Мэтт Гэлловей

СИЛА OBJECTIVE-C 2.0

Эффективное программирование для iOS и OS X

Addison-Wesley

ПИТЕР

OZON.ru

Matt Galloway

Effective Objective-C 2.0

52 Specific Ways to Improve Your iOS and OS X Programs

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Мэтт Гэлловей

СИЛА OBJECTIVE-C 2.0

Эффективное программирование для iOS и OS X

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

ББК 32.973.2- 018.1
УДК 004.43
Г98

Гэлловей М.

Г98 Сила Objective-C 2.0. Эффективное программирование для iOS и OS X. — СПб.: Питер, 2014. — 304 с.: ил. — (Серия «Библиотека специалиста»).

ISBN 978-5-496-00963-8

Эта книга поможет вам освоить всю мощь языка программирования Objective-C 2.0 и научит применять его максимально эффективно при разработке мобильных приложений для iOS и OS X. Автор описывает работу языка на понятных практических примерах, которые помогут как начинающим программистам, так и опытным разработчикам повысить уровень понимания Objective-C и существенно обогатить опыт его применения в своей работе.

В книге содержится 52 проверенных подхода для написания «чистого» и работающего кода на Objective-C, которые можно легко использовать на практике. Автор рассматривает такие темы, как проектирование интерфейсов и API, управление памятью, блоки и GCD, системные фреймворки и другие аспекты программирования на Objective-C, понимание которых поможет в эффективной разработке приложений для iOS или OS X.

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

ББК 32.973.2-018.1
УДК 004.43

Authorized translation from the English language edition, entitled Effective Objective-C 2.0: 52 Specific Ways to Improve Your iOS and OS X Programs; ISBN 978-0321917010; by Galloway, Matt; published by Pearson Education, Inc, publishing as Addison-Wesley Professional. Copyright © 2012 Pearson Education, Inc

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Права на издание получены по соглашению с Pearson Education, Inc. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

ISBN 978-0321917010 англ.
ISBN 978-5-496-00963-8

© Pearson Education, Inc., 2012
© *Перевод на русский язык* ООО Издательство «Питер», 2014
© *Издание на русском языке, оформление* ООО Издательство «Питер», 2014

ОГЛАВЛЕНИЕ

Предисловие 8

О книге. 8

Для кого написана эта книга. 9

Какие темы рассматриваются в книге 10

Благодарности. 12

Об авторе 14

Глава 1. Осваиваем Objective-C 15

1. Познакомьтесь с истоками Objective-C 15

2. Минимизируйте импортирование в заголовках. 19

3. Используйте литеральный синтаксис вместо эквивалентных методов 23

4. Используйте типизованные константы вместо препроцессорных директив #define. 29

5. Используйте перечисления для состояний, флагов и кодов ошибок 34

Глава 2. Объекты, сообщения и исполнительная среда 42

6. Разберитесь, что такое свойства 42

7. Используйте прямое обращение к переменным экземпляров при внутренних операциях 51

8. Разберитесь, что такое равенство объектов 54

9. Используйте паттерн «Группа классов» и сокрытие подробностей реализации 61

10. Используйте ассоциированные объекты для присоединения пользовательских данных к существующим классам 66

11. Разберитесь с objc_msgSend 70

12. Разберитесь с перенаправлением сообщений 74

13. Используйте замены для отладки непрозрачных методов 83

14. Разберитесь с объектами классов 87

Глава 3. Проектирование интерфейса и API 93

15. Используйте префиксы для предотвращения конфликтов имен 93

16. Используйте основной инициализатор 98

17. Реализуйте метод description 105

18. Выбирайте неизменяемые объекты 111

19. Используйте четкие и последовательные схемы формирования имен 117

20. Разберитесь с префиксами в именах закрытых методов 125

21. Разберитесь с моделью ошибок Objective-C 128

22. Разберитесь с протоколом NSCopying 133

Глава 4. Протоколы и категории 139

23. Используйте протоколы делегатов и источников данных для взаимодействия между объектами 139

24. Используйте категории для разбиения классов 148

25. Всегда используйте префиксы имен категорий в классах, предназначенных для внешнего использования 152

26. Избегайте использования свойств в категориях 155

27. Используйте категории продолжения классов для сокрытия подробностей реализации 159

28. Используйте протоколы для создания анонимных объектов 166

Глава 5. Управление памятью 171

29. Разберитесь с механизмом подсчета ссылок 171

30. Используйте ARC для упрощения подсчета ссылок 179

31. Освобождайте ссылки и зачищайте состояние наблюдения только в dealloc 189

32. Защищайте управление памятью с помощью безопасного кода 193

33. Используйте слабые ссылки, чтобы избежать удерживающих циклов.. 197

34. Используйте пулы автоматического освобождения, чтобы уменьшить затраты памяти 201

35. Используйте объекты-зомби для решения проблем, связанных с управлением памятью. 206

36. Остерегайтесь метода retainCount 213

Глава 6. Блоки и Grand Central Dispatch 217

37. Разберитесь с блоками 218

38. Создайте typedef для часто используемых типов блоков.. 225

39. Используйте блоки в обработчиках, чтобы уменьшить логическое разбиение кода. 228

40. Избегайте циклов удержания между блоками и объектами, которым они принадлежат. 235

41. Используйте очереди диспетчеризации для синхронизации 240

42. Используйте GCD вместо метода performSelector и его семейства 246

43. Научитесь выбирать: GCD или очереди операций 250

44. Используйте группы диспетчеризации для платформенного масштабирования.. 254

45. Используйте dispatch_once для потоково-безопасного одноразового выполнения кода. 259

46. Остерегайтесь функции dispatch_get_current_queue 261

Глава 7. Системные фреймворки 268

47. Познакомьтесь поближе с системными фреймворками 268

48. Используйте перебор с выполнением блоков вместо циклов for.. .. 272

49. Используйте упрощенное преобразование для коллекций с нестандартной семантикой управления памятью.. 279

50. Используйте NSCache вместо NSDictionary для кэша. 285

51. Придерживайтесь компактных реализаций initialize и load.. 290

52. Запомните, что NSTimer удерживает приемник.. 296

Предисловие

Objective-C громоздок. Objective-C неуклюж. Objective-C уродлив. Я сам слышал, как все это говорили об Objective-C. Напротив, я нахожу этот язык элегантным, гибким и красивым. Но для того чтобы ваш код заслуживал всех этих эпитетов, необходимо понимать не только основные концепции, но и все нюансы, скрытые ловушки и странности: этой теме и посвящена книга.

О книге

Книга не научит вас синтаксису Objective-C. Предполагается, что вы его уже знаете. Вместо этого книга научит вас полноценно использовать этот язык для написания хорошего кода. Поскольку Objective-C произошел от Smalltalk, он чрезвычайно динамичен. Большая часть работы, которая в других языках, как правило, выполняется компилятором, в Objective-C обычно переходит на стадию выполнения. В результате код может нормально функционировать в ходе тестирования, но выдавать всевозможные странности на стадии реальной эксплуатации — например, при обработке некорректных данных. Конечно, лучший способ избежать подобных проблем — изначальное написание хорошего, качественного кода.

Многие рассматриваемые вопросы, строго говоря, не имеют прямого отношения к базовому синтаксису Objective-C. В тексте также упоминаются аспекты, относящиеся к системным библиотекам — например, Grand Central Dispatch, часть libdispatch. Также неоднократно встречаются ссылки на многие классы фреймворка Foundation, не исключая и корневой класс `NSObject`, поскольку современная разработка на Objective-C ориентирована на Mac OS X или iOS. Несомненно, при разработке для любой из этих систем вы

будете использовать системные фреймворки, объединенные под названием Cocoa и Cocoa Touch соответственно.

С первых дней появления iOS разработчики потянулись к Objective-C. Некоторые из них не имели дела с программированием, другие обладали опытом работы на Java или C++, третьи занимались веб-программированием. В любом случае все разработчики не должны жалеть времени на то, чтобы научиться эффективному использованию языка. Это сделает их код более производительным, упростит его сопровождение и снизит вероятность ошибок.

Хотя я работал над книгой около полугода, ее предыстория занимает несколько лет. Я купил iPod Touch просто так, без далеко идущих планов; затем, когда была выпущена первая версия SDK, я решил поэкспериментировать с разработкой. Так я построил свое первое приложение, которое было опубликовано под названием Subnet Calc и неожиданно получило намного больше загрузок, чем я мог рассчитывать. Я обрел уверенность в том, что мое будущее связано с красивым языком, с которым я только что познакомился. С того времени я вел исследования в области языка Objective-C и регулярно писал о нем в блоге на сайте *www.galloway.me.uk/*. Меня особенно интересуют подробности внутренней реализации — скажем, реализация блоков (blocks) и особенности работы ARC. Когда мне представилась возможность написать книгу об этом языке, я не упустил этот шанс.

Чтобы получить максимум пользы от книги, я рекомендую активно искать темы, которые вам особенно интересны или актуальны для того, над чем вы работаете в настоящее время. Каждый подход можно читать отдельно, используя перекрестные ссылки для перехода к сопутствующим проблемам. В каждой главе собраны взаимосвязанные темы, а название главы поможет вам быстро найти вопросы, относящиеся к определенной возможности языка.

ДЛЯ КОГО НАПИСАНА ЭТА КНИГА

Книга предназначена для разработчиков, которые хотят углубить свои знания Objective-C, а также стремятся писать код, простой в сопровождении, эффективный и содержащий меньше ошибок. Даже если вы еще не являетесь разработчиком Objective-C, но у вас имеется опыт работы на других объектно-ориентированных языках (например, Java или C++), вы все равно узнаете много полезного.

Впрочем, в таком случае неплохо заранее ознакомиться с синтаксисом Objective-C.

КАКИЕ ТЕМЫ РАССМАТРИВАЮТСЯ В КНИГЕ

В книге не рассматриваются основы Objective-C — для этого есть много других книг и ресурсов. Вместо этого книга учит эффективно использовать язык. Она состоит из подходов, каждый из которых содержит простую и доступную информацию. Подходы сгруппированы по темам.

Глава 1. Осваиваем Objective-C

Основные концепции, относящиеся к языку в целом.

Глава 2. Объекты, сообщения и исполнительная среда

Связи и взаимодействия между объектами — важная сторона любого объектно-ориентированного языка. В этой главе мы рассмотрим эти аспекты и изучим строение исполнительной среды (runtime).

Глава 3. Проектирование интерфейса и API

Код редко пишется в расчете на одноразовое использование. Даже если вы не станете публиковать его для стороннего использования, скорее всего, код будет задействован в нескольких проектах. В этой главе объясняется, как написать класс, который хорошо встраивается в систему связей Objective-C.

Глава 4. Протоколы и категории

Протоколы и категории входят в число важнейших возможностей языка. Их эффективное использование сделает ваш код более удобочитаемым, упростит его сопровождение и снизит вероятность ошибки. Эта глава поможет вам освоить их.

Глава 5. Управление памятью

Модель управления памятью Objective-C основана на подсчете ссылок. Этот факт давно создавал проблемы для начинающих, особенно имеющих опыт работы на языке с уборкой мусора. Введение автоматического подсчета ссылок (ARC, Automatic Reference Counting) упростило ситуацию, но разработчик должен учитывать много важных факторов, чтобы модель объектов работала правильно и не страдала от утечки памяти. В этой главе

читатель познакомится с основными проблемами, связанными с управлением памятью.

Глава 6. Блоки и Grand Central Dispatch

Блоки представляют собой лексические замыкания (closures) для языка C, введенные компанией Apple. Они обычно используются в Objective-C для решения задач, в которых интенсивно используется шаблонный код. GCD (Grand Central Dispatch) предоставляет простой интерфейс многопоточного программирования. Блоки рассматриваются как задачи GCD, которые могут выполняться — возможно, параллельно (в зависимости от системных ресурсов). Эта глава поможет вам извлечь максимум пользы из этих двух основополагающих технологий.

Глава 7. Системные фреймворки

Как правило, будем писать код Objective-C для Mac OS X или iOS. В таких случаях в вашем распоряжении будет полный набор системных фреймворков: Cocoa и Cocoa Touch соответственно. В этой главе приведен краткий обзор фреймворков, а также углубленно рассмотрены некоторые из их классов.

Если у вас появятся какие-либо вопросы или замечания по поводу книги, не стесняйтесь обращаться ко мне. Данные для связи можно найти на сайте книги по адресу *www.effectiveobjectivec.com*.

Благодарности

Когда меня спросили, не хочу ли я написать книгу по Objective-C, я немедленно загорелся энтузиазмом. Я уже читал другие книги этой серии и знал, что написать книгу по Objective-C будет непросто. И все же благодаря помощи многих людей эта книга увидела свет.

Многие идеи, использованные в книге, я почерпнул в превосходных блогах, посвященных Objective-C. Майк Эш (Mike Ash), Мэтт Галлахер (Matt Gallagher), «bbum» — это лишь некоторые из авторов, чьи блоги я читаю. За прошедшие годы эти блоги помогли мне углубить свое понимание языка. В блоге NSHipster Мэтта Томпсона (Matt Thompson) также опубликованы отличные статьи, которые дали мне пищу для размышлений во время подбора материала. Наконец, превосходная документация от Apple тоже принесла огромную пользу.

Я бы не смог написать эту книгу без навыков и знаний, приобретенных мной во время работы в MX Telecom. В частности, Мэттью Ходжсон (Matthew Hodgson) предоставил мне возможность разработать первое iOS-приложение компании на основе развитой кодовой базы C++. Знания, полученные мной в этом проекте, в значительной мере заложили фундамент для последующей работы.

За прошедшие годы у меня было много замечательных коллег, с которыми я поддерживал постоянный контакт — как из-за научной работы, так и просто ради того, чтобы посидеть за пивом и поболтать. Все это пригодилось мне в работе над книгой.

У меня остались прекрасные впечатления от работы с группой из издательства Pearson. Трина Макдональд (Trina MacDonald), Оливия Баседжио (Olivia Basegio), Скотт Мейерс (Scott Meyers) и Крис Зан (Chris Zahn) помогали и приободряли меня, когда это было необходимо. Они предоставили инструменты, которые позволяли мне

писать книгу, не отвлекаясь на посторонние дела, и отвечали на мои вопросы.

Технические рецензенты, с которыми я имел удовольствие работать, оказали огромную помощь. Их вопросы позволили поднять качество книги на самый высокий уровень. Их внимание к подробностям во время работы с рукописью было просто невероятным.

Наконец, я не смог бы написать эту книгу без понимания и поддержки со стороны Хелен. Наш первый ребенок родился в тот день, когда я должен был взяться за работу, поэтому, естественно, сдача книги немного задержалась. Хелен и Рози оказали мне неоценимую поддержку на протяжении всей работы.

Об авторе

Мэтт Гэлловей (Matt Galloway) — iOS-разработчик из Лондона (Великобритания). Он окончил Кембриджский университет (колледж Пемброк) в 2007 году с ученой степенью магистра технических наук со специализацией в области электроники и информатики. С тех пор занимается программированием, в основном на Objective-C. Мэтт занимался программированием для iOS еще со времен выпуска первого SDK. Он публикуется в Twitter как *@mattjgalloway* и регулярно поставляет материалы для Stack Overflow (*http://stackoverflow.com*).

Глава 1
ОСВАИВАЕМ OBJECTIVE-C

Objective-C дополняет C объектно-ориентированными возможностями, используя для этого совершенно новый синтаксис. Синтаксис Objective-C часто называют слишком громоздким, из-за того что в нем используются многочисленные квадратные скобки, а очень длинные имена методов являются рядовым явлением. Полученный исходный код легко читается, но программистам C++ и Java в нем часто бывает трудно разобраться.

Вы можете быстро научиться писать программы на Objective-C, но при этом необходимо помнить о многих тонкостях и функциях, о которых часто забывают. Вдобавок программисты часто злоупотребляют некоторыми возможностями или недостаточно хорошо понимают их, что приводит к трудностям при отладке и сопровождении такого кода. В этой главе рассматриваются основополагающие темы, а в последующих главах — конкретные области языка и соответствующих фреймворков.

1 ПОЗНАКОМЬТЕСЬ С ИСТОКАМИ OBJECTIVE-C

Язык Objective-C похож на другие объектно-ориентированные языки (такие, как C++ и Java), но также во многом отличается от них. Если у вас уже есть опыт работы на другом объектно-ориентированном языке, вы быстро поймете многие парадигмы и паттерны Objective-C. Впрочем, синтаксис на первых порах кажется непривычным, потому что вместо вызовов функций в нем используется механизм обмена сообщениями.

Objective-C происходит от Smalltalk, прародителя обмена сообщениями. Следующий пример демонстрирует различия между обменом сообщениями и вызовом функций:

```
// Messaging (Objective-C)
Object *obj = [Object new];
[obj performWith:parameter1 and:parameter2];

// Вызов функций (C++)
Object *obj = new Object;
obj->perform(parameter1, parameter2);
```

Принципиальное различие заключается в том, что в механизме обмена сообщениями выполняемый код выбирается исполнительной средой, тогда как при вызове функций выбор выполняемого кода осуществляется компилятором. При использовании полиморфизма в примере с вызовом функций используется разновидность динамического выбора кода по так называемой *виртуальной таблице* (virtual table). Но при обмене сообщениями выбор кода всегда осуществляется динамически. Более того, компилятор при этом даже не обращает внимания на тип передаваемого объекта. Он также определяется на стадии выполнения; при этом используется процесс *динамической привязки* (dynamic binding), более подробно описанный в подходе 11.

Большая часть «черной работы» выполняется исполнительной средой Objective-C, а не компилятором. Исполнительная среда содержит все структуры данных и функции, необходимые для работы объектно-ориентированной поддержки Objective-C. В частности, исполнительная среда включает в себя все методы управления памятью. Фактически это служебный код, который связывает воедино весь ваш код и существует в форме динамической библиотеки, с которой компонуется ваш код. При каждом обновлении исполнительной среды ваше приложение немедленно начинает пользоваться всеми преимуществами оптимизации. Если же язык выполняет большую часть работы на стадии компиляции, то улучшения вступят в силу только после перекомпиляции программы.

Objective-C образует надмножество C, поэтому в программах Objective-C доступны все возможности языка C. Следовательно, для написания эффективных программ на Objective-C необходимо понимать базовые концепции как C, так и Objective-C. В частности, понимание модели памяти C поможет вам понять модель памяти Objective-C и почему подсчет ссылок работает именно так, а не иначе. А для этого необходимо знать, что указатели используются

для обозначения объектов в Objective-C. Когда вы объявляете переменную, используемую для хранения ссылки на объект, синтаксис выглядит примерно так:

```
NSString *someString = @"The string";
```

В этом синтаксисе, в основном позаимствованном прямо из C, объявляется переменная с именем `someString` и типом `NSString*`. Это означает, что переменная содержит указатель на `NSString`. Все объекты в Objective-C должны объявляться таким способом, потому что память для объектов всегда выделяется из кучи (heap) и никогда — из стека. Объявление объекта с выделением памяти из стека в Objective-C недопустимо:

```
NSString stackString;
// Ошибка: память для интерфейсных типов не может выделяться
           статически
```

Переменная `someString` указывает на участок памяти, выделенной из кучи и содержащей объект `NSString`. Соответственно, при создании другой переменной, указывающей на тот же адрес, копирование не выполняется; вместо этого в программе появляются две переменные, указывающие на один объект:

```
NSString *someString = @"The string";
NSString *anotherString = someString;
```

В этом примере существует только один экземпляр `NSString`, но на него указывают две переменные. Эти две переменные относятся к типу `NSString*`; это означает, что в текущем кадре стека выделяются два участка памяти, размер которых соответствует размеру указателя (4 байта для 32-разрядной архитектуры, 8 байт для 64-разрядной). Эти участки памяти содержат одно и то же значение: адрес экземпляра `NSString` в памяти.

Такое распределение памяти продемонстрировано на рис. 1.1. В данные, хранящиеся для экземпляра `NSString`, включаются байты, необходимые для представления строки. Памятью, выделяемой из кучи, приходится управлять напрямую, тогда как память, выделенная в стеке для хранения переменных, автоматически очищается при извлечении кадра стека, в котором эти переменные были созданы.

Управление памятью кучи абстрагируется средой Objective-C. Соответственно, вы не можете выделять и освобождать память объектов вызовами `malloc` и `free`. Исполнительная среда Objective-C

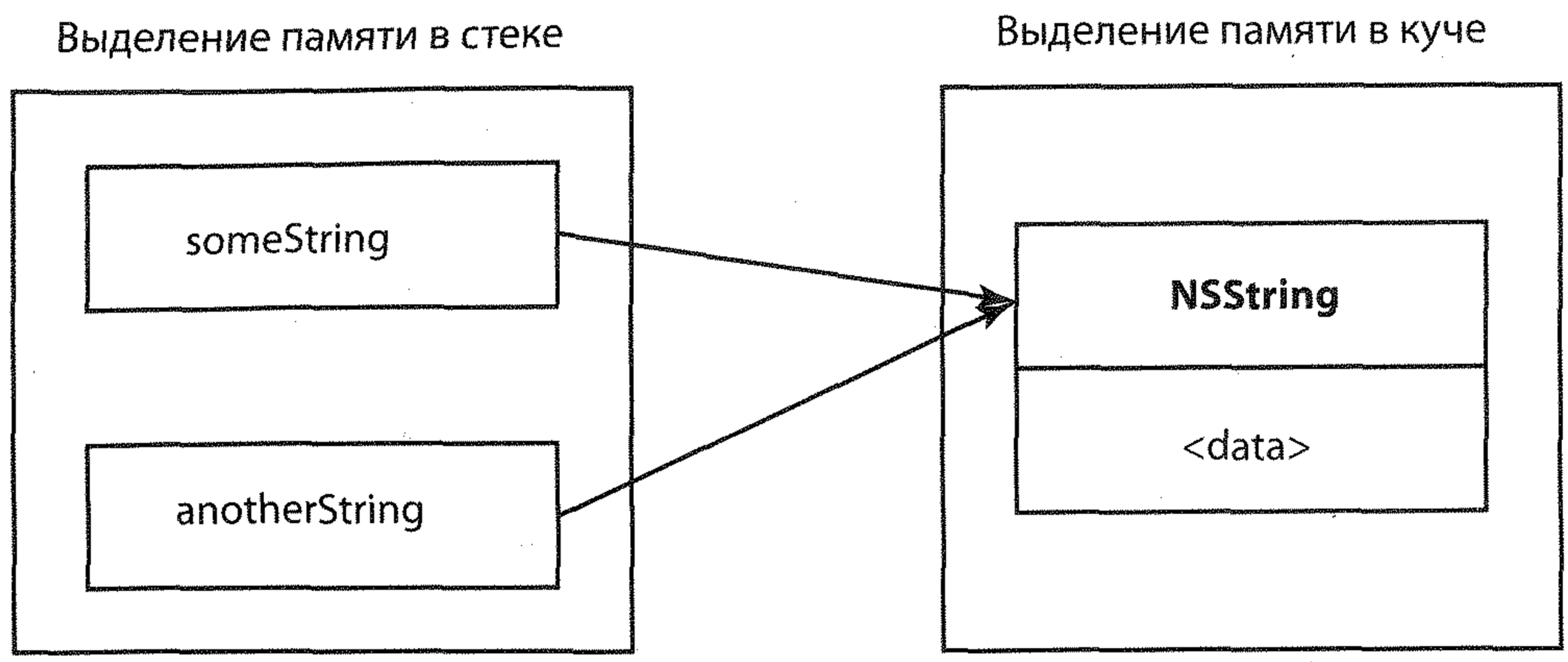


Рис. 1.1. Распределение памяти с экземпляром NSString в куче и двумя указателями на него в стеке

абстрагирует эти операции в архитектуре управления памятью, известной как *подсчет ссылок* (reference counting) — см. подход 29.

Иногда в Objective-C встречаются переменные, у которых в определении нет символа * и которые могут использовать память из стека. В таких переменных не могут храниться объекты Objective-C. В качестве примера можно привести тип `CGRect` из фреймворка CoreGraphics:

```
CGRect frame;
frame.origin.x = 0.0f;
frame.origin.y = 10.0f;
frame.size.width = 100.0f;
frame.size.height = 150.0f;
```

`CGRect` представляет собой структуру C, определяемую следующим образом:

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

Эти структурные типы используются в системных фреймворках, где дополнительные затраты на использование объектов Objective-C могут отрицательно повлиять на быстродействие. Создание объектов требует дополнительных операций, которые не нужны для

структур (например, выделение и освобождение памяти в куче). Если хранимые данные состоят только из необъектных типов (`int`, `float`, `double`, `char` и т. д.), обычно используется структура — такая, как `CGRect`.

Прежде чем браться за написание кода на Objective-C, я рекомендую что-нибудь почитать о языке C и освоиться с его синтаксисом. Если вы сразу возьметесь за Objective-C, некоторые составляющие синтаксиса могут показаться непонятными.

УЗЕЛКИ НА ПАМЯТЬ

- Objective-C представляет собой надмножество C с добавлением объектно-ориентированных возможностей. В Objective-C используется механизм обмена сообщениями с динамической привязкой; это означает, что тип объекта определяется во время выполнения. Код, выполняемый для конкретного сообщения, выбирается исполнительной средой, а не компилятором.
- Понимание базовых принципов C поможет вам писать эффективные программы на Objective-C. В частности, вы должны понимать модель памяти и смысл указателей.

2 МИНИМИЗИРУЙТЕ ИМПОРТИРОВАНИЕ В ЗАГОЛОВКАХ

В языке Objective-C, как в C и C++, используются заголовочные файлы и файлы реализации. При написании класса на Objective-C обычно создаются два файла, имена которых соответствует имени класса; заголовочный файл имеет суффикс .h, а файл реализации — суффикс .m.

Файлы, созданные при создании класса, могут выглядеть так:

```
// EOCPerson.h
#import <Foundation/Foundation.h>
@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@end

// EOCPerson.m
```

```
#import "EOCPerson.h"
@implementation EOCPerson
// Реализация методов
@end
```

Импортирование Foundation.h обязательно практически для всех классов, которые вы когда-либо будете создавать в Objective-C, — либо так, либо вы импортируете базовый заголовочный файл фреймворка, в котором находится суперкласс класса. Например, при создании приложений iOS вам придется часто субклассировать UIViewController. Заголовочные файлы этих классов импортируют файл UIKit.h.

Пока с этим классом все хорошо. Он полностью импортирует Foundation, но это неважно. Поскольку этот класс наследует от класса, являющегося частью Foundation, весьма вероятно, что большая его часть будет задействована пользователями EOCPerson. То же можно сказать о классе, наследующем от UIViewController: его пользователи будут потреблять большую часть UIKit.

Возможно, в будущем вы создадите новый класс EOCEmployer. Затем вы решаете, что в экземпляре EOCPerson должен храниться экземпляр EOCEmployer, и объявляете для него свойство:

```
// EOCPerson.h
#import <Foundation/Foundation.h>
@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, strong) EOCEmployer *employer;
@end
```

Однако здесь возникает проблема: класс EOCEmployer невидим при компиляции файла, импортирующего EOCPerson.h. Было бы неправильно требовать, чтобы каждый файл, который импортирует EOCPerson.h, также импортировал EOCEmployer.h. По этой причине в начало EOCPerson.h обычно включается директива:

```
#import "EOCEmployer.h"
```

Такое решение работает, но считается нежелательным. Для компиляции кода, использующего EOCPerson, не обязательно располагать полной информацией о EOCEmployer — достаточно просто знать о существовании класса с именем EOCEmployer. К счастью, существует способ передачи этой информации компилятору:

```
@class EOCEmployer;
```

Это называется *опережающим объявлением* класса. В результате заголовочный файл EOCPerson будет выглядеть так:

```
// EOCPerson.h
#import <Foundation/Foundation.h>
@class EOCEmployer;

@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, strong) EOCEmployer *employer;
@end
```

Файл реализации класса EOCPerson должен импортировать заголовочный файл EOCEmployer, так как для использования класса ему необходимо иметь всю информацию об его интерфейсе. Полученный файл реализации будет выглядеть так:

```
// EOCPerson.m
#import "EOCPerson.h"
#import "EOCEmployer.h"

@implementation EOCPerson
// Реализация методов
@end
```

Откладывая импортирование до того момента, когда оно действительно необходимо, вы ограничиваете состав информации, которую должен импортировать пользователь вашего класса. Если в нашем примере файл EOCEmployer.h импортировался в EOCPerson.h, любой компонент, импортирующий EOCPerson.h, также импортирует все содержимое EOCEmployer.h. Если цепочка импортирования продолжится, в результате может быть импортировано намного больше, чем вы рассчитывали, что, бесспорно, увеличит время компиляции.

Опережающие объявления также решают проблему двух классов, содержащих взаимные ссылки. Представьте, что произойдет, если EOCEmployer содержит методы добавления и удаления работников, которые определяются в заголовочном файле следующим образом:

```
- (void)addEmployee:(EOCPerson*)person;
- (void)removeEmployee:(EOCPerson*)person;
```

На этот раз класс EOCPerson должен быть видим компилятору — по тем же причинам, что и в обратном случае. Однако решение этой проблемы импортированием другого заголовка в каждом заголовке создаст классическую ситуацию циклических ссылок. В ходе разбора один заголовок импортирует другой, который импортирует первый. Использование директивы #import вместо #include не приводит к возникновению бесконечного цикла, но означает, что один из классов будет откомпилирован неправильно. Попробуйте сами, если не верите!

И все же иногда бывает нужно импортировать заголовок в заголовке. Скажем, вы должны импортировать заголовок, определяющий суперкласс, от которого вы наследуете. Аналогичным образом, при объявлении протоколов, которым должен соответствовать класс, эти протоколы должны быть полностью определены без опережающего объявления. Компилятор должен «видеть» методы, определяемые протоколом, а не просто узнать о существовании протокола из опережающего объявления.

Допустим, класс прямоугольника наследует от класса геометрической фигуры и реализует протокол, обеспечивающий его прорисовку:

```
// EOCRectangle.h
#import "EOCShape.h"
#import "EOCDrawable.h"

@interface EOCRectangle : EOCShape <EOCDrawable>
@property (nonatomic, assign) float width;
@property (nonatomic, assign) float height;
@end
```

Дополнительное импортирование неизбежно. В таких протоколах желательно вынести его в отдельный заголовочный файл. Если бы протокол EOCDrawable был частью большего заголовочного файла, пришлось бы импортировать его полностью, что создало бы проблемы зависимости и времени компиляции, о которых говорилось ранее.

Впрочем, не все протоколы должны размещаться в отдельных файлах — как, например, протоколы делегатов (см. подход 23). В таких случаях протокол имеет смысл только при определении вместе с классом, для которого он является делегатом. Часто бывает лучше объявить, что класс реализует делегата, в категории продолжения класса (см. подход 27). Это означает, что директива импортирования

заголовка, содержащего протокол делегата, может размещаться в файле реализации вместо открытого заголовочного файла.

Включая директиву импортирования в заголовочный файл, всегда спрашивайте себя, действительно ли это необходимо. Если опережающее объявление возможно — выбирайте этот вариант. Если импортирование относится к тому, что используется в свойстве, переменной экземпляра или реализации протокола, и может быть перемещено в категорию продолжения класса (см. подход 27) — выбирайте этот вариант. Такая стратегия ускоряет компиляцию и сокращает количество взаимозависимостей, которые могут создать проблемы при сопровождении или предоставлении доступа к отдельным частям кода в открытом API, если вдруг возникнет такая необходимость.

УЗЕЛКИ НА ПАМЯТЬ

- Всегда импортируйте заголовки в точке с максимальной глубиной. Обычно это подразумевает опережающее объявление классов в заголовке и импортирование соответствующих заголовков в реализации. Все это способствует снижению зависимостей между классами.
- Иногда опережающие объявления невозможны — скажем, при объявлении реализации протоколов. В таких ситуациях следует подумать о перемещении объявления реализации протокола в категорию продолжения класса, если это возможно. В других случаях импортируйте заголовок, содержащий только определение протокола.

ИСПОЛЬЗУЙТЕ ЛИТЕРАЛЬНЫЙ СИНТАКСИС ВМЕСТО ЭКВИВАЛЕНТНЫХ МЕТОДОВ

В ходе разработки на Objective-C вы будете постоянно сталкиваться с несколькими классами, входящими в фреймворк Foundation. Хотя формально код Objective-C можно писать и без Foundation, на практике такие программы встречаются редко. Речь идет о классах `NSString`, `NSNumber`, `NSArray` и `NSDictionary`. Вероятно, не нужно объяснять, какие структуры данных представляются каждым из этих классов.

Язык Objective-C известен пространностью своего синтаксиса. Однако еще начиная с Objective-C 1.0 существует очень простой способ создания объектов NSString. Это так называемые строковые литералы, которые выглядят примерно так:

```
NSString *someString = @"Effective Objective-C 2.0";
```

Без этого синтаксиса для создания объекта NSString пришлось бы выделить память и инициализировать объект NSString обычными вызовами методов alloc и init. К счастью, синтаксис литералов был расширен в последних версиях компилятора на экземпляры NSNumber, NSArray и NSDictionary. При использовании синтаксиса литералов код становится более компактным и удобочитаемым.

ЧИСЛОВЫЕ ЛИТЕРАЛЫ

Иногда бывает нужно упаковать в объекте Objective-C целое, вещественное или логическое значение. Для этого используется класс NSNumber, способный представлять различные числовые типы. Без литералов экземпляры пришлось бы создавать так:

```
NSNumber *someNumber = [NSNumber numberWithInt:1];
```

Команда создает число, которому присваивается целочисленное значение 1. Однако литеральный синтаксис делает команду более понятной:

```
NSNumber *someNumber = @1;
```

Как видите, литеральный синтаксис отличается большей наглядностью, но этим его достоинства не ограничены. Синтаксис также распространяется на все остальные типы данных, которые могут представляться экземплярами NSNumber, например:

```
NSNumber *intNumber = @1;
NSNumber *floatNumber = @2.5f;
NSNumber *doubleNumber = @3.14159;
NSNumber *boolNumber = @YES;
NSNumber *charNumber = @'a';
```

Литеральный синтаксис также работает и в выражениях:

```
int x = 5;
float y = 6.32f;
NSNumber *expressionNumber = @(x * y);
```

Числовые литералы чрезвычайно полезны. С ними операции с объектами NSNumber становятся более выразительными, так как основную часть объявления занимает значение, а не посторонний синтаксис.

ЛИТЕРАЛЬНЫЕ МАССИВЫ

Массивы принадлежат к числу наиболее распространенных структур данных. До появления литералов массив приходилось создавать следующим образом:

```
NSArray *animals =
    [NSArray arrayWithObjects:@"cat", @"dog",
                              @"mouse", @"badger", nil];
```

С появлением литералов синтаксис заметно упростился:

```
NSArray *animals = @[@"cat", @"dog", @"mouse", @"badger"];
```

Впрочем, заметное упрощение синтаксиса не исчерпывает всех возможностей литералов при работе с массивами. Одна из стандартных операций с массивами — получение элемента с заданным индексом. Литералы упрощают и ее. Обычно для этой цели используется метод objectAtIndex::

```
NSString *dog = [animals objectAtIndex:1];
```

С литералами достаточно следующей конструкции:

```
NSString *dog = animals[1];
```

Эта операция, как и в остальных примерах применения литерального синтаксиса, получается более компактной и понятной. Более того, она очень похожа на способы индексирования массивов в других языках.

Однако при создании массивов с использованием литерального синтаксиса следует учитывать одно обстоятельство. Если вместо какого-либо объекта указывается nil, происходит исключение, поскольку литеральный синтаксис в действительности представляет собой синтаксическое «украшение» для создания массива с последующим включением всех объектов в квадратных скобках. Исключение выглядит примерно так:

```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '***
```

```
-[__NSPlaceholderArray initWithObjects:count:]: attempt to
insert nil object from objects[0]'
```

Мы подошли к распространенной проблеме, связанной с переходом на литералы.

Следующий код создает два массива, по одному для каждого синтаксиса:

```
id object1 = /* … */;
id object2 = /* … */;
id object3 = /* … */;
NSArray *arrayA = [NSArray arrayWithObjects:
                        object1, object2, object3, nil];
NSArray *arrayB = @[object1, object2, object3];
```

Теперь рассмотрим сценарий, при котором `object1` и `object3` указывают на действительные объекты Objective-C, а `object2` содержит `nil`. Попытка создания литерального массива `arrayB` приведет к выдаче исключения. Массив `arrayA` при этом будет создан, но он будет содержать только `object1`. Дело в том, что метод `arrayWithObjects:` перебирает аргументы до тех пор, пока не обнаружит `nil`, а это происходит раньше, чем ожидалось.

Это тонкое различие означает, что литералы намного надежнее. Гораздо лучше получить исключение, возможно, с аварийным завершением приложения, чем создать усеченный массив. Скорее всего, вставка `nil` в массив произошла из-за ошибки программиста, а исключение поможет быстрее выявить эту ошибку.

ЛИТЕРАЛЬНЫЕ СЛОВАРИ

Словари (dictionaries) предоставляют ассоциативную структуру данных для добавления пар «ключ-значение». Как и массивы, словари очень часто используются в коде Objective-C. В традиционно синтаксисе создание словаря выглядит так:

```
NSDictionary *personData =
    [NSDictionary dictionaryWithObjectsAndKeys:
        @"Matt", @"firstName",
        @"Galloway", @"lastName",
        [NSNumber numberWithInt:28], @"age",
        nil];
```

Запись в порядке <объект>, <ключ>, <объект>, <ключ> и т. д. выглядит невразумительно. Обычно мы рассматриваем пары словаря

в другом направлении — от ключа к объекту, поэтому этот синтаксис оставляет желать лучшего. Однако литералы и на этот раз существенно упрощают запись:

```
NSDictionary *personData =
    @{@"firstName" : @"Matt",
      @"lastName" : @"Galloway",
      @"age" : @28};
```

Такая запись намного логичнее: ключи располагаются перед объектами, как и следовало ожидать. Также следует заметить, что числовой литерал в этом примере лишний раз демонстрирует полезность числовых литералов. И значения, и ключи должны быть объектами Objective-C, поэтому сохранить целое число 28 невозможно; оно должно быть упаковано в экземпляр `NSNumber`. Благодаря литеральному синтаксису это обойдется вам всего в один лишний символ.

Как и при работе с массивами, литеральный синтаксис словарей приводит к выдаче исключения, если какое-либо из значений содержит `nil`. Однако по причинам, описанным выше, эта особенность весьма полезна. Она означает, что вместо словаря с пропущенными данными (из-за метода `dictionaryWithObjectsAndKeys:`, останавливающегося на первом `nil`) будет выдано исключение.

Также по аналогии с массивами возможно обращение к словарям с использованием литерального синтаксиса. Старый способ получения значения, связанного с заданным ключом, выглядел так:

```
NSString *lastName = [personData objectForKey:@"lastName"];
```

Эквивалентный литеральный синтаксис:

```
NSString *lastName = personData[@"lastName"];
```

И снова литералы сокращают объем избыточного синтаксиса, оставляя удобочитаемую строку кода.

ИЗМЕНЯЕМЫЕ МАССИВЫ И СЛОВАРИ

Наряду с чтением по индексу массива или ключу словаря также можно выполнить операцию присваивания, если объект является изменяемым (mutable). Присваивание с использованием обычных методов изменяемых массивов и словарей выглядит так:

```
[mutableArray replaceObjectAtIndex:1 withObject:@"dog"];
[mutableDictionary setObject:@"Galloway" forKey:@"lastName"];
```

Сравните с присваиванием, использующим индексирование:

```
mutableArray[1] = @"dog";
mutableDictionary[@"lastName"] = @"Galloway";
```

ОГРАНИЧЕНИЯ

Основное ограничение литерального синтаксиса состоит в том, что за исключением строк класс созданного объекта должен входить в фреймворк Foundation. Если вам потребуется создать экземпляр собственного субкласса, придется использовать нелитеральный синтаксис. Но поскольку `NSArray`, `NSDictionary` и `NSNumber` являются группами классов (см. подход 9), они редко субклассируются, так как эта задача весьма нетривиальна. Кроме того, обычно стандартных реализаций оказывается вполне достаточно. Для строк можно использовать нестандартный класс, но эта возможность должна быть включена на уровне параметров компилятора. Использовать ее не рекомендуется, если только вы совершенно твердо не убеждены в обратном.

Кроме того, для строк, массивов и словарей литеральный синтаксис может использоваться только для создания неизменяемых версий. Если вам потребуется создать изменяемый экземпляр, создайте изменяемую копию:

```
NSMutableArray *mutable = [@[@1, @2, @3, @4, @5] mutableCopy];
```

Это потребует лишнего вызова метода и создания лишнего объекта, но преимущества литерального синтаксиса перевешивают недостатки.

УЗЕЛКИ НА ПАМЯТЬ

- Используйте литеральный синтаксис для создания строк, чисел, массивов и словарей. Он понятнее и компактнее традиционных методов создания объектов.
- Попытка вставки `nil` в массив или словарь с литеральным синтаксисом приведет к выдаче исключения. Всегда помните, что эти значения должны быть отличны от `nil`.

4 ИСПОЛЬЗУЙТЕ ТИПИЗОВАННЫЕ КОНСТАНТЫ ВМЕСТО ПРЕПРОЦЕССОРНЫХ ДИРЕКТИВ #DEFINE

В программах часто требуется определять константы. Например, возьмем класс представления (view), который отображает себя на экране, а затем закрывается с применением анимаций. Типичная константа, которую стоит определить для такого класса, — продолжительность анимации. Вы уже знаете, что язык Objective-C произошел от C, и определяете константу следующим образом:

```
#define ANIMATION_DURATION 0.3
```

Эта строка представляет собой директиву препроцессора; везде, где в вашем исходном коде встречается строка `ANIMATION_DURATION`, она заменяется на 0.3. На первый взгляд это именно то, что нужно, но это определение не содержит информации о типе. По имени можно предположить, что значение как-то связано со временем, но этот смысл не имеет явного выражения. Кроме того, препроцессор бездумно заменяет все вхождения `ANIMATION_DURATION`, поэтому, если объявление находится в заголовочном файле, все остальные файлы, импортирующие этот заголовок, увидят результат замены.

Для решения этих проблем следует задействовать компилятор. Константу всегда можно определить более правильным способом, чем с использованием препроцессора. Например, следующая строка определяет константу типа `NSTimeInterval`:

```
static const NSTimeInterval kAnimationDuration = 0.3;
```

Обратите внимание: в таком стиле объявления присутствует информация о типе, которая четко определяет, что собой представляет константа. В данном случае используется тип `NSTimeInterval`, поэтому использование переменной желательно документировать. При большом количестве определяемых констант документирование, безусловно, поможет вам и другим людям, которые будут читать код в будущем.

Также обратите внимание на то, что константе присвоено имя. По стандартным соглашениям, имена констант, локальных по отношению к единице трансляции (файлу реализации), снабжаются префиксом k. Префиксом имен констант, доступных за пределами класса, обычно является имя класса. Соглашения выбора имен более подробно рассматриваются в подходе 19.

Место определения констант тоже играет важную роль. Порой возникает искушение объявлять препроцессорные определения в заголовочных файлах, *но это* чрезвычайно скверная привычка, особенно если имена определений не выбираются для предотвращения возможных конфликтов. Например, имя `ANIMATION_DURATION` в заголовочном файле нежелательно, поскольку оно будет присутствовать во всех *остальных файлах*, импортировавших заголовок. Даже статическая константа сама по себе не должна присутствовать в заголовочном файле. Так как в Objective-C нет пространств имен, это приведет к объявлению глобальной переменной с именем `kAnimationDuration`. *Имя* должно иметь префикс, связанный с классом, с которым оно используется, — например, `EOCViewClassAnimationDuration`. Преимущества четкой и логичной схемы выбора имен описаны в подходе 19.

