приложение реагирует на его действия, внося соответствующие коррективы в эту настройку среди пользовательских настроек по умолчанию.

Интерфейс глобальных параметров настройки — не единственный объект, в котором используется настройка `@"hazyStripy"` среди пользовательских настроек по умолчанию. Она используется также в прикладном коде, где фактически рисуется объект с дымчатым или полосатым заполнением, и поэтому требуется знать, как рисовать этот объект. Следовательно, отпадает всякая необходимость в связи между объектом, который рисует объект с дымчатым

или полосатым заполнением, а также объектом, управляющим интерфейсом глобальных параметров настройки, чтобы они видели друг друга! Ведь оба эти объекта могут видеть общий объект, т.е. пользовательскую настройку по умолчанию `@"hazyStripy"` (рис. 13.1). Безусловно, пользовательскими настройками по умолчанию нередко злоупотребляют для хранения информации, которая используется не для поддержки пользовательских глобальных параметров настройки, а только как удобное место для размещения данных, глобально видимых для всех объектов.

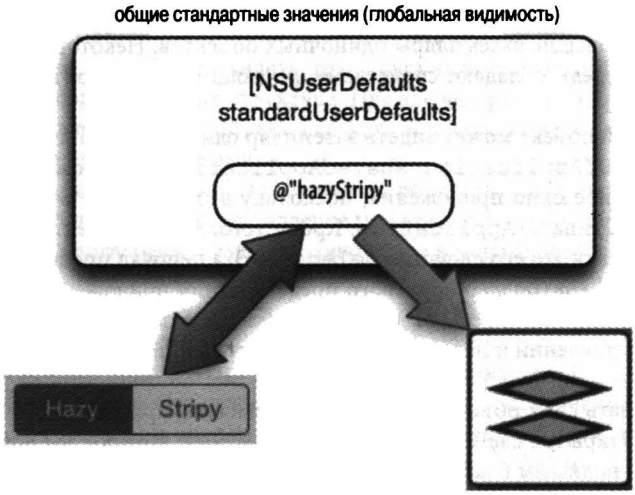


Рис. 13.1. Глобальная видимость пользовательских настроек по умолчанию

Уведомления

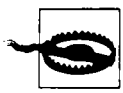
Уведомления (см. главу 11) могут, в частности, служить для связи между объектами, которые принципиально отдалены и не обязательно должны видеть друг друга. Для связи им требуется лишь одно: знать имя уведомления. Каждый объект может видеть центр уведомлений, и поэтому каждый объект можно организовать для отправки или получения уведомлений. Такое применение уведомлений может показаться пассивным уклонением от ответственности за благоразумное построение объектов. Иногда одному объекту совершенно не обязательно и даже не нужно знать, каким именно объектам он посылает сообщение.

Характерный тому пример был приведен в главе 11. Рассмотрим еще один пример. В одном из моих приложений его делегат может обнаружить потребность перестроить интерфейс сызнова. Если это должно произойти без утечек памяти (и всего связанного с этим опустошения), то каждый контроллер представления, который в настоящий момент управляет периодическим таймером типа `NSTimer`, должен сделать этот таймер недействительным (см. главу 12).

Вместо того чтобы выяснять, какие именно контроллеры представлений могут быть использованы для этой цели, и наделять каждый из них вызываемым методом, делегат моего приложения просто выдает всем контроллерам представлений команду остановить таймеры, посылая им соответствующее уведомление. Когда все контроллеры представлений, управляющие таймерами и зарегистрированные для получения такого уведомления, получат его, то они будут знать, что им делать дальше.