Константу, которую не нужно делать доступной извне, следует определять в файле реализации, в котором она *используется*. Например, если константа продолжительности анимации задействована в субклассе `UIView` для приложения iOS, использующего UIKit, она будет выглядеть так:

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

@interface EOCAnimatedView : UIView
- (void)animate;
@end

// EOCAnimatedView.m
#import "EOCAnimatedView.h"

static const NSTimeInterval kAnimationDuration = 0.3;

@implementation EOCAnimatedView
- (void)animate {
    [UIView animateWithDuration:kAnimationDuration
                        animations:^(){
                            // Выполнение анимаций
                        }];
}
@end
```

Важно, что в объявлении переменной присутствуют оба ключевых слова `static` и `const`. Квалификатор `const` означает, что компилятор должен выдать ошибку при попытке изменения значения. В этой

ситуации это именно то, что нам нужно: любые изменения значения запрещены. Квалификатор `static` указывает, что переменная является локальной по отношению к единице трансляции (translation unit), в которой она определяется. Единицей трансляции называются входные данные, получаемые компилятором для создания одного объектного файла. В случае Objective-C каждому классу обычно соответствует одна единица трансляции: файл реализации (.m). Таким образом, в предыдущем примере константа `kAnimationDuration` объявляется локально по отношению к объектному файлу, сгенерированному на базе файла EOCAnimatedView.m. Если бы переменная не была объявлена статической, компилятор создал бы для нее *внешнее символическое имя* (external symbol). Если в другой единице трансляции будет объявлена одноименная переменная, компоновщик выдаст ошибку с сообщением следующего вида:

```
duplicate symbol _kAnimationDuration in:
    EOCAnimatedView.o
    EOCOtherView.o
```

Фактически при объявлении переменной с квалификаторами `static` и `const` компилятор вообще не создает символическое имя, а заменяет вхождения в тексте, как это делает препроцессор. Однако при этом сохраняются преимущества наличия информации о типе.

Иногда требуется сделать константу видимой извне. Например, такая необходимость может возникнуть, если ваш класс использует оповещения `NSNotificationCenter`. В этом механизме один объект отправляет оповещения, а другие регистрируются для их получения. У оповещений имеется строковое имя, которое будет разумно объявить как константу с внешней видимостью. При этом объект, желающий зарегистрироваться для получения оповещений, не обязан знать фактическую строку имени; достаточно использовать константу.

Чтобы такие константы могли использоваться за пределами единицы трансляции, в которой они определяются, они должны быть включены в глобальную таблицу символических имен. Соответственно, их синтаксис должен отличаться от синтаксиса примера с `static const`:

```
// В заголовочном файле
extern NSString *const EOCStringConstant;
// В файле реализации
NSString *const EOCStringConstant = @"VALUE";
```

Константа «объявляется» в заголовочном файле и «определяется» в файле реализации. В типе константы положение квалификатора `const` играет важную роль. Определения читаются в обратном порядке; это означает, что в приведенном примере `EOCStringConstant` является «константным указателем на NSString». Это именно то, что нам нужно; изменение константы, при котором она будет указывать на другой объект `NSString`, запрещено.

Ключевое слово `extern` в заголовочном файле сообщает компилятору, как он должен поступить, обнаружив константу в импортирующем ее файле. Оно означает, что в глобальной таблице символических имен следует создать запись для `EOCStringConstant`. Это позволит использовать константу даже при том, что компилятор не видит ее определения. Компилятор просто знает, что константа будет существовать на момент компоновки двоичного файла.

Константа должна определяться один и только один раз. Обычно она определяется в файле реализации, связанном с заголовочным файлом, в котором она объявлена. Компилятор выделяет память для строки в разделе данных объектного файла, сгенерированного на основе файла реализации. Когда объектный файл компонуется с другими объектными файлами для получения итогового двоичного файла, компоновщик сможет разрешить глобальное символическое имя `EOCStringConstant` в месте его использования.

Сам факт присутствия символического имени в глобальной таблице означает, что к выбору имен констант следует подходить с осторожностью. Например, класс для ввода регистрационных данных в приложении может использовать оповещение, которое выдается после завершения ввода. Оповещение может выглядеть так:

```
// EOCLoginManager.h
#import <Foundation/Foundation.h>

extern NSString *const EOCLoginManagerDidLoginNotification;

@interface EOCLoginManager : NSObject
- (void)login;
@end

// EOCLoginManager.m
#import "EOCLoginManager.h"

NSString *const EOCLoginManagerDidLoginNotification =
    @"EOCLoginManagerDidLoginNotification";
```

```
@implementation EOCLoginManager

- (void)login {
    // Асинхронное выполнение с последующим вызовом
      'p_didLogin'.
}

- (void)p_didLogin {
    [[NSNotificationCenter defaultCenter]
      postNotificationName:EOCLoginManagerDidLoginNotification
                    object:nil];
}

@end
```

Обратите внимание на имя константы. Использование префикса с именем класса, к которому относится константа, поможет избежать конфликтов имен. Эта схема также широко используется в системных фреймворках. Например, в UIKit имена оповещений объявляются как глобальные константы по аналогичной схеме: `UIApplicationDidEnterBackgroundNotification`, `UIApplicationWillEnterForegroundNotification` и т. д.

То же самое может быть сделано с константами других типов. Если в приведенных примерах продолжительность анимации должна быть доступна за пределами класса `EOCAnimatedView`, ее можно объявить следующим образом:

```
// EOCAnimatedView.h
extern const NSTimeInterval EOCAnimatedViewAnimationDuration;

// EOCAnimatedView.m
const NSTimeInterval EOCAnimatedViewAnimationDuration = 0.3;
```

Такое определение константы намного лучше препроцессорного определения, потому что неизменяемость значения контролируется компилятором. После определения в файле EOCAnimatedView.m это значение используется повсеместно. Препроцессорное определение может быть случайно переопределено по ошибке, в результате чего в разных частях приложения будут использоваться разные значения.

Итак, старайтесь избегать использования директив препроцессора для констант. Используйте константы, которые видны компилятору, — как глобальные определения `static const`, объявляемые в файлах реализации.

УЗЕЛКИ НА ПАМЯТЬ

- Избегайте препроцессорных определений. Они не содержат информации о типе и по сути представляют собой простые операции поиска/замены перед компиляцией. Они могут переопределяться без выдачи предупреждений, что приводит к рассогласованию значений в приложении.
- Константы уровня единицы трансляции определяйте в файлах реализации, используя квалификаторы `static const`. Такие константы не попадут в глобальную таблицу символических имен, поэтому их имена не нужно снабжать *префиксами*.
- Объявляйте глобальные константы как внешние (`external`) в заголовочном файле и определяйте их в соответствующем файле реализации. Такие константы включаются в глобальную таблицу символических имен, поэтому их имена должны быть разбиты на пространства — как правило, с помощью префикса имени класса, которому они принадлежат.

5 ИСПОЛЬЗУЙТЕ ПЕРЕЧИСЛЕНИЯ ДЛЯ СОСТОЯНИЙ, ФЛАГОВ И КОДОВ ОШИБОК

Так как язык Objective-C базируется на C, в нем доступны все возможности C, в том числе и перечислимые типы (enum). Перечислимые типы широко применяются в системных фреймворках, но разработчики о них часто забывают. Тем не менее это исключительно полезный способ определения именованных констант — например, для кодов ошибок и наборов флагов, которые могут использоваться в комбинациях. Благодаря дополнениям стандарта C++11 в последних версиях системных фреймворков появилась возможность сильной типизации перечислимых типов. Да, стандарт C++11 оказался полезным и для Objective-C!

Перечисление представляет собой не что иное, как способ именования констант. Простой набор перечислимых значений может использоваться для определения состояний, в которых может находиться объект. Например, состояние подключения через сокет может быть представлено следующим перечислением:

```
enum EOCConnectionState {
    EOCConnectionStateDisconnected,
```

```
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
```

Перечисление упрощает чтение кода, поскольку каждое состояние представляется понятным, содержательным именем. Компилятор присваивает каждому элементу перечисления уникальное значение, начиная с 0 и с увеличением на 1 для каждого последующего элемента. Выбор типа зависит от компилятора, но количество битов в нем должно быть по крайней мере достаточным для полного представления перечисления. Для перечисления из предыдущего примера будет достаточно типа `char` (1 байт), так как максимальное значение равно 2.

Впрочем, этот стиль определения перечислений не слишком полезен и требует использования следующего синтаксиса:

```
enum EOCConnectionState state = EOCConnectionStateDisconnected;
```

Было бы намного проще, если бы можно было обойтись без постоянного ввода `enum`, а просто использовать `EOCConnectionState`. Для этого в определение перечисления добавляется ключевое слово `typedef`:

```
enum EOCConnectionState {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
typedef enum EOCConnectionState EOCConnectionState;
```

Это означает, что вместо полной записи `enum EOCConnectionState` может использоваться сокращенная запись `EOCConnectionState`:

```
EOCConnectionState state = EOCConnectionStateDisconnected;
```

В стандарте C++11 в области перечислений были внесены некоторые изменения. В частности, появилась возможность явного указания типа для хранения переменных перечислимого типа. Его преимущество заключается в том, что оно позволяет применять опережающие объявления перечислимых типов. Без указания базового типа опережающее объявление перечислимого типа невозможно, так как компилятор не знает размер базового типа — а следовательно, не знает, сколько памяти нужно выделить для переменной. Для указания типа используется следующий синтаксис:

```
enum EOCConnectionStateConnectionState : NSInteger { /* … */ };
```

Это означает, что значение, представляющее элементы перечисления, гарантированно будет относиться к типу `NSInteger`. При желании для типа можно использовать опережающее объявление:

```
enum EOCConnectionStateConnectionState : NSInteger;
```

Также можно явно указать, каким значением должен представляться тот или иной элемент перечисления, вместо того чтобы позволить компилятору выбрать его за вас. Синтаксис выглядит так:

```
enum EOCConnectionStateConnectionState {
    EOCConnectionStateDisconnected = 1,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
```

В этом примере для `EOCConnectionStateDisconnected` будет использоваться значение 1 вместо 0. Другие значения следуют по порядку с увеличением на 1, как и прежде. Так, `EOCConnectionStateConnected` будет соответствовать значение 3.

Также перечислимые типы могут использоваться для определения наборов флагов, особенно если флаги могут объединяться. Если перечисление будет правильно определено, его элементы могут «объединяться» поразрядным оператором ИЛИ. Например, следующий перечислимый тип из UI-фреймворка iOS используется для определения изменяемых размеров представления:

```
enum UIViewAutoresizing {
    UIViewAutoresizingNone                 = 0,
    UIViewAutoresizingFlexibleLeftMargin   = 1 << 0,
    UIViewAutoresizingFlexibleWidth        = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin  = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin    = 1 << 3,
    UIViewAutoresizingFlexibleHeight       = 1 << 4,
    UIViewAutoresizingFlexibleBottomMargin = 1 << 5,
}
```

Каждый флаг может находиться в установленном или снятом состоянии. Данный синтаксис позволяет это сделать, потому что каждый флаг представляется всего одним битом в значении, представляющем их комбинацию. Флаги объединяются поразрядным оператором ИЛИ: например, `UIViewAutoResizingFlexibleWidth | UIViewAutoresizingFlexibleHeight`. На рис. 1.2 показано двоичное

представление каждого элемента перечисления и комбинация двух элементов.

Чтобы проверить, установлен ли некоторый флаг в наборе, используйте поразрядный оператор И:

```
enum UIVewAutoresizing resizing =
    UIViewAutoresizingFlexibleWidth |
    UIViewAutoresizingFlexibleHeight;
if (resizing & UIViewAutoresizingFlexibleWidth) {
    // Флаг UIViewAutoresizingFlexibleWidth установлен
}
```

UIViewAutoresizingFlexibleLeftMargin	0	0	0	0	0	1
UIViewAutoresizingFlexibleWidth	0	0	0	0	1	0
UIViewAutoresizingFlexibleRightMargin	0	0	0	1	0	0
UIViewAutoresizingFlexibleTopMargin	0	0	1	0	0	0
UIViewAutoresizingFlexibleHeight	0	1	0	0	0	0
UIViewAutoresizingFlexibleBottomMargin	1	0	0	0	0	0
UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight	0	1	0	0	1	0

Рис. 1.2. Двоичное представление флагов и комбинации двух флагов, объединенных поразрядным оператором ИЛИ

Эта возможность широко используется в системных библиотеках. В другом примере из UIKit — UI-фреймворке для iOS — наборы флагов передают системе информацию об ориентациях устройства, поддерживаемых представлением. Для этого используется перечислимый тип с именем `UIInterfaceOrientationMask`, а разработчик реализует метод с именем `supportedInterfaceOrientations` для обозначения поддерживаемых ориентаций:

```
- (NSUInteger)supportedInterfaceOrientations {
    return UIInterfaceOrientationMaskPortrait |
           UIInterfaceOrientationMaskLandscapeLeft;
}
```

В фреймворке Foundation определена пара вспомогательных конструкций, которые упрощают определение перечислимых типов, а также позволяют задать целочисленный тип для хранения значений перечислимого типа. При этом обеспечивается обратная совместимость: если компилятор поддерживает новый стандарт, то используется новый синтаксис, а если нет — старый. Эти вспомогательные конструкции реализованы в форме макросов препроцессора `#define`. Один предназначен для нормальных перечислимых типов, как в примере с `EOCConnectionState`, а другой — для перечислений, определяющих набор флагов, как в примере с `UIViewAutoresizing`. Примеры их использования:

```
typedef NS_ENUM(NSUInteger, EOCConnectionState) {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
typedef NS_OPTIONS(NSUInteger, EOCPermittedDirection) {
    EOCPermittedDirectionUp    = 1 << 0,
    EOCPermittedDirectionDown  = 1 << 1,
    EOCPermittedDirectionLeft  = 1 << 2,
    EOCPermittedDirectionRight = 1 << 3,
};
```

А вот как выглядят определения макросов:

```
#if (__cplusplus && __cplusplus >= 201103L &&
        (__has_extension(cxx_strong_enums) ||
         __has_feature(objc_fixed_enum))
    ) ||
    (!__cplusplus && __has_feature(objc_fixed_enum))
        #define NS_ENUM(_type, _name)
                enum _name : _type _name; enum _name : _type
        #if (__cplusplus)
            #define NS_OPTIONS(_type, _name)
                    _type _name; enum : _type
        #else
            #define NS_OPTIONS(_type, _name)
                    enum _name : _type _name; enum _name : _type
        #endif
#else
```

```
    #define NS_ENUM(_type, _name) _type _name; enum
    #define NS_OPTIONS(_type, _name) _type _name; enum
#endif
```

Разные варианты определения макросов объясняются различиями в сценариях их использования. Сначала макрос проверяет, поддерживает ли компилятор новый стиль перечислений. Логика проверки на первый взгляд выглядит сложно, но в действительности она просто проверяет доступность обновленного синтаксиса. Если новый стиль не поддерживается, перечисления определяются в старом стиле.

Если новый стиль поддерживается, тип NS_ENUM определяется таким образом, что в расширенном виде он выглядит так:

```
typedef enum EOCConnectionState : NSUInteger
EOCConnectionState;
enum EOCConnectionState : NSUInteger {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};
```

Макрос NS_OPTIONS также определяется разными способами в зависимости от того, выполняется компиляция в режиме C++ или нет. Если код компилируется не в режиме C++, то расширение производится так же, как для NS_ENUM. Однако для C++ расширение производится по несколько иным правилам. Почему? При выполнении поразрядной операции ИЛИ с двумя перечислимыми значениями компилятор C++ действует иначе. Как упоминалось ранее, эта операция обычно выполняется с перечислениями, представляющими набор флагов. При объединении двух типов операцией ИЛИ C++ считает, что результат относится к типу, представляемому перечислением: NSUInteger. Он также не позволяет выполнять неявное преобразование к типу перечисления. Чтобы стало понятнее, посмотрим, что произойдет при расширении перечисления EOCPermittedDirection через NS_ENUM:

```
typedef enum EOCPermittedDirection : int EOCPermittedDirection;
enum EOCPermittedDirection : int {
    EOCPermittedDirectionUp    = 1 << 0,
    EOCPermittedDirectionDown  = 1 << 1,
    EOCPermittedDirectionLeft  = 1 << 2,
    EOCPermittedDirectionRight = 1 << 3,
};
```

Допустим, затем мы пытаемся выполнить следующую команду:

```
EOCPermittedDirection permittedDirections =
    EOCPermittedDirectionLeft | EOCPermittedDirectionUp;
```

Если компилятор работает в режиме C++ (а возможно, и в Objective-C++), *это приведет к следующей ошибке:*

```
error: cannot initialize a variable of type
'EOCPermittedDirection' with an rvalue of type 'int'
```

Результат ИЛИ необходимо явно преобразовать к `EOCPermittedDirection`. Для C++ перечисление `NS_OPTIONS` определяется иначе, чтобы это преобразование не было обязательным. По этой причине, если вы собираетесь объединять элементы перечисления операцией ИЛИ, всегда используйте `NS_OPTIONS`; если нет — используйте `NS_ENUM`.

Перечисление может использоваться во многих ситуациях. Наборы флагов и состояния уже были продемонстрированы выше; однако существует много других потенциальных применений. Коды ошибок или статуса также хорошо подходят для перечислений. В отличие от препроцессорных определений или констант, логически связанные коды можно сгруппировать в одно перечисление. Еще один хороший кандидат — стили. Например, если некоторый элемент пользовательского интерфейса может создаваться с разными стилями, перечислимый тип идеально подходит для такой ситуации.

Последняя тонкость, касающаяся перечислений, связана с использованием команды `switch`. Иногда бывает нужно использовать конструкцию следующего вида:

```
typedef NS_ENUM(NSUInteger, EOCConnectionState) {
    EOCConnectionStateDisconnected,
    EOCConnectionStateConnecting,
    EOCConnectionStateConnected,
};

switch (_currentState) {
    EOCConnectionStateDisconnected:
        // Обработка для состояния отключения
        break;
    EOCConnectionStateConnecting:
        // Обработка для состояния установления подключения
        break;
    EOCConnectionStateConnected:
```

```
        // Обработка для подключенного состояния
        break;
}
```

Возникает искушение включить в `switch` секцию `default`. Однако при переключении по значению перечислимого типа, определяющему конечный автомат, лучше обойтись без `default`. Дело в том, что, если в будущем вы добавите новое состояние и забудете обработать его в команде `switch`, компилятор предупредит вас об этом. А поскольку блок `default` обрабатывает новое состояние, предупреждения не будет. Это относится и к другим типам перечислений, определяемых с использованием макроса `NS_ENUM`. Например, при определении стилей элементов пользовательского интерфейса обычно нужно следить за тем, чтобы в команде `switch` обрабатывались все возможные стили.

УЗЕЛКИ НА ПАМЯТЬ

- Используйте перечисления для присваивания удобочитаемых имен значениям, представляющим состояния конечного автомата, флаги, передаваемые методам или кодам ошибок.
- Если перечислимый тип определяет флаги, которые могут передаваться одновременно, определите его значения степенями 2, чтобы несколько значений могли объединяться поразрядным оператором ИЛИ.
- Используйте макросы `NS_ENUM` и `NS_OPTIONS` для определения перечислений с явно заданным типом. Тем самым вы гарантируете, что для представления значений будет использоваться тип, выбранный вами, а не компилятором.
- Не включайте секцию `default` в команды `switch` с выбором по значению перечислимого типа. Это упростит последующие расширения перечисления, потому что компилятор предупредит вас о том, что `switch` обрабатывает не все возможные значения.

Глава 2
ОБЪЕКТЫ, СООБЩЕНИЯ И ИСПОЛНИТЕЛЬНАЯ СРЕДА

Объекты являются основными структурными элементами программирования в объектно-ориентированных языках, таких как Objective-C, и предоставляют средства для хранения и перемещения данных. *Обмен сообщениями* (messaging) — процесс взаимодействия объектов для передачи данных и достижения результатов. Глубокое понимание принципов их работы крайне важно для построения эффективного, простого в сопровождении кода.

Исполнительная среда Objective-C обеспечивает работу кода после запуска приложения. Она предоставляет важнейшие функции, которые делают возможным обмен сообщениями между объектами, а также всю логику создания экземпляров классов. Хорошее понимание того, как эти механизмы работают в сочетании друг с другом, повысит вашу квалификацию разработчика.

6 РАЗБЕРИТЕСЬ, ЧТО ТАКОЕ СВОЙСТВА

Свойства (properties) обеспечивают инкапсуляцию данных, содержащихся в объектах Objective-C. Объекты Objective-C обычно содержат набор переменных экземпляров (полей) для хранения данных, с которыми они работают. Для обращения к переменным экземпляров обычно используются *методы доступа* (accessor methods). Get-метод читает текущее значение переменной, а set-метод присваивает ей новое значение. Эта концепция вошла в стандарт и стала частью Objective-C 2.0 в форме свойств, благодаря чему

разработчик может приказать компилятору автоматически сгенерировать методы доступа. Вместе со свойствами был введен новый «точечный синтаксис», с которым обращения к данным, хранящимся в экземплярах, становится более компактным. Вероятно, вы уже использовали свойства в своей работе, но не знали о некоторых их возможностях. Кроме того, со свойствами связаны некоторые нюансы, о которых тоже следует знать. В подходе 6 изложена суть проблем, для решения которых были введены свойства, а также описаны их ключевые возможности.

В классе, описывающем отдельного человека, может храниться имя этого человека, дата его рождения, адрес и т. д. Переменные экземпляров можно объявить в открытом интерфейсе класса:

```
@interface EOCPerson : NSObject {
@public
    NSString *_firstName;
    NSString *_lastName;
@private
    NSString *_someInternalData;
}
@end
```

Такое решение знакомо программистам с опытом работы на Java и C++, где можно определять область действия (scope) переменных экземпляров. Тем не менее в современном Objective-C это решение используется редко. У такого подхода имеется недостаток: он требует определения строения объекта на стадии компиляции. При каждом обращении к переменной `_firstName` компилятор задает фиксированное смещение в области памяти, занимаемой объектом. Такое решение работает хорошо, пока в объект не будет добавлена другая переменная экземпляра. Допустим, перед `_firstName` вставляется еще одна переменная экземпляра:

```
@interface EOCPerson : NSObject {
@public
    NSDate *_dateOfBirth;
    NSString *_firstName;
    NSString *_lastName;
@private
    NSString *_someInternalData;
}
@end
```

Смещение, которое ранее указывало на `_firstName`, теперь указывает на `_dateOfBirth`. Код, в котором используется фиксированное

смещение, прочитает неверное значение. Для наглядности на рис. 2.1 изображено представление класса в памяти до и после добавления переменной экземпляра `_dateOfBirth` (предполагается, что размер указателя равен 4 байтам).

	Person
+0	_firstName
+4	_lastName
+8	_someInternalData

	Person
+0	_dateOfBirth
+4	_firstName
+8	_lastName
+12	_someInternalData

Рис. 2.1. Представление класса в памяти до и после добавления переменной экземпляра

Если изменение определения класса не сопровождается перекомпиляцией, код, вычисляющий смещение на стадии компиляции, становится неработоспособным. Допустим, в библиотеке существует код, использующий старое определение класса. При компоновке с кодом, использующим новое определение класса, возникает несовместимость на стадии выполнения. Для решения этой проблемы были изобретены разные приемы. В Objective-C переменные экземпляров были реализованы как специальные переменные, содержащие смещения и хранящиеся в объектах классов (см. подход 14). Даже если определение класса изменится, во время выполнения будет использовано правильное смещение. Разработчик даже может добавлять переменные экземпляров прямо на стадии выполнения. Это называется *устойчивостью двоичного интерфейса приложения* (ABI, Application Binary Interface). Среди прочего, ABI определяет соглашения, по которым должен генерироваться код. Кроме того, устойчивость ABI также подразумевает, что переменные экземпляров могут определяться в категориях продолжения классов (см. подход 27) или в реализации. Таким образом, вам уже не приходится объявлять все переменные экземпляров в интерфейсе, а следовательно, раскрывать внутреннюю информацию о реализации в открытом интерфейсе.

Предпочтительнее использовать методы доступа вместо прямого обращения к переменным экземпляров — еще один фактор, направленный на решение этой проблемы. Свойства базируются на переменных экземпляров, но предоставляют удобную абстракцию для работы с ними. Вы можете написать методы доступа самостоятельно, но в стиле Objective-C имена методов доступа строятся по жестко заданной схеме. Это позволило ввести языковую конструкцию для автоматического генерирования методов доступа. Так мы подошли к синтаксису `@property`.

Включение свойств в определение интерфейса объекта предоставляет стандартный механизм обращения к данным, инкапсулированным объектом. Соответственно, свойства можно рассматривать как сокращенное объявление о том, что здесь будут определены методы доступа к переменной заданного типа с заданным именем. Для примера возьмем следующий класс:

```
@interface EOCPerson : NSObject
@property NSString *firstName;
@property NSString *lastName;
@end
```

Для пользователя класса это определение эквивалентно следующему:

```
@interface EOCPerson : NSObject
- (NSString*)firstName;
- (void)setFirstName:(NSString*)firstName;
- (NSString*)lastName;
- (void)setLastName:(NSString*)lastName;
@end
```

Для обращения к свойствам используется точечный синтаксис, аналогичный синтаксису обращения к полям структуры, созданной в стеке, в обычном языке C. Компилятор преобразует точечный синтаксис в вызовы методов доступа, как если бы эти методы вызывались напрямую. Следовательно, нет абсолютно никаких различий между точечным синтаксисом и прямым вызовом методов доступа. Эквивалентность двух вариантов записи продемонстрирована в следующем фрагменте кода:

```
EOCPerson *aPerson = [Person new];

aPerson.firstName = @"Bob"; // Эквивалентно:
[aPerson setFirstName:@"Bob"];
```

```
NSString *lastName = aPerson.lastName; // Эквивалентно:
NSString *lastName = [aPerson lastName];
```

Но преимущества свойств не ограничиваются компактностью записи. Если вы дадите разрешение, компилятор автоматически сгенерирует код этих методов; данный процесс называется *автосинтезом* (autosynthesis). Важно заметить, что эта операция выполняется компилятором во время компиляции, так что код синтезированных методов не будет виден в редакторе. Наряду с генерированием кода компилятор также автоматически добавляет в класс переменную экземпляра нужного типа с именем по схеме «подчеркивание+имя свойства». В приведенном примере будут созданы две переменные экземпляров: `_firstName` и `_lastName`.

Именем переменной экземпляра можно управлять при помощи синтаксиса `@synthesize` в реализации класса:

```
@implementation EOCPerson
@synthesize firstName = _myFirstName;
@synthesize lastName = _myLastName;
@end
```

В этом фрагменте будут созданы переменные экземпляров с именами `_myFirstName` и `_myLastName` вместо имен по умолчанию. Вообще говоря, необходимость в изменении имен переменных экземпляров встречается нечасто; но если вам не нравятся символы подчеркивания в именах переменных экземпляров, задайте им любые имена на свое усмотрение. Тем не менее я рекомендую использовать имена по умолчанию, потому что соблюдение единых соглашений об именах упрощает чтение кода.

Если вы не хотите, чтобы компилятор синтезировал методы доступа за вас, реализуйте методы самостоятельно. Однако если вы реализуете только один метод, компилятор синтезирует другой за вас. Также можно воспользоваться ключевым словом `@dynamic`, которое запрещает компилятору автоматически создавать переменную экземпляра для свойства и методы доступа. Кроме того, при обработке кода, обращающегося к свойству, компилятор игнорирует тот факт, что методы доступа могут оказаться неопределенными, и верит, что они будут доступны во время выполнения. Например, это обстоятельство используется при субклассировании класса `NSManagedObject` из библиотеки CoreData, где методы доступа создаются динамически на стадии выполнения. `NSManagedObject` выбирает этот подход, потому что свойства не обязаны базироваться

на переменных экземпляров. Данные поступают из того источника, который будет использован программой. Пример:

```
@interface EOCPerson : NSManagedObject
@property NSString *firstName;
@property NSString *lastName;
@end

@implementation EOCPerson
@dynamic firstName, lastName;
@end
```

В этом классе не будут синтезированы ни методы доступа, ни переменные экземпляров, а компилятор не будет выдавать предупреждения при попытке обращения к свойствам.

Атрибуты свойств

Другой аспект свойств, о котором следует знать, — атрибуты, которые могут использоваться для управления методами доступа, сгенерированными компилятором. Пример с использованием трех атрибутов выглядит так:

```
@property (nonatomic, readwrite, copy) NSString *firstName;
```

Атрибуты делятся на четыре категории.

Атомарность

По умолчанию синтезированные методы доступа включают блокировку (locking), обеспечивающую атомарность выполняемых операций. Если указать атрибут `nonatomic`, то блокировка не используется. Несмотря на отсутствие атрибута `atomic` (он предполагается при отсутствии атрибута `nonatomic`), он может быть применен без ошибок компиляции на случай, если вы хотите явно выразить свои намерения. Если вы определяете методы доступа самостоятельно, укажите нужную атомарность.

Чтение/запись

- `readwrite` — доступны как get-, так и set-метод. Если свойство синтезируется, то компилятор генерирует оба метода.
- `readonly` — доступен только get-метод, а компилятор генерирует его только при синтезировании свойства. Используйте

этот режим, если вы хотите предоставить внешний доступ к свойству «только для чтения», переобъявляя его для чтения/записи в категории продолжения класса. За дополнительной информацией обращайтесь к подходу 27.

Семантика управления памятью

Свойства инкапсулируют данные, а эти данные должны иметь четкую семантику принадлежности. Впрочем, это влияет только на set-метод. Например, должен ли set-метод увеличить счетчик для нового значения или просто присвоить его базовой переменной экземпляра? Когда компилятор синтезирует методы доступа, он использует эти атрибуты для определения генерируемого кода. Если вы пишете собственные методы доступа, придерживайтесь значения этого атрибута.

- `assign` — set-метод использует простую операцию присваивания для скалярных типов (таких, как `CGFloat` или `NSInteger`).
- `strong` — означает, что свойство определяет отношения принадлежности. При присваивании нового значения оно сначала удерживается в памяти вызовом `retain`, затем старое значение освобождается, и после этого происходит присваивание.
- `weak` — означает, что свойство определяет отношение без принадлежности. При присваивании нового значения не происходит ни его удержания вызовом `retain`, ни освобождения старого значения. Здесь происходит то же, что и в режиме `assign`, но при уничтожении объекта, на который указывает свойство, значению присваивается `nil`.
- `unsafe_unretained` — операция имеет ту же семантику, что и `assign`, но используется для объектных типов для обозначения отношений без принадлежности (без `retain`), которым не присваивается `nil` при уничтожении целевого объекта (в отличие от `weak`).
- `copy` — используется для отношений принадлежности, как и `strong`; но вместо удержания используется копирование значения. Часто используется с типом `NSString*` для обеспечения инкапсуляции, поскольку значение, переданное set-методу, может быть экземпляром субкласса `NSMutableString`. Если это изменяемый вариант, значение может быть изменено после задания свойства, причем объект об этом знать не будет. По этой причине создается неизменяемая копия, которая гарантирует, что строка не может измениться во время нахождения

в объекте. Для любого объекта, который может измениться, следует создавать копию.

Имена методов

Для управления именами методов доступа используются следующие атрибуты:

- `getter=<имя>` — задает имя get-метода. Этот метод обычно используется для логических свойств, для которых get-методу присваивается имя, начинающееся с префикса `is`. Например, для класса `UISwitch` свойство для получения информации о состоянии переключателя определяется следующим образом:

```
@property (nonatomic, getter=isOn) BOOL on;
```

- `setter=<имя>` — задает имя set-метода; используется редко.

При помощи этих атрибутов можно довольно точно управлять синтезированными методами доступа. Однако следует заметить, что, если вы реализуете собственные методы доступа, вы должны самостоятельно обеспечить их соответствие заданным атрибутам. Например, свойство, объявленное с атрибутом `copy`, должно обеспечить копирование объекта в set-методе. В противном случае пользователи свойства будут руководствоваться неверными предположениями, а несоблюдение контракта может привести к внесению ошибок.

Соблюдение семантики, заложенной в определение свойства, играет важную роль даже в других методах, которые могут задавать это свойство. Для примера возьмем расширение класса `EOCPerson`. Семантика управления памятью для свойств в нем объявлена с атрибутом `copy`, потому что значение может изменяться. Также добавляется инициализатор, который задает исходные значения имени и фамилии (`firstName` и `lastName`):

```
@interface EOCPerson : NSManagedObject

@property (copy) NSString *firstName;
@property (copy) NSString *lastName;

  (id)initWithFirstName:(NSString*)firstName
                lastName:(NSString*)lastName;

@end
```

В реализации пользовательского инициализатора важно придерживаться семантики копирования, заложенной в определениях свойств. Дело в том, что определения свойств документируют контракт класса с задаваемыми значениями. Следовательно, реализация инициализатора должна выглядеть так:

```
- (id)initWithFirstName:(NSString*)firstName
                 lastName:(NSString*)lastName
{
    if ((self = [super init])) {
        _firstName = [firstName copy];
        _lastName = [lastName copy];
    }
    return self;
}
```

Но почему бы просто не использовать set-методы свойств, которые всегда гарантируют использование правильной семантики? Никогда не используйте методы доступа в методе `init` (или `dealloc`) по причинам, изложенным в подходе 7.

Если вы уже читали подход 18, то знаете, что объект следует делать по возможности неизменяемым. Применительно к `EOCPerson` это означает, что оба свойства должны быть сделаны доступными только для чтения. Инициализатор задает значения, после чего они не могут изменяться. В этой ситуации важно объявить, какая семантика управления памятью будет использоваться для значений. Определения свойств в итоге будут выглядеть так:

```
@property (copy, readonly) NSString *firstName;
@property (copy, readonly) NSString *lastName;
```

И хотя set-методы не создаются, поскольку свойства объявлены с атрибутом `readonly`, важно документировать семантику при выполнении инициализатора. Без этого пользователь класса не может рассчитывать на то, что инициализатор создаст копию, и может сам создать лишнюю копию перед вызовом инициализатора. Лишнее копирование снижает эффективность кода.

Возможно, вас интересует, чем отличаются атомарные и неатомарные методы доступа. Как упоминалось ранее, атомарные методы доступа используют блокировки для обеспечения атомарности. Это означает, что если два программных потока читают и записывают одно свойство, то в любой момент времени значение свойства будет действительным. Без блокировок возможна ситуация, когда значение свойства читается в одном потоке, пока другой поток находится

на середине записи. При таком совпадении прочитанное значение свойства может оказаться недействительным.

Занимаясь разработкой для iOS, вы заметите, что все свойства объявляются с атрибутом `nonatomic`. Дело в том, что исторически блокировка в iOS создавала такие дополнительные затраты, что это порождало проблемы с быстродействием. Обычно атомарность все равно не обязательна, потому что она не гарантирует потоковой безопасности, которая обычно требует более глубокой блокировки. Например, даже с атомарностью один поток может несколько раз подряд прочитать свойство и получить разные значения, если другой поток в то же время осуществляет запись. По этой причине в iOS обычно используются неатомарные свойства. Но в Mac OS X атомарные обращения к свойствам, как правило, не становятся критическим фактором быстродействия.

7 ИСПОЛЬЗУЙТЕ ПРЯМОЕ ОБРАЩЕНИЕ К ПЕРЕМЕННЫМ ЭКЗЕМПЛЯРОВ ПРИ ВНУТРЕННИХ ОПЕРАЦИЯХ

Свойства всегда должны использоваться для внешних обращений к внутренним переменным экземпляров объекта, но способ внутренних обращений к переменным стал темой бурных обсуждений в сообществе Objective-C. Одни считают, что обращения к переменным экземпляров всегда должны осуществляться через свойства, другие рекомендуют всегда обращаться к переменным экземпляров напрямую, а третьи предпочитают смешанные решения. Я настоятельно рекомендую всегда читать переменные экземпляров напрямую, но задавать их с использованием свойств — с несколькими оговорками.

Возьмем следующий класс:

```
@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
// Вспомогательные методы для firstName + " " + lastName:
- (NSString*)fullName;
- (void)setFullName:(NSString*)fullName;
@end
```

Вспомогательные методы `fullName` и `setFullName:` могут быть реализованы следующим образом:

```
- (NSString*)fullName {
    return [NSString stringWithFormat:@"%@ %@",
            self.firstName, self.lastName];
}

/** Следующая реализация предполагает, что все полные имена
 *  состоят ровно из 2 частей. Метод можно переписать для
поддержки
 *  более экзотических имен.
 */
- (void)setFullName:(NSString*)fullName {
    NSArray *components =
        [fullName componentsSeparatedByString:@" "];
    self.firstName = [components objectAtIndex:0];
    self.lastName = [components objectAtIndex:1];
}
```

И в get-, и в set-методе мы обращаемся к переменным экземпляров через методы доступа с использованием точечного синтаксиса. Теперь предположим, что методы переписаны для прямого обращения к переменным экземпляров:

```
- (NSString*)fullName {
    return [NSString stringWithFormat:@"%@ %@",
            _firstName, _lastName];
}

- (void)setFullName:(NSString*)fullName {
    NSArray *components =
        [fullName componentsSeparatedByString:@" "];
    _firstName = [components objectAtIndex:0];
    _lastName = [components objectAtIndex:1];
}
```

Между этими стилями существует ряд различий.

- Безусловно, прямой доступ к переменным экземпляров будет быстрее, поскольку в нем не будет задействована диспетчеризация методов Objective-C (*см. подход 11*). Компилятор генерирует код с прямыми обращениями к памяти, в которой хранятся переменные экземпляров объекта.
- Прямые обращения работают в обход семантики управления памятью, определяемой set-методом свойства. Например, если свойство объявляется с атрибутом copy, прямое присваивание переменной экземпляра не приведет к созданию копии. Новое значение будет захвачено, а старое освобождено.

- Оповещения KVO (Key-Value Observing) не будут инициироваться при прямых обращениях к переменным экземпляров. Это может создать проблемы в зависимости от желаемого поведения объектов.
- Обращения через свойства упрощают отладку проблем, связанных со свойствами, поскольку вы можете добавить точку прерывания в get- и/или set-метод для определения, кто и когда может обращаться к свойствам.

Хорошее компромиссное решение — запись в переменные экземпляров через set-метод и чтение посредством прямого доступа. Его преимуществом является быстрое чтение с сохранением контроля над записью. Самый важный аргумент в пользу записи через set-метод заключается в том, что она сохраняет семантику управления памятью. Впрочем, и у этого решения есть свои нюансы, на которые следует обращать внимание.

Первый нюанс связан с заданием значений в методе-инициализаторе. Здесь всегда следует использовать прямой доступ к переменным экземпляров, потому что set-метод может переопределяться субклассами. Допустим, у `EOCPerson` имеется субкласс `EOCSmithPerson`, предназначенный для хранения информации исключительно о людях с фамилией «Smith». Этот субкласс может переопределить set-метод для `lastName` следующим образом:

```
- (void)setLastName:(NSString*)lastName {
    if (![lastName isEqualToString:@"Smith"]) {
        [NSException raise:NSInvalidArgumentException
                    format:@"Last name must be Smith"];
    }
    self.lastName = lastname;
}
```

Базовый класс `EOCPerson` может присвоить фамилии пустую строку в своем инициализаторе по умолчанию. Если бы он сделал это в set-методе, то был бы вызван set-метод субкласса, что привело бы к выдаче исключения. Однако в некоторых ситуациях необходимо использовать set-метод в инициализаторе — например, при объявлении переменной экземпляра в суперклассе; обратиться к переменной экземпляра напрямую все равно нельзя, поэтому приходится использовать set-метод.

Другой нюанс связан с использованием отложенной инициализации. В этом случае приходится обращаться через get-метод; в противном случае может оказаться, что у переменной экземпляра не

будет возможности выполнить инициализацию. Например, у класса `EOCPerson` может быть свойство для доступа к сложному объекту, описывающему строение мозга конкретного человека. Если обращения к этому свойству относительно редки и сопряжены с высокими затратами, свойство может инициализироваться позднее в get-методе:

```
- (EOCBrain*)brain {
    if (!_brain) {
        _brain = [Brain new];
    }
    return _brain;
}
```

Если мы обратимся к переменной экземпляра напрямую, а get-метод еще не был вызван, может оказаться, что описание мозга еще не создано. Следовательно, для всех обращений к свойству `brain` должен вызываться метод доступа.

УЗЕЛКИ НА ПАМЯТЬ

- Во внутренних операциях рекомендуется использовать прямое чтение данных через переменные экземпляров и прямую запись данных через свойства.
- В инициализаторах и `dealloc` всегда используйте прямое чтение и запись данных через переменные экземпляров.
- Чтение данных через свойства иногда оказывается необходимым при отложенной инициализации данных.

РАЗБЕРИТЕСЬ, ЧТО ТАКОЕ РАВЕНСТВО ОБЪЕКТОВ

Возможность проверки равенства объектов чрезвычайно полезна. Однако простой оператор == обычно делает не то, что нужно, потому что он сравнивает указатели, а не объекты, на которые они указывают. Вместо него для проверки равенства двух объектов следует использовать метод `isEqual:`, объявленный в протоколе `NSObject`. Как правило, два объекта разных классов всегда считаются неравными. Некоторые объекты также предоставляют специальные

методы проверки равенства, которые могут использоваться, если два проверяемых объекта заведомо принадлежат к одному классу. Для примера возьмем следующий фрагмент:

```
NSString *foo = @"Badger 123";
NSString *bar = [NSString stringWithFormat:@"Badger %i", 123];
BOOL equalA = (foo == bar); //< equalA = NO
BOOL equalB = [foo isEqual:bar]; //< equalB = YES
BOOL equalC = [foo isEqualToString:bar]; //< equalC = YES
```

Здесь видны различия между == и методами проверки равенства. `NSString` — пример класса, реализующего собственный метод проверки равенства; этот метод называется `isEqualToString:`. Объект, передаваемый методу, также должен относиться к типу `NSString`; в противном случае результат проверки не определен. Этот метод спроектирован так, что он работает быстрее `isEqual:`, которому приходится выполнять дополнительную работу, потому что он не знает класс сравниваемых объектов.

В основе проверки равенства из протокола `NSObject` лежат два объекта:

```
- (BOOL)isEqual:(id)object;
- (NSUInteger)hash;
```

Реализация по умолчанию из класса `NSObject` считает, что два объекта равны только в том случае, если равны значения их указателей. Чтобы понять, как переопределить эти методы для ваших объектов, важно понимать их контракт. Любые два объекта, равные по результату вызова `isEqual:`, должны возвращать одинаковые значения при вызове метода `hash`. Однако два объекта, для которых `hash` возвращает одинаковые значения, не обязаны быть равными согласно вызову метода `isEqual:`.

Для примера возьмем следующий класс:

```
@interface EOCPerson : NSObject
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, assign) NSUInteger age;
@end
```

Два объекта `EOCPerson` считаются равными только в случае равенства всех полей. Таким образом, метод `isEqual:` должен выглядеть так:

```
- (BOOL)isEqual:(id)object {
    if (self == object) return YES;
```

```
    if ([self class] != [object class]) return NO;

    EOCPerson *otherPerson = (EOCPerson*)object;
    if (![_firstName isEqualToString:otherPerson.firstName])
        return NO;
    if (![_lastName isEqualToString:otherPerson.lastName])
        return NO;
    if (_age != otherPerson.age)
        return NO;
    return YES;
}
```

Сначала объект проверяется на равенство указателей с `self`. Если указатели равны, то объекты должны быть равны, потому что это один и тот же объект! Затем сравниваются классы двух объектов. Если классы не равны, то и объекты не могут быть равными — в конце концов, объект `EOCPerson` не может быть равен объекту `EOCDog`. Конечно, может оказаться, что экземпляр `EOCPerson` равен экземпляру своего субкласса: например, `EOCSmithPerson`. В этом проявляется стандартная проблема иерархий наследования с равенством, которую необходимо учитывать при реализации методов `isEqual:`. И наконец, проверяются на равенство все свойства объекта. Если какие-либо свойства не равны, то два объекта считаются неравными; в противном случае они равны.

Остается метод `hash`. Вспомните, что по контракту равные объекты должны возвращать одинаковые хеш-коды, но объекты с одинаковыми хеш-кодами не обязаны быть равными. Следовательно, при переопределении `isEqual:` необходимо переопределить и `hash`. Следующая реализация `hash` вполне допустима:

```
- (NSUInteger)hash {
    return 1337;
}
```

Однако такое решение может вызвать проблемы производительности при включении объектов в коллекцию, потому что хеш-код используется как индекс в хеш-таблицах коллекций. Реализация множества может использовать хеш-код для распределения объектов по разным массивам. Затем при добавлении объекта в множество происходит перебор массива, соответствующего его хеш-коду, для проверки наличия в массиве равных объектов. Если равные объекты будут найдены, значит объект уже присутствует в множестве. Таким образом, если все объекты будут возвращать одинаковые хеш-коды, при добавлении в множество миллиона объектов каждое последу-

ющее добавление в множество будет приводить к сканированию всего миллиона объектов.

Другая реализация метода hash может выглядеть так:

```
- (NSUInteger)hash {
    NSString *stringToHash =
        [NSString stringWithFormat:@"%@:%@:%i",
            _firstName, _lastName, _age];
    return [stringToHash hash];
}
```

На этот раз алгоритм метода hash класса NSString базируется на создании строки и возвращении ее хеш-кода. Такая реализация соответствует контракту, поскольку два равных объекта EOCPerson всегда будут возвращать одинаковые хеш-коды. Недостаток такого подхода заключается в том, что он значительно уступает по скорости возвращению одного значения из-за затрат на создание строки. Это снова может вызвать проблемы с производительностью при добавлении объектов в коллекцию, поскольку хеш приходится вычислять для каждого добавляемого объекта.

Третье и последнее решение основано на создании хеш-кода по следующему принципу:

```
- (NSUInteger)hash {
    NSUInteger firstNameHash = [_firstName hash];
    NSUInteger lastNameHash = [_lastName hash];
    NSUInteger ageHash = _age;
    return firstNameHash ^ lastNameHash ^ ageHash;
}
```

Это решение занимает промежуточное положение между эффективностью и созданием хотя бы какого-то диапазона хеш-кодов. Конечно, при создании хеш-кодов по такому алгоритму неизбежны коллизии, но по крайней мере с ним возможны множественные возвращаемые значения. Выбирая компромисс между частотой коллизий и вычислительной мощностью метода hash, поэкспериментируйте и посмотрите, какой вариант лучше подходит для вашего объекта.

СПЕЦИАЛИЗИРОВАННЫЕ МЕТОДЫ ПРОВЕРКИ РАВЕНСТВА

Кроме упоминавшегося класса NSString, специализированные методы проверки равенства также предоставляются классами NSArray (isEqualToArray:) и NSDictionary (isEqualToDictionary:). Оба метода выдают исключение, если сравниваемый объект не является

массивом или словарем соответственно. Objective-C не имеет сильной проверки типов во время компиляции, поэтому разработчик легко может передать объект неправильного типа. Следовательно, необходимо следить за тем, чтобы передаваемый объект действительно относился к правильному типу.

Одна из причин для создания специализированного метода проверки равенства — ее частое выполнение (когда отказ от проверки типов приводит к значительной экономии ресурсов). Также собственные методы проверки равенства могут определяться по чисто косметическим соображениям: если вы считаете, что такой вызов лучше смотрится и читается; вероятно, отчасти поэтому был создан метод `isEqualToString:` в `NSString`.

Создавая специализированный метод проверки равенства, также переопределите метод `isEqual:` и осуществите сквозную передачу управления, если класс сравниваемого объекта совпадает с классом получателя. В противном случае обычно применяется передача управления реализации суперкласса. Например, реализация класса `EOCPerson` может выглядеть так:

```
- (BOOL)isEqualToPerson:(EOCPerson*)otherPerson {
    if (self == object) return YES;

    if (![_firstName isEqualToString:otherPerson.firstName])
        return NO;
    if (![_lastName isEqualToString:otherPerson.lastName])
        return NO;
    if (_age != otherPerson.age)
        return NO;
    return YES;
}

- (BOOL)isEqual:(id)object {
    if ([self class] == [object class]) {
        return [self isEqualToPerson:(EOCPerson*)object];
    } else {
        return [super isEqual:object];
    }
}
```

ГЛУБОКОЕ И ПОВЕРХНОСТНОЕ РАВЕНСТВО

При создании метода проверки равенства необходимо решить, проверять ли весь объект или ограничиться несколькими полями. `NSArray` проверяет, содержат ли два сравниваемых массива одина-

ковое количество объектов, и если они совпадают — вызывает для каждого объекта `isEqual:`. Если все объекты равны, то и массивы считаются равными. Такая реализация называется *глубоким равенством* (deep equality). Однако в некоторых ситуациях равенство определяется совпадением подмножества данных и проверять все данные до последнего бита не обязательно.

Например, для класса `EOCPerson` экземпляры, загруженные из базы данных, могут содержать дополнительное свойство — уникальный идентификатор, используемый в качестве первичного ключа базы данных:

```
@property NSUInteger identifier;
```

В такой ситуации можно ограничиться проверкой совпадения идентификаторов, особенно если для внешнего пользователя свойства объявлены с атрибутом `readonly`. Если два объекта содержат одинаковые идентификаторы, значит, они представляют один объект, а следовательно, должны считаться равными. Это избавит разработчика от необходимости проверять все данные, содержащиеся в объектах `EOCPerson`.

Итак, необходимость проверки всех полей зависит исключительно от проверяемого объекта. Никто, кроме вас, не может знать, что должно пониматься под равенством двух экземпляров вашего класса.

Равенство изменяемых классов в контейнерах

Есть еще одна важная ситуация, которую необходимо учитывать: размещение изменяемых классов в контейнерах. После того как объект будет добавлен в коллекцию, его хеш-код не должен изменяться. Ранее я объяснял, как объекты распределяются по гнездам в зависимости от хеш-кода. Если хеш изменится после распределения, объект окажется в неправильном гнезде. Чтобы решить эту проблему, следует либо проследить за тем, чтобы хеш-код не зависел от изменяемых частей объекта, либо просто отказаться от изменения объектов после их включения в коллекцию. В подходе 18 я объясняю, почему объекты следует делать неизменяемыми; это отличный пример такого рода.

Проблему можно наглядно продемонстрировать на примере с `NSMutableSet` и несколькими объектами `NSMutableArray`. Сначала добавьте в множество один массив:

```
NSMutableSet *set = [NSMutableSet new];
NSMutableArray *arrayA = [@[@1, @2] mutableCopy];
```

```
[set addObject:arrayA];
NSLog(@"set = %@", set);
// Результат: set = {((1,2))}
```

Множество содержит один объект: массив, в котором находятся два объекта. Теперь добавьте массив, содержащий равные объекты в таком же порядке, чтобы массив, уже находящийся в множестве, и новый объект были равны:

```
NSMutableArray *arrayB = [@[@1, @2] mutableCopy];
[set addObject:arrayB];
NSLog(@"set = %@", set);
// Результат: set = {((1,2))}
```

Множество содержит всего один объект, поскольку добавленный объект равен уже хранящемуся объекту. Теперь мы добавляем в множество массив, не равный хранимому массиву:

```
NSMutableArray *arrayC = [@[@1] mutableCopy];
[set addObject:arrayC];
NSLog(@"set = %@", set);
// Результат: set = {((1),(1,2))}
```

Как и ожидалось, множество теперь содержит два массива: исходный и новый, потому что массив `arrayC` не равен массиву из множества. Наконец, мы изменяем массив `arrayC`, чтобы он был равен хранимому массиву:

```
[arrayC addObject:@2];
NSLog(@"set = %@", set);
// Результат: set = {((1,2),(1,2))}
```

Сюрприз — множество теперь содержит два одинаковых массива! Семантика множеств не должна допускать таких ситуаций, но мы не смогли обеспечить эту семантику, потому что один из объектов был изменен уже во время нахождения в множестве. При копировании множества ситуация становится еще более неловкой:

```
NSSet *setB = [set copy];
NSLog(@"setB = %@", setB);
// Результат: setB = {((1,2))}
```

Скопированное множество теперь содержит всего один объект, как если бы оно было создано пустым, а потом в него последовательно добавлялись элементы оригинала. Возможно, именно так и было задумано — а может, и нет. Возможно, вы ожидали, что оригинал будет

скопирован «один в один», с поврежденной внутренней структурой. А может, думали, что все произойдет именно так, как произошло. Оба алгоритма копирования будут действительны; это лишний раз демонстрирует то обстоятельство, что множество было повреждено и при работе с ним ни на что рассчитывать не приходится.

Какой же вывод можно сделать? Тщательно продумайте, что может произойти при модификации объектов, находящихся в коллекции. Это не значит, что такого в принципе не должно быть, но вы должны знать о потенциальных проблемах и учитывать их в своем коде.

УЗЕЛКИ НА ПАМЯТЬ

- Представьте методы `isEqual:` и `hash` для объектов, которые должны проверяться на равенство.
- Равные объекты должны иметь одинаковые хеш-коды, но объекты с одинаковыми хеш-кодами не обязательно равны.
- Определите, что реально необходимо для проверки на равенство, вместо бездумного сравнения всех свойств.
- Пишите методы хеширования, которые работают быстро, но обеспечивают достаточно низкий уровень коллизий.

ИСПОЛЬЗУЙТЕ ПАТТЕРН «ГРУППА КЛАССОВ» И СОКРЫТИЕ ПОДРОБНОСТЕЙ РЕАЛИЗАЦИИ

Группа классов (class cluster) — отличный способ маскировки подробностей реализации за абстрактным базовым классом. Этот паттерн широко распространен в системных фреймворках Objective-C. В качестве примера из UIKit, фреймворка пользовательского интерфейса для iOS, можно привести класс `UIButton`. Чтобы создать кнопку, разработчик вызывает следующий метод:

```
+ (UIButton*)buttonWithType:(UIButtonType)type;
```

Тип возвращаемого объекта зависит от типа переданной кнопки; при этом все классы наследуют от одного базового класса `UIButton`. Пользователя класса `UIButton` не интересуют тип создаваемой кнопки

и подробности реализации относительно того, как кнопка себя прорисовывает. Все, что ему нужно знать, — как создать кнопку; как задать атрибуты (например, надпись) и как добавить приемники для операций касания.

Вообще говоря, эту проблему также можно решить с одним классом, который обрабатывает всю прорисовку кнопок и осуществляет переключение в зависимости от типа:

```
- (void)drawRect:(CGRect)rect {
    if (_type == TypeA) {
        // Прорисовка кнопки TypeA
    } else if (_type == TypeB) {
        // Прорисовка кнопки TypeB
    } /* … */
}
```

Однако совершенно ясно, что этот подход станет очень громоздким, если слишком много методов потребует переключения по типу. Хороший программист на этой стадии может выполнить рефакторинг с созданием субклассов, выполняющих специализированную работу для каждой разновидности кнопок. Однако в этом случае пользователь должен будет знать обо всех различных субклассах. В таких ситуациях паттерн «Группа классов» занимает свое место и предоставляет гибкость множественных субклассов с сохранением стройности интерфейса, для чего субклассы скрываются за абстрактным базовым классом. Вы не создаете экземпляры субклассов, а позволяете базовому классу создать их за вас.

СОЗДАНИЕ ГРУППЫ КЛАССОВ

В качестве примера создания группы классов возьмем класс, представляющий работников. Каждый работник обладает именем, зарплатой и может получить приказ на выполнение своей повседневной работы. Однако что именно происходит в ходе его ежедневной работы — зависит от типа работника. Руководитель организации, в которой трудятся работники, не интересуется тем, какую работу выполняет каждый из его подчиненных, а просто приказывает заняться своим делом.

Для начала нужно определить абстрактный базовый класс:

```
typedef NS_ENUM(NSUInteger, EOCEmployeeType) {
    EOCEmployeeTypeDeveloper,
    EOCEmployeeTypeDesigner,
    EOCEmployeeTypeFinance,
```

```
};

@interface EOCEmployee : NSObject

@property (copy) NSString *name;
@property NSUInteger salary;

// Вспомогательный метод для создания объектов Employee
+ (EOCEmployee*)employeeWithType:(EOCEmployeeType)type;

// Объекты Employee выполняют свою повседневную работу
- (void)doADaysWork;
@end

@implementation EOCEmployee

+ (EOCEmployee*)employeeWithType:(EOCEmployeeType)type {
    switch (type) {
        case EOCEmployeeTypeDeveloper:
            return [EOCEmployeeDeveloper new];
            break;
        case EOCEmployeeTypeDesigner:
            return [EOCEmployeeDesigner new];
            break;
        case EOCEmployeeTypeFinance:
            return [EOCEmployeeFinance new];
            break;
    }
}

- (void)doADaysWork {
    // Subclasses implement this.
}
@end
```

Каждый конкретный субкласс наследует от базового класса, например:

```
@interface EOCEmployeeDeveloper : EOCEmployee
@end

@implementation EOCEmployeeDeveloper

- (void)doADaysWork {
    [self writeCode];
}

@end
```

В этом примере базовый класс реализует метод, объявленный как метод класса, который в зависимости от типа работника создает и выделяет память под экземпляр нужного класса. Паттерн «Фабрика» — один из способов создания группы классов.

К сожалению, в Objective-C не существует языковых средств для указания того, что базовый класс является абстрактным. Вместо этого правила использования класса должны быть указаны в документации. В нашем примере в интерфейсе не определен «семейный» метод `init`; это означает, что экземпляры, возможно, не должны создаваться напрямую. Другой способ предотвратить использование экземпляров базового класса — выдача исключения в методе `doADaysWork` базового класса. Впрочем, это крайняя мера, которая обычно оказывается излишней.

Будьте осторожны при использовании объектов, входящих в группу классов, в ходе интроспекции (см. подход 14). Даже если вы уверены, что создали экземпляр некоторого класса, на самом деле может быть создан экземпляр субкласса. В примере с `Employee` вызов `[employee isMemberOfClass:[EOCEmployee class]]` вроде бы должен вернуть `YES`, но в действительности возвращается не объект `Employee`, поэтому вызов вернет `NO`.

Группы классов в Cocoa

В системных фреймворках существуют многочисленные группы классов. Большинство классов коллекций являются группами классов — как, например, `NSArray` и его изменяемый аналог `NSMutableArray`. Таким образом, фактически существуют два абстрактных базовых класса: один — для изменяемых массивов, другой — для неизменяемых. То есть мы имеем дело с группой классов с двумя открытыми интерфейсами. Неизменяемый класс определяет методы, общие для всех массивов, а изменяемый класс определяет методы, присутствующие только в изменяемых массивах. Наличие группы классов означает простоту совместного использования кода между двумя типами массивов и поддержку создания копий с модификацией изменяемости.

В случае `NSArray` при создании экземпляра при вызове `alloc` в действительности выделяется память для экземпляра другого класса, называемого *массивом-заместителем* (placeholder array). Массив-заместитель затем преобразуется в экземпляр другого класса, являющегося конкретным субклассом `NSArray`. К сожалению, подробное описание этой темы выходит за рамки книги.

То, что NSArray и большинство других классов коллекций, если на то пошло, являются группами классов — важный факт, потому что в противном случае можно было бы написать код следующего вида:

```
id maybeAnArray = /* … */;
if ([maybeAnArray class] == [NSArray class]) {
    // Никогда не выполняется
}
```

Зная, что NSArray является группой классов, вы поймете, почему этот код неправилен, а условие if никогда не будет истинным. Класс, возвращаемый из [maybeAnArray class], никогда не будет классом NSArray, поскольку экземпляры, возвращаемые инициализаторами NSArray, являются экземплярами внутренних типов за общедоступным фасадом группы классов.

Это не значит, что проверить класс экземпляра из группы классов невозможно. Вместо приведенного выше кода следует использовать методы интроспекции, описанные в подходе 14. Вместо проверки равенства объектов классов используйте следующую конструкцию:

```
id maybeAnArray = /* … */;
if ([maybeAnArray isKindOfClass:[NSArray class]]) {
    // Может быть выполнено
}
```

Необходимость добавления конкретной реализации в группу классов встречается часто, но при этом необходима внимательность. В примере Employee добавить новый тип работников невозможно без доступа к исходному коду фабричного метода. В случае с группами классов Cocoa (такими как NSArray) это возможно, но разработчик должен выполнить ряд правил.

- *Субкласс должен наследовать от абстрактного базового класса группы классов.* В случае NSArray этот базовый класс может быть неизменяемым или изменяемым.
- *Субкласс должен определять собственный способ хранения данных.* Эта часть часто вызывает проблемы при субклассировании таких классов, как NSArray. В вашем субклассе должна присутствовать переменная экземпляра для хранения объектов, содержащихся в массиве. На первый взгляд это выглядит противоестественно — наверняка это должен делать сам класс NSArray. Но вспомните, что NSArray представляет собой простую обертку для других скрытых

объектов, обеспечивающую «чистое» определение интерфейса к массивам. Хорошим выбором объекта для хранения экземпляров пользовательского субкласса массива будет сам `NSArray`.

- *Субкласс должен переопределять документированный набор методов суперкласса.* Каждый абстрактный базовый класс содержит набор методов, которые должны быть реализованы субклассом. В случае `NSArray` обязательными для реализации являются методы `count` и `objectAtIndex:`. Реализовывать другие методы (такие, как `lastObject`) не обязательно, потому что они сами используют эти два метода.

Вся конкретная информация о субклассировании групп классов должна быть приведена в документации класса. Всегда читайте ее заранее.

УЗЕЛКИ НА ПАМЯТЬ

- Паттерн «Группа классов» используется для сокрытия подробностей реализации за простым общедоступным фасадом.
- Группы классов часто используются в системных фреймворках.
- Будьте внимательны при субклассировании открытого абстрактного класса группы кластеров. Всегда читайте документацию, если она доступна.

10 ИСПОЛЬЗУЙТЕ АССОЦИИРОВАННЫЕ ОБЪЕКТЫ ДЛЯ ПРИСОЕДИНЕНИЯ ПОЛЬЗОВАТЕЛЬСКИХ ДАННЫХ К СУЩЕСТВУЮЩИМ КЛАССАМ

Иногда с объектом требуется связать дополнительную информацию. Обычно для этого вы применяете субклассирование и используете субкласс вместо класса объекта. Тем не менее такое решение возможно не всегда — иногда экземпляры класса создаются не вами, а какой-то другой стороной, которой нельзя приказать создать экземпляры вашего класса. В таких ситуациях вам пригодится мощный механизм *ассоциированных объектов* Objective-C.

Объекты ассоциируются с другими объектами и идентифицируются по ключу. Им также назначается *политика хранения данных*, определяющая семантику управления памятью для хранимого значения. Политика хранения данных определяется перечислением `objc_AssociationPolicy`, значения которого приведены в таблице 2.1; они эквивалентны соответствующим значениям атрибута `@property` так, как если бы ассоциация была свойством (за дополнительной информацией о свойствах обращайтесь к подходу 6).

Для управления ассоциациями используются следующие методы:

```
void objc_setAssociatedObject(id object, void *key, id value,
objc_AssociationPolicy policy)
```

Создает ассоциацию объекта со значением, с заданным ключом и политикой.