Наблюдение за значениями по ключам

Наблюдение за значениями по ключам (или сокращенно KVO) — это механизм, позволяющий одному объекту быть зарегистрированным другим объектом, чтобы автоматически уведомлять об изменении значения во втором объекте. Для того чтобы выполнить действие регистрации, требуется, чтобы в какой-то момент стал видимым, прежде всего, второй объект или чтобы оба эти объекта были видимы третьему объекту. Впоследствии второй объект может оказаться в состоянии посылать сообщения первому объекту, не прибегая к переменной экземпляра для ссылки на первый объект и даже ничего *не* зная о классе первого объекта или открыто объявленных методах. В архитектурном отношении этот механизм похож на пары "цель-действие" (см. главу 11), но он подходит для любых двух объектов. (Механизм KVO предоставляется через неформальный протокол `NSKeyValueObserving`, который фактически представляет собой ряд категорий в классе `NSObject` и других классах.) Второй объект совсем не обязательно должен быть экземпляром вашего собственного класса. Это может быть экземпляр встроенного класса Cocoa. Первый объект должен содержать ваш собственный код, чтобы вовремя реагировать на уведомления.



Предупреждение для программистов в системе OS X

Характерные для системы OS X привязки отсутствуют в системе iOS, но иногда для достижения аналогичных целей можно воспользоваться механизмом KVO.

Механизм KVO можно разделить на три категории.

Регистрация

Для того чтобы отреагировать на изменение значения, принадлежащего объекту A, объект B должен быть зарегистрирован объектом A.

Изменение

Изменение значения, принадлежащего объекту A, должно происходить особым образом, совместимым с механизмом KVO. Как правило, это означает применение метода доступа, совместимого с механизмом доступа к значениям по ключам, для внесения изменений. (Если объект A относится к вашему собственному классу, вы можете дополнительно написать код, который будет вручную вызывать другие виды изменений, которые считаются совместимыми с механизмом KVO.)

Уведомление

Объект B автоматически уведомляется, что значение в объекте A изменилось и что он может теперь отреагировать на него так, как посчитает нужным.

Ниже приведен довольно простой пример, но он в достаточной степени демонстрирует действие механизма KVO. В этом примере имеется класс `MyClass1`, которому принадлежит объект `objectA`, а также класс `MyClass2`, которому принадлежит объект `objectB`. Получив ссылки на эти объекты, мы регистрируем объект `objectB` для реагирования на изменения в переменной экземпляра `value` объекта `objectA`, а затем изменяем значение в этой переменной. Безусловно, объект `objectB` автоматически уведомляется обо всех изменениях.

```
// MyClass1.h:
@interface MyClass1 : NSObject
```

```
@property (nonatomic, copy) NSString* value;
@end

// MyClass2.m:
- (void) observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context {
   NSLog(@"I heard about the change!");
}

// где-то еще объекты определяются полностью:
MyClass1* objectA = [MyClass1 new];
MyClass2* objectB = [MyClass2 new];
// зарегистрировать для механизма KVO
[objectA addObserver:objectB forKeyPath:@"value" options:0 context:nil]; ❶
// change the value in a KVO compliant way
objectA.value = @"Hello, world!"; ❷
// в результате вызывается метод observeValueForKeyPath:...
// для объекта objectB ❸
```

❶ Для регистрации объекта `objectB` с целью реагировать на изменения в значении объекта `objectA` вызывается метод `addObserver:forKeyPath:`. Его параметры `options:` и `context:`, рассматриваемые далее, не используются. (В частности, параметр `context:` служит для передачи значения, которое должно возвратиться как часть уведомления; см. главу 12.)

На практике объект B должен был бы, скорее всего, *зарегистрироваться*, чтобы реагировать на изменения в пути к ключу объекта A. В данном примере это организовано искусственно для максимального упрощения кода.

❷ Значение объекта `objectA` изменяется совместимым с механизмом KVO способом, а именно: передачей этого значения через метод установки, поскольку установка значения свойства равнозначна его передаче через метод установки. Это еще одна причина для применения методов доступа (и свойств), так как они помогают обеспечить совместимость с механизмом KVO при изменении значения.

❸ При изменении значения объекта `objectA` автоматически наступает третья стадия, когда вызывается метод `observeValueForKeyPath:....` для объекта `objectB`. Этот метод реализован в классе `MyClass2`, чтобы получить уведомление. В данном простом примере предполагается получить только одно уведомление, и поэтому далее происходит регистрация того, что оно действительно получено. На практике один объект может быть зарегистрирован для получения более чем одного уведомления в соответствии с механизмом KVO. Для различения разных уведомлений и принятия соответствующих решений используются входящие параметры.

При вызове метода `observeValueForKeyPath:...` требуется хотя бы знать, что собой представляет новое значение. Это нетрудно выяснить, поскольку объекту, в котором произошли изменения, была передана ссылка наряду с путем к ключу для доступа к значению в этом объекте. Следовательно, с помощью механизма KVC можно сделать запрос объекта, в котором произошли изменения, в самом общем виде, как показано ниже.

```
- (void) observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
```

```
                        context:(void *)context {
    id newValue = [object valueForKeyPath:keyPath];
    NSLog(@"The key path %@ changed to %@", keyPath, newValue);
}
```

Можно также сделать запрос, чтобы включить новое значение в уведомление. Это зависит от аргумента `options:`, передаваемого при первоначальной регистрации. В приведенном ниже фрагменте кода делается запрос на включение как старого, так и нового значения в уведомление.

```
objectA.value = @"Hello";
[objectA addObserver:objectB forKeyPath:@"value"
    options: NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
    context: nil];
objectA.value = @"Goodbye"; // инициируется уведомление
```

При получении уведомления старое и новое значения извлекаются из словаря `change` следующим образом:

```
- (void) observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context {
    id newValue = change[NSKeyValueChangeNewKey];
    id oldValue = change[NSKeyValueChangeOldKey];
    NSLog(@"The key path %@ changed from %@ to %@",
    keyPath, oldValue, newValue);
}
```

В процессе регистрации никакого управления памятью не происходит, поэтому именно на вас возлагается обязанность снять объект B с регистрации, прежде чем он будет уничтожен. В противном случае объект A может впоследствии попытаться отправить уведомление по висячему указателю (см. главу 12). Для того чтобы снять объект A с регистрации, ему посылается сообщение `removeObserver:forKeyPath:`. Наблюдатель должен быть снят с регистрации явным образом по каждому пути к ключу, по которому он зарегистрирован. В качестве второго аргумента посылаемого метода нельзя использовать пустое значение (`nil`), означающее все пути к ключам. На практике там, где объект B, вероятно, зарегистрировался с помощью объекта A, он должен быть снят с регистрации с помощью того же самого объекта A. Это, возможно, должно быть сделано в его реализации метода `dealloc`. Следует, однако, иметь в виду, что для этого у объекта B должна быть ссылка на объект A.

Кроме того, объект B должен сняться с регистрации на получение уведомлений от объекта A, если последний собирается прекратить свое существование! Данное требование кажется обременительным, а его причины не совсем ясны. По-видимому, это требование связано с управлением памятью, которое происходит в механизме KVO и обеспечивает его работоспособность. Правда, среда выполнения аккуратно предупредит в журнале регистрации, если объект, наблюдаемый с помощью механизма KVO, собирается прекратить свое существование. (Однако вы не получите такое предупреждение, если объект *наблюдателя* прекращает свое существование. Об этом вы узнаете лишь после того, как сообщение, отправленное висячему указателю, приведет к аварийному завершению вашего приложения.)

Начинающим программировать в операционной системе iOS нередко оказывается непонятно, как пользоваться механизмом KVO для наблюдения за изменениями, происходящими в изменяемом массиве, чтобы вовремя получить уведомление о вводе, удалении или замене объекта в этом массиве. Ввести наблюдатель в массив нельзя, и поэтому наблюдение за изменениями в нем можно организовать через объект, у которого имеется путь к ключу для данного

массива, используя, например, методы доступа. В таком случае простейшее решение состоит в том, чтобы получить доступ к массиву с помощью метода `mutableArrayValueForKey:`, который предоставляет наблюдаемый объект-заместитель. В приведенном ниже примере кода устанавливается объект со свойством `theData` в виде массива словарей (см. главу 12).

```
(
   {
      description = "The one with glasses.";
      name = Manny;
   },
   {
      description = "Looks a little like Governor Dewey.";
      name = Moe;
   },
   {
      description = "The one without a mustache.";
      name = Jack;
   }
)
```