```
id objc_getAssociatedObject(id object, void *key)
```

Получает ассоциированное с объектом значение с заданным ключом.

Таблица 2.1. Типы ассоциаций

ТИП АССОЦИАЦИИ	ЭКВИВАЛЕНТНЫЕ АТРИБУТЫ @PROPERTY
OBJC_ASSOCIATION_ASSIGN	assign
OBJC_ASSOCIATION_RETAIN_NONATOMIC	nonatomic, retain
OBJC_ASSOCIATION_COPY_NONATOMIC	nonatomic, copy
OBJC_ASSOCIATION_RETAIN	retain
OBJC_ASSOCIATION_COPY	copy

С функциональной точки зрения обращение к ассоциированным объектам выглядит так, словно объект принадлежит к классу `NSDictionary`, для которого вызываются `[object setObject:value forKey:key]` и `[object objectForKey:key]`. Однако при этом следует обратить внимание на важное различие: ключ рассматривается исключительно как непрозрачный указатель. Если для словарей ключи считаются равными, когда для них `isEqual:` возвращает `YES`, ключи ассоциированных объектов равны лишь в случае точного совпадения содержащихся в них указателей. По этой причине для ключей часто используются статические глобальные переменные.

ПРИМЕР ИСПОЛЬЗОВАНИЯ АССОЦИИРОВАННЫХ ОБЪЕКТОВ

В разработке для iOS часто используется класс `UIAlertView`, который обеспечивает стандартное представление для вывода сигнальных сообщений для пользователя. Протокол делегата обеспечивает обработку нажатия кнопки для закрытия сигнала; однако делегирование отделяет код создания сигнального окна от кода обработки касания. Это обстоятельство несколько усложняет чтение программы, так как код распределяется между двумя местами. Нормальный вариант использования `UIAlertView` выглядит примерно так:

```
- (void)askUserAQuestion {
    UIAlertView *alert = [[UIAlertView alloc]
                              initWithTitle:@"Question"
                               message:@"What do you want to do?"
                            delegate:self
                   cancelButtonTitle:@"Cancel"
                   otherButtonTitles:@"Continue", nil];
      [alert show];
}

// Метод протокола UIAlertViewDelegate
- (void)alertView:(UIAlertView *)alertView
        clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == 0) {
        [self doCancel];
    } else {
        [self doContinue];
    }
}
```

Код еще больше усложняется, если вам потребуется вывести более одного сигнала окна в одном классе — придется проверять параметр `alertView`, переданный методу делегата, и переключаться в зависимости от его значения. Было бы намного проще, если бы логика выбора действий по нажатию кнопки могла быть определена при создании сигнала. Для этой цели можно воспользоваться ассоциированным объектом. При создании сигнала назначается блок, который затем читается при выполнении метода делегата. Реализация может выглядеть примерно так:

```
#import <objc/runtime.h>

static void *EOCMyAlertViewKey = "EOCMyAlertViewKey";
```

```
- (void)askUserAQuestion {
    UIAlertView *alert = [[UIAlertView alloc]
                              initWithTitle:@"Question"
                               message:@"What do you want to do?"
                                   delegate:self
                         cancelButtonTitle:@"Cancel"
                         otherButtonTitles:@"Continue", nil];

    void (^block)(NSInteger) = ^(NSInteger buttonIndex){
      if (buttonIndex == 0) {
          [self doCancel];
      } else {
          [self doContinue];
      }
    };

    objc_setAssociatedObject(alert,
                             EOCMyAlertViewKey,
                             block,
                             OBJC_ASSOCIATION_COPY);

    [alert show];
}

// Метод протокола UIAlertViewDelegate
- (void)alertView:(UIAlertView*)alertView
        clickedButtonAtIndex:(NSInteger)buttonIndex
{
    void (^block)(NSInteger) =
        objc_getAssociatedObject(alertView, EOCMyAlertViewKey);
    block(buttonIndex);
}
```

При таком подходе код создания сигнального окна и обработки результата находится в одном месте. Благодаря этому он лучше читается, поскольку пользователю не приходится переключаться между двумя частями кода, пытаясь понять, почему используется сигнальное представление. Тем не менее от разработчика требуется определенная осторожность, так как захват блока может легко привести к возникновению циклов удержания (retain cycles). Дополнительная информация об этой проблеме приведена в подходе 40.

Как видите, решение весьма мощное, однако применять его следует только тогда, когда нужный результат не достигается другими способами. Его чрезмерное применение быстро выходит из-под контроля

и усложняет отладку. С ним труднее разобраться в циклах удержания — формального определения отношений между ассоциированными объектами не существует, поскольку семантика управления памятью определяется в момент ассоциации, а не в момент определении интерфейса. Итак, будьте осмотрительны и не применяйте это решение только потому, что можете. Альтернативный способ достижения того же результата с `UIAlertView` — субклассирование с добавлением свойства для хранения блока. Я рекомендую использовать его вместо ассоциированных объектов, если представление будет использоваться многократно.

УЗЕЛКИ НА ПАМЯТЬ

- Ассоциированные объекты предоставляют механизм связывания двух объектов.
- Семантика управления памятью ассоциированных объектов может определяться для имитации отношений с принадлежностью и без.
- Ассоциированные объекты следует использовать только тогда, когда другое решение невозможно, потому что они часто приводят к появлению коварных ошибок.

11 РАЗБЕРИТЕСЬ С OBJC_MSGSEND

Одна из самых типичных операций, выполняемых в Objective-C, — вызов методов объектов. В терминологии Objective-C это называется *передачей сообщения*. Сообщения обладают именами, которые называются *селекторами*; они получают аргументы и могут возвращать значение.

Поскольку Objective-C является надмножеством C, для начала следует понять, что при вызове функции в C используется так называемая *статическая привязка* (static binding); это означает, что вызываемая функция известна на стадии компиляции. Для примера возьмем следующий код:

```
#import <stdio.h>
void printHello() {
   printf("Hello, world!\n");
}
```

```
void printGoodbye() {
    printf("Goodbye, world!\n");
}

void doTheThing(int type) {
    if (type == 0) {
        printHello();
    } else {
        printGoodbye();
    }
    return 0;
}
```

Если исключить подстановку кода (inlining), при компиляции методы `printHello` и `printGoodbye` известны, а компилятор генерирует инструкции для прямого вызова функций. Фактически адреса функций жестко программируются в инструкциях. А теперь посмотрим, что будет, если записать программу в следующей форме:

```
#import <stdio.h>
void printHello() {
   printf("Hello, world!\n");
}
void printGoodbye() {
    printf("Goodbye, world!\n");
}

void doTheThing(int type) {
    void (*fnc)();
    if (type == 0) {
        fnc = printHello;
    } else {
        fnc = printGoodbye;
    }
    fnc();
    return 0;
}
```

Здесь используется динамическая привязка, так как вызываемая функция остается неизвестной до момента выполнения. Чем же различаются инструкции, генерируемые компилятором? В первом примере функция вызывается как в команде `if`, так и в секции `else`. Второй пример ограничивается одним вызовом функции, но за это приходится читать адрес вызываемой функции вместо использования жестко запрограммированного адреса.

Динамическая привязка — механизм активизации методов в Objective-C при передаче сообщения объекту. Все методы по сути представляют собой обычные функции C, но решение о том, какой именно метод будет вызван для заданного сообщения, принимается исключительно во время выполнения. Выбор даже может изменяться в процессе выполнения приложения, что делает Objective-C по-настоящему динамическим языком.

Вызов сообщения для объекта выглядит так:

```
id returnValue = [someObject messageName:parameter];
```

В этом примере `someObject` является *получателем* (receiver), а `messageName` — *селектором* (selector). Селектор в сочетании с параметрами называется *сообщением* (message). Встречая сообщение, компилятор преобразует его в стандартный вызов функции C `objc_msgSend`, прототип которой выглядит так:

```
void objc_msgSend(id self, SEL cmd, ...)
```

Эта функция получает два или более параметров. В первом параметре передается получатель, во втором — селектор (`SEL` — тип селектора), а в остальных параметрах содержатся параметры сообщения в порядке их следования. Селектором называется имя для обозначения метода. В предыдущем примере сообщение преобразуется в следующий фрагмент:

```
id returnValue = objc_msgSend(someObject,
                              @selector(messageName:),
                              parameter);
```

Функция `objc_msgSend` вызывает метод, выбираемый в зависимости от типа получателя и селектора. Для этого она просматривает список методов, реализованных классом получателя. При обнаружении метода, соответствующего имени селектора, происходит переход к его реализации. В противном случае для поиска метода функция поднимается вверх по иерархии наследования. Если подходящий метод так и не обнаружен, вступает в действие механизм перенаправления сообщений. За дополнительной информацией о перенаправлении сообщений обращайтесь к подходу 12.

Казалось бы, для одного вызова метода приходится выполнять слишком большой объем работы. К счастью, `objc_msgSend` кэширует результат в быстром ассоциативном массиве (по одному для каждого класса), также будущие сообщения для той же комбинации класса

и селектора выполняются очень быстро. Даже этот быстрый путь оказывается медленнее вызова функции со статической привязкой, но после кэширования селектора потеря скорости незначительна. На самом деле диспетчеризация сообщений не является узким местом приложений, но даже если вдруг возникнут проблемы, вы всегда можете написать функцию C и вызвать ее, передав ей все необходимое состояние от объекта Objective-C.

Приведенное описание относится только к определенным сообщениям. Для некоторых особых случаев исполнительная среда Objective-C предоставляет дополнительные функции:

- `objc_msgSend_stret` — для отправки сообщений, возвращающих структуру. Функция может обрабатывать только сообщения, возвращаемый тип которых помещается в регистрах процессора. Если возвращаемый тип в регистрах не помещается, вызывается другая функция для выполнения диспетчеризации. В данном случае эта функция вернет структуру через переменную, выделенную в стеке.
- `objc_msgSend_fpret` — для отправки сообщений, возвращающих вещественные значения. Некоторые архитектуры требуют специальной обработки вещественных регистров между вызовами функций, из-за чего стандартной функции `objc_msgSend` оказывается недостаточно. Эта функция предназначена для обработки несколько необычных ситуаций, возникающих в таких архитектурах, как x86.
- `objc_msgSendSuper` — для отправки сообщений суперклассу (например, `[super message:parameter]`). Также для super-вызовов существуют аналоги `objc_msgSend_stret` и `objc_msgSend_fpret`.

Ранее я уже упоминал о том, что `objc_msgSend` и его аналоги «переходят» к правильной реализации метода после завершения подбора. Каждый метод объекта Objective-C можно представить в виде простой функции C, прототип которой имеет следующую форму:

```
<return_type> Class_selector(id self, SEL _cmd, …)
```

Имя функции выглядит не совсем так; я обозначил его как комбинацию класса и селектора просто для того, чтобы пояснить суть происходящего. Для таких указателей на функции в каждом классе хранится таблица, в качестве ключа для которой используется имя селектора. Именно по ней семейство методов `objc_msgSend` ищет реализацию, к которой следует перейти. Обратите внимание на странное сходство

прототипа с функцией `objc_msgSend`. Это сходство неслучайно — оно упрощает переходы к методам и позволяет эффективно использовать оптимизации хвостовых (завершающих) вызовов.

Оптимизация *хвостового вызова* (tail-call) возможна в том случае, если работа функции завершается вызовом другой функции. Вместо создания нового кадра в стеке компилятор генерирует код перехода к следующей функции. Это можно сделать только тогда, когда последним, что происходит в функции, является вызов другой функции, а возвращаемое значение в вызывающей функции не используется. Такая оптимизация критична для `objc_msgSend`, потому что без нее в трассировке стека каждому методу Objective-C будет предшествовать вызов `objc_msgSend`. Кроме того, ее отсутствие ускорит переполнение стека.

На практике вам не нужно думать обо всем этом при написании кода Objective-C, но понимать суть происходящего «за кулисами» безусловно полезно. Если вы осознаете, что происходит при передаче сообщения, вы можете лучше понять, как выполняется ваш код и почему при отладке в трассировке постоянно встречаются вызовы `objc_msgSend`.

УЗЕЛКИ НА ПАМЯТЬ

- Сообщение состоит из получателя, селектора и параметров. Передача сообщения эквивалентна вызову метода объекта.
- Все сообщения проходят через динамическую систему диспетчеризации, основанную на поиске и последующем выполнении реализации.

12 РАЗБЕРИТЕСЬ С ПЕРЕНАПРАВЛЕНИЕМ СООБЩЕНИЙ

В подходе 11 объясняется, почему так важно понимать механизм отправки сообщений объектам. В подходе 12 рассказано о том, почему необходимо знать, что происходит при отправке объекту сообщения, которое им не опознается.

Класс «понимает» только те сообщения, на понимание которых он был запрограммирован посредством реализации методов. Однако

отправка классу сообщения, которое он не опознает, не приведет к ошибке компиляции, потому что методы могут добавляться в классы на стадии выполнения и компилятор не знает заранее, будет ли существовать реализация метода. При получении объектом сообщения, которое им не поддерживается, запускается процесс *перенаправления* (forwarding) сообщения — процесс, который позволяет разработчику указать, как должно обрабатываться незнакомое сообщение.

Скорее всего, вы уже наблюдали процесс перенаправления сообщений, даже если не сознавали этого. Каждый раз, когда на консоли выводится приведенное ниже сообщение, это означает, что вы отправили объекту непонятное сообщение и вступила в силу реализация перенаправления по умолчанию из `NSObject`:

```
-[__NSCFNumber lowercaseString]: unrecognized selector sent to
instance 0x87
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '-[__NSCFNumber
lowercaseString]: unrecognized selector sent to instance 0x87'
```

Это исключение инициируется методом `NSObject` с именем `doesNotRecognizeSelector`: оно сообщает, что получатель сообщения относится к типу `__NSCFNumber` и не опознает селектор `lowercaseString`. В данном случае это неудивительно (`__NSCFNumber` — внутренний класс, создаваемый при создании `NSNumber`; см. подход 49). В нашем случае путь перенаправления привел к аварийному завершению приложения, но вы можете подключиться к нему в своих классах, чтобы вместо сбоя выполнялась любая нужная логика. Путь перенаправления разделяется на две ветви. Первая ветвь предоставляет классу, к которому принадлежит получатель, возможность динамически добавить метод для неизвестного селектора. Это называется *динамическим разрешением метода* (dynamic method resolution). Вторая ветвь содержит собственно механизм перенаправления. Если исполнительная среда дошла до текущей точки, она знает, что у самого получателя больше нет возможности ответить на селектор, поэтому она предлагает получателю попытаться обработать вызов самостоятельно. Процедура состоит из двух шагов. Сначала исполнительная среда спрашивает, не должен ли другой объект получить сообщение. Если проверка дает положительный результат, исполнительная среда меняет путь доставки и все продолжается обычным образом. Если другого получателя нет, то в действие вступает полный механизм перенаправления, с использованием объекта `NSInvocation` для инкапсуляции подробной

информации о необработанном сообщении, а получателю дается последняя возможность обработать его.

Динамическое разрешение метода

Первым методом, вызываемым при передаче объекту сообщения, которое он не опознает, является метод класса данного объекта:

```
+ (BOOL)resolveInstanceMethod:(SEL)selector
```

Этот метод получает неопознанный селектор и возвращает логический признак, указывающий, был ли добавлен в класс метод экземпляра, способный обработать этот селектор. Таким образом, классу предоставляется вторая возможность добавить реализацию, прежде чем переходить к основной части механизма перенаправления. Аналогичный метод с именем `resolveClassMethod:` вызывается в том случае, если нереализованный метод является методом класса, а не методом экземпляра.

Использование этого способа требует, чтобы реализация метода была доступна и готова для динамического подключения к классу. Он часто используется для реализации свойств `@dynamic` (см. подход 6), как происходит в CoreData при обращении к свойствам объектов `NSManagedObject`, поскольку методы, необходимые для реализации таких свойств, могут быть известны во время компиляции.

Реализация `resolveInstanceMethod:` для использования со свойствами `@dynamic` может выглядеть так:

```
id autoDictionaryGetter(id self, SEL _cmd);
void autoDictionarySetter(id self, SEL _cmd, id value);

+ (BOOL)resolveInstanceMethod:(SEL)selector {
    NSString *selectorString = NSStringFromSelector(selector);
    if ( /* селектор относится к свойству @dynamic */ ) {
        if ([selectorString hasPrefix:@"set"]) {
            class_addMethod(self,
                            selector,
                            (IMP)autoDictionarySetter,
                            "v@:@");
        } else {
            class_addMethod(self,
                            selector,
                            (IMP)autoDictionaryGetter,
                            "@@:");
```

```
        }
        return YES;
    }
    return [super resolveInstanceMethod:selector];
}
```

Метод получает селектор в строковом виде и проверяет его. Если селектор начинается с префикса set, предполагается, что это set-метод; в противном случае он интерпретируется как get-метод. В любом случае в класс для заданного селектора добавляется метод, указывающий на реализацию, определенную в формате обычной функции C. Код, содержащийся в этих функциях C, работает со структурами данных, используемыми классом для хранения этих свойств. Например, в случае CoreData эти методы взаимодействуют с подсистемой баз данных для выполнения соответствующих операций чтения или обновления значений.

АЛЬТЕРНАТИВНЫЙ ПОЛУЧАТЕЛЬ

При второй попытке обработки неизвестного селектора у получателя запрашивается информация о том, доступен ли альтернативный получатель для обработки сообщения. Метод, выполняющий эту операцию, выглядит так:

```
- (id)forwardingTargetForSelector:(SEL)selector
```

Неизвестный селектор передается в параметре; предполагается, что получатель вернет объект, который станет его «заместителем», или `nil`, если найти замену не удалось. Метод в сочетании с композицией может использоваться для реализации некоторых преимуществ множественного наследования. Объект может содержать внутренний набор других объектов; он возвращает эти объекты для селекторов, которые они могут обрабатывать. В результате все выглядит так, словно обработка осуществляется самим объектом.

Учтите, что при использовании этой ветви перенаправления с сообщением ничего нельзя сделать. Если сообщение должно быть изменено до его отправки альтернативному получателю, придется использовать полный механизм перенаправления.

Полный механизм перенаправления

Если алгоритм перенаправления добрался до этой точки, остается только одно: применение полного механизма перенаправления. Все начинается с создания объекта `NSInvocation`, инкапсулирующего

подробную информацию о сообщении, которое осталось необработанным. В объекте содержатся селектор, приемник и аргументы. Объект `NSInvocation` может быть активизирован (invoked); при этом механизм диспетчеризации сообщений приходит в движение и доставляет сообщение его приемнику.

Метод, который пытается выполнить перенаправление, выглядит так:

```
- (void)forwardInvocation:(NSInvocation*)invocation
```

Простейшая реализация меняет приемника и передает ему сообщение. Впрочем, ситуация будет эквивалентна использованию альтернативного получателя, поэтому простая реализация редко используется на практике. Другая, более полезная реализация будет каким-то образом изменять сообщение перед активизацией — например, присоединять к нему другой аргумент или изменять селектор.

Реализация метода всегда должна вызывать реализацию суперкласса для активизаций, которые она не обрабатывает сама. Это означает, что после того, как всем суперклассам в иерархии будет предоставлена возможность обработать сообщение, будет вызвана реализация `NSObject`. Она вызывает `doesNotRecognizeSelector:` для выдачи исключения необработанного селектора.

Общая картина

На рис. 2.2 изображена блок-схема процесса перенаправления.

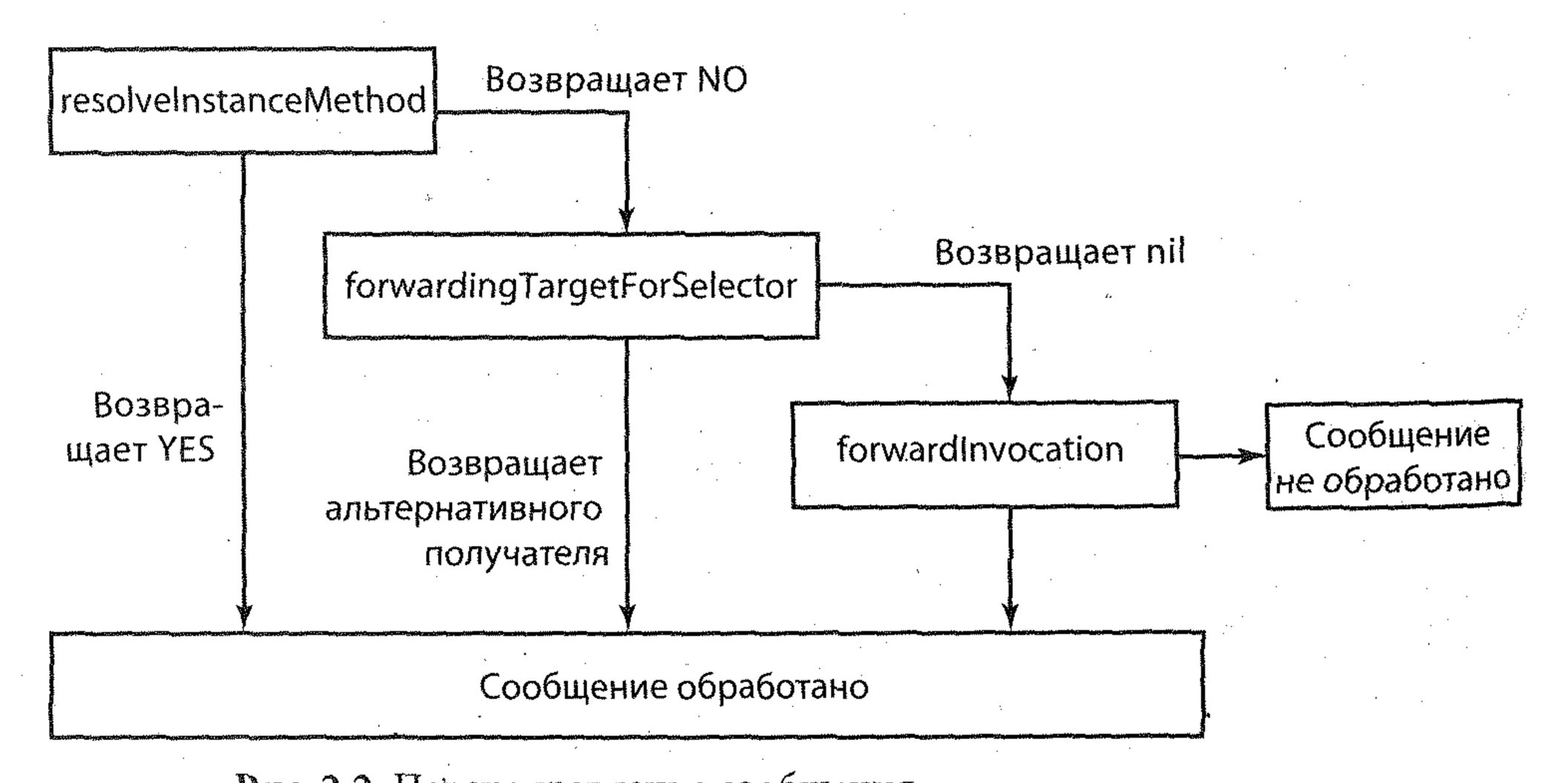


Рис. 2.2. *Перенаправление сообщения*

На каждом шаге получателю предоставляется возможность обработать сообщение. Каждый последующий шаг обходится дороже предыдущего. Лучше всего, если метод будет определен на первом шаге, так как этот метод будет кэширован исполнительной средой и при последующих обращениях к тому же селектору для экземпляра того же класса не придется задействовать перенаправление. Перенаправление сообщения другому получателю на втором шаге представляет собой простую оптимизацию третьего шага для случая, когда объект может предоставить альтернативного получателя. В этом случае изменяется только приемник, а обработка значительно проще последнего шага, на котором приходится создавать объект `NSInvocation` и обрабатывать процедуру его активизации.

Полный пример динамического разрешения метода

Чтобы вы лучше поняли, как работает перенаправление, мы рассмотрим пример, демонстрирующий применение динамического разрешения метода для свойств `@dynamic`. Возьмем объект, который позволяет сохранять в нем любые объекты, — как в словаре, но с обращением через свойства. Идея класса заключается в том, что вы добавляете определение свойства и объявляете его с атрибутом `@dynamic`, а класс как по волшебству обеспечивает сохранение и чтение значения. Звучит невероятно, не правда ли?

Интерфейс класса будет выглядеть так:

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

@interface EOCAutoDictionary : NSObject
@property (nonatomic, strong) NSString *string;
@property (nonatomic, strong) NSNumber *number;
@property (nonatomic, strong) NSDate *date;
@property (nonatomic, strong) id opaqueObject;
@end
```

Для этого примера не так уж важно, что это за свойства. Вообще говоря, я привожу разные типы просто для того, чтобы продемонстрировать всю мощь этого решения. Во внутренней реализации значения свойств будут храниться в словаре, так что начало реализации класса будет выглядеть так (с объявлениями свойств с атрибутом `@dynamic`, чтобы переменные экземпляров и методы доступа не создавались для них автоматически):

```
#import "EOCAutoDictionary.h"
#import <objc/runtime.h>
```

```
@interface EOCAutoDictionary ()
@property (nonatomic, strong) NSMutableDictionary
*backingStore;
@end

@implementation EOCAutoDictionary

@dynamic string, number, date, opaqueObject;
- (id)init {
    if ((self = [super init])) {
        _backingStore = [NSMutableDictionary new];
    }
    return self;
}
```

Мы подходим к самому интересному реализации `resolveInstanceMethod:`:

```
+ (BOOL)resolveInstanceMethod:(SEL)selector {
    NSString *selectorString = NSStringFromSelector(selector);
    if ([selectorString hasPrefix:@"set"]) {
        class_addMethod(self,
                        selector,
                        (IMP)autoDictionarySetter,
                        "v@:@");
    } else {
        class_addMethod(self,
                        selector,
                        (IMP)autoDictionaryGetter,
                        "@@:");
    }
    return YES;
}

@end
```

При первом обращении к свойству экземпляра `EOCAutoDictionary` исполнительная среда не находит соответствующий селектор, потому что он не реализован напрямую и не синтезирован. Например, при записи в свойство `opaqueObject` предшествующий метод будет вызван с селектором `setOpaqueObject:`. Аналогичным образом, при чтении свойства он будет вызван с селектором `opaqueObject`. Метод обнаруживает различия между set- и get-селекторами по префиксу `set`. В каждом случае в класс добавляется метод для заданного селектора, указывающий на функцию с именем `autoDictionarySetter`

или `autoDictionaryGetter` соответственно. При этом используется метод `class_addMethod`, который динамически добавляет в класс метод для заданного селектора с заданием реализации при помощи указателя на функцию. Последним параметром функции является кодированное представление типа реализации. Оно состоит из символов, представляющих возвращаемый тип, за которыми следуют параметры, получаемые функцией.

Функция get-селектора реализуется следующим образом:

```
id autoDictionaryGetter(id self, SEL _cmd) {
    // Получить от объекта хранилище данных
    EOCAutoDictionary *typedSelf = (EOCAutoDictionary*)self;
    NSMutableDictionary *backingStore = typedSelf.backingStore;

    // В качестве ключа используется имя селектора
    NSString *key = NSStringFromSelector(_cmd);

    // Вернуть значение
    return [backingStore objectForKey:key];
}
```

Наконец, set-функция реализуется следующим образом:

```
void autoDictionarySetter(id self, SEL _cmd, id value) {
    // Получить от объекта хранилище данных
    EOCAutoDictionary *typedSelf = (EOCAutoDictionary*)self;
    NSMutableDictionary *backingStore = typedSelf.backingStore;

    /** Примерный вид селектора: "setOpaqueObject:".
     *  Необходимо удалить "set" и ":", а также привести
     *  к нижнему регистру первую букву остатка.
     */
    NSString *selectorString = NSStringFromSelector(_cmd);
    NSMutableString *key = [selectorString mutableCopy];

    // Удалить завершающий символ ':'
    [key deleteCharactersInRange:NSMakeRange(key.length - 1, 1)];

    // Удалить префикс 'set'
    [key deleteCharactersInRange:NSMakeRange(0, 3)];

    // Привести первый символ к нижнему регистру
    NSString *lowercaseFirstChar =
        [[key substringToIndex:1] lowercaseString];
    [key replaceCharactersInRange:NSMakeRange(0, 1)
```

```
                                    withString:lowercaseFirstChar];

        if (value) {
            [backingStore setObject:value forKey:key];
        } else {
            [backingStore removeObjectForKey:key];
        }
    }
```

Работать с EOCAutoDictionary очень просто:

```
EOCAutoDictionary *dict = [EOCAutoDictionary new];
dict.date = [NSDate dateWithTimeIntervalSince1970:475372800];
NSLog(@"dict.date = %@", dict.date);
// Вывод: dict.date = 1985-01-24 00:00:00 +0000
```

Обращения к другим свойствам в словаре выглядят так же, как обращение к свойству date; вы можете создавать новые свойства, добавляя определение @property и объявляя его с атрибутом @dynamic. Аналогичный подход используется в классе CALayer из фреймворка CoreAnimation для iOS. Благодаря ему CALayer представляет собой контейнерный класс для хранения данных по принципу «ключ-значение». CALayer использует эту возможность для добавления пользовательских свойств; хранение значений свойств обеспечивается базовым классом, но определения свойств могут добавляться в субклассе.

УЗЕЛКИ НА ПАМЯТЬ

- Перенаправление сообщений — процесс, через который проходит объект, неспособный отреагировать на селектор.
- Динамическое разрешение метода используется для добавления методов в класс непосредственно при их использовании.
- Объект может объявить, что за обработку некоторых селекторов, не распознаваемых им, отвечает другой объект.
- Полное перенаправление используется только в том случае, если способ обработки селектора, упомянутый в предыдущем пункте, не обнаружен.

13 ИСПОЛЬЗУЙТЕ ЗАМЕНЫ ДЛЯ ОТЛАДКИ НЕПРОЗРАЧНЫХ МЕТОДОВ

Как объяснялось в подходе 11, метод, вызываемый при отправке сообщения объекту в Objective-C, определяется во время выполнения. Резонно предположить, что метод, вызываемый для заданного имени селектора, может быть изменен во время выполнения, — и это действительно так. Эту возможность можно применять с большой пользой, потому что она позволяет изменять функциональность классов, исходный код которых вам недоступен, без субклассирования и переопределения методов. Новая функциональность может использоваться всеми экземплярами класса, а не только экземплярами субкласса с переопределенными методами. Такое решение часто называется *заменой методов* (method swizzling).

Список методов класса содержит список соответствий между именами селекторов и реализациями, по которым система динамической передачи сообщений находит реализацию заданного метода. Реализации хранятся в виде указателей на функции, которые называются IMP (от implementation) и имеют следующий прототип:

```
id (*IMP)(id, SEL, ...)
```

Класс `NSString` реагирует, среди прочего, на селекторы с именами `lowercaseString`, `uppercaseString` и `capitalizedString`. Все селекторы указывают на разные IMP, в результате чего создается таблица наподобие изображенной на рис. 2.3.

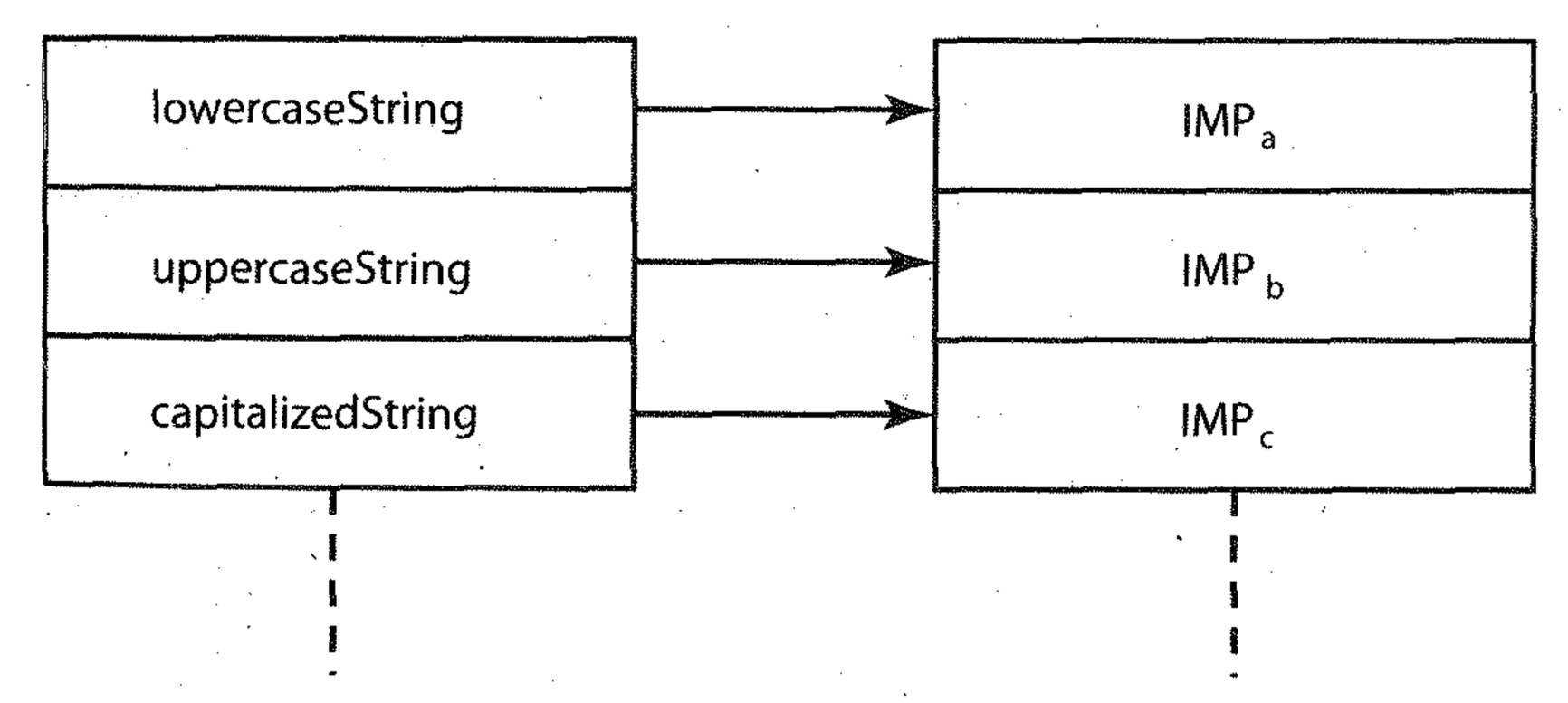


Рис. 2.3. Таблица селекторов NSString

Для работы с таблицей используются функции, предоставляемые исполнительной средой Objective-C. Вы можете добавлять селекторы в список, изменять реализацию, связанную с конкретным селектором,

или менять местами реализации, связанные с двумя селекторами. После выполнения нескольких подобных операций таблица методов класса может выглядеть так, как показано на рис. 2.4.

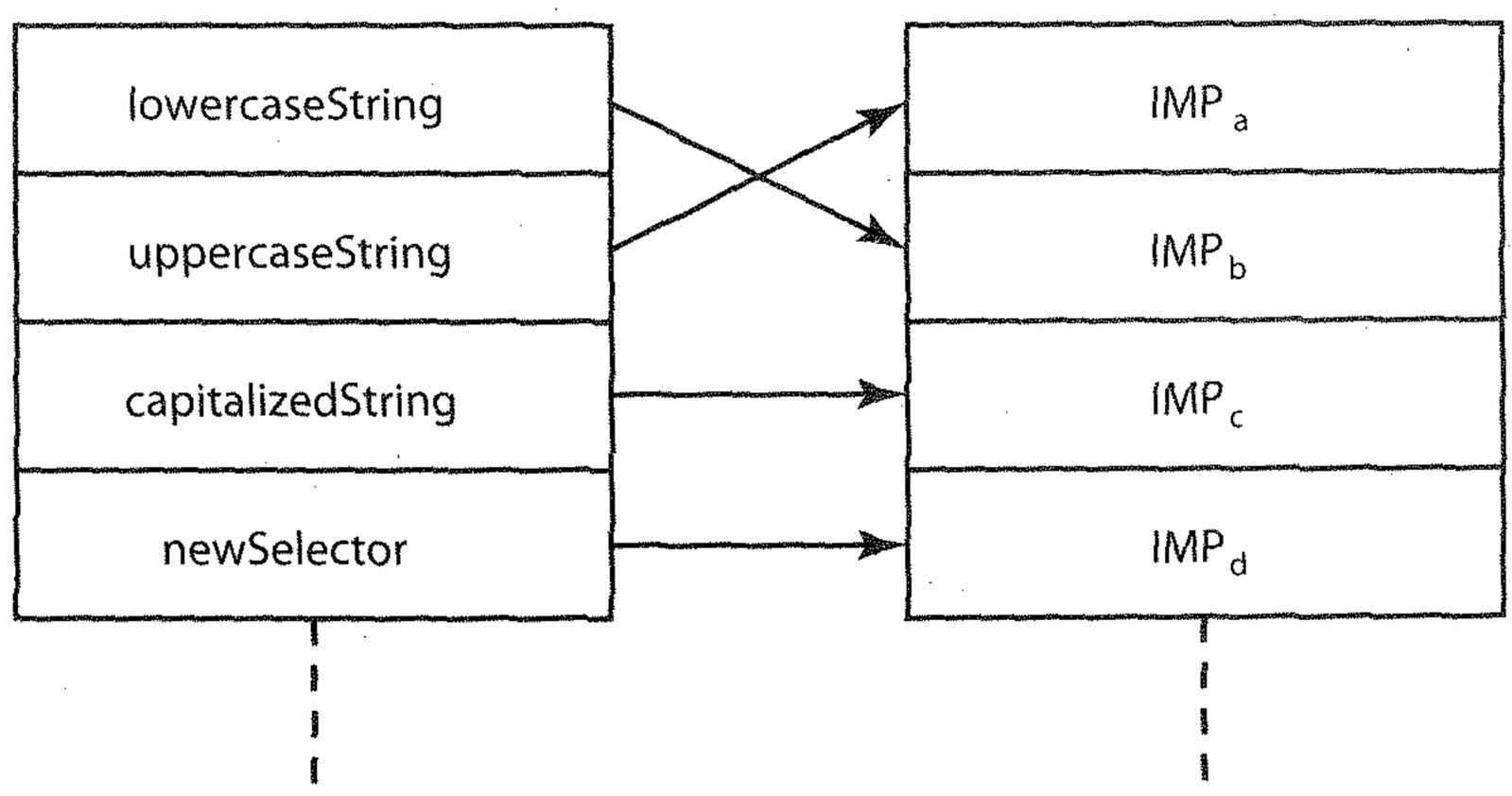


Рис. 2.4. Таблица селекторов NSString после выполнения нескольких операций

В таблицу был добавлен новый селектор с именем `newSelector`, реализация `capitalizedString` была изменена, а реализации `lowercaseString` и `uppercaseString` поменялись местами. Все это было сделано без написания субклассов, а новая структура таблицы методов будет использоваться для каждого экземпляра `NSString` в приложении. Согласитесь, это исключительно мощный механизм!

Название этого подхода относится к переключению реализаций, позволяющему добавлять новую функциональность в методы. Однако прежде чем объяснять, каким образом замена реализаций расширяет функциональность, нужно объяснить, как поменять местами две существующие реализации методов. Для этого используется следующая функция:

```
void method_exchangeImplementations(Method m1, Method m2)
```

В аргументах функции передаются две реализации, которые могут быть получены при помощи следующей функции:

```
Method class_getInstanceMethod(Class aClass, SEL aSelector)
```

Функция получает от класса метод, соответствующий заданному селектору. Например, переключение реализаций `lowercaseString`

и uppercaseString в предыдущем примере выполняется следующим образом:

```
Method originalMethod =
    class_getInstanceMethod([NSString class],
                            @selector(lowercaseString));
Method swappedMethod =
    class_getInstanceMethod([NSString class],
                            @selector(uppercaseString));
method_exchangeImplementations(originalMethod, swappedMethod);
```

Начиная с этого момента при вызове lowercaseString для экземпляра NSString будет вызываться исходная реализация uppercaseString, и наоборот:

```
NSString *string = @"ThIs iS tHe StRiNg";

NSString *lowercaseString = [string lowercaseString];
NSLog(@"lowercaseString = %@", lowercaseString);
// Вывод: lowercaseString = THIS IS THE STRING

NSString *uppercaseString = [string uppercaseString];
NSLog(@"uppercaseString = %@", uppercaseString);
// Вывод: uppercaseString = this is the string
```

Из этого фрагмента становится понятно, как переключать реализации методов, но на практике простое переключение двух реализаций достаточно бессмысленно. В конце концов, реализации uppercaseString и lowercaseString делают то, что они делают, не просто так! Зачем менять их местами? Однако описанный способ может использоваться для добавления новой функциональности в существующие реализации методов. Представьте, что каждый вызов lowercaseString должен регистрироваться в системном журнале. Рассмотренный механизм позволит сделать это. Нужно лишь добавить другой метод, который реализует нужную функциональность, а затем передает управление исходной реализации.

Для добавления нового метода можно воспользоваться категорией:

```
@interface NSString (EOCMyAdditions)
- (NSString*)eoc_myLowercaseString;
@end
```

Метод будет меняться местами с исходным методом lowercaseString. В результате таблица методов будет выглядеть так, как показано на рис. 2.5.