Допустим, что это массив типа `NSMutableArray`. В таком случае рассматриваемый здесь объект можно зарегистрировать для наблюдения за путем к ключу `@"theData"` следующим образом:

```
[objectA addObserver:objectB forKeyPath:@"theData" options:0 context:nil];
```

Теперь объект B будет получать уведомления об изменениях, происходящих в данном изменяемом массиве, но только если эти изменения делаются с помощью метода `mutableArrayValueForKey:` для объекта-заместителя, как показано ниже.

```
[[objectA mutableArrayValueForKeyPath:@"theData"] removeObjectAtIndex:0];
// уведомление инициировано
```

По-видимому, было бы обременительно требовать от клиентов вызывать метод `mutableArrayValueForKey:` самостоятельно. В качестве простого решения объект A должен сам предоставить метод получения, вызывающий метод `mutableArrayValueForKey:`. Ниже приведена одна из возможных реализаций такого решения.

```
// MyClass1.h:
@interface MyClass1 : NSObject
@property (nonatomic, strong, getter=theDataGetter) NSMutableArray* theData;
@end

// MyClass1.m:
- (NSMutableArray*) theDataGetter {
   return [self mutableArrayValueForKey:@"theData"];
}
```

В итоге любому клиенту становится известно, что у данного объекта имеется ключ `@"theData"` и свойство `theData` и что по этому ключу можно зарегистрироваться для наблюдения и затем получить доступ к изменяемому массиву через свойства `theData`, как показано ниже.

```
[objectA addObserver:objectB forKeyPath:@"theData"
   options: NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
   context:nil];
[objectA.theData removeObjectAtIndex:0]; // уведомление инициировано
```

Если вы собираетесь принять именно такой подход, то должны также реализовать (в классе MyClass1) четыре метода, совместимых с механизмом KVC, в качестве фасада изменяемого массива (см. главу 12), хотя можно, по-видимому, обойтись и без них. Несмотря на то что эти методы кажутся тривиальными, поскольку они лишь делегируют переменной экземпляра self->_theData эквивалентные вызовы, тем не менее они будут вызываться поставляемым объектом-заметителем ради повышения его эффективности (а некоторые считают, что и надежности). В отсутствие этих методов объект-заместитель прибегает к непосредственной установке переменной экземпляра, заменяя весь изменяемый массив всякий раз, когда клиент вносит изменения в этот массив:

```
- (NSUInteger) countOfTheData {
  return [self->_theData count];
}

- (id) objectInTheDataAtIndex: (NSUInteger) ix {
   return self->_theData[ix];
}

- (void) insertObject: (id) val inTheDataAtIndex: (NSUInteger) ix {
   [self->_theData insertObject:val atIndex:ix];
}

- (void) removeObjectFromTheDataAtIndex: (NSUInteger) ix {
   [self->_theData removeObjectAtIndex: ix];
}
```

Дело усложняется, если требуется наблюдать за изменениями в отдельном элементе массива. Допустим, имеется изменяемый массив словарей. Для наблюдения за изменениями значения ключа @"description" любого словаря в массиве потребуется зарегистрироваться по данному ключу для *каждого* массива словарей в отдельности. Это можно сделать эффективно с помощью метода экземпляра addObserver:toObjectsAtIndexes:forKeyPath:options:context:, но если *сам* массив является изменяемым, то придется зарегистрироваться по данному ключу для каждого *нового* словаря, последовательно вводимого в массив, а также сниматься с регистрации, когда словарь удаляется из массива, что кажется слишком обременительным.

Свойства встроенных классов, предоставляемых компанией Apple, как правило, совместимы с механизмом KVO. На самом деле это относится ко многим классам, в которых свойства, по существу, не применяются. Например, класс NSUserDefaults совместим с механизмом KVO. К сожалению, компания Apple предупреждает, что рассчитывать на недокументированную совместимость с механизмом KVO не следует.