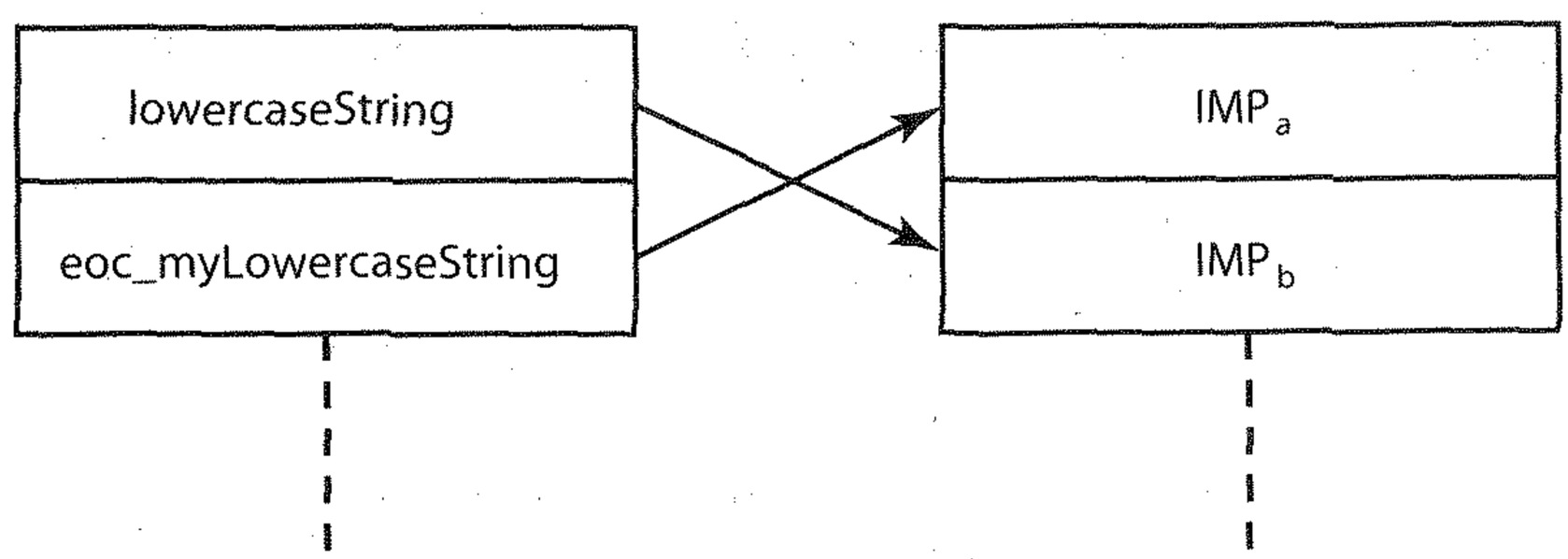


Рис. 2.5. Переключение реализаций lowercaseString и eoc_myLowercaseString

Реализация нового метода выглядит так:

```
@implementation NSString (EOCMyAdditions)
- (NSString*)eoc_myLowercaseString {
    NSString *lowercase = [self eoc_myLowercaseString];
    NSLog(@"%@ => %@", self, lowercase);
    return lowercase;
}
@end
```

На первый взгляд это напоминает рекурсивный вызов, но вспомните, что реализации будут переставлены. Таким образом, во время выполнения при поиске селектора `eoc_myLowercaseString` будет вызвана реализация `lowercaseString`. Наконец, перестановка реализаций методов осуществляется следующим образом:

```
Method originalMethod =
    class_getInstanceMethod([NSString class],
                            @selector(lowercaseString));
Method swappedMethod =
    class_getInstanceMethod([NSString class],
                            @selector(eoc_myLowercaseString));
method_exchangeImplementations(originalMethod, swappedMethod);
```

С этого момента каждый раз, когда для экземпляра `NSString` вызывается метод `lowercaseString`, в журнал будет выводиться трассировочная строка:

```
NSString *string = @"ThIs iS tHe StRiNg";
NSString *lowercaseString = [string lowercaseString];
// Вывод: ThIs iS tHe StRiNg => this is the string
```

Возможность включения отладочного вывода в методы, полностью непрозрачные для вас, может оказаться очень полезной. Впрочем,

обычно она применяется только при отладке. Замена методов для глобального изменения функциональности класса редко когда находит другие применения. Не старайтесь использовать данную возможность только потому, что она существует. Злоупотребления быстро приводят к появлению невразумительного кода, создающего массу проблем в сопровождении.

УЗЕЛКИ НА ПАМЯТЬ

- Реализации методов для заданного селектора класса могут добавляться и переключаться во время выполнения.
- Заменой (swizzling) называется процесс переключения реализаций методов, обычно с целью добавления функциональности к исходной реализации.
- Манипуляции с методами на стадии выполнения обычно применяются только в целях отладки. Не используйте их только потому, что такая возможность существует.

14 РАЗБЕРИТЕСЬ С ОБЪЕКТАМИ КЛАССОВ

Язык Objective-C имеет исключительно динамичную природу. В подходе 11 объясняется, как организовать поиск реализации метода на стадии выполнения, а в подходе 14 — как работает механизм перенаправления, если класс не может немедленно ответить на некоторый селектор. Но как насчет получателя сообщения — самого объекта? Как исполнительная среда узнает, к какому типу относится объект? Привязка объекта к типу осуществляется не во время компиляции, а во время выполнения. Более того, специальный тип `id` может использоваться для обозначения любого типа объектов Objective-C. Впрочем, в общем случае тип следует по возможности указывать, чтобы компилятор не выдавал предупреждения об отправке сообщений, которые, как он полагает, могут оказаться непонятными для получателя. И наоборот, предполагается, что любой объект типа `id` способен отреагировать на все сообщения.

Но, как вы знаете из подхода 12, компилятор не может знать все селекторы, поддерживаемые типом, поскольку они могут вставляться динамически на стадии выполнения. Впрочем, даже при использо-

вании этой механики компилятор ожидает увидеть прототип метода определенным где-то в заголовке, чтобы на основании полной сигнатуры метода сгенерировать правильный код диспетчеризации сообщений.

Анализ типа объекта на стадии выполнения называется *интроспекцией* (introspection); это мощный и полезный механизм, встроенный во фреймворк Foundation как часть протокола `NSObject`, поддерживаемого всеми объектами, производными от общих корневых классов (`NSObject` и `NSProxy`). Как будет показано далее, использование методов интроспекции вместо прямого сравнения классов обладает своими преимуществами. Но прежде чем рассматривать инстроспекцию, стоит поближе познакомиться с тем, что же собой представляет объект Objective-C.

Каждый экземпляр класса Objective-C представляет собой указатель на область памяти. Именно поэтому при объявлении переменной рядом с типом ставится символ *:

```
NSString *pointerVariable = @"Some string";
```

Читатели с опытом программирования на C сразу поймут, о чем идет речь. Для остальных поясню, что переменная `pointerVariable` содержит адрес памяти, а данными, которые хранятся по этому адресу, является сам объект `NSString`. Таким образом, переменная «указывает» на экземпляр `NSString`. Собственно, так устроены все ссылки на объекты Objective-C; если вы попытаетесь выделить память для объекта в стеке, компилятор выдаст сообщение об ошибке:

```
NSString stackVariable = @"Some string";
// ошибка: статическое выделение памяти для интерфейсного типа
невозможно
```

Обобщенный объектный тип `id` уже является указателем сам по себе, поэтому он используется следующим образом:

```
id genericTypedString = @"Some string";
```

Это определение семантически эквивалентно определению переменной типа `NSString*`. Единственное отличие от полного указания заключается в том, что компилятор может помочь и выдать предупреждение о попытке вызова метода, не существующего для экземпляров этого класса.

Структура данных за каждым объектом определяется в заголовках исполнительной среды вместе с определением самого типа `id`:

```
typedef struct objc_object {
    Class isa;
} *id;
```

Таким образом, первым членом каждого объекта является переменная типа `Class`, определяющая класс объекта (имя `isa` происходит от «is a», то есть «является»; например, объект «является» `NSString`). Объект `Class` тоже определяется в заголовках исполнительной среды:

```
typedef struct objc_class *Class;
struct objc_class {
    Class isa;
    Class super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
};
```

Структура содержит метаданные о классе — какие методы реализуются экземплярами класса, какие переменные экземпляров содержатся в экземплярах и т. д. Тот факт, что в первой переменной этой структуры содержится указатель `isa`, означает, что сам объект `Class` тоже является объектом Objective-C. Структура также содержит другую переменную с именем `super_class`, определяющую родителя класса. Тип класса (то есть класс, на который ссылается указатель `isa`) представляет собой другой класс, называемый *метаклассом* (metaclass), который используется для описания метаданных об экземплярах самого класса. Именно здесь определяются методы класса, поскольку они могут рассматриваться как «методы экземпляров» экземпляров класса. Однако существует только один экземпляр класса и только один экземпляр соответствующего метакласса.

На рис. 2.6 изображена иерархия класса с именем `SomeClass`, производного от `NSObject`.

Указатель `super_class` создает иерархию, а указатель `isa` описывает тип экземпляра. Вы можете оперировать с этой иерархией для выполнения интроспективного анализа. В частности, можно узнать, отвечает ли объект на некоторый селектор или поддерживает ли некоторый протокол, а также определить информацию о том, к какой части иерархии классов принадлежит объект.

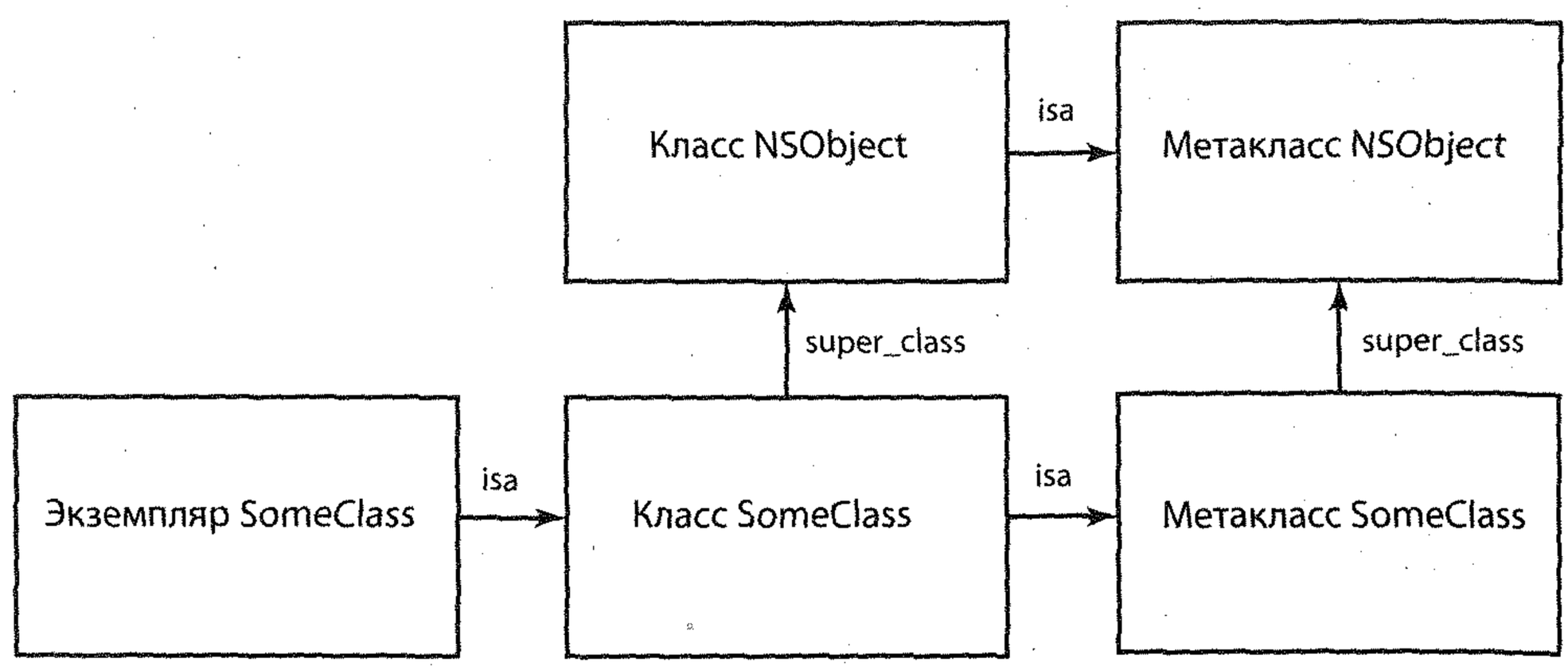


Рис. 2.6. Иерархия классов для экземпляров класса SomeClass, производного от NSObject (включая иерархию метаклассов)

ИЕРАРХИЯ КЛАССОВ

Методы интроспекции могут использоваться для анализа иерархии классов. Например, можно проверить, является ли объект экземпляром некоторого класса, при помощи метода `isMemberOfClass:`, или является ли объект экземпляром некоторого класса или производных от него классов, при помощи метода `isKindOfClass:`. Пример:

```
NSMutableDictionary *dict = [NSMutableDictionary new];
[dict isMemberOfClass:[NSDictionary class]]; ///< NO
[dict isMemberOfClass:[NSMutableDictionary class]]; ///< YES
[dict isKindOfClass:[NSDictionary class]]; ///< YES
[dict isKindOfClass:[NSArray class]]; ///< NO
```

Интроспекция такого рода основана на использовании указателя `isa` для получения класса объекта с последующим проходом по иерархии наследования по указателю `super_class`. С учетом динамической природы объектов эта возможность чрезвычайно полезна. Без интроспекции невозможно полностью определить тип объекта (в отличие от других языков, которые вам, вероятно, известны).

Интроспекция класса объекта особенно полезна с учетом динамической типизации, используемой в Objective-C. Интроспекция часто используется при чтении объектов из коллекций; обычно такие объекты не имеют сильной типизации (то есть объекты, прочитанные из коллекций, обычно относятся к типу `id`). Если потребуется определить фактический тип объекта, можно воспользоваться интроспекцией — допустим, вы хотите сгенерировать

строку из объектов массива, разделенных запятыми, для сохранения в текстовом файле. В такой ситуации можно воспользоваться следующим кодом:

```
- (NSString*)commaSeparatedStringFromObjects:(NSArray*)array {
    NSMutableString *string = [NSMutableString new];
    for (id object in array) {
        if ([object isKindOfClass:[NSString class]]) {
            [string appendFormat:@"%@,", object];
        } else if ([object isKindOfClass:[NSNumber class]]) {
            [string appendFormat:@"%d,", [object intValue]];
        } else if ([object isKindOfClass:[NSData class]]) {
            NSString *base64Encoded = /* base64 encoded data */;
            [string appendFormat:@"%@,", base64Encoded];
        } else {
            // Тип не поддерживается
        }
    }
    return string;
}
```

Также объекты классов можно проверять на равенство. Для этой цели используется оператор == вместо метода `isEqual:`, обычно применяемого при сравнении объектов Objective-C (см. подход 8). Дело в том, что эти классы являются синглетными, то есть в приложении существует только один экземпляр объекта `Class` каждого класса. Таким образом, другой способ проверки того, что объект является «точным» экземпляром класса, выглядит так:

```
id object = /* … */;
if ([object class] == [EOCSomeClass class]) {
    // 'object' является экземпляром EOCSomeClass
}
```

Тем не менее методы интроспекции всегда предпочтительнее прямой проверки равенства объектов классов, потому что методы интроспекции способны учитывать объекты, использующие перенаправление сообщений (см. подход 12). Представьте объект, который перенаправляет все свои селекторы другому объекту. Такой объект называется *заместителем* (proxy); `NSProxy` — корневой класс, предназначенный специально для таких объектов.

Обычно при вызове метода `class` объекты-заместители возвращают класс заместителя (то есть субкласс `NSProxy`) вместо класса замещаемого объекта. Однако при вызове методов интроспекции (таких, как `isKindOfClass:`) они передают сообщение замещаемому

объекту. Результат будет отличен от инспектирования объекта класса, возвращаемого методом `class`, потому что вместо класса замещаемого объекта будет возвращен класс-заместитель.

УЗЕЛКИ НА ПАМЯТЬ

- Иерархия классов моделируется при помощи объектов `Class`. Каждый экземпляр содержит указатель на такой объект, определяющий его тип.
- Если тип объекта неизвестен на стадии компиляции, используйте интроспекцию.
- Всегда старайтесь по возможности использовать методы интроспекции вместо прямого сравнения объектов классов, поскольку объект может реализовать перенаправление сообщений.

Глава 3
ПРОЕКТИРОВАНИЕ ИНТЕРФЕЙСА И API

Когда ваше приложение будет построено, вы, вероятно, захотите сами использовать некоторые его части в будущих проектах. Возможно, вы опубликуете часть кода, чтобы его могли использовать другие. И даже если вы не собираетесь этим заниматься, когда-нибудь код все равно может понадобиться. И тогда вам сильно поможет, если ваши интерфейсы будут написаны в расчете на повторное использование. Это значит, что вы должны задействовать парадигмы, часто встречающиеся в Objective-C, и понимать типичные проблемы.

За последние годы использование кода, написанного другими разработчиками, стало повсеместным явлением — особенно в сообществе открытого кода и в области компонентов, ставших популярными с появлением iOS. Другие пользователи также могут использовать ваш код, поэтому четкость и логичность кода поможет им быстрее и проще интегрировать ваши разработки. И кто знает — возможно, вы напишете библиотеку, которая будет применяться в тысячах приложений!

15 ИСПОЛЬЗУЙТЕ ПРЕФИКСЫ ДЛЯ ПРЕДОТВРАЩЕНИЯ КОНФЛИКТОВ ИМЕН

В отличие от других языков, Objective-C не имеет встроенной поддержки пространств имен. По этой причине в программах легко могут возникнуть конфликты имен, если только вы не примете меры

для их предотвращения. Как правило, из-за конфликтов имен приложения не проходят компоновку с выдачей ошибок о дубликатах символических имен:

```
duplicate symbol _OBJC_METACLASS_$_EOCTheClass in:
    build/something.o
    build/something_else.o
duplicate symbol _OBJC_CLASS_$_EOCTheClass in:
    build/something.o
    build/something_else.o
```

Ошибка возникла из-за того, что символические имена класса и метакласса (см. подход 14) для класса с именем `EOCTheClass` были определены дважды, в двух реализациях `EOCTheClass` в двух разных частях кода приложения — возможно, в двух разных библиотеках, которые вы подключили к своему проекту.

Впрочем, ошибка компоновки — еще не худший вариант. Представьте, что одна из библиотек, содержащих дубликаты, загружается на стадии выполнения. В этом случае динамический загрузчик обнаружит ошибку, что, скорее всего, приведет к аварийному завершению всего приложения.

Проблему можно предотвратить только одним способом: использованием примитивного подобия пространств имен, при котором все имена в программе начинаются с определенного префикса. Выбранный префикс должен быть связан с вашей компанией и/или приложением. Например, если компания называется Effective Widgets, вы можете использовать префикс `EWS` в коде, общем для всех приложений, и префикс `EWB` в коде приложения Effective Browser. Префиксы не исключают конфликты имен полностью, но существенно снижают их вероятность.

Если вы создаете приложения на базе Cocoa, следует помнить, что компания Apple зарезервировала право использования двухбуквенных префиксов для себя, поэтому в этом случае определенно стоит выбирать трехбуквенные префиксы. Например, если разработчик не соблюдал эти рекомендации и решал использовать префикс `TW`, это могло создать проблему. В пакет iOS 5.0 SDK была включена поддержка Twitter, которая использует префикс `TW`; в ней имеется класс `TWRequest` для создания запросов HTTP к Twitter API. Разработчик мог очень легко создать собственный класс `TWRequest` при создании собственного API — например, для компании Tiny Widgets.

Префиксы не должны ограничиваться именами классов. Применяйте их ко всем именам, встречающимся в приложениях. В подходе 25

объясняется важность префиксов имен категорий и их методов, если категория относится к существующему классу. Также часто забывают о другой важной области потенциальных конфликтов — функциях C и глобальных переменных, используемых в файлах реализации классов. Разработчики часто забывают, что эти имена будут присутствовать в таблице символических имен верхнего уровня в откомпилированных объектных файлах. Например, фреймворк AudioToolbox в iOS SDK содержит функцию для воспроизведения звукового файла. Для нее можно назначить функцию обратного вызова, которая будет вызвана по завершении воспроизведения. Возможно, вы решите написать класс для включения этой функциональности в класс Objective-C, который вызывает делегата по завершении звукового файла:

```
// EOCSoundPlayer.h
#import <Foundation/Foundation.h>

@class EOCSoundPlayer;
@protocol EOCSoundPlayerDelegate <NSObject>
- (void)soundPlayerDidFinish:(EOCSoundPlayer*)player;
@end

@interface EOCSoundPlayer : NSObject
@property (nonatomic, weak) id <EOCSoundPlayerDelegate>
delegate;
- (id)initWithURL:(NSURL*)url;
- (void)playSound;
@end

// EOCSoundPlayer.m

#import "EOCSoundPlayer.h"
#import <AudioToolbox/AudioToolbox.h>

void completion(SystemSoundID ssID, void *clientData) {
    EOCSoundPlayer *player =
        (__bridge EOCSoundPlayer*)clientData;
    if ([player.delegate
            respondsToSelector:@selector(soundPlayerDidFinish:)])
    {
        [player.delegate soundPlayerDidFinish:player];
    }
}

@implementation EOCSoundPlayer {
```

```
    SystemSoundID _systemSoundID;
}

- (id)initWithURL:(NSURL*)url {
    if ((self = [super init])) {
        AudioServicesCreateSystemSoundID((__bridge CFURLRef)url,
                                         &_systemSoundID);
    }
    return self;
}

- (void)dealloc {
    AudioServicesDisposeSystemSoundID(_systemSoundID);
}

- (void)playSound {
    AudioServicesAddSystemSoundCompletion(
        _systemSoundID,
        NULL,
        NULL,
        completion,
        (__bridge void*)self);
    AudioServicesPlaySystemSound(_systemSoundID);
}

@end
```

Выглядит вполне безобидно, но, заглянув в таблицу символических имен объектного файла, созданного на базе класса, мы видим следующее:

```
00000230 t -[EOCSoundPlayer .cxx_destruct]
0000014c t -[EOCSoundPlayer dealloc]
000001e0 t -[EOCSoundPlayer delegate]
0000009c t -[EOCSoundPlayer initWithURL:]
00000198 t -[EOCSoundPlayer playSound]
00000208 t -[EOCSoundPlayer setDelegate:]
00000b88 S _OBJC_CLASS_$_EOCSoundPlayer
00000bb8 S _OBJC_IVAR_$_EOCSoundPlayer._delegate
00000bb4 S _OBJC_IVAR_$_EOCSoundPlayer._systemSoundID
00000b9c S _OBJC_METACLASS_$_EOCSoundPlayer
00000000 T _completion
00000bf8 s l_OBJC_$_INSTANCE_METHODS_EOCSoundPlayer
00000c48 s l_OBJC_$_INSTANCE_VARIABLES_EOCSoundPlayer
00000c78 s l_OBJC_$_PROP_LIST_EOCSoundPlayer
00000c88 s l_OBJC_CLASS_RO_$_EOCSoundPlayer
00000bd0 s l_OBJC_METACLASS_RO_$_EOCSoundPlayer
```

Обратите внимание на прячущееся в середине имя `_completion`. Это функция завершения, которая должна обрабатывать завершение воспроизведения звука. И хотя она определяется в файле реализации без объявления в заголовочном файле, она все равно остается символическим именем верхнего уровня. Если в будущем будет создана другая функция с именем `completion`, во время компоновки произойдет ошибка дублирования символических имен:

```
duplicate symbol _completion in:
    build/EOCSoundPlayer.o
    build/EOCAnotherClass.o
```

Еще хуже, если ваш код будет распространяться как библиотека, которую другие разработчики будут использовать в своих приложениях. Получается, что пользователи такой библиотеки уже не смогут создать функцию с именем `completion`; вряд ли они похвалят вас за это.

Итак, функции C тоже всегда должны снабжаться префиксами. Например, в приведенном примере обработчику завершения можно присвоить имя `EOCSoundPlayerCompletion`. У правильного выбора имени также имеется побочный эффект: если символическое имя появится в данных трассировки, вы сможете легко определить, в каком коде возникли проблемы.

К проблемам дублирования символических имен следует относиться особенно осторожно в том случае, если вы используете стороннюю библиотеку в коде, который сам будет распространяться в виде библиотеки. Если используемые вами сторонние библиотеки также используются в приложении, это сильно повышает риск ошибок дублирования символических имен. В таких ситуациях обычно применяется редактирование всего кода используемых библиотек с заменой префикса. Например, если в вашей библиотеке EOCLibrary используется библиотека с именем XYZLibrary, префиксы всех имен в XYZLibrary заменяются префиксами `EOC`. После этого приложение может использовать библиотеку XYZLibrary без риска коллизий имен, как показано на рис. 3.1.

Конечно, замена всех имен в коде — дело довольно утомительное, но это разумная мера для предотвращения конфликтов имен. Почему все это необходимо, почему приложение не может просто не включать XYZLibrary и использовать вашу реализацию? Конечно, это возможно, но представьте ситуацию, в которой приложение подключает третью библиотеку ABCLibrary, в которой также

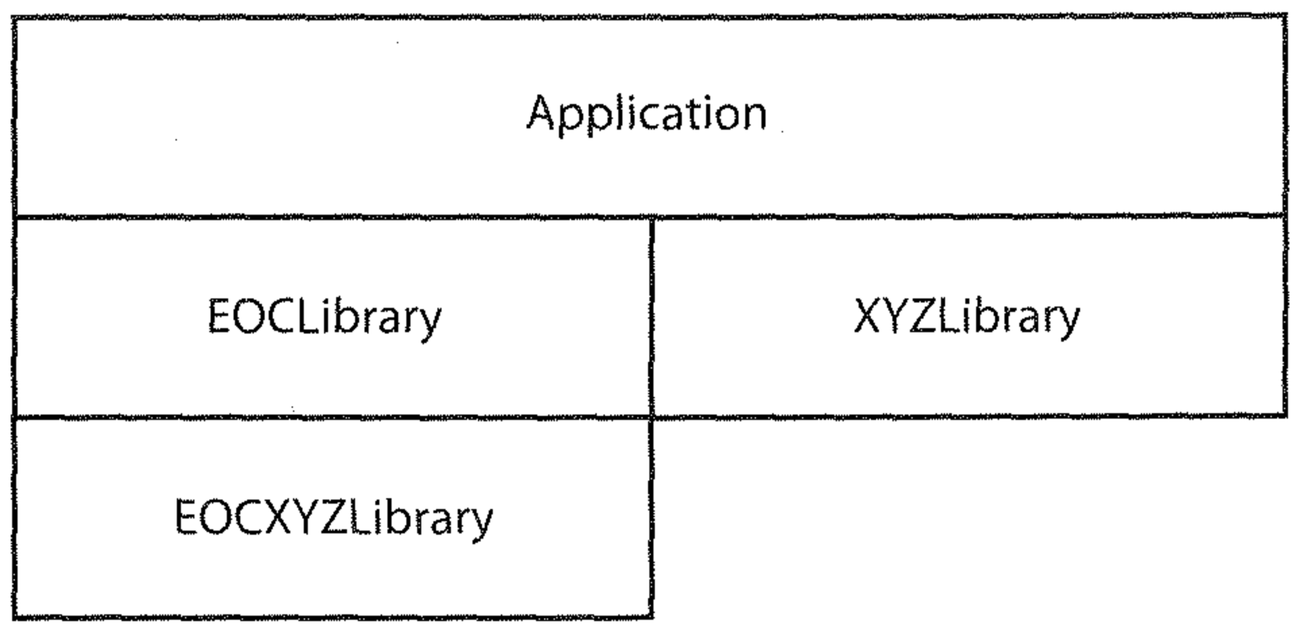


Рис. 3.1. Предотвращение конфликтов имен при повторном включении сторонней библиотеки (самим приложением и другой библиотекой)

используется библиотека XYZLibrary. В этом случае префиксы не используются ни вами, ни автором ABCLibrary и в приложении все равно появятся ошибки дублирующихся символических имен. Или если вы используете версию X библиотеки XYZLibrary, а приложению необходима функциональность из версии Y, ему все равно потребуется своя копия. Если у вас есть опыт использования популярных сторонних библиотек в программировании для iOS, такие префиксы вам наверняка попадались.

УЗЕЛКИ НА ПАМЯТЬ

- Выберите префикс класса по названию компании и/или приложения и используйте его во всем коде.
- Если вы используете стороннюю библиотеку в своей библиотеке, рассмотрите возможность замены ее имен с включением своего префикса.

16 ИСПОЛЬЗУЙТЕ ОСНОВНОЙ ИНИЦИАЛИЗАТОР

Все объекты должны инициализироваться. В некоторых случаях объекты инициализируются в состоянии по умолчанию, но часто объект не может выполнить инициализацию без получения дополнительной информации. Например, класс `UITableViewCell` из UIKit (фреймворк пользовательского интерфейса для iOS) при

своей инициализации должен получать стиль и идентификатор для группировки ячеек разных типов; это позволяет эффективно использовать объекты ячеек, создание которых сопряжено с большими затратами. Инициализатор, который передает объекту необходимую информацию для выполнения этой операции, называется *основным* (designated) инициализатором.

Если экземпляры класса могут создаваться несколькими способами, класс может иметь более одного инициализатора. Это абсолютно нормально, но при этом у него все равно должен быть только один основной инициализатор, используемый всеми остальными инициализаторами. Например, для класса `NSDate` определяются следующие инициализаторы:

```
- (id)init
- (id)initWithString:(NSString*)string
- (id)initWithTimeIntervalSinceNow:(NSTimeInterval)seconds
- (id)initWithTimeInterval:(NSTimeInterval)seconds
                  sinceDate:(NSDate*)refDate
- (id)initWithTimeIntervalSinceReferenceDate:
                              (NSTimeInterval)seconds
- (id)initWithTimeIntervalSince1970:(NSTimeInterval)seconds
```

Основным инициализатором в данном случае является `initWithTimeIntervalSinceReferenceDate:`, как указано в документации класса. Это означает, что все остальные инициализаторы в конечном итоге передают ему управление. Таким образом, сохранение внутренних данных осуществляется только в основном инициализаторе. Если данные по какой-либо причине изменятся, то изменения достаточно внести всего в одном месте.

Для примера возьмем класс, представляющий прямоугольник. Его интерфейс выглядит так:

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

@interface EOCRectangle : NSObject
@property (nonatomic, assign, readonly) float width;
@property (nonatomic, assign, readonly) float height;

@end
```

В соответствии с рекомендациями из подхода 18 свойства доступны только для чтения. Но это означает, что свойства объекта `Rectangle` не могут быть заданы извне, поэтому в класс включается инициализатор:

```
- (id)initWithWidth:(float)width
          andHeight:(float)height
{
    if ((self = [super init])) {
        _width = width;
        _height = height;
    }
    return self;
}
```

Но что, если кто-то захочет создать прямоугольник вызовом `[[EOCRectangle alloc] init]`? Это вполне законно, поскольку суперкласс `EOCRectangle` — `NSObject` — реализует метод с именем `init`, который просто задает всем переменным экземпляров значение 0 (или эквивалент 0 для фактического типа данных). При вызове этого метода ширина и высота прямоугольника сохранят значение 0 от момента создания экземпляра `EOCRectangle` (в вызове `alloc`), когда все переменные экземпляра были обнулены. И хотя может оказаться, что именно такое поведение вам и нужно, с большей вероятностью вы захотите либо задать значения по умолчанию самостоятельно, либо выдать исключение, указывающее, что экземпляр должен инициализироваться только с использованием основного инициализатора. В случае `EOCRectangle` это означает переопределение метода `init`:

```
// Использование значений по умолчанию
- (id)init {
    return [self initWithWidth:5.0f andHeight:10.0f];
}

// Выдача исключения
- (id)init {
    @throw [NSException
            exceptionWithName:NSInternalInconsistencyException
             reason:@"Must use initWithWidth:andHeight:
instead."
           userInfo:nil];
}
```

Обратите внимание на то, как версия, задающая значения по умолчанию, передает управление основному инициализатору. Ее также можно было бы реализовать непосредственным заданием переменных экземпляра `_width` и `_height`. Но если способ хранения данных изменится (например, с использованием структуры для хранения ширины и высоты вместе), логику присваивания придется изменять в обоих инициализаторах. В этом простом примере это не создает особых проблем, но представьте себе более сложный класс

с существенно большим количеством инициализаторов и более сложными данными. В этом случае повышается вероятность того, что разработчик забудет изменить один из инициализаторов, что приведет к логическим несоответствиям.

Теперь представьте, что вам потребовалось субклассировать класс прямоугольника `EOCRectangle` и создать класс с именем `EOCSquare`, представляющий квадрат. Понятно, что такая иерархия разумна, но где должен находиться инициализатор? Естественно, ширина и высота должны совпадать — ведь это следует из самого определения квадрата! Возможно, вам захочется создать инициализатор следующего вида:

```
#import "EOCRectangle.h"

@interface EOCSquare : EOCRectangle
- (id)initWithDimension:(float)dimension;
@end

@implementation EOCSquare

- (id)initWithDimension:(float)dimension {
    return [super initWithWidth:dimension andHeight:dimension];
}

@end
```

Он становится основным инициализатором `EOCSquare`. Обратите внимание на вызов основного инициализатора суперкласса. Обратившись к реализации `EOCRectangle`, мы видим, что она тоже вызывает основной инициализатор своего суперкласса. Очень важно, чтобы эта цепочка вызовов основных инициализаторов сохранялась, но вызывающая сторона может инициализировать объект `EOCSquare` с использованием как `initWithWidth:andHeight:`, так и метода `init`. Такая ситуация нежелательна, поскольку в ней может быть создан «квадрат», у которого ширина и высота не совпадают. В этом проявляется один важный аспект субклассирования: вы всегда должны переопределять основной инициализатор своего суперкласса, если вы используете основной инициализатор с другим именем. В случае `EOCSquare` переопределение основного инициализатора `EOCRectangle` может выглядеть так:

```
- (id)initWithWidth:(float)width andHeight:(float)height {
    float dimension = MAX(width, height);
    return [self initWithDimension:dimension];
}
```

Обратите внимание на вызов основного инициализатора EOCSquare. С этой реализацией, как по волшебству, также начинает работать метод init. Вспомните, что в EOCRectangle его реализация передавала управление основному инициализатору EOCRectangle со значениями по умолчанию. Она по-прежнему это делает, но так как метод initWithWidth:andHeight: был переопределен, вызывается реализация EOCSquare, которая в свою очередь передает управление основному инициализатору. Получается, что все работает, и создать объект EOCSquare с разными сторонами не удастся.

Иногда переопределение основного инициализатора суперкласса нежелательно, поскольку оно не имеет смысла. Например, вы можете решить, что присваивание объекту EOCSquare, созданному методом initWithWidth:andHeight:, размера, равного большему из двух переданных значений, является ошибкой пользователя. В таких ситуациях обычно применяется переопределение основного инициализатора с выдачей исключения:

```
- (id)initWithWidth:(float)width andHeight:(float)height {
    @throw [NSException
        exceptionWithName:NSInternalInconsistencyException
                  reason:@"Must use initWithDimension: instead."
                  userInfo:nil];
}
```

Такое решение выглядит довольно радикально, но иногда оно неизбежно, потому что созданный объект содержит противоречивые внутренние данные. В примере с EOCRectangle и EOCSquare это будет означать, что вызов init также приведет к выдаче исключения, потому что init передает управление initWithWidth:andHeight:. В таком случае можно переопределить init и вызвать initWithDimension: с разумным значением по умолчанию:

```
- (id)init {
    return [self initWithDimension:5.0f];
}
```

Однако выдача исключения в Objective-C означает, что ошибка фатальна (см. подход 21), так что выдача исключения из инициализатора должна стать последней мерой, если экземпляр не может быть инициализирован другим способом.

В некоторых ситуациях одного основного инициализатора оказывается недостаточно — например, если экземпляр может создаваться двумя разными способами, которые должны обрабатываться

по отдельности. В качестве примера можно привести протокол `NSCoding` — механизм сериализации, позволяющий объектам управлять процессом своего кодирования и декодирования. Этот механизм широко используется в AppKit и UIKit, двух фреймворках пользовательского интерфейса для Mac OS X и iOS соответственно, для управления сериализацией объектов в формате XML и созданием так называемых NIB-файлов, обычно используемых для хранения макета контроллера представлений. При загрузке из NIB-файла контроллер представления проходит процесс распаковки.

Согласно протоколу `NSCoding`, должен быть реализован следующий метод инициализатора:

```
- (id)initWithCoder:(NSCoder*)decoder;
```

Этот метод обычно не может передать управление основному инициализатору, потому что ему приходится выполнять разную работу для распаковки объекта через декодер. Кроме того, если суперкласс тоже реализует `NSCoding`, должен быть вызван его метод `initWithCoder:`. В результате у класса появляются два основных инициализатора в формальном понимании, так как более чем один инициализатор передает управление инициализатору суперкласса.

Применительно к `EOCRectangle` мы получаем следующее:

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

@interface EOCRectangle : NSObject <NSCoding>
@property (nonatomic, assign, readonly) float width;
@property (nonatomic, assign, readonly) float height;
- (id)initWithWidth:(float)width
          andHeight:(float)height;
@end

@implementation EOCRectangle

// Основной инициализатор
- (id)initWithWidth:(float)width
          andHeight:(float)height
{
    if ((self = [super init])) {
        _width = width;
        _height = height;
    }
    return self;
}
```

```
// Основной инициализатор суперкласса
- (id)init {
    return [self initWithWidth:5.0f andHeight:10.0f];
}

// Инициализатор от NSCoding
- (id)initWithCoder:(NSCoder*)decoder {
    // Передача управления основному инициализатору суперкласса
    if ((self = [super init])) {
        _width = [decoder decodeFloatForKey:@"width"];
        _height = [decoder decodeFloatForKey:@"height"];
    }
    return self;
}

@end
```

Обратите внимание на то, как инициализатор `NSCoding` передает управление основному инициализатору суперкласса (вместо своего собственного). Но если суперкласс также реализует `NSCoding`, он передаст управление этому инициализатору. Для примера рассмотрим `EOCSquare`:

```
#import "EOCRectangle.h"

@interface EOCSquare : EOCRectangle
- (id)initWithDimension:(float)dimension;
@end

@implementation EOCSquare

// Основной инициализатор
- (id)initWithDimension:(float)dimension {
    return [super initWithWidth:dimension andHeight:dimension];
}

// Основной инициализатор суперкласса
- (id)initWithWidth:(float)width andHeight:(float)height {
    float dimension = MAX(width, height);
    return [self initWithDimension:dimension];
}

// Основной инициализатор NSCoding
- (id)initWithCoder:(NSCoder*)decoder {
    if ((self = [super initWithCoder:decoder])) {
        // EOCSquare's specific initializer
```

```
    }
    return self;
}

@end
```

По аналогии с тем, как любой инициализатор по цепочке передает управление реализации суперкласса, `initWithCoder:` сначала вызывает ее, прежде чем делать что-либо конкретное. В этом отношении класс `EOCSquare` полностью совместим с `NSCoding`. Если бы мы передавали управление собственному инициализатору или другому инициализатору суперкласса, инициализатор `initWithCoder:` класса `EOCRectangle` никогда бы не вызывался для экземпляров `EOCSquare`, а переменные экземпляров `_width` и `_height` не были бы декодированы.

УЗЕЛКИ НА ПАМЯТЬ

- Реализуйте основной инициализатор в своих классах и документируйте его. Все остальные инициализаторы должны передавать ему управление.
- Если основной инициализатор отличен от основного инициализатора суперкласса, проследите за тем, чтобы его основной инициализатор был переопределен.
- Выдавайте исключение в переопределенных инициализаторах суперклассов, которые не должны использоваться в субклассах.

17 РЕАЛИЗУЙТЕ МЕТОД DESCRIPTION

В ходе отладки часто бывает полезно вывести содержимое объекта для анализа. В одном из способов написания кода трассировки выводятся все свойства объекта. Но чаще используются конструкции, которые выглядят примерно так:

```
NSLog(@"object = %@", object);
```

При построении строки, которая должна быть выведена в журнал, объекту отправляется сообщение `description`; результат подставляется на место %@ в форматной строке. Таким образом, например, если бы объект был массивом, команда вывода могла бы выглядеть так:

```
NSArray *object = @[@"A string", @(123)];
NSLog(@"object = %@", object);
```

А выводимый результат выглядит так:

```
object = (
  "A string",
  123
)
```

Но если вы попытаетесь проделать нечто подобное со своим классом, результат будет совсем другим:

```
object = <EOCPerson: 0x7fd9a1600600>
```

Совсем не так удобно, как с массивом! Если вы не переопределите метод `description` в своем классе, будет вызвана реализация по умолчанию из `NSObject`. Метод определяется в протоколе `NSObject`, но реализуется классом `NSObject`. Многие методы являются частью протокола `NSObject`, потому что `NSObject` — не единственный корневой класс. Скажем, `NSProxy` также является корневым классом, соответствующим протоколу `NSObject`. Поскольку такие методы, как `description`, определяются в протоколе, субклассы других корневых классов также должны реализовать их. Как видно из примера, эта реализация делает не так уж много: она выводит имя класса и адрес объекта в памяти. Эта информация будет полезна, если вы хотите только узнать, являются ли два объекта одним, — и только. Вероятно, вы бы предпочли получить более развернутую информацию об объекте.

Чтобы сделать вывод более полезным, достаточно переопределить метод `description`, чтобы он возвращал нужную строку с представлением объекта. Допустим, имеется класс для описания человека:

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

@interface EOCPerson : NSObject

@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName;

@end

@implementation EOCPerson
```

```
- (id)initWithFirstName:(NSString*)firstName
                lastName:(NSString*)lastName
{
    if ((self = [super init])) {
        _firstName = [firstName copy];
        _lastName = [lastName copy];
    }
    return self;
}

@end
```

Типичная реализация метода `description` для этого класса выглядит так:

```
- (NSString*)description {
    return [NSString stringWithFormat:@"<%@: %p, \"%@ %@\">",
            [self class], self, _firstName, _lastName];
}
```

При использовании этой реализации отладочный вывод для объекта типа `EOCPerson` выглядит примерно так:

```
EOCPerson *person = [[EOCPerson alloc]
                     initWithFirstName:@"Bob"
                               lastName:@"Smith"];
NSLog(@"person = %@", person);
// Вывод:
// person = <EOCPerson: 0x7fb249c030f0, "Bob Smith">
```

Результат содержит гораздо больше полезной информации. Я рекомендую выводить имя класса и адрес, как в реализации по умолчанию, просто потому, что эта информация иногда бывает полезной. Как было показано ранее, `NSArray` этого не делает, и, разумеется, никаких единых правил на этот счет не существует. Включайте в описание ту информацию, которая имеет смысл для конкретного объекта.

Простой способ написания метода `description` с большим количеством разнообразной информации основан на использовании реализации `description` класса `NSDictionary`, возвращающей результат следующего вида:

```
{
    key: value;
    foo: bar;
}
```

Чтобы воспользоваться этой компактной формой описания, создайте в своем методе `description` словарь и верните строку с результатом вызова метода `description` этого словаря. Например, следующий класс описывает географическую точку с названием и координатами (широтой и долготой):

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

@interface EOCLocation : NSObject
@property (nonatomic, copy, readonly) NSString *title;
@property (nonatomic, assign, readonly) float latitude;
@property (nonatomic, assign, readonly) float longitude;
- (id)initWithTitle:(NSString*)title
          latitude:(float)latitude
         longitude:(float)longitude;
@end

@implementation EOCLocation

- (id)initWithTitle:(NSString*)title
          latitude:(float)latitude
         longitude:(float)longitude
{
  if ((self = [super init])) {
    _title = [title copy];
    _latitude = latitude;
    _longitude = longitude;
  }
  return self;
}

@end
```

Мы хотим, чтобы метод `description` для этого класса выводил название и широту с долготой. При использовании `NSDictionary` реализация `description` может выглядеть так:

```
- (NSString*)description {
    return [NSString stringWithFormat:@"<%@: %p, %@>",
            [self class],
            self,
            @{@"title":_title,
              @"latitude":@(_latitude),
              @"longitude":@(_longitude)}
           ];
}
```

Примерный вид вывода:

```
location = <EOCLocation: 0x7f98f2e01d20, {
    latitude = "51.506";
    longitude = 0;
    title = London;
}>
```

Эта информация намного полезнее указателя и имени класса, а все свойства в объекте представлены в удобном виде. Также можно сформировать строку из переменных экземпляров, но решение с `NSDictionary` упростит сопровождение кода при добавлении в класс новых свойств, которые также потребуется включить в выходные данные `description`.

Другой метод, о котором следует знать, — `debugDescription` — также является частью протокола `NSObject`. Он предназначен для тех же целей, что и `description`. Два метода различаются тем, что метод `debugDescription` вызывается при выполнении команды вывода объекта в отладчике. Реализация по умолчанию из класса `NSObject` просто передает управление `description`. Допустим, в примере с классом `EOCPerson` мы запускаем приложение в отладчике (LLDB в данном случае) и вставляем точку прерывания сразу же за созданием экземпляра:

```
EOCPerson *person = [[EOCPerson alloc]
                        initWithFirstName:@"Bob"
                                 lastName:@"Smith"];
NSLog(@"person = %@", person);
// Здесь ставится точка прерывания
```

При достижении точки прерывания отладочная консоль готова к выводу данных. Команда po в LLDB выводит следующий результат:

```
EOCTest[640:c07] person = <EOCPerson: 0x712a4d0, "Bob Smith">
(lldb) po person
(EOCPerson *) $1 = 0x0712a4d0 <EOCPerson: 0x712a4d0, "Bob
Smith">
```

Учтите, что строка `(EOCPerson *) $1 = 0x712a4d0` добавлена отладчиком. Метод отладочного вывода возвращает строку, которая следует далее.

Возможно, вы решите, что нормальное описание `EOCPerson` должно состоять только из имени, а метод отладочного вывода должен

предоставлять более подробную информацию. В этом случае два метода будут выглядеть так:

```
- (NSString*)description {
    return [NSString stringWithFormat:@"%@ %@",
            _firstName, _lastName];
}

- (NSString*)debugDescription {
    return [NSString stringWithFormat:@"<%@: %p, \"%@ %@\">",
            [self class], self, _firstName, _lastName];
}
```

На этот раз после выполнения приведенного выше кода и команды отладочного вывода будет получен следующий результат:

```
EOCTest[640:c07] person = Bob Smith
(lldb) po person
(EOCPerson *) $1 = 0x07117fb0 <EOCPerson: 0x7117fb0, "Bob
Smith">
```

Такой вывод особенно полезен, если вы не хотите, чтобы дополнительная информация (имя класса, адрес в памяти) была видна в нормальном описании, но была легкодоступна в ходе отладки. Например, такая возможность реализована в классе `NSArray` из фреймворка Foundation:

```
NSArray *array = @[@"Effective Objective-C 2.0", @(123),
                   @(YES)];
NSLog(@"array = %@", array);
// Здесь ставится точка прерывания
```

Если теперь запустить программу, остановить ее на точке прерывания и вывести объект массива, вы получите следующий результат:

```
EOCTest[713:c07] array = (
    "Effective Objective-C 2.0",
    123,
    1
)
(lldb) po array
(NSArray *) $1 = 0x071275b0 <__NSArrayI 0x71275b0>(
Effective Objective-C 2.0,
123,
1
)
```

УЗЕЛКИ НА ПАМЯТЬ

- Реализуйте метод `description`, чтобы предоставить содержательное описание экземпляров в форме строки.
- Если в описание объекта желательно включить дополнительную информацию в ходе отладки, реализуйте метод `debugDescription`.

18 ВЫБИРАЙТЕ НЕИЗМЕНЯЕМЫЕ ОБЪЕКТЫ

В идеале при проектировании класса следует использовать свойства (см. подход 6) для инкапсуляции данных. Свойства могут ограничиваться доступом только для чтения. Оставляя им уровень доступа по умолчанию (чтение/запись), вы делаете свои классы изменяемыми (mutable). Однако часто моделируемые данные не обязаны быть изменяемыми. Например, если данные представляют объект, полученный от веб-службы с доступом только для чтения (скажем, список точек для отображения на карте), изменяемость такого объекта не имеет смысла. Даже если такой объект будет изменен, это не приведет к обновлению данных на сервере. Как показано в подходе 8, при хранении изменяемых объектов в коллекциях изменение объектов легко приводит к нарушению целостности внутренних структур данных контейнеров. По этой причине объекты следует делать изменяемыми только тогда, когда это действительно необходимо.

На практике это означает, что свойства при внешних обращениях доступны только для чтения, а доступ предоставляется только к тем данным, для которых это действительно необходимо. Для примера рассмотрим класс для обработки точек на карте, полученных от веб-службы. Первая версия класса может выглядеть примерно так:

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

@interface EOCPointOfInterest : NSObject

@property (nonatomic, copy) NSString *identifier;
@property (nonatomic, copy) NSString *title;
@property (nonatomic, assign) float latitude;
@property (nonatomic, assign) float longitude;

- (id)initWithIdentifier:(NSString*)identifier
```

```
                    title:(NSString*)title
                 latitude:(float)latitude
                longitude:(float)longitude;

@end
```

Все значения поступают от веб-службы, а для обозначения нужной точки во взаимодействиях со службой используется идентификатор. После того как точка будет создана в ходе взаимодействия со службой, разрешать изменение ее данных не имеет смысла. Вероятно, в других языках вы бы объявили переменные экземпляров закрытыми, используя соответствующие языковые конструкции, и определили для переменной только get-метод. Однако благодаря свойствам Objective-C открывается отличная возможность не лениться и обойтись вообще без закрытых переменных.

Чтобы сделать класс `EOCPointOfInterest` неизменным, следует добавить во все свойства атрибут `readonly`:

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

@interface EOCPointOfInterest : NSObject

@property (nonatomic, copy, readonly) NSString *identifier;
@property (nonatomic, copy, readonly) NSString *title;
@property (nonatomic, assign, readonly) float latitude;
@property (nonatomic, assign, readonly) float longitude;

- (id)initWithIdentifier:(NSString*)identifier
                   title:(NSString*)title
                latitude:(float)latitude
               longitude:(float)longitude;
@end
```

Такое решение гарантирует, что при попытке изменения одного из значений свойств будет выдана ошибка компилятора. Значения по-прежнему можно читать, но нельзя изменять, так что данные `EOCPointOfInterest` не могут потерять внутренней целостности. В случае с картой, на которой отображаются объекты `EOCPointOfInterest`, широта и долгота отображаемых точек останутся неизменными.

Возможно, вас интересует, почему нужно беспокоиться о назначении атрибутов семантики управления памятью для свойства — ведь set-метода все равно не будет? Конечно, можно оставить определения по умолчанию:

```
@property (nonatomic, readonly) NSString *identifier;
@property (nonatomic, readonly) NSString *title;
@property (nonatomic, readonly) float latitude;
@property (nonatomic, readonly) float longitude;
```

Тем не менее семантику управления памятью, используемую в реализации, полезно документировать; вдобавок это упростит возможное преобразование свойства для чтения/записи, если возникнет такая необходимость.

Возможно, вы захотите сохранить возможность изменения данных, инкапсулированных в объекте, средствами самого объекта — но не снаружи. В этом случае обычно применяется внутреннее переобъявление свойства `readonly` с атрибутом `readwrite`. Конечно, при этом появляется вероятность состояния гонки у неатомарных свойств. Теоретически наблюдатель может читать свойство одновременно с его внутренней записью. Чтобы избежать подобных ситуаций, необходимо позаботиться о том, чтобы все обращения, включая внутренние, были синхронизированы — например, за счет использования очереди диспетчеризации (см. подход 41).

Внутреннее переобъявление свойства с атрибутом `readwrite` может быть выполнено при помощи категории продолжения класса (см. подход 27); вы можете объявить свойство, уже присутствующее в открытом интерфейсе, при условии, что оно обладает теми же атрибутами и расширяет свой статус `readonly/readwrite`. В случае `EOCPointOfInterest` категория продолжения класса может выглядеть так:

```
#import "EOCPointOfInterest.h"

@interface EOCPointOfInterest ()
@property (nonatomic, copy, readwrite) NSString *identifier;
@property (nonatomic, copy, readwrite) NSString *title;
@property (nonatomic, assign, readwrite) float latitude;
@property (nonatomic, assign, readwrite) float longitude;
@end

@implementation EOCPointOfInterest

/* … */

@end
```

Теперь свойства могут задаваться только «изнутри», из реализации `EOCPointOfInterest`. А если говорить точнее, внешний по отноше-

нию к объекту пользователь может задать эти свойства при помощи механизма KVC (Key-Value Coding), используя `setValue:forKey:` следующим образом:

```
[pointOfInterest setValue:@"abc" forKey:@"identifier"];
```

Эта команда изменит значение свойства `identifier`, потому что KVC найдет в вашем классе метод `setIdentifier:`, который существует, хотя и не предоставляется в открытом интерфейсе. Однако происходящее является вторжением в API вашего класса, и если с подобными трюками что-то пойдет не так — пускай разработчик сам разбирается.

Метод «грубой силы» позволяет даже обойти отсутствие set-метода — в классе посредством интроспекции определяется смещение переменной экземпляра свойства в двоичном представлении экземпляра. Далее разработчик может вручную задать переменную экземпляра. Впрочем, это является еще более грубым вторжением в открытый API класса. Таким образом, хотя техническая возможность обойти отсутствие открытого set-метода существует, вам все же стоит соблюдать рекомендации и делать объекты по возможности неизменяемыми.

Другое обстоятельство, о котором следует помнить при определении открытого API класса, — изменяемость и неизменяемость свойств классов коллекций. Если, например, в классе, представляющем человека, могут храниться ссылки на друзей, возможно, вам понадобится свойство для получения списка всех друзей этого человека. Если друзья могут добавляться и удаляться из списка, в класс включается изменяемое множество, поставляющее данные для свойства. В таком случае обычный способ предоставления доступа к списку друзей заключается в создании свойства `readonly`, возвращающего неизменяемое множество, являющееся копией внутреннего изменяемого множества. Например, возьмем следующее определение класса:

```
// EOCPerson.h
#import <Foundation/Foundation.h>

@interface EOCPerson : NSObject

@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;
@property (nonatomic, strong, readonly) NSSet *friends;

- (id)initWithFirstName:(NSString*)firstName
```

```
            andLastName:(NSString*)lastName;
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;

@end

// EOCPerson.m
#import "EOCPerson.h"

@interface EOCPerson ()
@property (nonatomic, copy, readwrite) NSString *firstName;
@property (nonatomic, copy, readwrite) NSString *lastName;
@end

@implementation EOCPerson {
    NSMutableSet *_internalFriends;
}

- (NSSet*)friends {
    return [_internalFriends copy];
}

- (void)addFriend:(EOCPerson*)person {
    [_internalFriends addObject:person];
}

- (void)removeFriend:(EOCPerson*)person {
    [_internalFriends removeObject:person];
}

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName {
    if ((self = [super init])) {
        _firstName = firstName;
        _lastName = lastName;
        _internalFriends = [NSMutableSet new];
    }
    return self;
}

@end
```

Свойство `friends` можно было объявить с классом `NSMutableSet` и разрешить пользователям класса добавлять людей в множество самим, вместо того чтобы вызывать методы `addFriend:` и `removeFriend:`. Но это приведет к ослаблению контроля над

данными и откроет возможность для внедрения ошибок. В этом случае внутреннее множество друзей `EOCPerson` может быть изменено «изнутри», что может создать проблемы — допустим, если объект `EOCPerson` захочет что-то сделать при добавлении или удалении друга. Если множество изменяется без ведома объекта, это может привести к нарушению логической целостности данных.

Также важно не применять к возвращаемым объектам интроспекцию для определения возможности их изменения. Допустим, вы взаимодействуете с библиотекой, содержащей класс `EOCPerson`. Вместо возвращения копии внутреннего изменяемого множества разработчик библиотеки решил вернуть само внутреннее изменяемое множество. Такое решение может быть вполне законным и даже предпочтительным при очень большом размере множества, если создание копии обойдется слишком дорого. Так как класс `NSMutableSet` является субклассом `NSSet`, появляется искушение использовать конструкцию следующего вида:

```
EOCPerson *person = /* … */;
NSSet *friends = person.friends;
if ([friends isKindOfClass:[NSMutableSet class]]) {
    NSMutableSet *mutableFriends = (NSMutableSet*)friends;
    /* изменение множества */
}
```

Избегайте таких решений любой ценой. Они не соответствуют контракту класса `EOCPerson`, поэтому применять интроспекцию подобным образом не следует. Причина все та же: объект может не справиться с внутренними изменениями данных, на которых он основан.

УЗЕЛКИ НА ПАМЯТЬ

- По возможности старайтесь создавать неизменяемые объекты.
- Если свойство должно изменяться во внутренней реализации, расширяйте свойства `readonly` до `readwrite` в категориях продолжения класса.
- Предоставляйте методы для изменения коллекций, хранящихся в объектах (вместо того чтобы предоставлять доступ к изменяемой коллекции в виде свойства).

19 ИСПОЛЬЗУЙТЕ ЧЕТКИЕ И ПОСЛЕДОВАТЕЛЬНЫЕ СХЕМЫ ФОРМИРОВАНИЯ ИМЕН

Имена классов, методов, переменных и т. д. являются важным аспектом Objective-C. Новички часто считают этот язык излишне пространным, потому что он использует синтаксическую структуру, с которой команды читаются как предложения. Имена часто включают связки («in», «for», «with» и т. д.), которые в других языках считаются избыточными. Для примера рассмотрим следующий фрагмент кода:

```
NSString *text = @"The quick brown fox jumped over the lazy
dog";
NSString *newText =
    [text stringByReplacingOccurrencesOfString:@"fox"
                                       withString:@"cat"];
```

Казалось бы, это довольно длинная запись для того, что выглядит как простое выражение, — метод выполнения замены состоит из 48 символов! Но зато он определенно читается как предложение: «Взять `text` и создать новую строку, заменив вхождения строки "fox" строкой "cat"».

Это предложение идеально описывает суть происходящего. В более компактных языках аналогичная процедура может выглядеть так:

```
string text = "The quick brown fox jumped over the lazy dog";
string newText = text.replace("fox", "cat");
```

Но в каком порядке следуют параметры `text.replace()`? Будет ли «fox» заменяться «cat», или наоборот? Кроме того, заменяет ли функция `replace` dct вхождения, или только первое? С ходу и не скажешь. Напротив, имена Objective-C предельно ясны — но за счет компактности.

Также обратите внимание на чередование регистра символов (camel casing) в именах методов и переменных, начинающихся со строчной буквы. Имена классов также используют чередование, но начинаются с прописной буквы и обычно имеют префикс из двух или трех букв (см. подход 15). Этот стиль является наиболее распространенным в коде Objective-C. При желании вы можете использовать собственную схему выбора имен, но чередование регистра гарантирует, что ваши имена будут уместно и естественно выглядеть в мире Objective-C.

ИМЕНА МЕТОДОВ

Программисты с опытом работы на C++ и Java привыкли присваивать функциям лаконичные имена и заглядывать в прототипы функций для определения смысла параметров. Однако это сильно затрудняет чтение кода, потому что пользователю приходится часто обращаться к прототипу, чтобы вспомнить, что делает функция.

Для примера возьмем класс, представляющий прямоугольник. В C++ его определение может выглядеть так:

```
class Rectangle {
public:
    Rectangle(float width, float height);
    float getWidth();
    float getHeight();
private:
    float width;
    float height;
};
```

Если вы не знаете C++, это неважно. Достаточно понимать, что в этом фрагменте определяется класс с двумя переменными экземпляров: `width` и `height`. Также определяется один способ создания экземпляров класса — конструктор, получающий размеры прямоугольника. Для размеров также задаются методы доступа. При использовании класса экземпляр создается следующим образом:

```
Rectangle *aRectangle = new Rectangle(5.0f, 10.0f);
```

Когда вы через какое-то время вернетесь к этому коду, вряд ли вам будет очевидно, что означают 5.0f и 10.0f. Наверное, размеры прямоугольника, но что сначала — `width` или `height`? Чтобы узнать это, придется возвращаться к определению функции.

Objective-C решает эту проблему за счет использования более подробных имен методов. Первая версия эквивалентного класса на Objective-C, написанная программистом, знакомым с C++, может выглядеть так:

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

@interface EOCRectangle : NSObject

@property (nonatomic, assign, readonly) float width;
@property (nonatomic, assign, readonly) float height;
```

```
- (id)initWithSize:(float)width :(float)height;

@end
```

Человек, написавший этот класс, явно увидел, что методы семейства `init` являются аналогами конструкторов, и решил использовать имя метода `initWithSize::`. На первый взгляд, это довольно странно, а кто-то даже подумает, что отсутствие символов перед вторым двоеточием является синтаксической ошибкой. Вообще говоря, синтаксис абсолютно законный, но ему присущ тот же недостаток, что и имени функции C++: при использовании класса вы снова не знаете, какая переменная чему соответствует:

```
EOCRectangle *aRectangle =
    [[EOCRectangle alloc] initWithSize:5.0f :10.0f];
```

Следующее имя метода намного лучше:

```
- (id)initWithWidth:(float)width andHeight:(float)height;
```

Оно длиннее предыдущего, но по крайней мере при использовании абсолютно однозначно видно, что означает та или иная переменная:

```
EOCRectangle *aRectangle =
    [[EOCRectangle alloc] initWithWidth:5.0f andHeight:10.0f];
```

Неофитам Objective-C часто трудно привыкнуть к длинным именам методов, хотя они и упрощают чтение кода. Не бойтесь использовать длинные имена. Присваивая методу нужное имя, вы проследите за тем, чтобы ваш метод делал то, что заявлено в его имени. Впрочем, абсурдно длинные имена тоже не приветствуются. Имена методов должны быть точными и однозначными.

Возьмем для примера класс `EOCRectangle`. В качестве удачно выбранных имен методов можно привести следующие примеры:

```
- (EOCRectangle*)unionRectangle:(EOCRectangle*)rectangle
- (float)area
```

Пара примеров неудачных имен этих методов:

```
- (EOCRectangle*)union:(EOCRectangle*)rectangle // Непонятно
- (float)calculateTheArea // Слишком длинно
```

Хорошие имена методов читаются слева направо, как фрагмент текста. Соблюдение многочисленных правил, определяющих имена

методов, не является обязательным, но оно, безусловно, упростит сопровождение вашего кода и сделает его более понятным для других.

Примеры хорошего выполнения рекомендаций по выбору имен можно найти в классе `NSString`. Далее приведены некоторые методы этого класса с краткими пояснениями относительно выбора имен:

`+ string`

Фабричный метод для создания новой пустой строки. Тип возвращаемого значения четко обозначен в имени метода.

`+ stringWithString:`

Фабричный метод для создания новой строки, идентичной существующей. Как и в случае с фабричным методом для создания новой строки, возвращаемый тип обозначается первым словом в имени метода.

`+ localizedStringWithFormat:`

Фабричный метод для создания новой локализованной строки в заданном формате. Возвращаемый тип обозначается вторым словом в имени метода, потому что к возвращаемому типу применяется модификатор. Метод также возвращает строку, но ее более конкретную разновидность (локализованная строка).

`- lowercaseString`

Преобразует строку, приводя все символы верхнего регистра в строке к нижнему регистру. Метод создает новую строку вместо преобразования получателя, поэтому он следует правилу включения возвращаемого типа в имя метода. К типу применяется модификатор, который предшествует ему.

`- intValue`

Разбирает строку как целое число. Первым словом имени является возвращаемый тип `int`. Обычно сокращенные имена типов не используются (например, `string` никогда не сокращается до `str`), но `int` является именем типа, поэтому здесь это допустимо. Имя метода снабжено суффиксом `Value`, чтобы оно не состояло из одного слова. Одиночные слова обычно используются для свойств; так как `int` не является свойством `string`, для уточнения смысла добавляется суффикс `Value`.

- `length`

  Получает длину строки (то есть количество символов). Имя состоит из одного слова, так как по сути представляет свойство строки. Вряд ли оно будет базироваться на переменной экземпляра, но от этого не перестает быть свойством. Пример плохого имени для этого метода — `stringLength`. Строка избыточна, поскольку получателем метода заведомо является строка.

- `lengthOfBytesUsingEncoding:`

  Получает длину массива байтов при кодировании строки в заданной кодировке (ASCII, UTF8, UTF16 и т. д.). По своему назначению метод аналогичен `length`, поэтому для его имени справедливо все, что было сказано о методе `length`. Кроме того, методу должен передаваться параметр. В имени метода параметр указывается непосредственно после описания типа.

- `getCharacters:range:`

  Получает символы из заданного диапазона строки. Это один из примеров использования префикса `get`, который обычно не применяется в методах доступа (в отличие от других языков). Здесь он используется из-за того, что символы возвращаются в массиве, передаваемом в первом аргументе. Полная сигнатура метода выглядит так:

```
- (void)getCharacters:(unichar*)buffer
                range:(NSRange)aRange
```

  Первый параметр `buffer` должен содержать указатель на массив, размер которого достаточен для хранения символов из заданного диапазона. Метод возвращает результат через параметр (часто называемый *выходным параметром*), а не в возвращаемом значении, потому что это более разумно с точки зрения управления памятью. Вызывающая сторона осуществляет все управление памятью — вместо создания объекта в методе и его освобождения на стороне вызова. Второй параметр снабжается префиксом, описывающим его тип, как и для любого другого параметра. Иногда имена параметров снабжаются префиксами-частицами; например, этот метод также мог называться `getCharacters:inRange:`. Обычно это делается тогда, когда требуется подчеркнуть важность параметра по сравнению с другими параметрами.

- `hasPrefix:`

  Определяет, имеет ли строка префикс, заданный другой строкой. Метод возвращает логический тип, поэтому обычно разработчики стараются использовать конструкции, читающиеся как предложения, например:

  ```
  [@"Effective Objective-C" hasPrefix:@"Effective"] == YES
  ```

  Если бы метод назывался просто `prefix:`, он бы хуже читался. Также можно было бы назвать его `isPrefixedWith:`, но такое имя получается более длинным и громоздким.

- `isEqualToString:`

  Определяет, равны ли две строки. Метод возвращает логический тип, поэтому, как и в случае с `hasPrefix:`, имя начинается с `is`, чтобы метод читался как предложение. Также префикс `is` используется для логических свойств. Например, если бы свойству было присвоено имя `enabled`, методы доступа обычно назывались бы `setEnabled:` и `isEnabled`.

Ниже приведена сводка правил, которые помогут вам в выборе хороших имен методов.

- Если метод возвращает только что созданное значение, первым словом в имени метода должен быть его тип (если только перед ним не нужно поставить уточнение, как в случае с методами `localizedString`). Правило не распространяется на методы доступа свойств, поскольку логически они не создают новый объект, хотя и могут возвращать копию внутреннего объекта. Имена методов доступа выбираются по именам представляемых ими свойств.
- Параметру должно немедленно предшествовать существительное, описывающее его тип.
- Методы, выполняющие действие с объектом, должны содержать глагол, за которым следует существительное (или несколько существительных), если методу для выполнения своего действия требуются параметры.
- Избегайте сокращений (таких, как `str`) и используйте полные имена (`string`).
- Используйте в логических свойствах префикс `is`. Методы, возвращающие логическое значение, но непосредственно не

являющиеся свойствами, снабжаются префиксом `has` или `is` в зависимости от того, что имеет смысл для данного метода.

- Префикс `get` резервируется для методов, возвращающих значения через выходной параметр (например, заполняющих массивы в стиле C).

ЛЕНА КЛАССОВ И ПРОТОКОЛОВ

В именах классов и протоколов используются префиксы, предотвращающие конфликты пространств имен (см. подход 15); они должны читаться слева направо, как имена методов. Например, в классе `NSArray` и его изменяемом аналоге — субклассе `NSMutableArray` — квалификатор изменяемости следует перед `Array`, потому что он описывает специализацию типа.

Для демонстрации правил выбора имен рассмотрим следующую выборку из UIKit, UI-библиотеки для iOS.

`UIView` (класс)

Класс, от которого наследуют все *представления* (views). Представления являются структурными элементами пользовательского интерфейса и обеспечивают прорисовку кнопок, текстовых полей и таблиц. Имя класса не требует разъяснений; в нем используется префикс `UI`, встречающийся во всех компонентах фреймворка UIKit.

`UIViewController` (класс)

Класс `UIView` обеспечивает прорисовку представлений, но не отвечает за содержимое, отображаемое в представлении. Этим занимается класс *контроллера представления*. Его имя выбрано таким образом, чтобы сохранить порядок чтения слева направо.

`UITableView` (класс)

Конкретный тип представления для вывода списка элементов в таблице. Имя выбирается по имени суперкласса с квалификатором, уточняющим тип представления. Имя суперкласса с префиксом — стандартная схема в правилах выбора имен. Если бы класс назывался просто `UITable`, из имени не было бы понятно, что это представление. Вам пришлось бы заглядывать в объявление интерфейса, чтобы убедиться в этом. При создании конкретного табличного представления, предназначенного, допустим, для вывода графики, можно создать субкласс с именем `EOCImageTableView`. Всегда присоединяйте собственный

префикс вместо использования префикса суперкласса (для UIKit это префикс UI в данном случае). Это объясняется тем, что вы *не имеете прав для расширения* пространства имен другой библиотеки, так как в последней когда-нибудь может появиться класс с таким именем.

`UITableViewController` (класс)

По аналогии с тем, как `UITableView` является конкретной разновидностью представлений, также существует разновидность контроллера представлений, предназначенная специально для управления табличным представлением. Соответственно, и имя ее выбирается аналогичным образом.

`UITableViewDelegate` (протокол)

Протокол определяет интерфейс, через который табличное представление может взаимодействовать с другим объектом. Имя протокола определяется именем класса, для которого он определяет интерфейс делегата с сохранением порядка чтения слева направо (за дополнительной информацией о паттерне «Делегат» обращайтесь к подходу 23).

Прежде всего будьте последовательны в выборе имен. Кроме того, если вы субклассируете класс в другой библиотеке, придерживайтесь его правил. Например, если вы создаете пользовательское представление субклассированием `UIView`, проследите за тем, чтобы последним словом в имени класса было `View`. Аналогичным образом при создании собственного протокола делегата его имя строится из имени класса, для которого он является делегатом, и суффикса `Delegate`. Соблюдение этих правил гарантирует, что код будет понятен вам *или* кому-нибудь другому, кто позднее будет читать его.

УЗЕЛКИ НА ПАМЯТЬ

- Соблюдайте правила выбора имен, которые стали стандартными для Objective-C. Это позволит вам создавать интерфейсы, которые естественно выглядят в существующем коде.
- Следите за тем, чтобы имена методов были лаконичными, но точно описывали их назначение и читались слева направо, как предложения.
- Избегайте сокращенных имен типов в именах методов.
- Самое важное — следите за тем, чтобы имена методов согласовались с вашим кодом (или тем кодом, с которым он интегрируется).

20 РАЗБЕРИТЕСЬ С ПРЕФИКСАМИ В ИМЕНАХ ЗАКРЫТЫХ МЕТОДОВ

Классы очень часто делают больше того, что видно внешнему пользователю. В реализациях классов часто присутствуют методы, предназначенные только для внутреннего использования. Я рекомендую включать в их имена специальные префиксы. Если открытые методы будут визуально отличаться от закрытых, это упростит отладку.

Другая причина для специальной пометки закрытых методов — возможные изменения имени и сигнатуры методов. Если метод является открытым, такие изменения приходится продумывать более тщательно, потому что изменение открытого API класса нежелательно — возможно, пользователям класса придется переделывать свой код. Но если метод используется только во внутренних операциях, изменяться будет только собственный код класса без каких-либо последствий для открытого API. Пометка закрытых методов позволит вам быстро выделить их при внесении таких изменений.

Выбор префикса зависит от ваших личных предпочтений. Часто в него включаются символы подчеркивания и буква p. Лично я использую префикс p_, потому что «p» означает «private» (закрытый), а подчеркивание отделяет префикс от имени. Далее следует имя метода с обычным чередованием регистра, начиная со строчной буквы. Например, класс `EOCObject` с закрытыми методами может выглядеть так:

```
#import <Foundation/Foundation.h>
@interface EOCObject : NSObject
- (void)publicMethod;
@end

@implementation EOCObject

- (void)publicMethod {
    /* … */
}

- (void)p_privateMethod {
    /* … */
}

@end
```

В отличие от открытых методов, закрытый метод не включается в определение интерфейса. Иногда закрытый метод объявляется

в категории продолжения класса (см. подход 27); тем не менее последние изменения в компиляторе уже не требуют обязательного объявления методов перед использованием. Итак, закрытые методы обычно объявляются только в реализации.

У разработчиков с опытом программирования на C++ или Java может возникнуть вопрос: зачем все это, почему нельзя просто объявить метод закрытым? В Objective-C не существует способа пометить метод как закрытый. Все объекты могут реагировать на все сообщения (см. подход 12), и объект можно проанализировать во время выполнения, чтобы узнать, на какие сообщения он реагирует (см. подход 14). Поиск метода для заданного сообщения выполняется во время выполнения (см. подход 11), и не существует механизма для ограничения того, кем и когда может вызываться конкретный метод. Такая семантика, как закрытость методов, определяется на уровне выбора имен. Возможно, новичкам это покажется непривычным, но в динамизме Objective-C проявляется его мощь. Впрочем, этот динамизм следует держать под контролем, и правила выбора имен — один из способов обеспечения этого контроля.

Компания Apple в своих закрытых методах обычно использует префикс из единственного символа — подчеркивания. Казалось бы, разработчику тоже стоит последовать примеру Apple и использовать этот префикс. Тем не менее такое решение может привести к катастрофическим последствиям; использование этого префикса в субклассе класса Apple может привести к случайному переопределению одного из методов. По этой причине компания Apple официально документировала, что символ подчеркивания не следует использовать в качестве префикса. Возможно, отсутствие средств для ограничения области действия метода является недостатком языка, но это часть мощной динамической системы диспетчеризации методов (см. подход 11), у которой также имеется масса достоинств.

Эта ситуация не так уж маловероятна, как может показаться. Допустим, при создании контроллера представления в приложении iOS применяется субклассирование `UIViewController`. Контроллер представления может хранить значительное состояние. Вам нужен метод для очистки этого состояния, который должен выполняться при каждом появлении контроллера представления на экране. Вы пишете реализацию метода, которая выглядит так:

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

@interface EOCViewController : UIViewController
@end
```

```
@implementation EOCViewController
- (void)_resetViewController {
    // Сброс состояния и представлений
}
@end
```

И тут вдруг выясняется, что `UIViewController` тоже реализует метод `_resetViewController`! Версия `EOCViewController` будет выполняться повсюду, где вы ее вызываете, а также везде, где должна выполняться исходная версия. И вы даже не будете знать об этом без внимательного изучения кода библиотеки, потому что имя метода не раскрывается в интерфейсе. Ведь это закрытый метод, на что указывает подчеркивание! В такой ситуации начинаются всевозможные странности с контроллером представления, потому что исходная реализация не вызывается вообще, а ваша версия вызывается намного чаще, чем следует.

Короче говоря, при субклассировании классов в библиотеках, не принадлежащих ни вам, ни Apple, вы не можете знать, какой суффикс закрытых методов в них используется (и используется ли) — если только он не документирован. В такой ситуации вы можете использовать свой префикс классов (см. подход 15) в качестве префикса закрытых методов, чтобы значительно снизить риск любых потенциальных конфликтов. Аналогичным образом следует подумать о том, как другие люди будут субклассировать ваши классы; собственно, поэтому и нужно использовать префиксы в именах закрытых методов. Без исходного кода реализации невозможно (без применения довольно сложных инструментов) узнать, какие методы реализуются классом — кроме тех, что документированы в определении открытого интерфейса.

УЗЕЛКИ НА ПАМЯТЬ

- Включайте префиксы в имена закрытых методов, чтобы они отличались от открытых методов.
- Не используйте одиночный символ подчеркивания как префикс метода, поскольку этот префикс зарезервирован Apple.

21 РАЗБЕРИТЕСЬ С МОДЕЛЬЮ ОШИБОК OBJECTIVE-C

Во многих современных языках, включая Objective-C, поддерживаются исключения. Программисту с опытом работы на Java, скорее всего, использование исключений для обработки ошибок покажется вполне привычным. Если вы принадлежите к их числу, постарайтесь поскорее забыть все, что знаете об исключениях, и начать заново.

Первое, на что следует обратить внимание, — что механизм автоматического подсчета ссылок ARC (Automatic Reference Counting, см. подход 30) по умолчанию не обладает безопасностью к исключениям. На практике это означает, что любые объекты, которые должны освобождаться в конце области действия, выдавшей исключение, освобождены не будут. Разработчик может установить флаг компилятора для генерирования кода, безопасного к исключениям, но тогда дополнительный код будет генерироваться даже в том случае, если исключение не выдается. Этот режим включается флагом компилятора -fobjc-arc-exceptions.

Даже если ARC не используется, написать код, безопасный к утечкам памяти при использовании исключений, достаточно сложно. Допустим, создан ресурс, который должен освобождаться, когда он перестает быть необходимым. Если до освобождения ресурса будет выдано исключение, то ресурс так и останется неосвобожденным:

```
id someResource = /* … */;
if ( /* check for error */ ) {
    @throw [NSException exceptionWithName:@"ExceptionName"
                                   reason:@"There was an error"
                                 userInfo:nil];
}
[someResource doSomething];
[someResource release];
```

Конечно, проблему можно решить освобождением someResource перед выдачей исключения; но при большом количестве освобождаемых ресурсов и более сложной структуре кода программа быстро загромождается. Кроме того, если в такой код будет введен новый ресурс, разработчик может забыть о необходимости добавления команды освобождения до выдачи исключения.

В последнее время в Objective-C стало принято использовать исключения в редкой ситуации, когда восстановление нежелательно, а исключение должно приводить к выходу из приложения.

Это означает, что сложный код, безопасный к исключениям, не понадобится.

Итак, если исключения должны использоваться только для фатальных ошибок, рассмотрим пример с созданием абстрактного базового класса, который должен субклассироваться перед использованием. В Objective-C, в отличие от других языков, не существует языковой конструкции для обозначения абстрактных классов. Соответственно, лучшим способом ее имитации является выдача исключения в любом методе, который должен переопределяться в субклассах. При попытке создания экземпляра абстрактного базового класса и его использования будет выдано исключение:

```
- (void)mustOverrideMethod {
    NSString *reason = [NSString stringWithFormat:
                        @"%@ must be overridden",
                        NSStringFromSelector(_cmd)];
    @throw [NSException
        exceptionWithName:NSInternalInconsistencyException
                   reason:reason
                 userInfo:nil];
}
```

Но если исключения должны использоваться только для фатальных ошибок, что делать с другими ошибками? В парадигме, принятой в Objective-C, нефатальные ошибки обозначаются либо возвращением `nil` / `0` из методов, в которых произошла ошибка, либо использованием `NSError`. В качестве примера возвращения `nil` / `0` можно привести ситуацию, когда в инициализаторе экземпляр невозможно инициализировать переданным параметром:

```
- (id)initWithValue:(id)value {
    if ((self = [super init])) {
        if ( /* С этим значением экземпляр не может быть создан
*/ ) {
            self = nil;
        } else {
            // Инициализация экземпляра
        }
    }
    return self;
}
```

Если команда `if` определяет, что экземпляр не может быть создан с переданным значением (возможно, значение должно быть отлично от `nil`), `self` присваивается `nil` и это значение возвращается

методом. Сторона, вызвавшая инициализатор, поймет, что произошла ошибка, потому что экземпляр не был создан.

Объекты `NSError` обладают существенно большей гибкостью, потому что они позволяют вернуть информацию о причине ошибки. В объекте `NSError` инкапсулируются три информационные составляющие.

Область возникновения ошибки (`String`)

Область, в которой произошла ошибка. Обычно это глобальная переменная, которая может использоваться для однозначного определения источника ошибки. Например, подсистема обработки URL-адресов использует область `NSURLErrorDomain` для всех ошибок, возникающих при разборе или получении данных из URL-адресов.

Код ошибки (`Integer`)

Код, однозначно идентифицирующий ошибку в области, в которой она произошла. Часто для определения набора возможных ошибок используется перечисление. Например, для неудачных запросов HTTP в качестве кода ошибки может использоваться код состояния HTTP.

Информация для пользователя (`Dictionary`)

Дополнительная информация об ошибке: локализованное описание; другая ошибка, ставшая причиной возникновения текущей, и т. д.

Первый распространенный способ использования ошибок при проектировании API — протоколы делегатов. При возникновении ошибки объект может передать своему делегату ошибку через один из методов протокола. Например, у класса `NSURLConnection` в протокол делегата `NSURLConnectionDelegate` включается следующий метод:

```
- (void)connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error
```

Когда подключение решает, что произошла ошибка (например, тайм-аут при связи с удаленным сервером), оно вызывает этот метод с передачей кода ошибки, описывающего ситуацию. Реализация этого метода делегата не обязательна; пользователь класса `NSURLConnection` сам решает, необходимо ли ему знать об ошибке. Такой механизм лучше выдачи исключения, потому что *пользователь сам решает, нужно ли ему получать уведомления об ошибке.*

Еще один распространенный вариант использования NSError — передача объекта в выходном параметре метода:

```
- (BOOL)doSomething:(NSError**)error
```

Переменная error, передаваемая методу, содержит указатель на NSError. Фактически это позволяет методу вернуть объект NSError в дополнение к возвращаемому значению. Пример использования:

```
NSError *error = nil;
BOOL ret = [object doSomething:&error];
if (error) {
    // Произошла ошибка
}
```

Методы, возвращающие ошибку, также часто возвращают логический признак успеха или неудачи. Таким образом, если пользователя не интересует конкретная ошибка, он проверяет логический статус, а если интересует — обращается к возвращенной ошибке. Параметр error также может быть равен nil, если вы не хотите получить информацию об ошибке, как в следующем фрагменте. Пример использования метода:

```
BOOL ret = [object doSomething:nil];
if (ret) {
    // Произошла ошибка
}
```

На практике при использовании ARC компилятор преобразует NSError** в сигнатуре метода в NSError*__autoreleasing*; это означает, что объект, на который ссылается указатель, будет автоматически освобожден в конце метода. Это необходимо, потому что метод doSomething: не может гарантировать, что вызывающая сторона сможет освободить созданный ею объект NSError, и поэтому для него приходится добавить автоматическое освобождение. Итоговая семантика эквивалентна семантике возвращаемых значений для большинства методов (исключая, конечно, методы, имена которых начинаются с new, alloc, copy и mutableCopy).

Метод передает объект ошибки через выходной параметр следующим образом:

```
- (BOOL)doSomething:(NSError**)error {
    // Действия, которые могут привести к ошибке

    if ( /* ошибка произошла */ ) {
```

```
        if (error) {
            // Передать 'error' в выходном параметре
            *error = [NSError errorWithDomain:domain
                                         code:code
                                     userInfo:userInfo];
        }
        return NO; ///< Вернуть код неудачи
    } else {
        return YES; ///< Вернуть код успеха
    }
}
```

Параметр `error` разыменовывается с использованием синтаксиса `*error`, то есть значение, на которое указывает `error`, заменяется новым объектом `NSError`. Сначала необходимо убедиться в том, что параметр `error` отличен от `nil`, поскольку разыменование неопределенного указателя приведет к ошибке сегментации и сбою программы. Так как вызывающая сторона может передать `nil`, эту проверку необходимо выполнить, на случай если ее не интересует информация об ошибке.

Поля области, кода и информации для пользователя объекта ошибки должны быть заполнены данными, имеющими смысл для возникшей ошибки. Это позволяет вызывающей стороне выбирать поведение в зависимости от типа ошибки. Область лучше всего определить глобальной константой `NSString`, а коды ошибок — перечислимым типом. Например, определение может выглядеть так:

```
// EOCErrors.h
extern NSString *const EOCErrorDomain;

typedef NS_ENUM(NSUInteger, EOCError) {
    EOCErrorUnknown                 = -1,
    EOCErrorInternalInconsistency = 100,
    EOCErrorGeneralFault            = 105,
    EOCErrorBadInput                = 500,
};
// EOCErrors.m
NSString *const EOCErrorDomain = @"EOCErrorDomain";
```

Определение области возникновения ошибки для вашей библиотеки — разумный шаг. Пользователь будет уверен в том, что создаваемые и возвращаемые объекты `NSError` происходят от вашей библиотеки. Создание перечислимого типа для кодов ошибок также полезно, поскольку он документирует ошибки и связывает коды с содержательными именами. Возможно, в заголовочный файл,

в котором они определяются, стоит добавить еще более подробные описания всех видов ошибок.

УЗЕЛКИ НА ПАМЯТЬ

- Используйте исключения только для фатальных ошибок, которые приводят к аварийному завершению приложения.
- Для нефатальных ошибок либо предоставьте метод делегата для обработки ошибки, либо передайте объект `NSError` в выходном параметре.

22 РАЗБЕРИТЕСЬ С ПРОТОКОЛОМ NSCOPYING

Копирование относится к числу самых распространенных операций с объектами. В Objective-C оно выполняется методом `copy`. Чтобы обеспечить поддержку копирования ваших классов, реализуйте протокол `NSCopying`, который содержит всего один метод:

```
- (id)copyWithZone:(NSZone*)zone
```

Сигнатура напоминает о тех днях, когда класс `NSZone` использовался для сегментации памяти по зонам, а объекты создавались в определенной зоне. В наши дни каждое приложение использует всего одну зону. Таким образом, даже несмотря на необходимость реализации этого метода, вам не нужно беспокоиться о смысле параметра `zone`.

Метод `copy` реализуется в `NSObject` и просто вызывает `copyWithZone:` с зоной по умолчанию. Не забывайте, что, как бы вам ни хотелось переопределить `copy`, реализовать нужно `copyWithZone:`.

Итак, для поддержки копирования необходимо заявить о реализации протокола `NSCopying` и предоставить реализацию единственного метода этого протокола. Для примера возьмем класс, представляющий человека. В определении интерфейса вы объявляете о том, что ваш класс реализует `NSCopying`:

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

@interface EOCPerson : NSObject <NSCopying>

@property (nonatomic, copy, readonly) NSString *firstName;
```

```
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;

@end
```

Затем реализуется единственный обязательный метод этого протокола:

```
- (id)copyWithZone:(NSZone*)zone {
    EOCPerson *copy = [[[self class] allocWithZone:zone]
                       initWithFirstName:_firstName
                             andLastName:_lastName];
    return copy;
}
```

В этом примере вся работа по инициализации копии просто передается основному инициализатору. Иногда с копией требуется выполнить дополнительную работу — скажем, если класс содержит другие структуры данных, не задаваемые в инициализаторе. Допустим, класс `EOCPerson` содержит массив, с которым выполняются операции включения и исключения друзей (другие объекты `EOCPerson`). В этом случае вместе с объектом также необходимо скопировать массив друзей. Далее приводится полный пример такой реализации:

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

@interface EOCPerson : NSObject <NSCopying>

@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;

- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;

@end

@implementation EOCPerson {
    NSMutableSet *_friends;
}

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName {
```

```
    if ((self = [super init])) {
        _firstName = [firstName copy];
        _lastName = [lastName copy];
        _friends = [NSMutableSet new];
    }
    return self;
}

- (void)addFriend:(EOCPerson*)person {
    [_friends addObject:person];
}

- (void)removeFriend:(EOCPerson*)person {
    [_friends removeObject:person];
}

- (id)copyWithZone:(NSZone*)zone {
    EOCPerson *copy = [[[self class] allocWithZone:zone]
                       initWithFirstName:_firstName
                              andLastName:_lastName];
    copy->_friends = [_friends mutableCopy];
    return copy;
}

@end
```

На этот раз метод копирования дополнительно задает переменной экземпляра `_friends` объекта-копии копию своей переменной экземпляра `_friends`. Обратите внимание на использование синтаксиса `->`, так как переменная экземпляра `_friends` является внутренней. Для нее также можно было бы объявить свойство, но поскольку переменная никогда не используется извне, в таком объявлении нет необходимости.

В связи с этим примером возникает интересный вопрос: зачем создавать копию переменной экземпляра `_friends`? С таким же успехом можно было этого не делать, чтобы оба объекта совместно использовали одно изменяемое множество. Но тогда при добавлении друга к исходному объекту он, словно по волшебству, становился бы другом копии. Безусловно, в данной ситуации это нежелательно. Но если бы множество было неизменяемым, от копирования можно было бы отказаться. Это избавило бы нас от необходимости хранить в памяти два идентичных множества.

Обычно для инициализации копии используется основной инициализатор, как в приведенном примере. Исключение составляют ситуации с наличием у основного инициализатора побочного

эффекта, который не должен применяться к копии (скажем, создание сложных внутренних структур данных, которые немедленно будут перезаписаны).

Вернувшись к методу `copyWithZone:`, мы видим, что множество друзей копируется методом `mutableCopy`. Он входит в другой протокол, который называется `NSMutableCopying`. Этот протокол похож на `NSCopying`, но в нем определяется следующий метод:

```
- (id)mutableCopyWithZone:(NSZone*)zone
```

Вспомогательный метод `mutableCopy` похож на `copy`; он тоже вызывает предыдущий метод с зоной по умолчанию. Реализуйте `NSMutableCopying`, если ваш класс существует в изменяемом и неизменяемом вариантах. При использовании этой схемы не следует переопределять `copyWithZone:` в изменяемом классе для возвращения изменяемой копии. Вместо этого, если вам потребуется создать изменяемую копию изменяемого или неизменяемого экземпляра, используйте `mutableCopy`. Аналогичным образом для создания неизменяемой копии используйте `copy`.

Следующие условия истинны для `NSArray` и `NSMutableArray`:

```
-[NSMutableArray copy] => NSArray
-[NSArray mutableCopy] => NSMutableArray
```

Обратите внимание на тонкость с вызовом `copy` для изменяемого объекта и получением экземпляра другого класса — неизменяемого варианта. Это делается для того, чтобы упростить переключение между изменяемым и неизменяемым вариантами. Того же эффекта можно добиться с использованием трех методов: `copy`, `immutableCopy` и `mutableCopy`, где `copy` всегда возвращает тот же класс, а два других метода возвращают конкретные варианты. Однако такое решение уже не так хорошо работает, если вы не знаете, является ли имеющийся экземпляр неизменяемым. Вы можете вызвать `copy` для объекта, который был возвращен как `NSArray`, но на самом деле представляет собой `NSMutableArray`. В этом случае вы предполагаете, что был возвращен неизменяемый массив, тогда как на самом деле он является изменяемым.

Тип экземпляра можно определить посредством интроспекции (см. подход 14), но это усложнит код во всех местах, где выполняется копирование. Таким образом, в итоге для надежности вы всегда будете использовать `immutableCopy` или `mutableCopy`, и все возвращается к наличию всего двух методов — что равносильно использованию только `copy` и `mutableCopy`. Преимущество использования

имени copy вместо immutableCopy заключается в том, что протокол NSCopying предназначен не только для классов с изменяемыми и неизменяемыми вариантами, но и для случаев, когда такого различия не существует, и тогда имя immutableCopy выглядит явно неуместно.

Также необходимо выбрать, должен метод копирования выполнять глубокое или поверхностное копирование. При глубоком копировании также копируются все данные, задействованные в объекте. По умолчанию для классов коллекций в Foundation используется поверхностное копирование, при котором копируется только контейнер, но не хранящиеся в нем данные. В основном это связано с тем, что копирование объектов в контейнере может оказаться невозможным; кроме того, обычно копировать каждый объект нежелательно. Различие между глубоким и поверхностным копированием продемонстрировано на рис. 3.2.

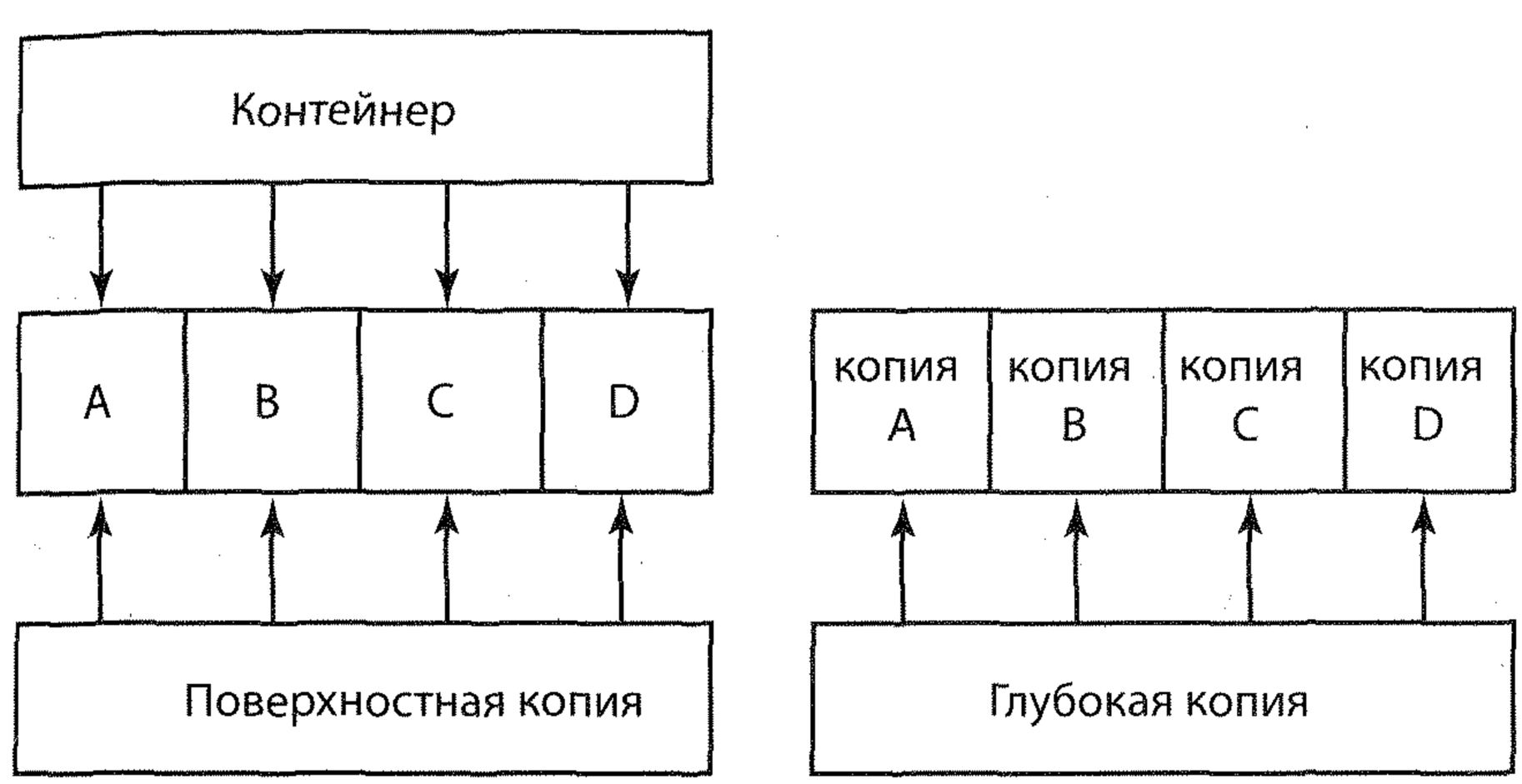


Рис. 3.2. Поверхностное и глубокое копирование. Содержимое поверхностной копии указывает на те же объекты, что и оригинал. Содержимое глубокой копии указывает на копии исходного содержимого

Обычно в ваших классах следует использовать ту же схему, что и в системных библиотеках, — с выполнением поверхностного копирования в copyWithZone:. Но при необходимости можно добавить метод для выполнения глубокого копирования. В случае NSSet эта задача решается следующим методом-инициализатором:

```
- (id)initWithSet:(NSArray*)array copyItems:(BOOL)copyItems
```

Если copyItems задано значение YES, то для формирования нового возвращаемого множества элементам массива отправляется сообщение copy для создания копий.

В примере с классом EOCPerson множество, содержащее информацию о друзьях, копируется в copyWithZone:, но, как упоминалось ранее, сами элементы множества при этом не копируются. Но если бы глубокое копирование было необходимым, можно было бы предоставить метод следующего вида:

```
- (id)deepCopy {
    EOCPerson *copy = [[[self class] alloc]
                        initWithFirstName:_firstName
                                andLastName:_lastName];
    copy->_friends = [[NSMutableSet alloc] initWithSet:_friends
                                                copyItems:YES];
    return copy;
}
```

Глубокое копирование никаким протоколом не определяется, поэтому каждый класс должен сам определить, как будет создаваться такая копия. Просто решите, нужно ли предоставлять метод глубокого копирования в вашей ситуации. Кроме того, никогда не следует полагать, что объект, соответствующий NSCopying, будет выполнять глубокое копирование. В подавляющем большинстве случаев создаваемая копия будет поверхностной. Если вам потребуется глубокая копия объекта, то либо найдите соответствующий метод, либо считайте, что вы должны создать собственную реализацию (если только в документации не сказано, что ее реализация NSCopying предоставляет глубокую копию).

УЗЕЛКИ НА ПАМЯТЬ

- Если объекты вашего класса должны копироваться, реализуйте протокол NSCopying.
- Если ваш объект существует в изменяемом и неизменяемом вариантах, реализуйте протоколы NSCopying и NSMutableCopying.
- Решите, должно копирование быть поверхностным или глубоким. В обычных ситуациях по возможности используйте поверхностное копирование.
- Рассмотрите возможность добавления метода глубокого копирования, если создание глубоких копий ваших объектов принесет реальную пользу.

Глава 4
ПРОТОКОЛЫ И КАТЕГОРИИ

Протоколы (protocols) представляют собой функцию языка, сходную с интерфейсами в Java. В Objective-C нет множественного наследования, поэтому протоколы позволяют определять наборы методов, которые должны быть реализованы классом. Протоколы чаще всего используются в реализации паттерна «Делегат» (см. подход 23), но также существуют и другие применения. Если вы будете знать и использовать их, это существенно упростит сопровождение вашего кода, так как протоколы являются хорошим средством документирования интерфейса вашего кода.

Категории также входят в число важных языковых возможностей Objective-C. Они предоставляют механизм добавления методов в класс без применения субклассирования, необходимого в других языках. Динамическая природа исполнительной среды сделала возможным применение категорий, но в ней же кроются некоторые ловушки, которые необходимо изучить перед использованием категорий.

23 ИСПОЛЬЗУЙТЕ ПРОТОКОЛЫ ДЕЛЕГАТОВ И ИСТОЧНИКОВ ДАННЫХ ДЛЯ ВЗАИМОДЕЙСТВИЯ МЕЖДУ ОБЪЕКТАМИ

Объектам часто требуется взаимодействовать друг с другом; существует много вариантов организации таких взаимодействий. В частности, разработчики Objective-C часто применяют паттерн

«Делегат». Суть этого паттерна заключается в определении интерфейса, который должен поддерживаться объектом, для того чтобы стать делегатом другого объекта. Этот другой объект обращается к своему делегату за информацией или сообщает ему о том, что происходит что-то важное.

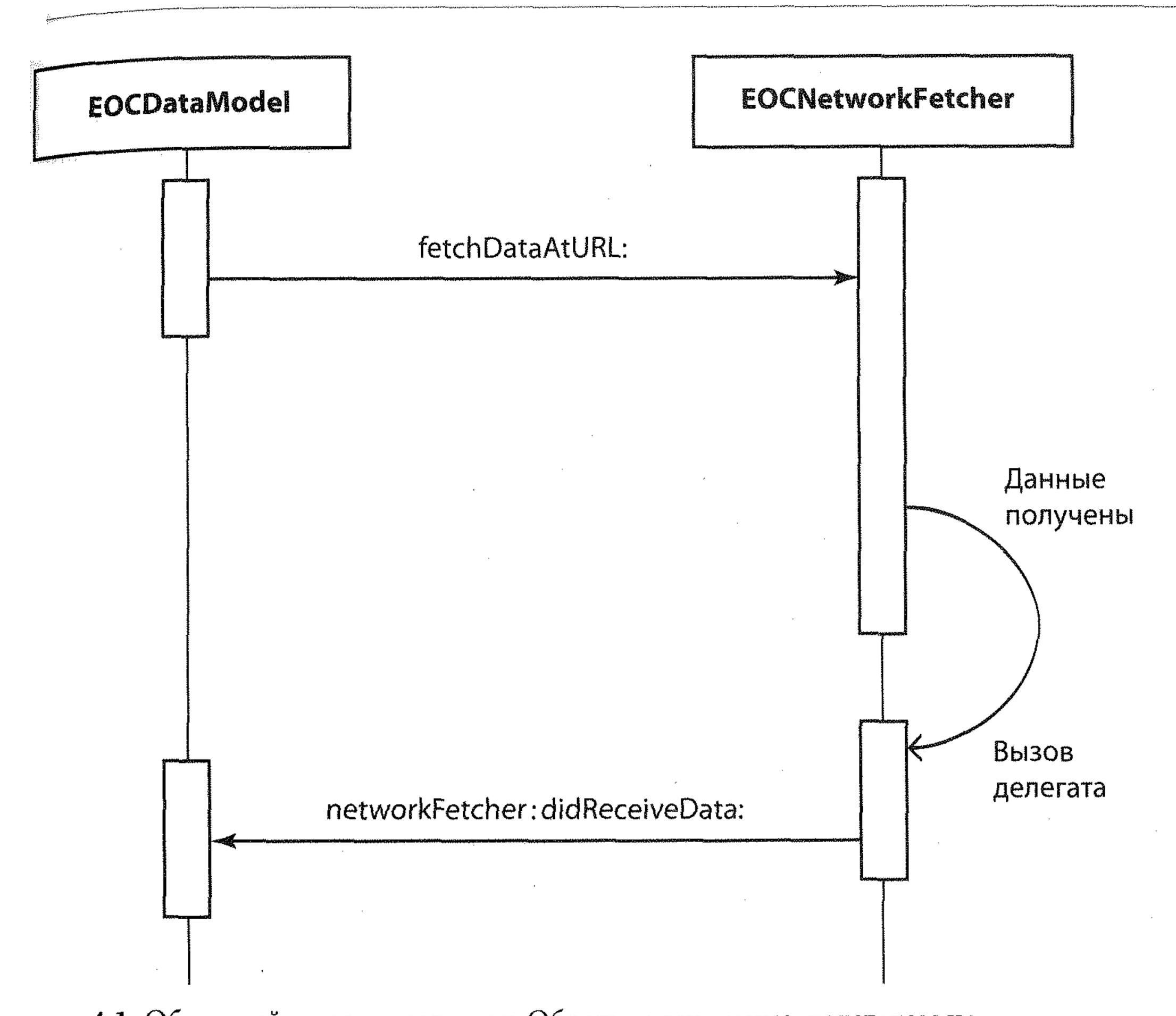
Применение паттерна способствует отделению данных от бизнес-логики. Например, представление, предназначенное для вывода списка данных, должно отвечать только за логику вывода данных, а не за принятие решений о том, какие данные должны выводиться или что должно происходить при взаимодействии пользователя с данными. Объект представления может обладать свойствами, в которых содержатся объекты, ответственные за работу с данными и обработку событий. Эти объекты называются *источником данных* (data source) и *делегатом* (delegate) соответственно.