С другой стороны, в некоторых классах Cocoa явно приветствуется применение механизма KVO. Они трактуют его как основной механизм уведомлений вместо центра уведомлений и класса NSNotification. Например, класс AVPlayer, отвечающий за воспроизведение мультимедийного содержимого, включая фильмы, обладает различными свойствами, в том числе status и rate, которые служат для уведомления о готовности мультимедийных средств к воспроизведению или же о факте воспроизведения ими мультимедийного содержимого. Как утверждается в документации *AV Foundation Programming Guide*, предоставляемой компанией Apple, для контроля над воспроизведением мультимедийного содержимого средствами класса AVPlayer следует применять к его свойствам механизм наблюдения за значениями по ключам.

Механизм наблюдения за значениями по ключам довольно основателен, и поэтому за полным его описанием обращайтесь к документации *Key-Value Observing Guide*, предоставляемой компанией Apple. Ему присущи и некоторые серьезные недостатки. Например, все уведомления, к сожалению, поступают в результате вызова одного и того же метода `observeValueForKeyPath:...`, становящегося из-за этого критическим элементом с точки зрения производительности. Следить за тем, кто за кем наблюдает, обеспечивать надлежащие срок действия наблюдателя и наблюдаемого и снимать наблюдатель с регистрации, прежде чем он прекратит свое существование, будет не так-то просто. Однако, как правило, механизм KVO приносит пользу, когда требуется поддерживать согласованность значений в разных объектах.

Шаблон проектирования "модель–представление–контроллер"

В документации компании Apple и в других источниках можно найти ссылки на термин *шаблон проектирования "модель–представление–контроллер"*, или сокращенно *шаблон MVC*. Этим термином обозначается архитектурная цель для поддержания отличий между тремя функциональными аспектами программы, дающими пользователю возможность просматривать и править информацию. Это означает, что программа, по существу, обладает графическим пользовательским интерфейсом. Упоминание о шаблоне MVC относится еще к временам языка Smalltalk, и на тему его применения в разработке прикладного программного обеспечения написано немало. Неформально термин *шаблон проектирования "модель–представление–контроллер"* обозначает следующее.

Модель

Описывает данные и управление ими и нередко называется также бизнес-логикой прикладной программы, т.е. самой ее сердцевиной.

Представление

Описывает то, что пользователь видит и с чем он может взаимодействовать в прикладной программе.

Контроллер

Служит в качестве посредника между моделью и представлением.

В качестве примера рассмотрим игровое приложение, где текущий счет в игре отображается для пользователя. Для этого обязанности в игровом приложении распределяются следующим образом.

- Класс `UILabel`, отвечающий за отображение для пользователя текущего счета по ходу игры, выполняет функции *представления*. По существу, он лишь воспроизводит информацию по отдельным пикселям и должен знать, как это делается. Обязанность знать, что *именно* следует воспроизводить (в данном случае — счет в игре), возлагается на другую составляющую рассматриваемого здесь шаблона.
- Начинающий программист может попытаться воспользоваться счетом в игре, отображаемым средствами класса `UILabel`, как конкретным числом, чтобы увеличить его, прочитав его как символьную строку типа `UILabel`, преобразовав эту строку в число, увеличив последнее, преобразовав его обратно в символьную строку и представив ее

вместо прежней строки. Однако это было бы грубым нарушением самого принципа, положенного в основу шаблона MVC. Представление, которое предоставляется пользователю, должно *отражать* счет в игре, но не *хранить* этот счет.

- Счет — это данные, поддерживаемые внутренним образом, т.е. в *модели*. Модель может быть простой, как переменная экземпляра с открытым методом увеличения счета в игре, или же сложной, как объект класса `Score` с целым рядом методов специального назначения.
- Счет относится к числовому типу данных, тогда как класс `UILabel` отображает символьную строку. Одного этого достаточно, чтобы показать, что представление и модель имеют естественные отличия.
- В обязанности *контроллера* входит уведомление о моменте изменения счета и управление отображением обновленного счета в пользовательском интерфейсе. Это особенно очевидно, если представить, что числовое значение счета в игре требуется, так или иначе, преобразовать в форму, удобную для представления пользователю.
- Допустим, что класс `UILabel`, представляющий счет, сообщает следующее: "Ваш текущий счет равен 20". Предположительно, обязанность за сохранение и предоставление счета 20 пользователю возлагается на контроллер.