В Objective-C реализация этого паттерна обычно базируется на протоколах — в частности, эта функция языка широко используется в библиотеках, входящих в Cocoa. Используя протоколы в своем коде, вы увидите, что этот паттерн естественно подходит для многих ситуаций.

В качестве примера рассмотрим класс, предназначенный для загрузки данных из сети. Данные могут загружаться из ресурса на удаленном сервере. Сервер может ответить не сразу, поэтому блокировка выполнения до получения данных нежелательна. В таких ситуациях часто используется паттерн «Делегат»: у класса, загружающего данные, определяется делегат, к которому он обращается с обратным вызовом после получения данных. Эта концепция продемонстрирована на рис. 4.1; объект `EOCDataModel` является делегатом для `EOCNetworkFetcher`. Объект `EOCDataModel` приказывает `EOCNetworkFetcher` выполнить задачу в асинхронном режиме, а `EOCNetworkFetcher` сообщает своему делегату `EOCDataModel` о том, что работа завершена.

Этот паттерн легко реализуется на Objective-C с использованием протокола. Для примера на рис. 4.1 протокол может выглядеть так:

```
@protocol EOCNetworkFetcherDelegate
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data;
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didFailWithError:(NSError*)error;
@end
```



4.1. Обратный вызов к делегату. Обратите внимание: делегатом не ан быть экземпляр EOCDataModel, это может быть другой объект

Имя протокола делегата обычно состоит из имени класса и суффикса `Delegate`, с чередованием регистра символов. При соблюдении стандартной схемы протокол делегата будет выглядеть знакомо для разработчиков, которые его используют.

Протокол предоставляет свойство в классе, имеющем делегата. В нашем примере это класс `EOCNetworkFetcher`. Следовательно, интерфейс будет выглядеть так:

```
@interface EOCNetworkFetcher : NSObject
@property (nonatomic, weak)
    id <EOCNetworkFetcherDelegate> delegate;
@end
```

Важно проследить за тем, чтобы свойство определялось с атрибутом `weak`, а не `strong`, потому что создается отношение без принадлежности. Обычно объект, который является делегатом, удерживает

делегирующий объект. Например, объект, желающий использовать объект `EOCNetworkFetcher`, удерживает его до завершения работы с ним. Если бы свойство, удерживающее делегата, было объявлено с атрибутом `strong` (отношение принадлежности), то в программе образовался бы цикл удержания. По этой причине свойство делегата всегда объявляется с атрибутом `weak`, чтобы использовать преимущества автоматического обнуления (autonilling) (см. подход 6) или `unsafe_unretained`, если автоматическое обнуление не нужно. Диаграмма принадлежности на рис. 4.2 поясняет сказанное.

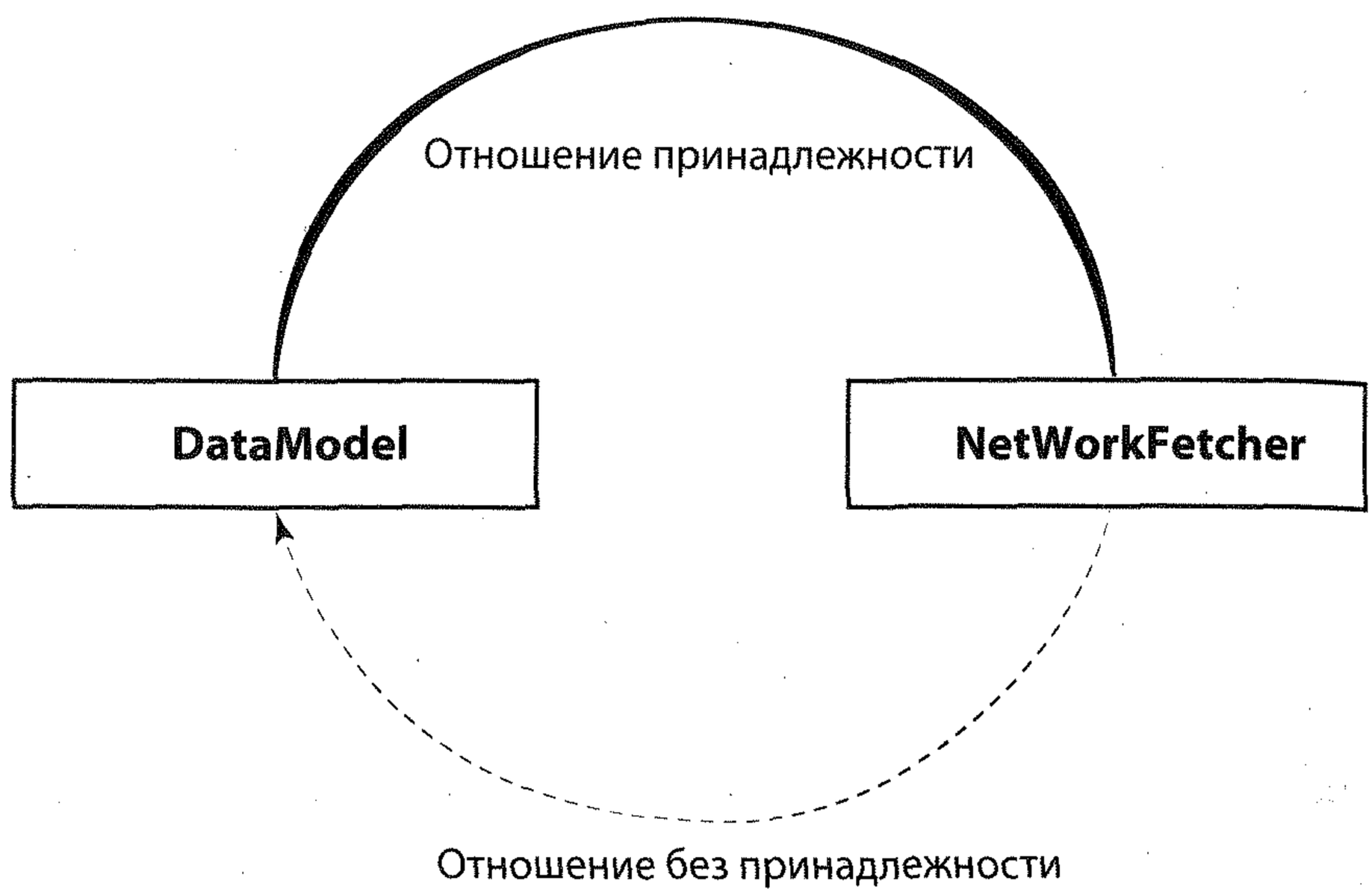


Рис. 4.2. Диаграмма принадлежности с предотвращением цикла удержания

Реализация делегата сводится к объявлению того, что ваш класс реализует протокол делегата, и реализации всех нужных методов протокола. О том, что класс реализует протокол, можно объявить в интерфейсе или в категории продолжения класса (см. подход 27). Объявление в интерфейсе полезно, если вы хотите сообщить пользователям о реализации протокола; тем не менее в случае делегатов эта информация обычно используется только внутри класса. Соответственно на практике обычно используется объявление в категории продолжения класса:

```
@implementation EOCDataModel () <EOCNetworkFetcherDelegate>
@end
@implementation EOCDataModel
```

```
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data {
    /* Обработка данных */
}
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didFailWithError:(NSError*)error {
    /* Обработка ошибки */
}
@end
```

Обычно методы, входящие в протокол делегата, не являются обязательными, поскольку некоторые методы могут не представлять интереса для делегата. В нашем примере класс `DataModel` может не обращать внимания на ошибку и не реализовать метод `networkFetcher:didFailWithError:`. По этой причине в протоколах делегатов многие (или все) методы объявляются необязательными при помощи ключевого слова `@optional`:

```
@protocol EOCNetworkFetcherDelegate
@optional
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data;
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didFailWithError:(NSError*)error;
@end
```

Прежде чем вызывать необязательный метод делегата, следует определить, реагирует ли делегат на этот селектор, при помощи интроспекции (см. подход 14). В случае `EOCNetworkFetcher` это выглядит так:

```
NSData *data = /* данные, полученные из сети */;
if ([_delegate respondsToSelector:
          @selector(networkFetcher:didReceiveData:)])
{
    [_delegate networkFetcher:self didReceiveData:data];
}
```

Метод `respondsToSelector:` используется для проверки того, реализуется ли метод делегатом. Если метод реализуется, то он вызывается; в противном случае не происходит ничего. При такой реализации метод делегата действительно необязателен, и, если он отсутствует, в программе ничего не сломается. И даже если делегат не назначен, все будет работать, потому что при отправке сообщения `nil` условие `if` будет ложным.

Очень важно правильно сформировать имена методов делегатов. В имени должно быть четко указано, что происходит и почему делегату передается информация. В нашем примере имя метода делегата предельно понятно: оно сообщает, что объект получения данных из Сети только что получил какие-то данные. Также всегда следует передавать делегирующий экземпляр, как в приведенном примере, чтобы реализацию метода делегата можно было переключать в зависимости от конкретного экземпляра. Пример:

```
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data
{
    if (fetcher == _myFetcherA) {
        /* Обработка данных */
    } else if (fetcher == _myFetcherB) {
        /* Обработка данных */
    }
}
```

Здесь объект, являющийся делегатом, использует два объекта получения данных, поэтому он должен знать, какой из двух объектов сообщает ему о поступлении данных. Без этой информации он мог бы использовать только один объект получения данных, а это нежелательно.

Методы делегата также могут использоваться для получения информации о делегате. Например, при перенаправлении в ходе загрузки данных класс может запросить у своего делегата, следует ли разрешить перенаправление. Соответствующий метод делегата может выглядеть примерно так:

```
- (BOOL)networkFetcher:(EOCNetworkFetcher*)fetcher
        shouldFollowRedirectToURL:(NSURL*)url;
```

Пример наглядно показывает, почему паттерн называется «Делегат»: объект делегирует ответственность за выполнение действия другому классу.

Протоколы также могут предоставлять интерфейс, через который передаются данные, необходимые другому классу. Это применение паттерна «Делегат» обычно называется паттерном «Источник данных», так как его целью является предоставление данных классу. Таким образом данные передаются в направлении класса, тогда как у традиционных делегатов они передаются в направлении от класса (рис. 4.3).

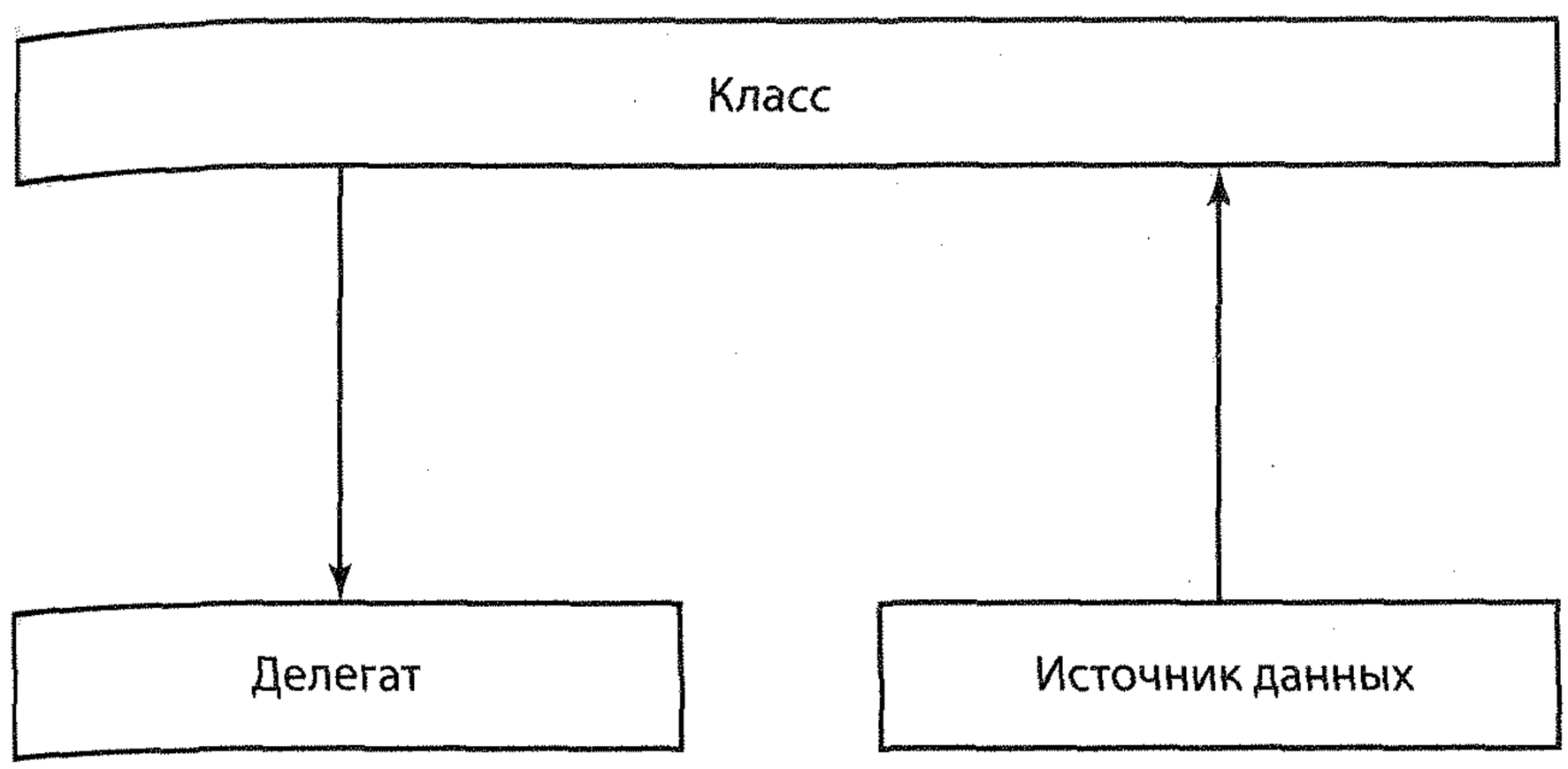


Рис. 4.3. Делегат передает информацию в направлении от класса, а источник данных — в направлении к классу

Например, объект представления списка в библиотеке пользовательского интерфейса может использовать протокол источника данных для поставки данных, выводимых в списке. У представления списка также может быть определен делегат, обрабатывающий действия пользователя со списком. Разделение протоколов источника данных и делегата способствует изоляции разнородной логики и как следствие — формированию более четкого интерфейса. Кроме того, источником данных может быть один объект, а делегатом — другой. Впрочем, на практике обе функции обычно выполняются одним объектом.

В паттернах «Делегат» и «Источник данных», в которых реализация многих методов не является обязательной, часто встречается код следующего вида:

```
if ([_delegate respondsToSelector:
          @selector(someClassDidSomething:)])
{
    [_delegate someClassDidSomething];
}
```

Проверить, реагирует ли делегат на некоторый селектор, недолго, но при многократном выполнении этой операции все ответы после первого становятся избыточными. Если делегат не изменялся, крайне маловероятно, что он вдруг неожиданно начал или перестал реагировать на селектор. По этой причине обычно применяется оптимизация кэширования информации о том, реагирует ли делегат на методы протокола. Например, у объекта получения данных в нашем примере имеется метод, обратные вызовы которого

передают делегату информацию о прогрессе операции. Этот метод может многократно вызываться в ходе жизненного цикла объекта. Каждый раз проверять, отвечает ли делегат на этот селектор, было бы неэффективно.

Рассмотрим расширенный протокол делегата, определяемый следующим образом:

```
@protocol EOCNetworkFetcherDelegate
@optional
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didReceiveData:(NSData*)data;
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didFailWithError:(NSError*)error;
- (void)networkFetcher:(EOCNetworkFetcher*)fetcher
        didUpdateProgressTo:(float)progress;
@end
```

Здесь вызывается один необязательный метод `networkFetcher:didUpdateProgressTo:`. Кэширование лучше всего реализовать на базе типа данных битового поля. Об этой возможности C часто забывают, но для этой цели она подходит просто идеально: разработчик определяет, что некоторое поле структуры должно занимать определенное количество бит. Это выглядит так:

```
struct data {
    unsigned int fieldA : 8;
    unsigned int fieldB : 4;
    unsigned int fieldC : 2;
    unsigned int fieldD : 1;
};
```

В этой структуре для хранения `fieldA` используются ровно 8 бит, для `fieldB` — 4 бита, `fieldC` — 2 бита и `fieldD` — 1 бит. Таким образом, в `fieldA` могут храниться значения от 0 до 255, а в `fieldD` — только 0 или 1. И последняя возможность пригодится для кэширования информации о том, какие методы реализуются делегатом. Создав структуру, состоящую из однобитовых полей, можно упаковать большое количество логических флагов в малый объем данных. В примере с получением данных из Сети в одной из переменных экземпляра размещается структура с битовым полем, по одной переменной для каждого метода делегата. Структура выглядит примерно так:

```
@interface EOCNetworkFetcher () {
    struct {
        unsigned int didReceiveData      : 1;
```

```
    unsigned int didFailWithError    : 1;
    unsigned int didUpdateProgressTo : 1;
  } _delegateFlags;
}
@end
```

Здесь для добавления переменной экземпляра используется категория продолжения класса (см. подход 27). Добавленная переменная экземпляра представляет собой структуру из трех полей, по одному для каждого из необязательных методов делегата. Пример чтения и записи полей структуры в классе `EOCNetworkFetcher`:

```
// Установка флага
_delegateFlags.didReceiveData = 1;
// Проверка флага
if (_delegateFlags.didReceiveData) {
    // Да, флаг установлен
}
```

Эта структура кэширует информацию о том, реагирует ли делегат на селекторы. Проверка может осуществляться в set-методе свойства делегата:

```
- (void)setDelegate:(id<EOCNetworkFetcher>)delegate {
    _delegate = delegate;
    _delegateFlags.didReceiveData =
      [delegate respondsToSelector:
                @selector(networkFetcher:didReceiveData:)];
    _delegateFlags.didFailWithError =
      [delegate respondsToSelector:
                @selector(networkFetcher:didFailWithError:)];
    _delegateFlags.didUpdateProgressTo =
      [delegate respondsToSelector:
               @selector(networkFetcher:didUpdateProgressTo:)];
}
```

Затем вместо того, чтобы проверять селектор при каждом вызове метода делегата, мы проверяем соответствующий флаг:

```
if (_delegateFlags.didUpdateProgressTo) {
    [_delegate networkFetcher:self
          didUpdateProgressTo:currentProgress];
}
```

При большом количестве вызовов такая оптимизация оправданна. Ее эффективность зависит от специфики кода. Выполните

профилирование своего кода, найдите в области активного использования участки кода и придержите их на будущее для возможного повышения быстродействия. Скорее всего, оптимизация хорошо сработает в протоколах источников данных, в которых источник данных постоянно получает запросы для отдельных блоков данных.

УЗЕЛКИ НА ПАМЯТЬ

- Используйте паттерн «Делегат» для формирования интерфейса к объектам, которые должны сообщать другим объектам об актуальных событиях.
- Определите протокол с (возможно) необязательными методами для определения интерфейса, который должен поддерживаться делегатами.
- Используйте паттерн «Делегат», когда объект должен получать данные от другого объекта. В таких ситуациях часто используется термин «протокол источника данных».
- При необходимости реализуйте структуру с битовыми полями для кэширования информации о том, на какие методы протокола реагирует делегат.

24 ИСПОЛЬЗУЙТЕ КАТЕГОРИИ ДЛЯ РАЗБИЕНИЯ КЛАССОВ

Классы быстро разрастаются в один огромный файл реализации, заполненный многочисленными методами. Иногда с этим сделать ничего не удается, и никакой рефакторинг не улучшит ситуацию. В таких случаях можно воспользоваться категориями Objective-C для разбиения класса на логические подразделы, упрощающие не только разработку, но и отладку.

Для примера возьмем класс, моделирующий человека. В таком классе могут быть определены следующие методы:

```
#import <Foundation/Foundation.h>
@interface EOCPerson : NSObject

@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;
@property (nonatomic, strong, readonly) NSArray *friends;
```

```
- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;

/* Методы управления друзьями */
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;
- (BOOL)isFriendsWith:(EOCPerson*)person;

/* Методы для работы */
- (void)performDaysWork;
- (void)takeVacationFromWork;

/* Методы для развлечения */
- (void)goToTheCinema;
- (void)goToSportsGame;

@end
```

Реализация такого класса представляет собой длинный список методов в одном большом файле. С добавлением в класс новых методов файл становится только длиннее и неудобнее, поэтому классы бывает полезно разделить на изолированные части. Например, приведенный пример можно переписать с использованием категорий:

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

@interface EOCPerson : NSObject
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;
@property (nonatomic, strong, readonly) NSArray *friends;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;
@end

@interface EOCPerson (Friendship)
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;
- (BOOL)isFriendsWith:(EOCPerson*)person;
@end

@interface EOCPerson (Work)
- (void)performDaysWork;
- (void)takeVacationFromWork;
@end
```

```
@interface EOCPerson (Play)
- (void)goToTheCinema;
- (void)goToSportsGame;
@end
```

Класс разбивается на несколько частей, каждая из которых содержит отдельную *категорию* методов. В приведенном примере основа класса, включая свойства и инициализатор, объявляется в главной реализации. Дополнительные наборы методов, соответствующие разным типам действий, разбиваются на категории.

Как и прежде, весь класс может быть определен в одном файле интерфейса и одном файле реализации, но с увеличением количества категорий один файл реализации быстро выходит из-под контроля. В этом случае категории могут выделяться в собственные файлы. Например, для категорий класса `EOCPerson` может быть определен следующий набор файлов:

- EOCPerson+Friendship(.h/.m)
- EOCPerson+Work(.h/.m)
- EOCPerson+Play(.h/.m)

Категория управления друзьями будет выглядеть примерно так:

```
// EOCPerson+Friendship.h
#import "EOCPerson.h"

@interface EOCPerson (Friendship)
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;
- (BOOL)isFriendsWith:(EOCPerson*)person;
@end

// EOCPerson+Friendship.m
#import "EOCPerson+Friendship.h"

@implementation EOCPerson (Friendship)
- (void)addFriend:(EOCPerson*)person {
    /* … */
}
- (void)removeFriend:(EOCPerson*)person {
    /* … */
}
- (BOOL)isFriendsWith:(EOCPerson*)person {
    /* … */
}
@end
```

Класс разделен на более удобные блоки кода, которые можно обрабатывать по отдельности. Конечно, вы должны помнить о том, что заголовок EOCPerson.h должен импортироваться везде, где нужны методы категории, и все же это хороший способ упростить работу с кодом.

Даже если класс не слишком велик, использование категорий может стать полезным способом разбиения кода на функциональные области. Например, в Cocoa такой подход использует класс `NSURLRequest` и его изменяемый аналог `NSMutableURLRequest`. Этот класс выполняет запросы для получения данных от URL-адреса. В основном он используется с протоколом HTTP для получения данных с серверов в Интернете, но вообще класс является универсальным и может использоваться с другими протоколами. Однако для запросов HTTP должна задаваться дополнительная информация помимо той, которая необходима для стандартных запросов URL-адресов: метод HTTP (GET, POST и т. д.), заголовки HTTP и т. д.

Но субклассировать `NSURLRequest` достаточно сложно, потому что этот класс инкапсулирует функции C для работы со структурой данных `CFURLRequest`, включая полный набор методов HTTP. Таким образом, методы HTTP просто добавляются в `NSURLRequest` как категория с именем `NSHTTPURLRequest`, а изменяющие методы добавляются в `NSMutableURLRequest` как категория с именем `NSMutableHTTPURLRequest`. Таким образом, все нижележащие функции `CFURLRequest` инкапсулируются в одном классе Objective-C, а методы, относящиеся к HTTP, выделяются в отдельную область для предотвращения путаницы (чтобы пользователи не удивлялись, почему они могут применить метод HTTP к объекту запроса, использующему протокол FTP).

Разбиение класса на категории также приносит пользу для отладки; имя категории включается в символические имена всех методов категории. Например, метод `addFriend:` обладает следующим символическим именем:

```
-[EOCPerson(Friendship) addFriend:]
```

В данных трассировки отладчика он выглядит примерно так:

```
frame #2: 0x00001c50 Test'-[EOCPerson(Friendship) addFriend:]
+ 32 at main.m:46
```

Имя категории может быть чрезвычайно полезно для уточнения функциональной области класса, к которой принадлежит метод, — особенно если некоторые методы должны рассматриваться как

закрытые. В таком случае может быть полезно создать категорию `Private`, в которую включаются все эти методы. Такие методы категорий обычно используются только во внутренних операциях класса или библиотеки и не предназначаются для внешнего использования. Если пользователь обнаруживает такой метод (например, во время чтения данных трассировки), слово `private` в имени поможет ему понять, что метод не должен вызываться напрямую. В каком-то смысле это механизм создания самодокументируемого кода.

Категория `Private` полезна при создании библиотеки, которая будет использоваться другими разработчиками. Часто некоторые методы не должны входить в открытый API, но вполне могут использоваться в самой библиотеке. В такой ситуации создание категории `Private` окажется хорошим вариантом, поскольку ее заголовок может импортироваться во всех местах использования методов в библиотеке. Если заголовок категории не публикуется при выпуске библиотеки, пользователи библиотеки понятия не имеют о существовании этих закрытых методов.

УЗЕЛКИ НА ПАМЯТЬ

- Используйте категории для разбиения реализации класса на более удобные фрагменты.
- Создайте категорию `Private` для сокрытия реализации методов, которые должны рассматриваться как закрытые.

25 ВСЕГДА ИСПОЛЬЗУЙТЕ ПРЕФИКСЫ ИМЕН КАТЕГОРИЙ В КЛАССАХ, ПРЕДНАЗНАЧЕННЫХ ДЛЯ ВНЕШНЕГО ИСПОЛЬЗОВАНИЯ

Категории часто используются для расширения функциональности существующих классов, исходный код которых вам недоступен. Это исключительно мощная возможность, но при ее использовании очень легко упустить одну потенциальную проблему. Эта проблема обусловлена тем фактом, что методы категории включаются в класс так, как если бы они были частью самого класса. Это происходит во время выполнения при загрузке категории. Исполнительная среда

перебирает методы, реализованные категорией, и добавляет их в список методов класса. Если метод из категории уже существует, то реализация из категории заменяет существующую реализацию. Такое переопределение может происходить снова и снова, так что метод из одной категории может переопределить метод из другой категории, которая переопределяет метод из основной реализации класса. Категория, загружаемая последней, замещает остальные определения.

Предположим, вы решили добавить в `NSString` категорию со вспомогательными методами для кодирования/декодирования строк URL-адресов протокола HTTP. Определение категории может выглядеть так:

```
@interface NSString (HTTP)

// URL-кодирование строки
- (NSString*)urlEncodedString;

// URL-декодирование
- (NSString*)urlDecodedString;
@end
```

Определение выглядит абсолютно нормально, но представьте, что произойдет, если другая категория также добавляет методы в `NSString`. Вторая категория может добавить метод с именем `urlEncodedString`, реализованный несколько иначе и не подходящий для ваших целей. Если вторая категория будет загружаться после вашей, она «победит» — и ваш код будет вызывать реализацию, определенную в этой категории. В результате ваш код начинает работать некорректно, и вы получаете неожиданные результаты. Такие ошибки бывает трудно выявить, потому что вы можете не знать, что выполняемый код не является вашей реализацией `urlEncodedString`.

Обычно эта проблема решается формированием пространства имен для категории и определяемых ей методов. В Objective-C пространства имен моделируются при помощи префиксов. По аналогии с тем, как это делается для классов (см. подход 15), следует выбрать префикс, подходящий для ситуации. Часто он совпадает с префиксом, используемым в других частях приложения или библиотеки. Таким образом, с префиксом `ABC` категория для `NSString` принимает следующий вид:

```
@interface NSString (ABC_HTTP)

// URL-кодирование
```

```
- (NSString*)abc_urlEncodedString;

// URL-декодирование
- (NSString*)abc_urlDecodedString;

@end
```

Формально имя самой категории не обязательно снабжать префиксом. Даже если в программе используются две одноименные категории, ничего страшного не произойдет. Но поступать так не рекомендуется, и компилятор выдаст предупреждение следующего вида:

```
warning: duplicate definition of category 'HTTP' on interface
'NSString'
```

Переопределение вашего метода в другой категории остается возможным, но его вероятность заметно снижается, поскольку другая библиотека вряд ли будет использовать тот же префикс. Также снижается вероятность того, что разработчик класса при последующих обновлениях добавит метод, конфликтующий с добавленным методом. Например, если компания Apple добавит в `NSString` метод `urlEncodedString`, ваш метод с тем же именем заменит метод Apple. Разумеется, это нежелательно, потому что другие пользователи `NSString` могут рассчитывать на результат реализации Apple. Или реализация Apple может включать побочные эффекты, которые отсутствуют в вашем методе; все это может порождать внутренние расхождения с возникновением трудноуловимых ошибок.

Также необходимо помнить, что методы, добавленные в класс через категорию, становятся доступными для каждого экземпляра класса в приложении. Если вы добавите методы в системный класс — например, в `NSString`, `NSArray` или `NSNumber`, каждый экземпляр этих классов сможет вызывать добавленные методы даже в том случае, если он не был создан в вашем коде. А случайное переопределение метода с использованием категории или конфликт с категорией, добавленной сторонней библиотекой, может породить странные ошибки: вы думаете, что ваш код выполняется, тогда как на самом деле этого не происходит. Наконец, намеренно переопределять методы в категории не рекомендуется — особенно в коде библиотек, используемых другими разработчиками, которые могут рассчитывать на существующую функциональность. Еще хуже, если другой разработчик переопределит тот же метод; в этом случае невозможно предсказать, какой метод «победит» в процессе переопределения.

Это еще один довод в пользу применения пространств имен к методам в категориях.

УЗЕЛКИ НА ПАМЯТЬ

- Всегда включайте префикс в имена категорий, которые вы добавляете к классам, вам не принадлежащим.
- Всегда включайте префикс в имена методов в категориях, которые вы добавляете к классам, вам не принадлежащим.

26 ИЗБЕГАЙТЕ ИСПОЛЬЗОВАНИЯ СВОЙСТВ В КАТЕГОРИЯХ

Свойство представляет собой способ инкапсуляции данных (см. подход 6). Хотя формально объявления свойств в категориях не запрещены, старайтесь избегать их, насколько это возможно. Дело в том, что в категориях, особенно в категориях продолжения классов (см. подход 27), в класс не могут добавляться новые переменные. Следовательно, категория также не может синтезировать переменную экземпляра, на базе которой работает свойство.

Предположим, после чтения подхода 24 вы решили использовать категории для разбиения вашей реализации класса, представляющего человека, на фрагменты. Для всех методов, относящихся к управлению списком друзей, выделяется отдельная категория. Не зная о проблеме, описанной выше, вы бы также могли поместить в категорию дружбы свойство для списка друзей:

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

@interface EOCPerson : NSObject
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
           andLastName:(NSString*)lastName;
@end

@implementation EOCPerson
```

```
// Методы
@end

@interface EOCPerson (Friendship)
@property (nonatomic, strong) NSArray *friends;
- (BOOL)isFriendsWith:(EOCPerson*)person;
@end

@implementation EOCPerson (Friendship)
// Методы
@end
```

При попытке откомпилировать этот код вы получите предупреждение от компилятора:

```
warning: property 'friends' requires method 'friends' to be
defined - use @dynamic or provide a method implementation in
this category [-Wobjc-property-implementation]
warning: property 'friends' requires method 'setFriends:' to be
defined - use @dynamic or provide a method implementation in
this category [-Wobjc-property-implementation]
```

Это загадочное предупреждение означает, что переменные экземпляров не могут синтезироваться категорией, а следовательно, в категории должны быть реализованы методы доступа. Также методы доступа могут объявляться с ключевым словом `@dynamic`, указывающим на то, что они будут доступны во время выполнения, но при этом не будут видны компилятору. Например, такая ситуация возникает при использовании механизма перенаправления сообщений (см. подход 12) для перехвата методов и предоставления реализации во время выполнения.

Чтобы обойти проблему невозможности синтезирования переменных экземпляров в категориях, можно воспользоваться ассоциированными объектами (см. подход 10). В нашем примере методы доступа в категории должны быть реализованы следующим образом:

```
#import <objc/runtime.h>

static const char *kFriendsPropertyKey = "kFriendsPropertyKey";

@implementation EOCPerson (Friendship)

- (NSArray*)friends {
    return objc_getAssociatedObject(self, kFriendsPropertyKey);
}
```

```
- (void)setFriends:(NSArray*)friends {
    objc_setAssociatedObject(self,
                             kFriendsPropertyKey,
                             friends,
                             OBJC_ASSOCIATION_RETAIN_
NONATOMIC);
}

@end
```

Такое решение работает, но оно отнюдь не идеально. Оно содержит много шаблонного кода и подвержено ошибкам управления памятью — слишком легко забыть, что свойство было реализовано подобным образом. Например, вы можете изменить семантику управления памятью, изменяя атрибуты свойств. Однако при этом также нельзя забыть об изменении семантики управления памятью сопутствующего объекта в set-методе. Итак, хотя это решение нельзя назвать плохим, я бы не стал его рекомендовать.

Или представьте, что вы захотели сделать переменную экземпляра, на которой базируется массив друзей, изменяемым массивом. Можно создать изменяемую копию, но это введет еще одну область потенциальных ошибок в кодовой базе. Определение свойств в главном определении интерфейса, а не в категориях, создаст куда меньше проблем.

В приведенном примере правильным решением будет хранение всех определений свойств в главном объявлении интерфейса. Все данные, инкапсулированные классом, должны определяться в главном интерфейсе — единственном месте, в котором могут определяться переменные экземпляров (данные). Так как свойства представляют собой синтаксические «украшения» для определения переменных экземпляров и связанных с ними методов доступа, они подчиняются тому же правилу. Категории следует рассматривать как способ расширения функциональности класса, а не инкапсулированных в нем данных.

С другой стороны, свойства, доступные только для чтения, могут успешно использоваться в категориях. Например, можно создать для `NSCalendar` категорию, возвращающую названия месяцев в строковом виде. Поскольку метод не обращается к данным, а свойство не базируется на переменной экземпляра, категорию можно реализовать следующим образом:

```
@interface NSCalendar (EOC_Additions)
@property (nonatomic, strong, readonly) NSArray *eoc_allMonths;
```

```
@end

@implementation NSCalendar (EOC_Additions)
- (NSArray*)eoc_allMonths {
    if ([self.calendarIdentifier
            isEqualToString:NSGregorianCalendar])
    {
        return @[@"January", @"February",
                 @"March", @"April",
                 @"May", @"June",
                 @"July", @"August",
                 @"September", @"October",
                 @"November", @"December"];
    } else if ( /* другие календарные идентификаторы */ ) {
        /* Возвращает названия месяцев для других календарей */
    }
}
@end
```

Автоматическое синтезирование переменной экземпляра, на которой базируется свойство, не сработает, потому что все обязательные методы (лишь один при доступе только для чтения) были реализованы. По этой причине компилятор не выдает предупреждений. Однако даже в такой ситуации лучше избегать использования свойств. Предполагается, что свойство базируется на данных, хранимых классом; свойства предназначены для инкапсуляции данных. В нашей ситуации лучше объявить метод для получения списка месяцев в категории:

```
@interface NSCalendar (EOC_Additions)
- (NSArray*)eoc_allMonths;
@end
```

УЗЕЛКИ НА ПАМЯТЬ

- Держите все объявления свойств для инкапсулированных данных в главном определении интерфейса.
- По возможности используйте методы доступа вместо объявлений свойств в категориях (кроме категорий продолжения классов).

27 ИСПОЛЬЗУЙТЕ КАТЕГОРИИ ПРОДОЛЖЕНИЯ КЛАССОВ ДЛЯ СОКРЫТИЯ ПОДРОБНОСТЕЙ РЕАЛИЗАЦИИ

Часто класс должен содержать больше методов и переменных экземпляров, чем вам хотелось бы представить внешнему пользователю. Можно сделать их доступными извне и документировать, что эти методы являются закрытыми и разработчик не должен на них полагаться. В конце концов, из-за особенностей работы динамического механизма обмена сообщениями в Objective-C нет «полноценных» закрытых методов или переменных экземпляров (см. подход 11). Тем не менее на практике рекомендуется предоставлять внешний доступ только к тем членам классов, для которых это действительно необходимо. Что же делать с методами и переменными экземпляров, которые не должны быть доступны извне, но существование которых вы все равно хотите документировать? На помощь приходит специальная категория — *категория продолжения класса* (class-continuation category).

Категории продолжения класса, в отличие от обычных категорий, должны определяться в файле реализации класса, который они продолжают. Важно отметить, что это единственная категория, в которой разрешается объявлять дополнительные переменные экземпляров. Кроме того, эта категория не имеет конкретной реализации. Предполагается, что каждый определенный в ней метод присутствует в основной реализации класса. В отличие от других категорий, категория продолжения класса не имеет имени. Категория продолжения класса для класса с именем `EOCPerson` выглядит так:

```
@interface EOCPerson ()
// Методы
@end
```

В чем польза таких категорий? В том, что в них могут определяться как методы, так и переменные экземпляров. Это возможно только благодаря устойчивости ABI (за дополнительной информацией обращайтесь к подходу 6) — этот термин означает, что для использования объекта не обязательно знать его размер. Следовательно, переменные экземпляров не обязательно определять в открытом интерфейсе, поскольку пользователи класса не обязаны знать его структуру. По этой причине переменные экземпляров могут добавляться как в категории продолжения класса, так и в его реализации.

Для этого достаточно добавить в нужном месте фигурные скобки и разместить в них переменные экземпляров:

```
@interface EOCPerson () {
    NSString *_anInstanceVariable;
}
// Объявления методов
@end

@implementation EOCPerson {
    int _anotherInstanceVariable;
}
// Реализации методов
@end
```

Для чего это нужно? Переменные экземпляров можно определять в открытом интерфейсе. Однако преимущество их сокрытия в категории продолжения класса или блоке реализации заключается в том, что эти переменные известны только во внутренней реализации. Даже если пометить их как закрытые в открытом интерфейсе, это по-прежнему приводит к утечке подробностей реализации. Предположим, вы не хотите, чтобы другие знали о существовании сверхсекретного класса, который используется только во внутренних операциях. Если одному из ваших классов принадлежит экземпляр такого класса и вам потребуется объявить переменную экземпляра в открытом интерфейсе, это будет выглядеть так:

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

@class EOCSuperSecretClass;

@interface EOCClass : NSObject {
@private
    EOCSuperSecretClass *_secretInstance;
}
@end
```

Факт существования класса с именем `EOCSuperSecretClass` раскрылся. Проблему можно обойти — отказавшись от сильной типизации переменной экземпляра и объявив ее с типом `id` вместо `EOCSuperSecretClass*`. Но такое решение отнюдь не идеально, потому что вы теряете всю помощь компилятора при внутреннем использовании. И почему вы должны ее лишаться только потому, что не хотите раскрывать какую-то информацию? В такой ситуации

может помочь категория продолжения класса. Она может быть объявлена следующим образом:

```
// EOCClass.h
#import <Foundation/Foundation.h>

@interface EOCClass : NSObject
@end

// EOCClass.m
#import "EOCClass.h"
#import "EOCSuperSecretClass.h"

@interface EOCClass () {
    EOCSuperSecretClass *_secretInstance;
}
@end

@implementation EOCClass
// Методы
@end
```

Аналогичным образом переменная экземпляра может добавляться в блок реализации; семантически это эквивалентно добавлению ее в категорию продолжения класса, а выбор в основном определяется личными предпочтениями. Я предпочитаю добавлять переменные в категории, чтобы все определения данных находились в одном месте. Также в категориях продолжений классов могут определяться свойства, поэтому переменные экземпляров тоже полезно объявлять в них. Такие переменные экземпляров не могут считаться «полноценно» закрытыми, потому что ограничения доступа всегда можно обойти хитрыми средствами исполнительной среды, но по сути они относятся к закрытым. Кроме того, так как объявления не включаются в открытый заголовок, информация гораздо надежнее скрывается при распространении кода в составе библиотеки.

Другим местом, в котором категории продолжения классов особенно удобны, является код Objective-C++. В этом гибриде Objective-C и C++ может использоваться код, написанный на обоих языках. Часто внутренняя часть игры пишется на C++ по соображениям портируемости. В других случаях C++ приходится использовать из-за взаимодействия со сторонней библиотекой, имеющей привязки только для C++. В таких случаях категория продолжения класса тоже может пригодиться. Допустим, ранее класс был записан в следующем виде:

```
#import <Foundation/Foundation.h>
#include "SomeCppClass.h"

@interface EOCClass : NSObject {
@private
    SomeCppClass _cppClass;
}
@end
```

Файл реализации для этого класса будет называться EOCClass.mm; расширение .mm сообщает компилятору, что файл должен компилироваться как код Objective-C++. Без этой информации включение SomeCppClass.h было бы невозможным. Однако обратите внимание на то, что класс C++ `SomeCppClass` должен импортироваться полностью — его определение должно быть полностью разрешено, чтобы компилятор знал, сколько памяти занимает переменная экземпляра `_cppClass`. Итак, любой файл, включающий EOCClass.h для использования класса, также должен компилироваться в режиме Objective-C, поскольку он также будет включать заголовочный файл `SomeCppClass`. Все эти зависимости быстро выходят из-под контроля, и все кончается тем, что все приложение компилируется в режиме Objective-C. В этом нет ничего страшного, но, на мой взгляд, решение выглядит уродливо, особенно если код распространяется в виде библиотеки для использования в других приложениях. Нехорошо заставлять стороннего разработчика переименовывать все файлы своего приложения с расширением .mm.

Можно подумать, что описанную проблему можно решить опережающим объявлением класса C++ вместо импортирования его заголовка, с созданием переменной экземпляра, содержащей указатель на экземпляр:

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

class SomeCppClass;

@interface EOCClass : NSObject {
@private
    SomeCppClass *_cppClass;
}
@end
```

Теперь переменная экземпляра должна быть указателем; в противном случае компилятор не сможет определить ее размер, что приведет к ошибке компиляции. Все указатели имеют фиксированный

размер, так что компилятору достаточно знать тип, на который ссылается указатель. Но тогда мы возвращаемся к той же проблеме: каждый класс, импортирующий заголовок `EOCClass`, столкнется с ключевым словом C++ `class` — а следовательно, должен компилироваться в режиме Objective-C++. Во-первых, это некрасиво; во-вторых, неестественно — переменная экземпляра все равно является закрытой, так почему другие классы должны беспокоиться о ее существовании? На помощь снова приходит категория продолжения класса. При ее использовании класс выглядит так:

```
// EOCClass.h
#import <Foundation/Foundation.h>

@interface EOCClass : NSObject
@end

// EOCClass.mm
#import "EOCClass.h"
#include "SomeCppClass.h"

@interface EOCClass () {
    SomeCppClass _cppClass;
}
@end

@implementation EOCClass
@end
```

Теперь заголовок `EOCClass` свободен от C++, а пользователи заголовка даже не подозревают, что где-то внизу прячется C++. Этот паттерн встречается в некоторых системных библиотеках — например, во фреймворке WebKit, написанном в основном на C++, для работы с которым предоставляется чистый интерфейс Objective-C. В CoreAnimation большая часть служебного кода тоже написана на C++, тогда как для работы с ней используется интерфейс Objective-C.

Также категории продолжения классов уместно использовать для расширения свойств, которые доступны только для чтения в открытом интерфейсе, но должны быть доступны для записи во внутренних операциях. Обычно присваивание должно выполняться set-методом вместо прямого обращения к переменной экземпляра (см. подход 7), потому что при этом срабатывают оповещения KVO (Key-Value Observing), которые могут прослушиваться другим объектом. Свойство, присутствующее в категории продолжения класса (или любой другой категории) и интерфейсе класса, должно

обладать точно совпадающими атрибутами, за исключением того, что доступ только для чтения может быть расширен до чтения/записи. Для примера рассмотрим класс, представляющий человека, с открытым интерфейсом следующего вида:

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

@interface EOCPerson : NSObject
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
                lastName:(NSString*)lastName;
@end
```

Обычно для расширения двух свойств до уровня чтения/записи определяется категория продолжения класса:

```
@interface EOCPerson ()
@property (nonatomic, copy, readwrite) NSString *firstName;
@property (nonatomic, copy, readwrite) NSString *lastName;
@end
```

Вот и все, что необходимо сделать. Теперь реализация `EOCPerson` сможет использовать set-методы, вызывая `setFirstName:` или `setLastName:`, или точечный синтаксис свойств. Это очень удобно, так как объекты остаются неизменяемыми в открытом интерфейсе, но при этом у вас остается возможность управлять ими во внутренней реализации так, как потребуется. Таким образом, инкапсулированные классом данные находятся под управлением экземпляра без возможности их внешнего изменения. За дополнительной информацией по этой теме обращайтесь к подходу 18. Учтите, что такой подход создает потенциальную возможность гонки (race condition), если наблюдатель читает свойство одновременно с его внутренней записью. Проблема решается разумным применением средств синхронизации (см. подход 41).

Еще одно уместное использование категории продолжения класса — объявление закрытых методов, которые будут использоваться только внутри реализации класса. Оно документирует методы, доступные в реализации класса. Выглядит это так:

```
@interface EOCPerson ()
- (void)p_privateMethod;
@end
```

Для обозначения закрытого метода здесь используется концепция префикса из подхода 20. В последних версиях компилятора объявление методов перед их использованием уже не обязательно. Тем не менее часто бывает разумно включать объявления в категорию продолжения класса для документирования существующих методов. Я предпочитаю записывать прототипы методов заранее, до реализации класса, а потом переходить к их реализации. Такой подход существенно упрощает чтение кода класса.

Наконец, в категории продолжения класса удобно указать, что объект реализует протоколы, которые рассматриваются как закрытые. Часто разглашение информации о реализации некоторых протоколов в открытом интерфейсе нежелательно — например, потому что протокол является частью закрытого API. Допустим, класс `EOCPerson` реализует протокол с именем `EOCSecretDelegate`. При использовании открытого интерфейса это будет выглядеть так:

```
#import <Foundation/Foundation.h>
#import "EOCSecretDelegate.h"

@interface EOCPerson : NSObject <EOCSecretDelegate>
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;

- (id)initWithFirstName:(NSString*)firstName
               lastName:(NSString*)lastName;
@end
```

Казалось бы, вместо импортирования протокола `EOCSecretDelegate` (а вернее, заголовочного файла, в котором он определяется) можно просто использовать опережающее объявление. В данном случае опережающее объявление выглядит так:

```
@protocol EOCSecretDelegate;
```

Однако во всех точках импортирования заголовка `EOCPerson` компилятор выдает следующее предупреждение:

```
warning: cannot find protocol definition for
'EOCSecretDelegate'
```

Компилятор не сможет узнать, какие методы входят в протокол, не видя его определения, — и предупреждает об этом вас. Но поскольку протокол является внутренним и закрытым, даже разглашение

его имени нежелательно. На помощь снова приходит категория продолжения класса! Вместо того чтобы объявлять о реализации EOCSecretDelegate классом EOCPerson в открытом интерфейсе, вы делаете это в категории продолжения класса:

```
#import "EOCPerson.h"
#import "EOCSecretDelegate.h"

@interface EOCPerson () <EOCSecretDelegate>
@end

@implementation EOCPerson
/* … */
@end
```

Из открытого интерфейса можно удалить все ссылки на EOCSecretDelegate. Закрытый протокол уже не виден извне, а пользователи могут узнать о его существовании только *после* глубокой интроспекции.

УЗЕЛКИ НА ПАМЯТЬ

- Используйте категории продолжения классов для добавления переменных экземпляров в класс.
- Свойства, объявленные с доступом только для чтения в главном интерфейсе, могут переобъявляться в категории продолжения класса как доступные для чтения/записи, если set-методы предназначены для использования только во внутренней реализации класса.
- Объявляйте прототипы закрытых методов в категориях продолжения классов.
- Используйте категории продолжения классов для объявления протоколов, факт реализации которых вашим классом вы не хотите разглашать.

28 ИСПОЛЬЗУЙТЕ ПРОТОКОЛЫ ДЛЯ СОЗДАНИЯ АНОНИМНЫХ ОБЪЕКТОВ

Протоколы определяют набор методов, которые могут (или должны) реализовываться объектом. С их помощью можно скрыть

подробности реализации в API своего кода, возвращая объекты с минимальным типом `id`, реализующие протокол. При этом имя конкретного класса в API не разглашается. Например, такая возможность может пригодиться, если за интерфейсом скрывается большое количество классов и вы не хотите разглашать лишнюю информацию о классе: допустим, этот класс может измениться или же возвращаться могут объекты многих разных классов, не укладывающихся в стандартную иерархию классов при указании общего базового класса в качестве типа.

Данная концепция, часто называемая *анонимными объектами* (anonymous objects), не похожа на анонимные объекты из других языков, в которых этим термином обозначается возможность создания «встроенного» класса без имени. В Objective-C это не так. Пример использования анонимных объектов встречается в подходе 23 при рассмотрении делегатов и источников данных. Например, определение свойства делегата может выглядеть так:

```
@property (nonatomic, weak) id <EOCDelegate> delegate;
```

При указанном типе `id<EOCDelegate>` класс объекта может быть абсолютно любым; он даже не обязан быть производным от `NSObject`. Годится любой класс при условии, что он реализует `EOCDelegate`. Для класса, обладающего этим свойством, делегат является анонимным. Он может при желании определить фактический класс объекта во время выполнения (см. подход 14). Тем не менее такая проверка считается нежелательной, поскольку контракт, определяемый типом свойства, указывает, что фактический класс неважен.

Другой пример применения этой концепции можно найти в `NSDictionary`. Стандартная семантика управления памятью для ключей словаря заключается в их копировании с удержанием их значений. Таким образом, в изменяемом словаре сигнатура метода для задания пары «ключ-значение» выглядит так:

```
- (void)setObject:(id)object forKey:(id<NSCopying>)key
```

Параметр ключа `key` указан с типом `id<NSCopying>`; это может быть любой объект, реализующий `NSCopying`, чтобы ему было успешно доставлено сообщение `copy` (см. подход 22). Можно считать, что ключ анонимен. Как и в случае со свойством делегата, словарь не знает, какой конкретный класс используется, — и его это не интересует. Ему достаточно знать, что экземпляру можно отправить сообщение `copy`.

Концепция анонимных объектов также может применяться к объектам, которые возвращаются библиотекой, управляющей подключениями к базе данных. Вероятно, вы не захотите разглашать класс, обеспечивающий подключение, поскольку для разных баз данных могут использоваться разные классы. Без осмысленной возможности наследования от общего базового класса придется возвращать объект типа `id`. Однако вы можете создать протокол, определяющий общие методы для всех подключений к базе данных, и объявить, что он реализуется объектом. Протокол может выглядеть примерно так:

```
@protocol EOCDatabaseConnection
- (void)connect;
- (void)disconnect;
- (BOOL)isConnected;
- (NSArray*)performQuery:(NSString*)query;
@end
```

Затем создается синглетный обработчик, предоставляющий подключения к базе данных. Его интерфейс может выглядеть так:

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

@protocol EOCDatabaseConnection;

@interface EOCDatabaseManager : NSObject
+ (id)sharedInstance;
- (id<EOCDatabaseConnection>)connectionWithIdentifier:
                                (NSString*)identifier;
@end
```

Имя класса, обеспечивающего подключение к базе данных, не разглашается, а метод может возвращать разные классы, которые теоретически могут принадлежать разным библиотекам. Для пользователя API важно лишь то, что возвращаемый объект может создавать и уничтожать подключения, а также выполнять запросы. Последнее обстоятельство особенно важно. В нашем примере внутренний код обработки подключений к базе данных может использовать разные сторонние библиотеки для подключения к базам данных разных типов (например, MySQL, PostgreSQL и т. д.). По этой причине создание иерархии, в которой классы подключений из разных сторонних библиотек наследуют от общего базового класса, может оказаться невозможным. В такой ситуации можно использовать решение с анонимными объектами, в котором создаются простые обертки (wrappers), субклассирующие все эти сторонние

классы и реализующие протокол `EOCDatabaseConnection`. Затем эти классы возвращаются методом `connectionWithIdentifier:`. В будущих версиях эти классы могут заменяться без каких-либо изменений в открытом API.

Анонимные типы также пригодятся в тех ситуациях, когда вы хотите подчеркнуть, что важен не тип объекта, а реализация им некоторых методов. Даже если тип всегда соответствует некоторому классу вашей реализации, анонимный тип с протоколом может использоваться для выражения того, что конкретный тип в данном случае неважен.

Пример такого подхода встречается в библиотеке CoreData. Класс с именем `NSFetchedResultsController` обрабатывает результаты запроса к базе данных CoreData и разбивает данные на секции, если потребуется. Для обращения к секциям используется свойство `sections` контроллера результатов. Вместо массива полностью типизованных объектов используется массив объектов, реализующих протокол `NSFetchedResultsSectionInfo`. Использование контроллера для получения информации о секции выглядит следующим образом:

```
NSFetchedResultsController *controller = /* контроллер */;
NSUInteger section = /* индекс секции */;

NSArray *sections = controller.sections;
id <NSFetchedResultsSectionInfo> sectionInfo =
sections[section];
NSUInteger numberOfObjects = sectionInfo.numberOfObjects;
```

Объект `sectionInfo` является анонимным. При таком оформлении API объект дает ясно понять, что он предоставляет доступ к информации о секции. Во внутренней реализации этот объект, скорее всего, представляет собой некий внутренний объект состояния, созданный контроллером результатов. Предоставлять доступ к открытому классу, представляющему данные, не обязательно, так как пользователя контроллера результата не интересует конкретный способ хранения данных секций. Все, что ему необходимо, — это иметь возможность выдачи запросов к данным. Использование протокола для создания анонимного объекта, как это сделано в нашем примере, означает, что внутренний объект состояния может возвращаться в массиве `sections`. Пользователь знает лишь то, что этот объект реализует некоторые методы, а остальная часть реализации объекта остается скрытой.

УЗЕЛКИ НА ПАМЯТЬ

- Протоколы могут использоваться для реализации некоторого уровня анонимности типов. Фактический тип сокращается до типа `id`, реализующего методы протокола.
- Используйте анонимные объекты, если тип (имя класса) требуется скрыть.
- Используйте анонимные объекты, если важен не фактический тип, а сам факт реакции объекта на определенные методы (определенные в протоколе).

Глава 5
УПРАВЛЕНИЕ ПАМЯТЬЮ

Управление памятью играет важную роль во всех объектно-ориентированных языках, к числу которых принадлежит Objective-C. Понимание специфики модели управления памятью необходимо для написания эффективного кода, свободного от ошибок.

Если разобраться в правилах, управление памятью в Objective-C оказывается не таким уж сложным. В частности, оно упростилось с появлением механизма автоматического подсчета ссылок ARC (Automatic Reference Counting). ARC передает практически все решения по управлению памятью компилятору, позволяя вам сосредоточиться на бизнес-логике.

29 РАЗБЕРИТЕСЬ С МЕХАНИЗМОМ ПОДСЧЕТА ССЫЛОК

В Objective-C для управления памятью используется механизм подсчета ссылок; это означает, что с каждым объектом связан счетчик. Значение счетчика увеличивается, когда вы регистрируете свой интерес к объекту (т. е. намерение выполнить с ним некоторую операцию), а уменьшается после завершения работы с ним. Когда счетчик ссылок объекта уменьшается до нуля, это означает, что объект никому больше не нужен и его можно уничтожить. Впрочем, это лишь краткий обзор; полное понимание темы необходимо для написания хорошего кода Objective-C, даже если вы собираетесь использовать ARC (см. подход 30).

Уборщик мусора, используемый в коде Objective-C для Mac OS X, официально объявлен устаревшим с выходом Mac OS X 10.8 и никогда не был доступен для iOS. Понимать принципы работы подсчета ссылок очень важно, потому что вы уже не можете полагаться на уборщик мусора в Mac OS X, а в iOS это просто невозможно. Если вы уже используете ARC, не слушайте ту часть мозга, которая говорит, что большая часть представленного кода не откомпилируется. Это действительно так… в мире ARC. Но в этом подходе подсчет ссылок рассматривается с точки зрения Objective-C, а для этого приходится демонстрировать код с использованием методов, которые однозначно недопустимы в ARC.

КАК РАБОТАЕТ ПОДСЧЕТ ССЫЛОК

В архитектурах с подсчетом ссылок с объектом связывается счетчик, который указывает, сколько сторон заинтересовано в продолжении существования объекта. В Objective-C он называется *счетчиком удержаний* (retain count), но термин «счетчик ссылок» тоже часто встречается. В протоколе `NSObject` объявлены следующие три метода, увеличивающие и уменьшающие значение счетчика:

- `retain` — увеличивает счетчик ссылок;
- `release` — уменьшает счетчик ссылок;
- `autorelease` — уменьшает счетчик ссылок позднее, при исчерпании пула автоматического освобождения. (Пул автоматического освобождения рассматривается далее в этом подходе, а также в подходе 34.)

Метод проверки счетчика ссылок `retainCount` обычно бесполезен даже во время отладки, поэтому я (и компания Apple) не рекомендую его использовать. За дополнительной информацией обращайтесь к подходу 36.

При создании объекта счетчик ссылок равен минимум 1. Интерес к продолжению существования объекта выражается вызовом метода `retain`. Когда этот интерес проходит, потому что объект становится ненужным для некоторой части кода, вызывается метод `release` или `autorelease`. Когда счетчик ссылок падает до 0, память объекта помечается как пригодная для повторного использования. После этого все ссылки на объект становятся недействительными.

На рис. 5.1 изображен объект, проходящий фазы создания, удержания и двух освобождений.

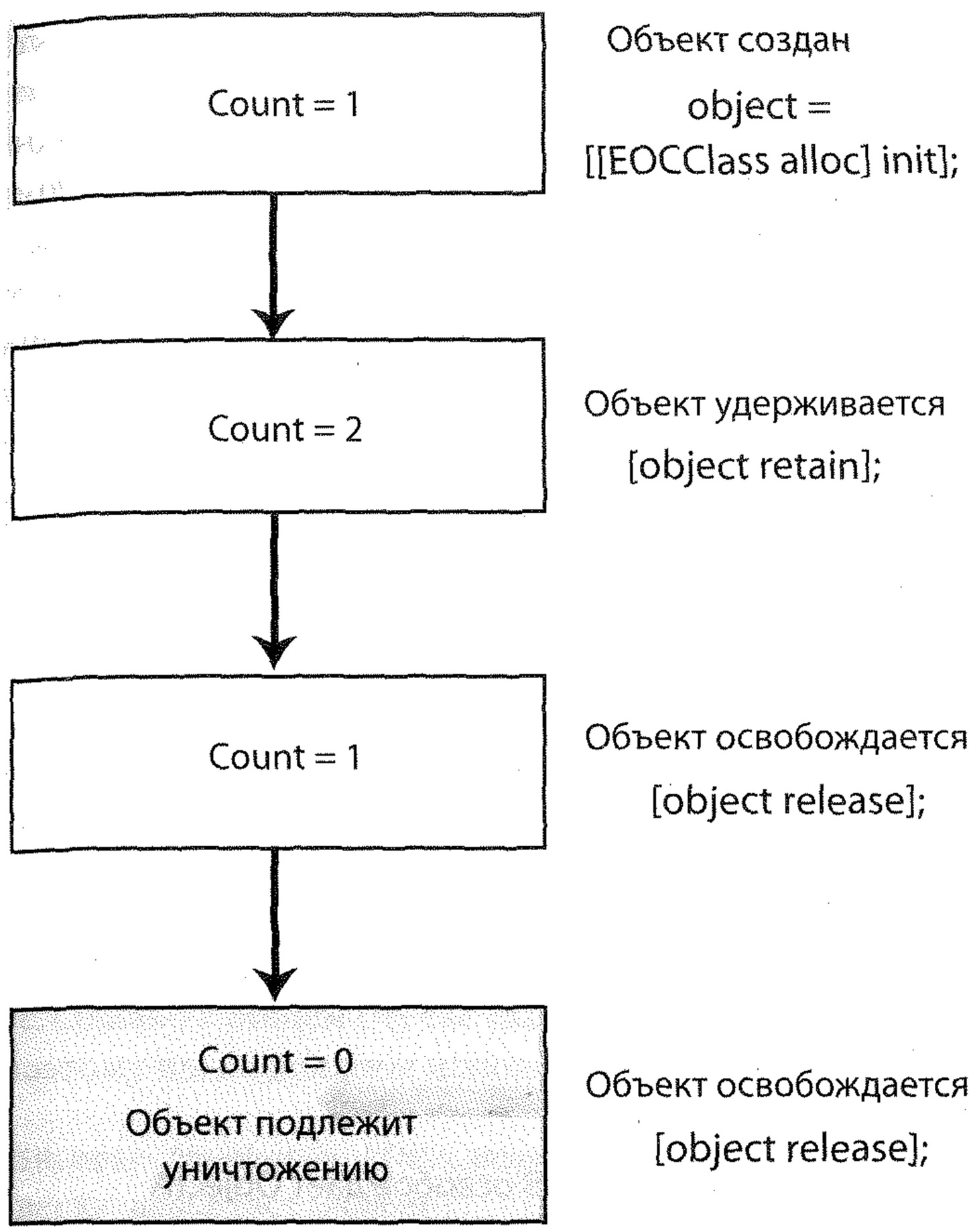


Рис. 5.1. Увеличение и уменьшение счетчика ссылок объекта в фазах жизненного цикла

В своем жизненном цикле приложение создает многочисленные объекты. Все эти объекты связаны друг с другом. Например, объект, представляющий человека, содержит ссылку на строковый объект с именем, а также может содержать ссылки на объекты других людей — например, множество, представляющее круг его друзей. Так формируется так называемый *граф объектов* (object graph). Говорят, что объект является *владельцем* другого объекта (и, соответственно, второй объект *принадлежит* первому), но если второй объект содержит сильную ссылку на первый. Это означает, что первый объект зарегистрировал свой интерес к тому, чтобы второй объект продолжал существовать. А когда работа будет завершена, он освободит его.

На графе объектов с рис. 5.2 объекты `ObjectB` и `ObjectC` содержат ссылки на `ObjectA`. Когда и `ObjectB`, и `ObjectC` закончат работать

с объектом ObjectA, его счетчик ссылок уменьшится до 0 и объект может быть уничтожен. Объекты ObjectB и ObjectC продолжают существовать из-за других объектов, которые в свою очередь продолжают существовать еще из-за каких-то объектов. В конечном счете при восхождении по дереву ссылок вы доберетесь до корневого объекта. Для приложений Mac OS X им может быть объект NSApplication; для приложений iOS — объект UIApplication. В обоих случаях это синглетные объекты, создаваемые при запуске приложения.

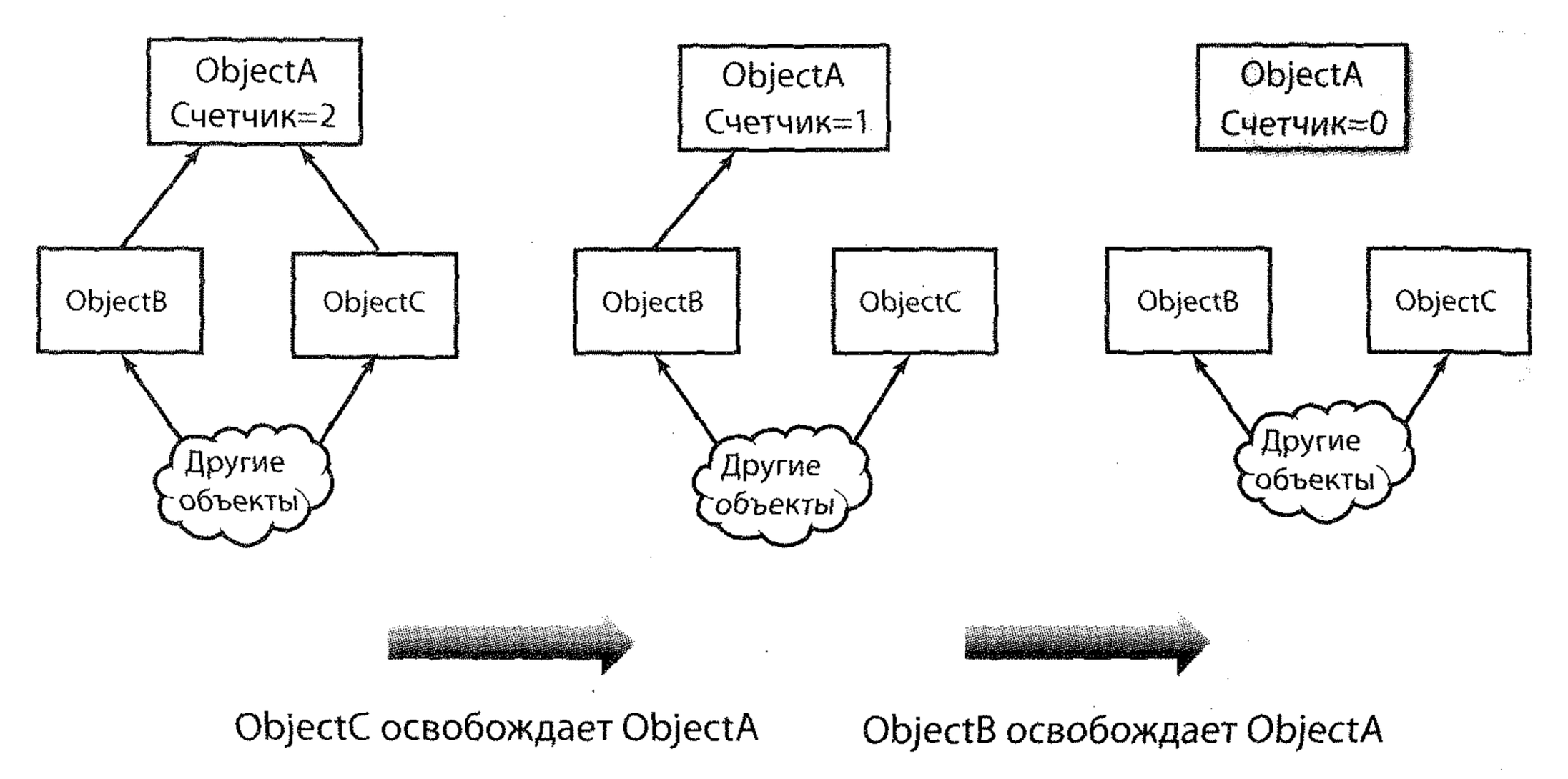


Рис. 5.2. Граф объектов и уничтожение объекта, после того как счетчик ссылок на него уменьшится до 0

Следующий код поможет вам понять все происходящее на практике:

```
NSMutableArray *array = [[NSMutableArray alloc] init];

NSNumber *number = [[NSNumber alloc] initWithInt:1337];
[array addObject:number];
[number release];

// Действия с 'array'

[array release];
```

Как упоминалось выше, этот код не будет компилироваться для ARC из-за явных вызовов release. В Objective-C вызов alloc возвращает объект, принадлежащий вызывающей стороне. Иначе говоря, интерес вызывающей стороны уже зарегистрирован при использовании

alloc. Однако следует учесть, что это не обязательно означает, что счетчик ссылок равен именно 1. Его значение может быть и больше, поскольку реализация alloc или initWithInt: может привести к дополнительному удержанию объекта. Гарантируется лишь то, что счетчик ссылок содержит значение не ниже 1. Именно так следует думать о счетчиках ссылок. Никогда не пытайтесь гарантировать конкретное значение счетчика ссылок; следите лишь за тем, к каким последствиям приводят ваши действия со счетчиком ссылок: уменьшают или увеличивают его.

Затем в массив добавляется объект number. При этом массив также регистрирует свой интерес, вызывая retain для объекта number в методе addObject:. В этой точке счетчик ссылок равен минимум 2. Далее надобность в объекте number пропадает, и он освобождается. Его счетчик ссылок снова уменьшается до минимум 1. В этой точке дальнейшее использование переменной number становится небезопасным. Вызов release означает, что существование объекта, на который указывает ссылка, не гарантировано. Конечно, из кода в данной ситуации видно, что объект будет жив после вызова release, потому что ссылка на него по-прежнему хранится в массиве. Однако в общем случае такие предположения делать не следует, поэтому вы не должны использовать решения вида:

```
NSNumber *number = [[NSNumber alloc] initWithInt:1337];
[array addObject:number];
[number release];
NSLog(@"number = %@", number);
```

И хотя здесь этот код работает, поступать так не рекомендуется. Если по какой-либо причине объект number будет уничтожен в результате вызова release, когда его счетчик ссылок уменьшается до 0, вызов NSLog приведет к потенциальному сбою. Причина, по которой я называю сбой «потенциальным», заключается в том, что память уничтоженного объекта просто возвращается в пул свободной памяти. Если память еще не была перезаписана к моменту обращения к NSLog, то объект все еще существует и сбой не происходит. По этой причине ошибки преждевременного освобождения объектов создают особенно много проблем при отладке.

Для предотвращения случайного использования недействительного объекта вызов release часто сопровождается присваиванием nil указателю. Тем самым гарантируется, что никто не обратится по указателю на потенциально недействительный объект. Например, это может быть сделано так:

```
NSNumber *number = [[NSNumber alloc] initWithInt:1337];
[array addObject:number];
[number release];
number = nil;
```

Управление памятью в методах доступа свойств

Как упоминалось ранее, связи между объектами образуют граф объектов. Массив из нашего примера удерживает содержащиеся в нем объекты, создавая ссылку на них. Аналогичным образом объекты удерживают в памяти другие объекты, часто при помощи свойств (см. подход 6), использующих методы доступа для чтения и записи переменных экземпляров. Если свойство создает сильную связь, то значение свойства удерживается (retained). Set-метод для такого свойства с именем `foo`, базирующегося на переменной экземпляра с именем `_foo`, будет выглядеть так:

```
- (void)setFoo:(id)foo {
    [foo retain];
    [_foo release];
    _foo = foo;
}
```

Новое значение удерживается, а старое освобождается. Затем переменная экземпляра обновляется указателем на новое значение. Порядок выполнения операций важен: если старое значение будет освобождено до удержания нового значения, а два значения точно совпадают, освобождение может привести к потенциальному преждевременному уничтожению объекта. Последующее удержание не воскресит уничтоженный объект, а переменная экземпляра будет содержать указатель на несуществующий объект.

Пулы автоматического освобождения

В архитектуре подсчета ссылок Objective-C важную роль играют так называемые *пулы автоматического освобождения* (autorelease pools). Вместо вызова `release` для немедленного уменьшения счетчика удержаний объекта (и его потенциального уничтожения) также можно вызвать метод `autorelease`, который выполняет освобождение позднее — обычно при следующем проходе цикла событий, но, возможно, и быстрее (см. подход 34).

Эта возможность чрезвычайно полезна, особенно при возвращении объектов из методов. В данном случае объект не всегда должен

возвращаться как принадлежащий вызывающей стороне. Для примера возьмем следующий метод:

```
- (NSString*)stringValue {
    NSString *str = [[NSString alloc]
                         initWithFormat:@"I am this: %@",
self];
    return str;
}
```

В этой ситуации `str` возвращается со счетчиком ссылок +1, поскольку вызов `alloc` возвращает управление со счетчиком +1, а парный вызов `release` отсутствует. Значение +1 означает, что вызывающая сторона несет ответственность за одно удержание, которое должно быть каким-то образом скомпенсировано. Впрочем, это не означает, что счетчик ссылок равен ровно 1. Он может содержать и большее значение, но это уже нюансы реализации в методе `initWithFormat:`. Вы должны беспокоиться только о компенсации одного удержания.

Однако освободить `str` в методе нельзя, поскольку тогда объект будет немедленно уничтожен до его возвращения. По этой причине метод `autorelease` указывает, что объект должен быть освобожден в будущем, но гарантированно проживет достаточно долго, чтобы возвращаемое значение могло быть удержано вызывающей стороной (*если это нужно*). Иначе говоря, объект гарантированно переживает переход через границу вызова метода. Освобождение произойдет при исчерпании внешнего пула автоматического освобождения (см. подход 34), что при отсутствии ваших собственных пулов автоматического освобождения произойдет при следующем проходе цикла событий текущего потока. Применительно к методу `stringValue` это выглядит так:

```
- (NSString*)stringValue {
    NSString *str = [[NSString alloc]
                         initWithFormat:@"I am this: %@", self];
    return [str autorelease];
}
```

Теперь возвращаемый объект `NSString` определенно будет существовать при возвращении вызывающей стороне, что позволяет использовать его следующим образом:

```
NSString *str = [self stringValue];
NSLog(@"The string is: %@", str);
```

Никакое дополнительное управление памятью при этом не требуется, так как объект `str` возвращается с автоматическим освобождением, а следовательно, лишнее удержание компенсировано. Так как освобождение из пула не произойдет до следующего прохода цикла событий, объект не нужно явно удерживать для использования в `NSLog`. Но если объект должен остаться (например, при присваивании переменной экземпляра), его необходимо удержать с последующим освобождением:

```
_instanceVariable = [[self stringValue] retain];
// …
[_instanceVariable release];
```

Таким образом автоматическое освобождение продлевает срок жизни объекта ровно настолько, чтобы он пережил переход через границу вызова метода.

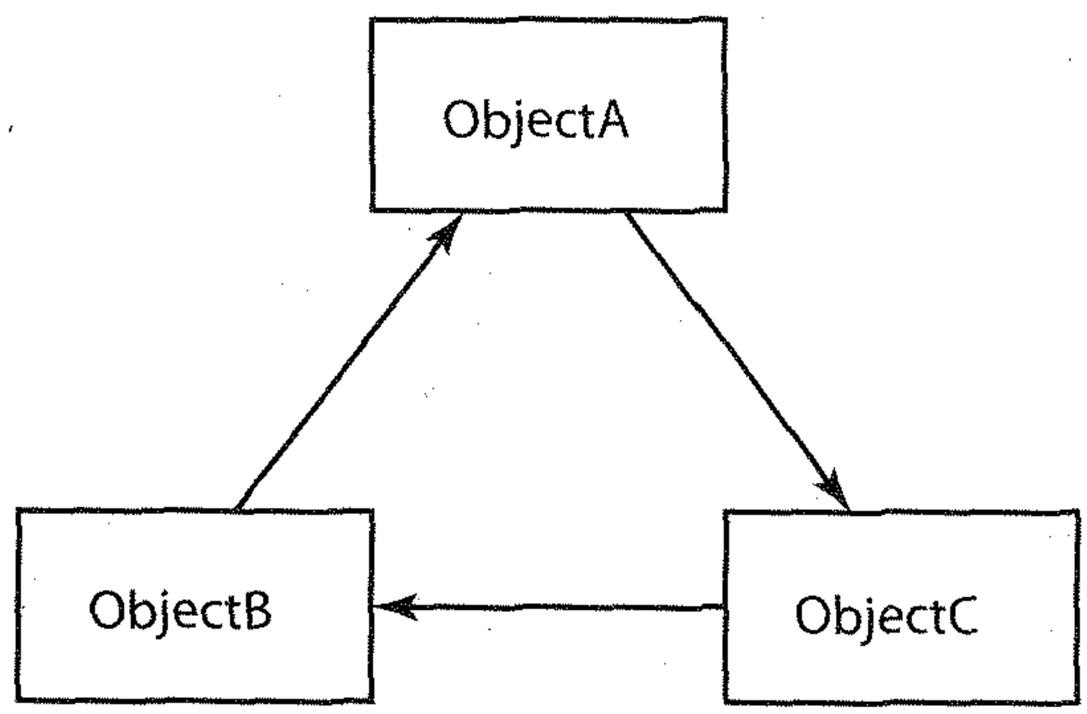


Рис. 5.3. Цикл удержания в графе объектов

Циклы удержания

Одной из стандартных проблем архитектуры подсчета ссылок являются так называемые циклы удержания (retain cycles). Они возникают тогда, когда несколько объектов поочередно ссылаются друг на друга. Это приводит к утечке памяти, поскольку ни у одного объекта в цикле счетчик ссылок не уменьшится до 0. У каждого объекта существует минимум еще один объект в цикле, поддерживающий ссылку на него. На рис. 5.3 каждый из трех объектов содержит ссылку на один из двух других. В таком цикле все счетчики ссылок не падают ниже 1.

В средах с уборкой мусора подобная проблема обычно выявляется как *изолированная зона* (island of isolation). В этом случае уборщик

мусора уничтожает все три объекта. В Objective-C с его архитектурой подсчета ссылок такая роскошь недоступна. Проблема обычно решается использованием слабых ссылок (см. подход 33) или внешним воздействием, заставляющим один из объектов отказаться от удержания другого объекта. В любом случае цикл удержания прерывается, а утечка памяти ликвидируется.

УЗЕЛКИ НА ПАМЯТЬ

- Управление памятью с подсчетом ссылок основано на счетчике, значение которого увеличивается и уменьшается. При создании объекта счетчик равен минимум 1. Объект с положительным счетчиком ссылок существует. Когда счетчик уменьшается до 0, объект уничтожается.
- В своем жизненном цикле объект удерживается и освобождается другими объектами, содержащими ссылки на него. Удержание и освобождение приводит к увеличению и уменьшению счетчика соответственно.

30 ИСПОЛЬЗУЙТЕ ARC ДЛЯ УПРОЩЕНИЯ ПОДСЧЕТА ССЫЛОК

В концепции подсчета ссылок нет ничего сложного (см. подход 29). Семантика местонахождения вызовов удержания и освобождения тоже выражается достаточно просто. Так, в проекте компилятора Cland появился статический анализатор, способный выявлять проблемы с подсчетом ссылок. Рассмотрим следующий фрагмент с ручным подсчетом ссылок:

```
if ([self shouldLogMessage]) {
    NSString *message = [[NSString alloc] initWithFormat:
                         @"I am object, %p", self];
    NSLog(@"message = %@", message);
}
```

Этот код создает утечку памяти, потому что объект сообщения не освобождается в конце команды `if`. Так как обращения к нему вне команды `if` невозможны, возникает утечка памяти. Правила, определяющие причину утечки, элементарны: вызов метода `alloc` для `NSString` возвращает объект со счетчиком ссылок +1, но у него нет компенсирующего освобождения. Эти правила легко выражаются,

а компьютер может легко применить их и сообщить об утечке памяти объекта. Именно это и делает статический анализатор.

Впрочем, это еще не всё. Так как статический анализатор может указать на проблемы управления памятью, он также легко может исправить их, добавив нужный вызов удержания или освобождения. Из этой идеи родился механизм автоматического подсчета ссылок ARC (Automatic Reference Counting). Механизм ARC делает в точности то, что заявлено в названии: он автоматизирует подсчет ссылок. Таким образом, в приведенном фрагменте кода для объекта `message` автоматически добавляется вызов `release` непосредственно перед завершением области действия команды `if`, а код автоматически приводится к следующему виду:

```
if ([self shouldLogMessage]) {
    NSString *message = [[NSString alloc] initWithFormat:
                         @"I am object, %p", self];
    NSLog(@"message = %@", message);
    [message release]; ///< Добавлено ARC
}
```

При использовании ARC важно помнить, что подсчет ссылок никуда не исчез; просто ARC добавляет вызовы удержания и освобождения за вас. Как вы увидите, ARC не ограничивается применением семантики управления памятью к методам, возвращающим объекты. Однако эта базовая семантика, ставшая стандартной в Objective-C, была заложена в основу для построения ARC.

Так как ARC добавляет вызовы `retain`, `release` и `autorelease` за вас, вызовы методов управления памятью в ARC запрещены. Конкретно не допускаются вызовы следующих методов:

- `retain`;
- `release`;
- `autorelease`;
- `dealloc`.

Прямой вызов любого из этих методов приведет к ошибке компиляции, потому что он помешает ARC определить необходимые вызовы управления памятью. Вам придется довериться ARC, что может быть непросто для разработчика, привыкшего к ручному подсчету ссылок.

ARC не вызывает эти методы через обычную систему диспетчеризации сообщений Objective-C, а использует низкоуровневые

аналоги C. Такое решение эффективно, потому что удержания и освобождения выполняются часто, а любая экономия процессорного времени приносит большой выигрыш. Например, аналог `retain` называется `objc_retain`. Кстати, это еще одна причина, по которой переопределение `retain`, `release` или `autorelease` недопустимо, — эти методы никогда не вызываются напрямую. В оставшейся части этого подхода в основном рассматриваются эквивалентные методы Objective-C, а не низкоуровневые аналоги C. Это должно помочь, если у вас имеется практический опыт ручного подсчета ссылок.

Правила выбора имен методов в ARC

Семантика управления памятью, предписанная именами методов, давно была принята в Objective-C, но в ARC она была закреплена в наборе жестких правил. Эти правила просты, и они относятся к именам методов. Метод, возвращающий объект, возвращает его принадлежащим вызывающей стороне, если имя метода начинается с одного из следующих префиксов:

- `alloc`;
- `new`;
- `copy`;
- `mutableCopy`.

Под «принадлежностью вызывающей стороне» следует понимать, что код, вызывающий любые из четырех перечисленных методов, отвечает за освобождение возвращенного объекта. Иначе говоря, объект будет иметь положительный счетчик ссылок, у которого ровно одно удержание должно быть скомпенсировано в вызывающем коде. Счетчик ссылок может быть больше 1, если к объекту применялись дополнительные удержания и автоматические освобождения; это одна из причин бесполезности метода `retainCount` (см. подход 36).

Любое другое имя метода означает, что возвращаемый объект не принадлежит вызывающему коду. В таких случаях объект возвращается с автоматическим освобождением, чтобы его значение пережило границу вызова метода. Если вызывающий код хочет, чтобы объект существовал более долгое время, он должен удержать его.

ARC автоматически реализует всё управление памятью, необходимое для соблюдения этих правил, включая код возвращения

объектов с автоматическим освобождением, как продемонстрировано в следующем коде:

```
+ (EOCPerson*)newPerson {
    EOCPerson *person = [[EOCPerson alloc] init];
    return person;
    /**
     * Имя метода начинается с 'new', а поскольку 'person'
     * уже имеет несбалансированное увеличение +1
     * счетчика ссылок от 'alloc',
     * при возвращении не требуются никакие вызовы
     * retain, release или autoreleases.
     */
}

+ (EOCPerson*)somePerson {
    EOCPerson *person = [[EOCPerson alloc] init];
    return person;
    /**
     * Имя метода не начинается с одного
     * из префиксов "принадлежности", поэтому ARC
     * добавляет autorelease при возвращении 'person'.
     * Эквивалентная команда для ручного подсчета ссылок:
     *   return [person autorelease];
     */
}

  (void)doSomething {
    EOCPerson *personOne = [EOCPerson newPerson];
    // …

    EOCPerson *personTwo = [EOCPerson somePerson];
    // …

    /**
     * В этой точке 'personOne' и 'personTwo' выходят из
     * видимости, поэтому механизм ARC должен обеспечить
     * необходимую зачистку. Объект 'personOne'
     * возвращался как принадлежащий этому блоку кода,
     *   поэтому его необходимо освободить.
     * Объект 'personTwo' возвращался без принадлежности
     * этому блоку кода, поэтому освобождать его не нужно.
     * Эквивалентная команда зачистки для ручного подсчета ссылок:
     *    [personOne release];
     */
}
```

ARC стандартизирует правила управления памятью на уровне выбора имен, хотя новичкам это часто кажется странным. В других языках выбору имен обычно не придается такого значения, как в Objective-C. Привыкайте к этой концепции, это очень важно для того, чтобы стать хорошим разработчиком Objective-C. ARC поможет вам в этом, выполняя значительную часть работы за вас.

Кроме автоматических удержаний и освобождений, ARC обладает и другими преимуществами. Например, ARC выполняет оптимизации, которые трудно или невозможно реализовать вручную. Скажем, во время компиляции ARC может выполнять свертку удержаний, освобождений и автоматических освобождений, чтобы по возможности исключить их. Если один объект несколько раз удерживается и несколько раз освобождается, ARC иногда может удалить пары удержаний/освобождений.

Также ARC включает компонент времени выполнения. Происходящие здесь оптимизации еще интереснее, и они наглядно показывают, почему весь будущий код следует писать для ARC. Вспомните, что некоторые объекты возвращаются методами с автоматическим освобождением. Иногда вызывающему коду требуется немедленно удержать объект, как в следующей ситуации:

```
// From a class where _myPerson is a strong instance variable
_myPerson = [EOCPerson personWithName:@"Bob Smith"];
```

Вызов `personWithName:` возвращает новый объект `EOCPerson` с автоматическим освобождением. Но компилятору также необходимо добавить `retain` при присваивании переменной экземпляра, поскольку при этом создается сильная ссылка. Следовательно, приведенный код эквивалентен следующему коду в мире ручного подсчета ссылок:

```
EOCPerson *tmp = [EOCPerson personWithName:@"Bob Smith"];
_myPerson = [tmp retain];
```

Здесь следует заметить, что вызов `autorelease` из метода `personWithName:` и вызов `retain` избыточны. Удаление обоих вызовов ускорит работу приложения. Но код, откомпилированный для ARC, должен быть совместим с кодом, не использующим ARC. Вообще говоря, в ARC можно было бы отказаться от концепции автоматического освобождения и потребовать, чтобы все объекты, возвращаемые методами, возвращались со счетчиком ссылок +1, но это нарушит обратную совместимость кода.

Однако ARC умеет на стадии выполнения обнаруживать ситуации с избыточными автоматическими освобождениями и немедленными удержаниями. Для этой цели используется специальная функция, выполняемая при возвращении объекта с автоматическим освобождением. Вместо простого вызова метода `autorelease` вызывается `objc_autoreleaseReturnValue`. Эта функция проверяет код, который должен быть выполнен немедленно при возвращении из текущего метода. Если проверка обнаруживает, что в нем выполняется удержание возвращенного объекта, вместо выполнения автоматического освобождения в глобальной структуре данных (зависящей от процессора) устанавливается флаг. Код на стороне вызова, удерживающий возвращенный методом объект с автоматическим освобождением, вместо прямого вызова `retain` использует функцию с именем `objc_retainAutoreleasedReturnValue`. Эта функция проверяет флаг и, если он установлен, не выполняет удержание. Операции установки и проверки флагов выполняются быстрее, чем автоматическое освобождение и удержание.

Следующий код показывает, как ARC используются эти спе функции для выполнения оптимизации:

```
// В классе EOCPerson
+ (EOCPerson*)personWithName:(NSString*)name {
    EOCPerson *person = [[EOCPerson alloc] init];
    person.name = name;
    objc_autoreleaseReturnValue(person);
}

// Код, использующий класс EOCPerson
EOCPerson *tmp = [EOCPerson personWithName:@"Matt Galloway"];
_myPerson = objc_retainAutoreleasedReturnValue(tmp);
```

Реализации этих специальных функций рассчитаны на конкретный процессор, чтобы повысить эффективность решения. Следующие реализации на псевдокоде объясняют, что при этом происходит:

```
id objc_autoreleaseReturnValue(id object) {
    if ( /* вызывающая сторона удержит объект */ ) {
        set_flag(object);
        return object; ///< Без автоматического освобождения
    } else {
        return [object autorelease];
    }
}
```

```
id objc_retainAutoreleasedReturnValue(id object) {
    if (get_flag(object)) {
        clear_flag(object);
        return object; ///< Без удержания
    } else {
        return [object retain];
    }
}
```

Способ, которым `objc_autoreleaseReturnValue` определяет, собирается ли вызывающий код немедленно удержать объект, зависит от процессора. Он может быть реализован только автором компилятора, потому что в нем используется анализ низкоуровневых инструкций машинного кода. Автор компилятора — единственный человек, который может гарантировать, что организация кода вызывающего метода сделает такой анализ возможным.

Это всего лишь одна из оптимизаций, возможных благодаря передаче управления памятью в руки компилятора и исполнительной среды. Она помогает понять, почему так желательно использовать ARC. Я убежден, что по мере «взросления» компилятора и исполнительной среды появятся другие оптимизации такого рода.

СЕМАНТИКА УПРАВЛЕНИЯ ПАМЯТЬЮ ДЛЯ ПЕРЕМЕННЫХ

ARC также обеспечивает управление памятью для локальных переменных и переменных экземпляров. По умолчанию каждая переменная содержит сильную ссылку на объект. Это важно особенно для переменных экземпляров, поскольку в особых разновидностях кода семантика может отличаться от семантики ручного подсчета ссылок. Например, возьмем следующий код:

```
@interface EOCClass : NSObject {
    id _object;
}

@implementation EOCClass
- (void)setup {
    _object = [EOCOtherClass new];
}
@end
```

Переменная экземпляра `_object` не удерживает свое значение автоматически при ручном подсчете ссылок, но делает это в ARC.

Следовательно, когда метод `setup` компилируется для ARC, метод преобразуется в следующую форму:

```
- (void)setup {
    id tmp = [EOCOtherClass new];
    _object = [tmp retain];
    [tmp release];
}
```

Конечно, в этой ситуации `retain` и `release` можно исключить. ARC так и делает, оставляя такой же код, как показано выше. Это бывает удобно при написании set-методов. До ARC их приходилось писать примерно так:

```
- (void)setObject:(id)object {
    [_object release];
    _object = [object retain];
}
```

Но здесь возникает проблема. Что, если переменной присваивается значение, уже хранящееся в переменной? Если ссылка на нее удерживалась только этим объектом, `release` в set-методе приведет к уменьшению счетчика ссылок до 0 — и объект будет уничтожен. Последующий вызов `retain` приведет к сбою приложения. С ARC такие ошибки невозможны. Эквивалентный set-метод в ARC выглядит так:

```
- (void)setObject:(id)object {
    _object = object;
}
```

ARC выполняет безопасное присваивание переменной экземпляра, удерживая новое значение с освобождением старого перед итоговым присваиванием. Возможно, вы разобрались во всем при ручном подсчете ссылок и правильно написали свои set-методы, но с ARC вам не придется беспокоиться о таких пограничных случаях.

Семантика локальных переменных и переменных экземпляров может быть изменена применением следующих квалификаторов:

- `__strong` — используется по умолчанию; значение удерживается;
- `__unsafe_unretained` — значение не удерживается и является потенциально небезопасным, так как объект может оказаться уничтоженным к моменту повторного использования переменной;

- `__weak` — значение не удерживается, но является безопасным, потому что ему автоматически присваивается `nil`, если текущий объект когда-либо будет уничтожен;
- `__autoreleasing` — специальный квалификатор, используемый при передаче объекта по ссылке. Значение автоматически освобождается при возвращении.

Чтобы переменная экземпляра использовала такое же поведение, как без ARC, следует применить атрибут `__weak` или `__unsafe_unretained`:

```
@interface EOCClass : NSObject {
    id __weak _weakObject;
    id __unsafe_unretained _unsafeUnretainedObject;
}
```

В обоих случаях при присваивании значения переменной экземпляра объект удерживаться не будет. Автоматическое обнуление слабых ссылок с квалификатором `__weak` доступно только в новейших версиях исполнительной среды (Mac OS X 10.7 и iOS 5.0), потому что оно зависит от функциональности, добавленной в этих версиях.

Применение квалификаторов к локальным переменным часто используется для разрыва циклов удержания, возникающих при работе с блоками (см. подход 40). Блок автоматически удерживает все захваченные им объекты, что иногда может приводить к формированию циклов удержания, если объект, удерживающий блок, в свою очередь удерживается блоком. Для разрыва цикла удержаний может использоваться локальная переменная с квалификатором `__weak`:

```
NSURL *url = [NSURL URLWithString:@"http://www.example.com/"];
EOCNetworkFetcher *fetcher =
    [[EOCNetworkFetcher alloc] initWithURL:url];
EOCNetworkFetcher * __weak weakFetcher = fetcher;
[fetcher startWithCompletion:^(BOOL success){
    NSLog(@"Finished fetching from %@", weakFetcher.url);
}];
```

УПРАВЛЕНИЕ ПАМЯТЬЮ ПЕРЕМЕННЫХ ЭКЗЕМПЛЯРОВ В ARC

Как объяснялось выше, ARC также обеспечивает управление памятью переменных экземпляров. Для этого ARC приходится автоматически генерировать необходимый «код зачистки» во время

уничтожения. Любые переменные, содержащие сильную ссылку, должны освобождаться, для чего ARC подключается к методу `dealloc`. При ручном подсчете ссылок вам придется самостоятельно писать методы `dealloc`, который выглядит примерно так:

```
- (void)dealloc {
    [_foo release];
    [_bar release];
    [super dealloc];
}
```

При использовании ARC такие методы `dealloc` не нужны; сгенерированная функция зачистки выполнит два освобождения за вас. Эта возможность позаимствована из Objective-C++. Объект Objective-C++ должен вызвать деструкторы для всех объектов C++, удерживаемых объектом, во время уничтожения. Когда компилятор видит, что объект содержит объекты C++, он генерирует метод с именем `.cxx_destruct`. ARC «пристраивается» к этому методу и включает в него необходимый код зачистки.

Впрочем, вам все равно придется зачистить все объекты, не являющиеся объектами Objective-C, если они присутствуют в программе (как, например, объекты CoreFoundation или память, выделенная из кучи). Но вам не нужно вызывать реализацию `dealloc` суперкласса, как это делалось ранее. Вспомните, что явные вызовы `dealloc` в ARC запрещены. ARC не только генерирует и выполняет метод `.cxx_destruct` за вас, но и автоматически вызывает метод `dealloc` суперкласса. В ARC метод `dealloc` может выглядеть примерно так:

```
- (void)dealloc {
    CFRelease(_coreFoundationObject);
    free(_heapAllocatedMemoryBlob);
}
```

Тот факт, что ARC генерирует код уничтожения, означает, что в общем случае метод `dealloc` необязателен. Нередко это приводит к существенному сокращению размера исходного кода проекта и сокращению доли шаблонного (boilerplate) кода.

Переопределение методов управления памятью

До перехода на ARC методы управления памятью могли переопределяться. Например, синглетная реализация часто переопределяла `release` как пустую операцию, так как синглетный объект не может быть освобожден. В ARC такое переопределение недопустимо,

поскольку оно может помешать ARC правильно оценить жизненный цикл объекта. Кроме того, поскольку вызов и переопределение методов недопустимы, ARC в порядке оптимизации не использует систему диспетчеризации сообщений Objective-C (см. подход 11), когда потребуется выполнить `retain`, `release` или `autorelease`. Вместо этого оптимизация реализуется на базе функций C глубоко в исполнительной среде. Это позволяет ARC выполнять оптимизации наподобие описанной выше для возвращения автоматически освобождаемого объекта, который немедленно удерживается.

УЗЕЛКИ НА ПАМЯТЬ

- Механизм автоматического подсчета ссылок ARC (Automatic Reference Counting) избавляет разработчика от многих хлопот по управлению памятью. Использование ARC сокращает объем шаблонного кода в классах.
- ARC практически полностью управляет жизненным циклом объекта, добавляя удержания и освобождения там, где считает нужным. Для обозначения семантики управления памятью могут использоваться квалификаторы переменных; ранее удержания и освобождения приходилось выполнять вручную.
- Имена методов всегда использовались для обозначения семантики управления памятью возвращаемых объектов. В ARC действовавшие соглашения были закреплены, и их нарушение стало невозможно.
- ARC работает только с объектами Objective-C. В частности, это означает, что объекты CoreFoundation не обрабатываются и к ним должны применяться соответствующие вызовы `CFRetain/CFRelease`.

31 ОСВОБОЖДАЙТЕ ССЫЛКИ И ЗАЧИЩАЙТЕ СОСТОЯНИЕ НАБЛЮДЕНИЯ ТОЛЬКО В DEALLOC

Объект, проходя по фазам своего жизненного цикла, в конечном итоге уничтожается; в этой фазе вступает в действие метод `dealloc`. Он вызывается в жизненном цикле объекта ровно один раз: когда счетчик ссылок уменьшается до нуля. Впрочем, точный момент вызова не гарантирован. Кроме того, не стоит полагать, что из наблюдения за удержаниями и освобождениями можно узнать, когда

он будет вызван. На практике любая библиотека может манипулировать объектами без вашего ведома, из-за чего освобождение может произойти в другой момент времени. Никогда не пытайтесь вызывать метод `dealloc` сами. Исполнительная среда вызовет его в нужный момент за вас. Кроме того, после вызова `dealloc` для объекта этот объект становится недействительным, и последующие вызовы методов для него тоже недействительны.

Что же нужно делать в `dealloc`? Прежде всего освободить любые ссылки, принадлежащие объекту. Это означает освобождение всех объектов Objective-C (то, что ARC автоматически добавляет за вас в метод `dealloc` при помощи автоматического метода `.cxx_destruct` — см. подход 30). Все остальные объекты, не являющиеся объектами Objective-C, тоже необходимо освободить. Например, объекты CoreFoundation должны освобождаться явно, так как они относятся к API языка C.

Еще одна стандартная операция, выполняемая в методе `dealloc`, — зачистка всего состояния, связанного с наблюдением. Если объект `NSNotificationCenter` использовался для регистрации объекта на определенные оповещения, часто `dealloc` хорошо подходит для отмены регистрации, оповещения не отправлялись уничтоженному объекту (что наверняка привело бы к сбою приложения).

Метод `dealloc` выглядит примерно так:

```
- (void)dealloc {
    CFRelease(coreFoundationObject);
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

При использовании ручного подсчета ссылок вместо ARC последней выполняемой операцией должен быть вызов `[super dealloc]`. ARC добавляет его автоматически; это еще одна причина, по которой такое решение намного проще и безопаснее ручного подсчета ссылок. С ручным подсчетом ссылок в этом методе также необходимо вручную освобождать каждый объект Objective-C, принадлежащий объекту.

Вместе с тем в `dealloc` не следует освобождать ресурсы, дефицитные или сопряженные с высокими затратами: дескрипторы файлов, сокеты, большие блоки памяти. Вы не можете рассчитывать на то, что метод `dealloc` будет вызван в какое-то определенное время, потому что объект может удерживаться чем-то, о существовании чего вам неизвестно. В такой ситуации вы будете удерживать

дефицитные системные ресурсы на более длительное время, чем необходимо, а это нежелательно. В таких ситуациях обычно реализуется другой метод, который должен вызываться по завершении работы с объектом. При этом жизненный цикл ресурсов становится детерминированным.

Например, зачистка может понадобиться объекту, управляющему подключением к серверу (например, к базе данных) через сокет. Интерфейс такого класса может выглядеть так:

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

@interface EOCServerConnection : NSObject
- (void)open:(NSString*)address;
- (void)close;
@end
```

В соответствии с контрактом этого класса, метод `open:` вызывается для открытия подключения; после завершения работы с подключением приложение вызывает `close`. Вызов `close` должен быть выполнен до уничтожения объекта подключения; обратное считается ошибкой программиста — такой же, как несбалансированность удержаний и освобождений при подсчете ссылок.