Даже такой простой пример (рис. 13.2) наглядно иллюстрирует преимущества шаблона MVC. Подобное разделение обязанностей позволяет описанным выше аспектам прикладной программы развиваться в значительной степени независимо друг от друга. Так, если для представления счета требуется другой тип и размер шрифта, достаточно изменить представление, а модели и контроллеру знать об этом совсем не обязательно, но они должны работать дальше, как и прежде. Если требуется внести изменения в контроллер, то изменять модель и представление для этого совсем не требуется.

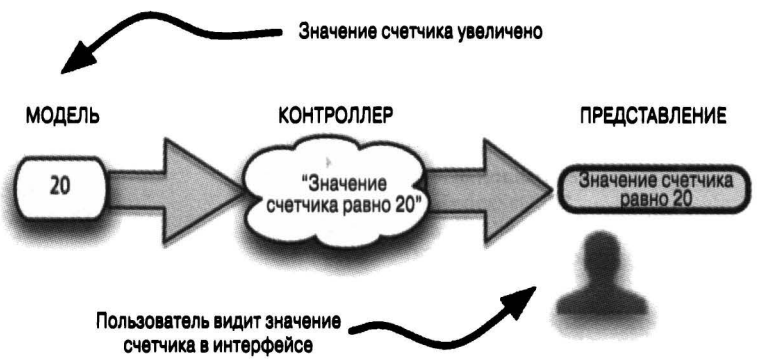


Рис. 13.2. Шаблон проектирования "модель-представление-контроллер"

Приверженность шаблону MVC особенно присуща приложениям Cocoa, поскольку этот шаблон поддерживается в самой среде Cocoa. Это видно из названий классов Cocoa. Например, класс `UIView` реализует представление, а класс `UIViewController` — контроллер, воплощающий логику управления отображением представления. В главе 11 было показано, что класс `UIPickerView` не содержит данные, которые он отображает. Он получает эти данные из источника данных. Следовательно, класс `UIPickerView` обозначает представление, а источник данных — модель.

В документации, предоставляемой компанией Apple, подчеркивается следующее дополнительное различие составляющих шаблона MVC: материал подлинной модели и подлинного представления должен быть в достаточной степени повторно используемым в том отношении, что они могут быть полностью перенесены в другое приложение. Материал контроллера обычно не используется повторно, поскольку в нем учитывается, каким образом *данное* приложение служит посредником между моделью и представлением.

Например, в одном из моих приложений загружается XML-документ из ленты новостей, а заглавия статей предоставляются пользователю в виде таблицы. Сохранение и синтаксический анализ XML-файла являются исключительно материалом модели, и поэтому они повторно используются настолько часто, что я даже не пытался писать эту часть прикладного кода, а воспользовался готовым кодом под названием FeedParser, написанным Кевином Баллардом (Kevin Ballard). Представление таблицы обеспечивается классом `UITableView` и также является повторно используемым как получаемое непосредственно из среды Cocoa. Однако когда класс `UITableView` обращается к коду моего приложения и запрашивает, что именно следует отображать в данной ячейке, или когда код моего приложения обращается к XML-документу и запрашивает заглавие статьи, соответствующее данной строке таблицы, то это уже материал, а точнее, код контроллера.

Шаблон MVC помогает найти ответы на вопросы, касающиеся тех объектов, которые должны быть видны другим объектам в приложении. Так, объект контроллера обычно должен видеть объекты модели и представления, а объекту модели или группе объектов модели обычно не требуется видеть все, что находится на пределами самой модели. Как правило, объекту представления также не требуется видеть все, что находится на пределами самого представления, но такие структурные средства, как делегирование, источник данных и пары "цель–действие", позволяют объекту представления связываться с контроллером независимым образом.

Предметный указатель

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э