Другая причина для освобождения ресурсов в отдельном методе зачистки заключается в том, что выполнение метода `dealloc` для каждого созданного объекта не гарантировано. Существует особый случай объектов, которые продолжают существовать по завершении приложения. Такие объекты не получают сообщение `dealloc`, а их уничтожение обусловлено возвратом ресурсов завершенного приложения операционной системе. Это оптимизация, основанная на отказе от вызова `dealloc`, но она означает, что вызов `dealloc` для каждого метода не гарантирован. В делегатах приложений Mac OS X и iOS присутствует метод, вызываемый по завершении приложения. Такие методы могут использоваться для выполнения методов зачистки для объектов, требующих гарантированной зачистки.

В случае Mac OS X по завершении приложения вызывается метод из экземпляра `NSApplicationDelegate`:

```
- (void)applicationWillTerminate:(NSNotification *)notification
```

Для iOS вызывается метод экземпляра `UIApplicationDelegate`:

```
(void)applicationWillTerminate:(UIApplication *)application
```

Что касается метода зачистки для объектов, управляющих ресурсами, он также должен вызываться в `dealloc` для исключения ситуации, в которой метод зачистки не был вызван. Если это произойдет, часто бывает полезно вывести в системный журнал строку с сообщением об ошибке программиста. А это именно ошибка программиста, потому что метод `close` должен быть вызван до уничтожения объекта. Эта строка укажет программисту на необходимость исправить ошибку. При этом закрывать ресурсы рекомендуется в `dealloc` для предотвращения утечек. Примеры таких методов `close` и `dealloc`:

```
- (void)close {
    /* clean up resources */
    _closed = YES;
}
- (void)dealloc {
    if (!_closed) {
        NSLog(@"ERROR: close was not called before dealloc!");
        [self close];
    }
}
```

Вместо простой регистрации ошибки при пропущенном вызове `close` можно выдать исключение, которое привлечет внимание программиста к серьезной ошибке.

В методах `dealloc` следует избегать вызова других методов. Конечно, в приведенном примере в `dealloc` вызывается метод, но это особый случай: выявление ошибки программиста. Вызывать любые другие методы нежелательно, потому что уничтожаемый объект находится в «предсмертном» состоянии. Если другой метод работает асинхронно или вызывает методы, которые работают асинхронно, уничтожаемый объект может полностью прекратить свое существование к тому моменту, когда эти методы завершат свою работу. В такой ситуации возникают всевозможные проблемы, которые часто приводят к сбою приложения из-за обратных вызовов, сообщающих объекту о завершении. Если объект не существует, такой вызов приведет к сбою.

Кроме того, метод `dealloc` вызывается в потоке, в котором произошло последнее освобождение, в результате которого счетчик ссылок упал до нуля. Некоторые методы должны выполняться в конкретном потоке — например, главном. Если эти методы вызываются из `dealloc`, не существует надежного способа гарантировать их выполнение в нужном потоке. Любой стандартный

код, обеспечивающий принудительное выполнение в нужном потоке, будет небезопасен, потому что объект находится в состоянии уничтожения, а исполнительная среда уже приступила к изменению своих внутренних структур данных для обозначения этого факта.

Нежелательность вызова методов в `dealloc` распространяется и на методы доступа к свойствам, которые могут переопределяться, — а следовательно, могут попытаться выполнить действия, небезопасные во время уничтожения. Также наблюдение за свойством может осуществляться через механизм KVO (Key-Value Observation), и наблюдатель может попытаться выполнить некоторую операцию (например, удержать объект) с объектом в процессе уничтожения. Это приведет к нарушению логической целостности состояния исполнительной среды и возникновению странных сбоев.

УЗЕЛКИ НА ПАМЯТЬ

- Метод `dealloc` следует использовать только для освобождения ссылок на другие объекты и отмены регистраций (например, регистрации на оповещения KVO (Key-Value Observing) или `NSNotificationCenter`.
- Если объект захватывает системные ресурсы (например, дескрипторы файлов), следует определить метод для освобождения таких ресурсов. Контракт пользователя должен предусматривать вызов пользователем метода `close` по завершении использования ресурсов.
- Постарайтесь обойтись без вызовов методов в методе `dealloc` на случай, если эти методы пытаются выполнять асинхронные операции или предполагают, что объект находится в нормальном состоянии.

32 ЗАЩИЩАЙТЕ УПРАВЛЕНИЕ ПАМЯТЬЮ С ПОМОЩЬЮ БЕЗОПАСНОГО КОДА

Механизм исключений поддерживается многими современными языками. Исключения не существуют в «чистом» C, но присутствуют в C++ и Objective-C. В современной исполнительной среде исключения C++ и Objective-C совместимы, то есть исключения, инициированные из одного языка, могут быть перехвачены обработчиком другого языка.

Несмотря на то что модель ошибок Objective-C (см. подход 21) предписывает использовать исключения только для фатальных ошибок, вам все равно может понадобиться код для перехвата и обработки всех исключений. В качестве примера можно привести код Objective-C++ или код, взаимодействующий со сторонней библиотекой так, что вы не можете управлять выдаваемыми исключениями. Кроме того, некоторые системные библиотеки все еще используют исключения — пережиток тех времен, когда исключения использовались повсеместно. Например, KVO выдает исключение при попытке отменить регистрацию наблюдателя, который не был зарегистрирован ранее.

Исключения создают интересную проблему в области управления памятью. Допустим, в блоке `try` объект удерживается, а затем до того, как он будет освобожден, выдается исключение. Такой объект создаст утечку памяти, если ситуация не будет обработана в блоке `catch`. В обработчиках исключений Objective-C выполняются деструкторы C++. Для C++ это важно, потому что любой объект, жизненный цикл которого был усечен выданным исключением, должен быть уничтожен; в противом случае возникнет утечка памяти, не говоря уже о других системных ресурсах — например, дескрипторах файлов. Среды с ручным подсчетом ссылок несколько затрудняют автоматическое уничтожение объектов в обработчиках исключений. Для примера возьмем следующий код Objective-C, использующий ручной подсчет ссылок:

```
@try {
    EOCSomeClass *object = [[EOCSomeClass alloc] init];
    [object doSomethingThatMayThrow];
    [object release];
}
@catch (...) {
    NSLog(@"Whoops, there was an error. Oh well…");
}
```

На первый взгляд, все правильно. Но что произойдет, если `doSomethingThatMayThrow` выдаст исключение? Вызов `release` в следующей строке не выполняется, потому что по исключению выполнение будет прервано с переходом к блоку `catch`. Таким образом, в этом сценарии выдача исключения приведет к утечке памяти объекта. Проблема решается при помощи блока `@finally`, который гарантированно будет выполнен один — и только один раз независимо от того, было выдано исключение или нет. Например, указанный код может быть приведен к следующему виду:

```
EOCSomeClass *object;
@try {
    object = [[EOCSomeClass alloc] init];
    [object doSomethingThatMayThrow];
}
@catch (...) {
    NSLog(@"Whoops, there was an error. Oh well…");
}
@finally {
    [object release];
}
```

Обратите внимание: объект приходится «вытаскивать» из блока `@try`, потому что на него необходимо ссылаться в блоке `@finally`. Очень быстро начинает утомлять, если это приходится проделывать для всех объектов, нуждающихся в освобождении. Кроме того, при использовании более сложной логики с несколькими командами в блоке `@try` легко упустить ситуацию, приводящую к утечке памяти объекта. Если такой объект является дефицитным ресурсом (или управляет им) — скажем, дескриптором файла или подключением к базе данных, утечка ведет к потенциальной катастрофе, потому что приложение начинает без необходимости удерживать все системные ресурсы.

С ARC ситуация только усугубляется. Эквивалентный код ARC для исходного кода выглядит так:

```
@try {
    EOCSomeClass *object = [[EOCSomeClass alloc] init];
    [object doSomethingThatMayThrow];
}
@catch (...) {
    NSLog(@"Whoops, there was an error. Oh well…");
}
```

Теперь проблема стала еще серьезнее: трюк с вынесением `release` в блоке `@finally` уже не может использоваться, потому что вызов `release` стал недопустимым. Но, конечно, ARC обрабатывает эту ситуацию, подумали вы? Вообще-то, по умолчанию не обрабатывает; для этого приходится добавлять большой объем шаблонного кода для отслеживания объектов, которым может понадобиться очистка в случае выдачи исключения. Этот код может заметно ухудшить быстродействие, даже если он не выдает исключений. Кроме того, весь дополнительный код значительно увеличивает размер приложения. Словом, побочные эффекты не из приятных.

Хотя по умолчанию такая возможность отключена, ARC поддерживает режим генерирования дополнительного кода для безопасной обработки исключений. Этот режим включается флагом компилятора -fobjc-arc-exceptions. А не включается по умолчанию он из-за того, что в программировании Objective-C исключения должны использоваться только в том случае, если в результате исключения приложение завершится (см. подход 21).

Таким образом, если приложение все равно завершится, потенциальная утечка памяти несущественна. Нет смысла добавлять код, необходимый для безопасности исключений, если приложение собирается завершиться.

Флаг -fobjc-arc-exceptions устанавливается по умолчанию только в одной ситуации: когда компилятор работает в режиме Objective-C++. В C++ уже должен присутствовать код, сходный с тем, который реализует ARC, так что, если ARC добавит собственный код для обеспечения безопасности исключений, потеря не так уж велика. Кроме того, исключения широко используются в C++, поэтому вероятность того, что разработчик Objective-C++ будет использовать исключения, тоже весьма велика.

Если вы используете ручной подсчет ссылок и вам необходимо перехватывать исключения, обязательно проследите за тем, чтобы код был написан с расчетом на правильное выполнение зачистки. Если вы используете ARC и вам необходимо перехватывать исключения, установите флаг -fobjc-arc-exceptions. Но самое главное — если вам приходится перехватывать много исключений, рассмотрите возможность рефакторинга для использования передачи ошибок в стиле `NSError`, как объясняется в подходе 21.

УЗЕЛКИ НА ПАМЯТЬ

- При перехвате исключений следует проследить за тем, чтобы для объектов, созданных в блоке `try`, выполнялась вся необходимая зачистка.
- По умолчанию ARC не генерирует код зачистки при выдаче исключений. Соответствующий режим можно включить флагом компилятора, но это приводит к увеличению размера кода и снижению быстродействия.

33 ИСПОЛЬЗУЙТЕ СЛАБЫЕ ССЫЛКИ, ЧТОБЫ ИЗБЕЖАТЬ УДЕРЖИВАЮЩИХ ЦИКЛОВ

В графе объектов часто возникает ситуация, в которой несколько объектов по кругу ссылаются друг на друга. В архитектурах с подсчетом ссылок (таких, как модель управления памятью Objective-C) это обычно приводит к утечке памяти, потому что в какой-то момент ни на один объект в цикле не существует внешних ссылок. Таким образом, обращение к объектам в цикле невозможно, но и уничтожаться они не будут, так как они «поддерживают жизнь» друг друга.

В простейшей разновидности циклов удержания два объекта ссылаются друг на друга. Пример показан на рис. 5.4.

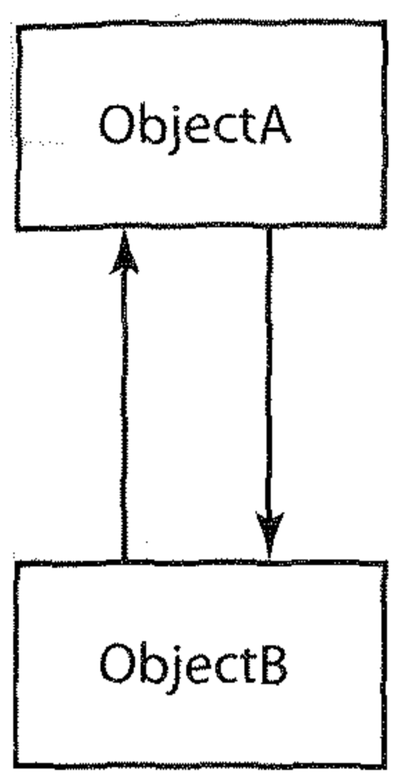


Рис. 5.4. Цикл удержания, в котором два объекта содержат сильные ссылки друг на друга

Такие циклы удержания просты для понимания, а чтобы их обнаружить, достаточно просмотреть код:

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

@class EOCClassA;
@class EOCClassB;

@interface EOCClassA : NSObject
@property (nonatomic, strong) EOCClassB *other;
@end

@interface EOCClassB : NSObject
```

```
@property (nonatomic, strong) EOCClassA *other;
@end
```

В этом коде легко угадывается потенциальный цикл удержания; если свойству `other` экземпляра `EOCClassA` присваивается экземпляр `EOCClassB` и наоборот с теми же экземплярами, образуется цикл удержания, показанный на рис. 5.4.

Результатом цикла удержания может быть утечка памяти. При освобождении последней ссылки на любой элемент цикла удержания весь цикл становится недоступным. На рис. 5.5 более сложный цикл удержания с участием четырех объектов приводит к утечке при удалении последней оставшейся ссылки на `ObjectB`.

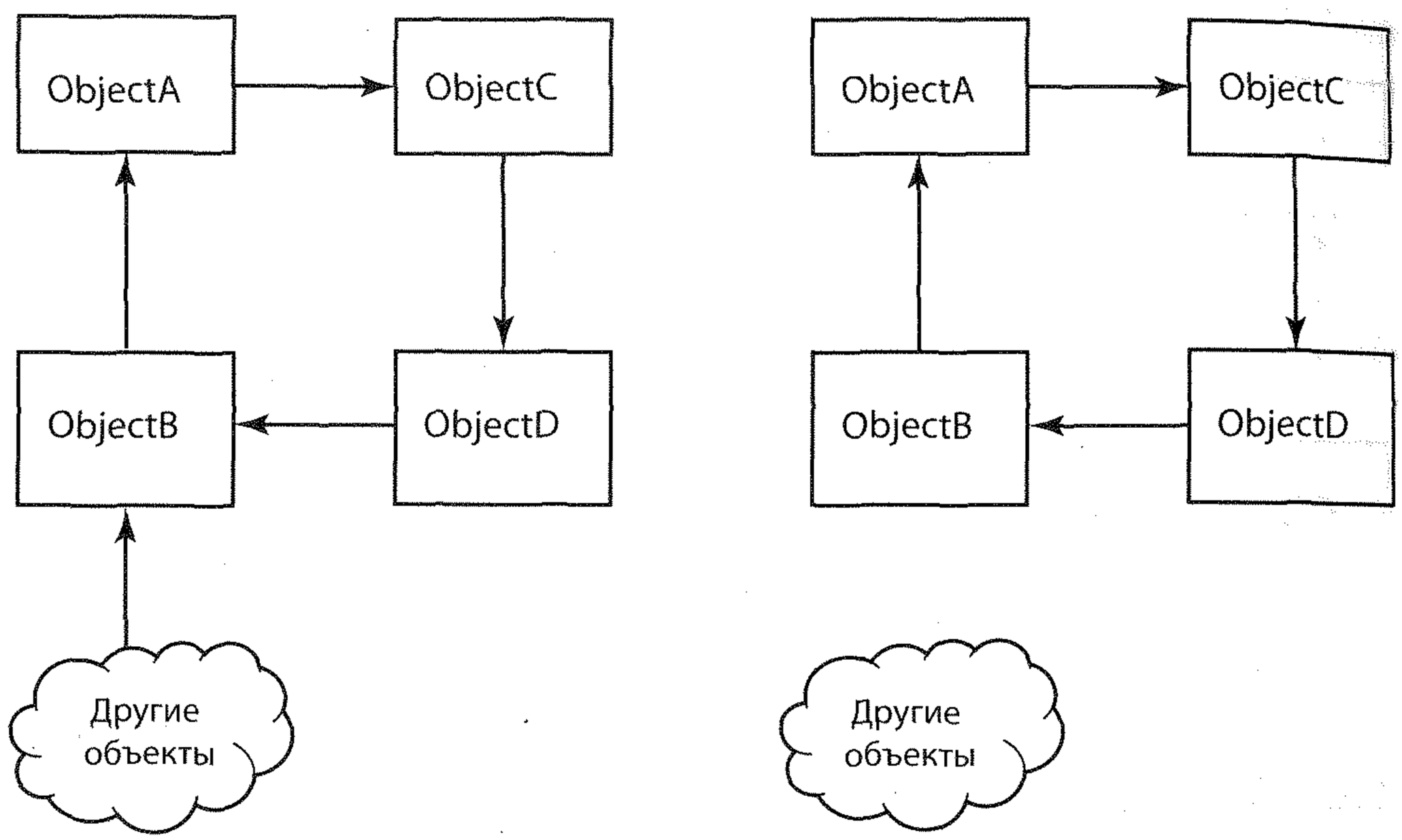


Рис. 5.5. В этом цикле удержания утечка возникает с удалением последней внешней ссылки на любой из объектов цикла

В приложениях Objective-C для Mac OS X можно использовать уборщик мусора, который обнаруживает циклы и уничтожает объекты, на которые нет других ссылок. Однако уборка мусора была официально объявлена устаревшей в Mac OS X 10.8, а в iOS она никогда не существовала. Таким образом, при написании кода необходимо помнить о проблеме циклов удержания и следить за тем, чтобы она не возникала.

Лучшим способом предотвращения циклов удержания является использование слабых ссылок. Также о таких ссылках говорят, что они представляют отношения без принадлежности. Задача может решаться при помощи атрибута `unsafe_unretained`. С применением этого атрибута приведенный пример выглядит так:

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

@class EOCClassA;
@class EOCClassB;

@interface EOCClassA : NSObject
@property (nonatomic, strong) EOCClassB *other;
@end

@interface EOCClassB : NSObject
@property (nonatomic, unsafe_unretained) EOCClassA *other;
@end
```

В этом случае экземпляр `EOCClassA` не принадлежит экземпляру `EOCClassB` через свойство `other`. Имя атрибута `unsafe_unretained` означает, что значение свойства потенциально небезопасно, и не удерживается экземпляром. Если этот объект был уничтожен, вызов его метода, скорее всего, приведет к сбою приложения.

Атрибут свойства `unsafe_unretained` семантически эквивалентен атрибуту `assign` (см. подход 6). Однако `assign` обычно используется только для целочисленных типов (`int`, `float`, `struct` и т. д.), а для объектных типов предпочтение отдается `unsafe_unretained`. Это своего рода самодокументирование, которое ясно показывает, что значение свойства потенциально небезопасно.

Вместе с ARC в исполнительной среде Objective-C появилась возможность создания безопасных слабых ссылок: новый атрибут свойств `weak` работает точно так же, как `unsafe_unretained`. С ним сразу же после уничтожения значения свойству немедленно автоматически присваивается `nil`. В нашем примере определение свойства `other` класса `EOCClassB` можно было бы заменить следующим:

```
@property (nonatomic, weak) EOCClassA *other;
```

Различия между свойствами `unsafe_unretained` и `weak` представлены на рис. 5.6.

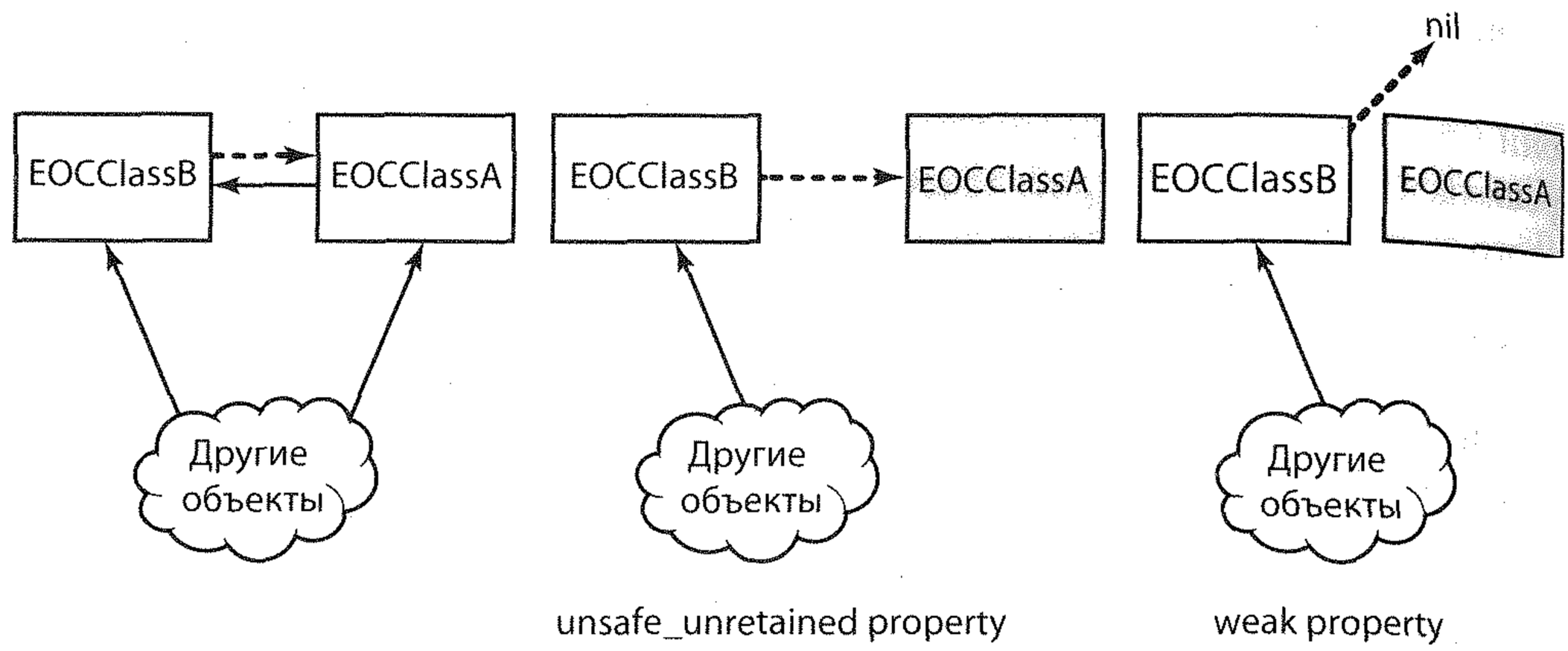


Рис. 5.6. Различия между атрибутами unsafe_unretained и weak при уничтожении объекта, на который указывает свойство

При удалении ссылки на экземпляр `EOCClassA` свойство `unsafe_unretained` продолжает указывать на экземпляр, который к этому моменту уже уничтожен. С атрибутом `weak` свойство содержит `nil`.

Впрочем, использование свойств `weak` — не оправдание для лентяев. В предыдущем примере тот факт, что экземпляр `EOCClassB` продолжает существовать при уничтожении объекта `EOCClassA`, на который указывает ссылка, следует рассматривать как ошибку. Если такое когда-либо произойдет, значит программа содержит ошибку. Постарайтесь следить за тем, чтобы такие ситуации не возникали. Тем не менее использование ссылок `weak` вместо `unsafe_unretained` повышает безопасность кода. Вместо сбоя приложение может вывести неправильные данные. Конечно, с точки зрения конечного пользователя этот способ куда более желателен, и все же сам факт преждевременного уничтожения объекта со ссылкой `weak` остается ошибкой. Допустим, у элемента пользовательского интерфейса имеется свойство источника данных, к которому он обращается для получения отображаемых данных. Обычно в таких свойствах используются слабые ссылки (см. подход 23). Если объект источника данных был уничтожен до того, как элемент завершит с ним работу, со слабой ссылкой приложение не «упадет», хотя и не будет выводить никакие данные.

Общее правило выглядит так: если объект вам не принадлежит, не удерживайте его. Одно из исключений составляют коллекции, которые часто не удерживают свое содержимое от имени объекта — владельца коллекции. Примером хранения в объекте ссылки на то, что ему не принадлежит, является паттерн «Делегат» (см. подход 23).

УЗЕЛКИ НА ПАМЯТЬ

- Для предотвращения циклов удержания можно объявить некоторые ссылки с атрибутом weak.
- Слабые ссылки могут использовать или не использовать автоматическое обнуление — новую возможность, представленную с ARC и реализованную в исполнительной среде. Автоматически обнуляемые слабые ссылки всегда безопасны для чтения, поскольку они никогда не содержат ссылок на уничтоженный объект.

ИСПОЛЬЗУЙТЕ ПУЛЫ АВТОМАТИЧЕСКОГО ОСВОБОЖДЕНИЯ, ЧТОБЫ УМЕНЬШИТЬ ЗАТРАТЫ ПАМЯТИ

К объектам Objective-C на протяжении их жизненного цикла применяется подсчет ссылок (см. подход 29). Одной из особенностей архитектуры подсчета ссылок Objective-C является концепция *пулов автоматического освобождения*. Освобождение объекта означает, что его счетчик ссылок либо уменьшается непосредственно вызовом `release`, либо добавляется в пул автоматического освобождения вызовом `autorelease`. В пуле автоматического освобождения собираются объекты, которые должны быть освобождены в какой-то момент в будущем. При заполнении пула всем объектам в пуле отправляется сообщение `release`.

Синтаксис создания пула автоматического освобождения выглядит так:

```
@autoreleasepool {
    // ...
}
```

Если при отправке объекту сообщения `autorelease` пул автоматического освобождения недоступен, на консоль выводится следующее сообщение:

```
Object 0xabcd0123 of class __NSCFString autoreleased
with no pool in place - just leaking - break on objc_
autoreleaseNoPool() to debug
```

Впрочем, скорее всего, вам не придется беспокоиться о пулах автоматического освобождения. Приложение, выполняемое в Mac OS X или iOS, работает в среде Cocoa (или Cocoa Touch для iOS). Потоки, создаваемые системой (например, главный поток или потоки, создаваемые в контексте GCD), имеют неявный пул автоматического освобождения, который очищается при каждом проходе цикла событий. Таким образом, вам не придется создавать собственные блоки пулов автоматического освобождения. Часто единственным блоком в приложении оказывается блок, который заключает основную точку входа приложения в функции `main`. Например, функция `main` в приложении iIS выглядит примерно так:

```
int main(int argc, char *argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc,
                                 argv,
                                 nil,
                                 @"EOCAppDelegate");
    }
}
```

Формально блок пула автоматического освобождения не является необходимым. Конец блока совпадает с завершением приложения, а в этой точке операционная система освобождает всю память, используемую приложением. Без блока у объектов, автоматически освобождаемых функцией `UIApplicationMain`, не будет пула автоматического освобождения — и вы получите предупреждение. Таким образом, этот пул может рассматриваться как внешний пул для перехвата всех объектов.

Фигурные скобки в следующем коде определяют область действия (scope) пула автоматического освобождения. Пул создается на первой фигурной скобке и автоматически очищается в конце области действия. Любому объекту, автоматически освобожденному в области действия, в ее конце отправляется сообщение `release`. Пулы автоматического освобождения могут быть вложенными. Автоматически освобождаемый объект добавляется во внутренний пул максимальной глубины, например:

```
@autoreleasepool {
    NSString *string = [NSString stringWithFormat:@"1 = %i", 1];
    @autoreleasepool {
        NSNumber *number = [NSNumber numberWithInt:1];
    }
}
```

В этом примере два объекта создаются фабричными методами, которые возвращают объекты автоматически освобождаемыми (см. подход 30). Объект `NSString` добавляется во внешний пул автоматического освобождения, а объект `NSNumber` — во внутренний пул. Вложением пулов автоматического освобождения можно воспользоваться для управления максимальными затратами памяти приложения.

Рассмотрим следующую часть кода:

```
for (int i = 0; i < 100000; i++) {
    [self doSomethingWithInt:i];
}
```

Если метод `doSomethingWithInt:` создает временные объекты, они с большой вероятностью попадают в пул автоматического освобождения (например, временные строки). Даже несмотря на то, что они не используются после конца метода, эти объекты могут быть «живыми», потому что они находятся в пуле автоматического освобождения, готовы к освобождению и последующему уничтожению. Однако пул не очищается до следующего прохождения цикла событий потока. Это означает, что в ходе выполнения цикла `for` новые объекты создаются и добавляются в пул автоматического освобождения. В конечном итоге, после его завершения объекты будут освобождены. Однако в ходе выполнения цикла объем потребляемой памяти стабильно увеличивается, после чего внезапно падает при освобождении временных объектов.

Ситуация неидеальна, особенно если продолжительность такого цикла зависит от пользовательского ввода. Допустим, вы читаете набор объектов из базы данных. Код чтения может выглядеть так:

```
NSArray *databaseRecords = /* … */;
NSMutableArray *people = [NSMutableArray new];
for (NSDictionary *record in databaseRecords) {
    EOCPerson *person = [[EOCPerson alloc]
                         initWithRecord:record];
    [people addObject:person];
}
```

Инициализатор `EOCPerson` может создавать дополнительные временные объекты, как в приведенном примере. При большом количестве записей возрастает количество временных объектов, живущих дольше, чем это действительно необходимо. В таких ситуациях стоит добавить пул автоматического освобождения. Если

внутренняя часть цикла упакована в блок пула автоматического освобождения, то все объекты, автоматически освобождаемые внутри цикла, включаются в этот пул вместо главного пула потока, например:

```
NSArray *databaseRecords = /* … */;
NSMutableArray *people = [NSMutableArray new];
for (NSDictionary *record in databaseRecords) {
    @autoreleasepool {
        EOCPerson *person =
            [[EOCPerson alloc] initWithRecord:record];
        [people addObject:person];
    }
}
```

Добавление нового пула автоматического освобождения приводит к снижению максимальных затрат памяти приложения в ходе выполнения цикла, поскольку в конце блока будут уничтожаться временные объекты.

Для наглядности можно считать, что пулы автоматического освобождения образуют стек. При создании пул автоматического освобождения добавляется в стек; при очистке он извлекается из стека. Объект, подлежащий автоматическому освобождению, помещается в пул, находящийся на вершине стека.

Необходимость в такой оптимизации полностью зависит от приложения. Прежде чем применять ее, безусловно, необходимо сначала проанализировать затраты памяти приложения и решить, стоит ли это делать. Пулы автоматического освобождения не создают особых дополнительных затрат, но, если эти затраты не оправданны, от них следует отказаться.

Если у вас имеется опыт программирования на Objective-C до перехода на ARC, вы наверняка помните старый синтаксис с использованием объекта `NSAutoreleasePool`. Это был специальный объект, имитировавший пул автоматического освобождения в новом блочном синтаксисе. Вместо того чтобы очищать пул при каждом прохождении цикла `for`, этот объект обычно создавал один пул и очищал его в заданный момент:

```
NSArray *databaseRecords = /* … */;
NSMutableArray *people = [NSMutableArray new];
int i = 0;

NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
```

```
for (NSDictionary *record in databaseRecords) {
    EOCPerson *person = [[EOCPerson alloc]
                                initWithRecord:record];
    [people addObject:person];

    // Пул очищается через каждые 10 итераций
    if (++i == 10) {
        [pool drain];
    }
}

// Очистка также выполняется в конце цикла на случай,
// если количество итераций не кратно 10
[pool drain];
```

Подобный стиль программирования ушел в прошлое. Вместе с новым синтаксисом в ARC были существенно снижены затраты на создание пулов автоматического освобождения. Таким образом, если у вас имеется код с очисткой пула через каждые n итераций цикла, его можно заменить заключением цикла `for` в блок пула автоматического освобождения; в этом случае пул будет создаваться и очищаться для каждой итерации.

Другое преимущество синтаксиса `@autoreleasepool` заключается в том, что с каждым пулом автоматического освобождения связывается область действия, что предотвращает случайное использование объектов, уничтоженных при очистке пула автоматического освобождения. Возьмем следующий код в старом стиле:

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
id object = [self createObject];
[pool drain];
[self useObject:object];
```

Конечно, это некоторое преувеличение, но суть понятна. При вызове `useObject:` передается потенциально уничтоженный объект. В новом стиле тот же код будет выглядеть так:

```
@autoreleasepool {
    id object = [self createObject];
}
[self useObject:object];
```

На этот раз код даже не откомпилируется, потому что объектная переменная недействительна за пределами блока, а следовательно, не может использоваться при вызове `useObject:`.

УЗЕЛКИ НА ПАМЯТЬ

- Пулы автоматического освобождения образуют стек. Объект, получающий сообщение `autorelease`, помещается в пул на вершине стека.
- Правильное применение пулов автоматического освобождения сокращает максимальные затраты памяти приложения.
- Современная реализация пулов автоматического освобождения с новым синтаксисом `@autoreleasepool` связана с минимальными затратами ресурсов.

ИСПОЛЬЗУЙТЕ ОБЪЕКТЫ-ЗОМБИ ДЛЯ РЕШЕНИЯ ПРОБЛЕМ, СВЯЗАННЫХ С УПРАВЛЕНИЕМ ПАМЯТЬЮ

Отладка проблем, связанных с управлением памятью, может быть весьма неприятным делом. Понятно, что отправка сообщения уничтоженному объекту является небезопасной операцией. Но иногда она работает, а иногда нет. Все зависит от того, было ли перезаписано содержимое памяти, в которой находился объект. Повторное использование памяти является недетерминированным фактором, поэтому сбои могут происходить непредсказуемо. В других ситуациях память может быть переписана частично, так что часть объекта остается действительной. Может случиться и так, что по чистому совпадению память будет перезаписана другим действительным, живым объектом. Исполнительная среда отправляет новому объекту сообщение, на которое он может реагировать или не реагировать. В первом случае приложение работает без сбоев, но объекты, которые не должны были получать сообщения, вдруг начинают получать их. Во втором случае приложение обычно завершается аварийно.

К счастью, в Cocoa существует весьма полезная возможность. При включении отладочного зомби-режима исполнительная среда преобразует все уничтожаемые экземпляры в специальный *объект-зомби* (вместо их фактического уничтожения). Память, в которой находится такой объект, не становится доступной для повторного использования; следовательно, ее содержимое не будет перезаписываться. При получении любого сообщения объект-зомби выдает исключение с точной информацией о том, какое сообщение было

отправлено и какие данные содержал объект, пока был жив. Зомби-режим — лучший способ отладки проблем, связанных с управлением памятью.

Чтобы включить этот режим, присвойте переменной окружения `NSZombieEnabled` значение `YES`. Например, если вы используете bash и запускаете приложение в Mac OS X, это делается следующим образом:

```
export NSZombieEnabled="YES"
./app
```

При отправке сообщения зомби-объекту информация выводится на консоль, а приложение завершается. Сообщение выглядит примерно так:

```
*** -[CFString respondsToSelector:]: message sent to
deallocated instance 0x7ff9e9c080e0
```

Также можно включить в Xcode режим автоматической установки переменной при запуске приложения из Xcode. Для этого включите редактирование схемы (scheme) приложения, выберите конфигурацию Run, перейдите на вкладку Diagnostics и установите флажок Enable Zombie Objects. На рис. 5.7 изображено диалоговое окно, которое отображается в Xcode, с установленным флажком включения зомби-объектов.

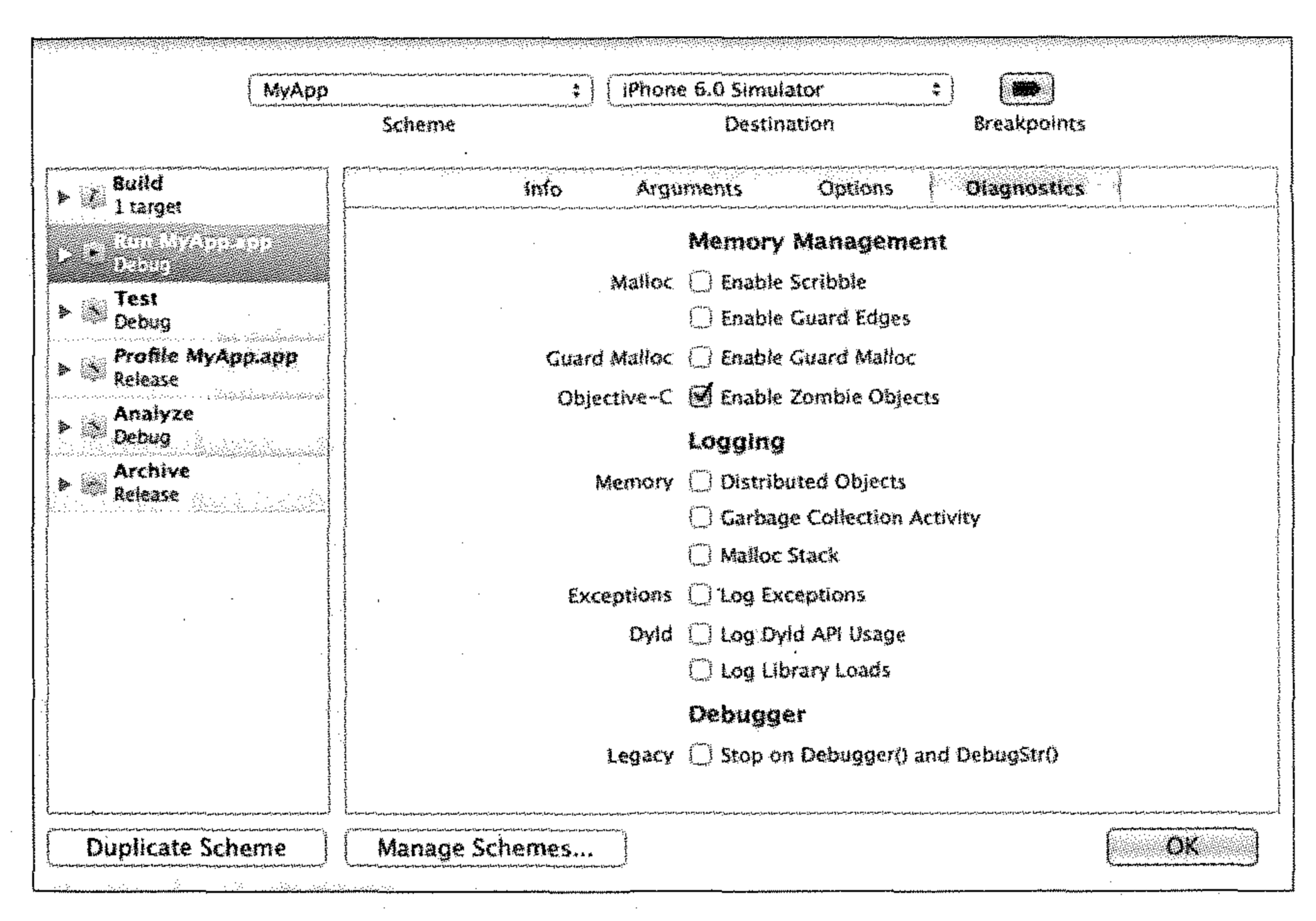


Рис. 5.7. Включение зомби-объектов в редакторе схемы Xcode

Как же работает режим объектов-зомби? Он реализован глубоко внутри исполнительной среды Objective-C, библиотек Foundation и CoreFoundation. Если переменная окружения установлена, то при уничтожении объекта выполняется дополнительное действие: вместо обычного уничтожения объект преобразуется в зомби.

Чтобы понять, что происходит на этом дополнительном шаге, рассмотрим следующий фрагмент:

```
#import <Foundation/Foundation.h>
#import <objc/runtime.h>

@interface EOCClass : NSObject
@end

@implementation EOCClass
@end

void PrintClassInfo(id obj) {
    Class cls = object_getClass(obj);
    Class superCls = class_getSuperclass(cls);
    NSLog(@"=== %s : %s ===",
          class_getName(cls), class_getName(superCls));
}

int main(int argc, char *argv[]) {
    EOCClass *obj = [[EOCClass alloc] init];
    NSLog(@"Before release:");
    PrintClassInfo(obj);

    [obj release];
    NSLog(@"After release:");
    PrintClassInfo(obj);
}
```

В этом коде используется ручной подсчет ссылок. Это поможет вам понять, что происходит, когда объект становится зомби. ARC следит за тем, чтобы объект `str` оставался живым столько, сколько необходимо; это означает, что он никогда не станет зомби в этом простом сценарии. Из сказанного не следует, что в ARC объекты не могут стать зомби. Такие ошибки управления памятью возможны и в ARC, но обычно они проявляются в более сложном коде.

В приведенном примере код содержит функцию для вывода имени класса и суперкласса для заданного объекта. Вместо отправки клас-

су сообщения Objective-C в коде используется функция времени выполнения `object_getClass()`.

Отправка любого сообщения объекту-зомби приводит к выводу сообщения об ошибке и сбою приложения. Результат выглядит примерно так:

```
Before release:
=== EOCClass : NSObject ===
After release:
=== _NSZombie_EOCClass : nil ===
```

Класс объекта изменился с `EOCClass` на `_NSZombie_EOCClass`. Но откуда взялся этот класс? В коде он не определяется. Кроме того, для компилятора было бы весьма неэффективно создавать лишний класс для каждого найденного класса просто на случай включения зомби-режима. На самом деле класс `_NSZombie_EOCClass` генерируется исполнительной средой в первый раз, когда объект класса `EOCClass` превращается в зомби. При этом используются мощные функции времени выполнения, оперирующие со списком классов.

Класс-зомби строится на основе шаблонного класса `_NSZombie_`. Классы-зомби в целом не делают ничего особенного, а просто действуют как маркеры. Вскоре вы увидите, как именно это происходит, но сначала рассмотрим фрагмент псевдокода, который показывает, как класс-зомби создается при необходимости и как он используется для преобразования уничтожаемого объекта в зомби.

```
// Получение класса уничтожаемого объекта
Class cls = object_getClass(self);

// Получение имени класса
const char *clsName = class_getName(cls);

// Присоединение префикса _NSZombie_ к имени класса
const char *zombieClsName = "_NSZombie_" + clsName;

// Проверка существования класса-зомби
Class zombieCls = objc_lookUpClass(zombieClsName);

// Если заданный класс-зомби не существует, его необходимо
создать
if (!zombieCls) {
    // Получение шаблонного класса-зомби с именем _NSZombie_
    Class baseZombieCls = objc_lookUpClass("_NSZombie_");
```

```
        // Дублирование базового класса-зомби
        // с присоединением имени нового класса
        zombieCls = objc_duplicateClass(baseZombieCls,
                                        zombieClsName, 0);
    }

    // Выполнение нормального уничтожения объекта
    objc_destructInstance(self);

    // Класс уничтожаемого объекта заменяется классом-зомби
    objc_setClass(self, zombieCls);

    // Теперь 'self' относится к классу _NSZombie_OriginalClass
```

Эта функция становится методом `dealloc` класса `NSObject`. Когда исполнительная среда видит, что переменная окружения `NSZombieEnabled` установлена, она заменяет (см. подход 13) метод `dealloc` версией, выполняющей этот фрагмент. В конце кода класс объекта заменяется на `_NSZombie_`*OriginalClass*, где *OriginalClass* — прежнее имя класса.

Принципиально то, что память, в которой находится объект, не освобождается (вызовом `free()`); следовательно, она недоступна для повторного использования. Это приводит к утечке памяти, но данный режим предназначен только для отладки; он никогда не должен включаться в реальных приложениях, поэтому утечка не так важна.

Но зачем создавать новый класс для каждого класса, преобразуемого в зомби? Чтобы при отправке сообщения зомби можно было определить исходный класс. Если бы все зомби относились к классу `_NSZombie_`, исходное имя класса было бы потеряно. Новый класс создается функцией `objc_duplicateClass()`, которая копирует весь класс и присваивает ему новое имя. По суперклассу, переменным экземпляров и методам класс-дубликат идентичен копируемому классу. Того же результата — сохранения имени старого класса — можно было достигнуть наследованием от `_NSZombie_` вместо копирования. Однако функции наследования существенно менее эффективны, чем функции прямого копирования.

Класс-зомби начинает участвовать в происходящем в функциях перенаправления (см. подход 12). Класс `_NSZombie_` (а следовательно, и все его копии) не реализует никаких методов. Класс не имеет суперкласса, то есть является корневым (как и `NSObject`); он содержит единственную переменную экземпляра `isa`, которая

должна присутствовать во всех корневых классах Objective-C. Этот минимальный класс не реализует методов, поэтому все отправляемые ему сообщения проходят через полный механизм перенаправления (см. подход 12).

В основе полного механизма перенаправления лежит функция `___forwarding___`, которую вы, возможно, встречали в данных трассировки во время отладки. Одной из первых операций, выполняемых этой функцией, является проверка имени класса объекта, которому отправляется сообщение. Если имя имеет префикс `_NSZombie_`, значит, это зомби, и для него запускается специальная обработка. В этот момент приложение уничтожается после вывода информации (см. начало подхода) с указанием отправленного сообщения и типа класса. Здесь-то вам и пригодится тот факт, что в имени класса-зомби присутствует имя исходного класса.

Префикс `_NSZombie_` удаляется из имени класса-зомби, и в нем остается только исходное имя. Следующий фрагмент псевдокода показывает, что при этом происходит:

```
// Получение класса объекта
Class cls = object_getClass(self);
// Получение имени класса
const char *clsName = class_getName(cls);

// Проверка наличия префикса _NSZombie_ в имени класса
if (string_has_prefix(clsName, "_NSZombie_") {
    // Если префикс присутствует, объект является зомби

    // Получение имени исходного класса (суффикс _NSZombie_
    // пропускается, т.е. извлекается подстрока
                                после 10 символа)
    const char *originalClsName = substring_from(clsName, 10);

    // Получение имени селектора сообщения
    const char *selectorName = sel_getName(_cmd);

    // Вывод информации с указанием того, какой селектор
    // и какому зомби отправляется
    Log("*** -[%s %s]: message sent to deallocated instance %p",
        originalClsName, selectorName, self);

    // Уничтожение приложения
    abort();
}
```

Чтобы понять, как работает этот код, расширим пример для отправки сообщения объекту-зомби `EOCClass`:

```
EOCClass *obj = [[EOCClass alloc] init];
NSLog(@"Before release:");
PrintClassInfo(obj);
[obj release];
NSLog(@"After release:");
PrintClassInfo(obj);
NSString *desc = [obj description];
```

Если выполнить этот код с включенным зомби-режимом, на консоль выводится следующая информация:

```
Before release:
=== EOCClass : NSObject ===
After release:
=== _NSZombie_EOCClass : nil ===
*** -[EOCClass description]: message sent to deallocated
instance 0x7fc821c02a00
```

Как видите, в выводе указывается отправленный селектор и исходный класс объекта, а также значение указателя мертвого объекта, которому было отправлено сообщение. Эта информация может использоваться в отладчике для дальнейшего анализа и оказать бесценную помощь с правильным инструментарием — например, программой Instruments из поставки Xcode.

УЗЕЛКИ НА ПАМЯТЬ

- При уничтожении объект может быть преобразован в объект-зомби. Данная возможность включается переменной окружения `NSZombieEnabled`.
- Преобразование в объект-зомби осуществляется посредством манипуляций с указателем `isa` и замены фактического класса объекта специальным классом-зомби. Этот класс реагирует на все селекторы, завершая работу приложения после выдачи информации с указанием того, какой селектор и какому объекту был отправлен.

36 ОСТЕРЕГАЙТЕСЬ МЕТОДА RETAINCOUNT

Objective-C использует подсчет ссылок для управления памятью (см. подход 29). С каждым объектом связан счетчик, определяющий, сколько других сторон заинтересовано в продолжении его существования. При создании объекта счетчик ссылок имеет положительное значение. Операции удержания и освобождения приводят соответственно к увеличению и уменьшению счетчика. Когда счетчик достигает нуля, объект уничтожается.

Метод, определяемый протоколом `NSObject`, позволяет узнать текущее значение счетчика ссылок для объекта:

```
- (NSUInteger)retainCount
```

Однако с появлением ARC этот метод считается устаревшим. Более того, компилятор выдает ошибку при попытке его использования с ARC, как и при попытке использования таких методов, как `retain`, `release` и `autorelease`. Даже при том, что этот метод официально считается устаревшим, разработчики часто понимают его неправильно, а использовать его не рекомендуется. Если вы не используете ARC (и совершенно напрасно), этот метод можно вызывать без получения ошибки компилятора. По этой причине важно понимать, почему использовать этот метод не рекомендуется.

На первый взгляд кажется, что метод полезен. Ведь он возвращает счетчик ссылок, а это, безусловно, важная информация о каждом объекте. Проблема в том, что абсолютное значение счетчика ссылок часто совершенно несущественно. Даже если вы используете этот метод исключительно в отладочных целях, пользы от него обычно не бывает.

Первая важная причина, по которой этот метод бесполезен, состоит в том, что он возвращает значение счетчика ссылок на конкретный момент времени. Так как значение не учитывает изменения, происходящие из-за последующей очистки пула автоматического освобождения (см. подход 34), это значение не обязательно отражает реальное состояние счетчика. Таким образом, следующий код очень плох:

```
while ([object retainCount]) {
    [object release];
}
```

Во-первых, этот код уменьшает счетчик ссылок вплоть до уничтожения объекта без учета всех незавершенных автоматических освобождений. Если объект также находится в пуле автоматического освобождения, при очистке пула объект будет освобожден лишний раз, а это наверняка приведет к сбою приложения.

Во-вторых, `retainCount` никогда не возвращает 0; из-за оптимизации в поведении освобождения объектов объект освобождается, если его счетчик ссылок равен 1. В противном случае происходит уменьшение счетчика. Следовательно, счетчик ссылок никогда «официально» не достигает 0. К сожалению, этот код иногда работает — в основном из-за чистого везения. В современных исполнительных средах при итерации цикла `while` после уничтожения объекта обычно просто происходит сбой.

Такой код ни при каких условиях не является необходимым. В программе не должно быть несбалансированных освобождений, оставляющих объект с положительным счетчиком ссылок, когда объект, по вашему мнению, должен быть уничтожен. В такой ситуации следует заняться выяснением того, что именно до сих пор удерживает объект, не позволяя освободить его.

При попытке использования счетчика ссылок разработчики также часто удивляются, почему его значение кажется несуразно большим. Возьмем следующий код:

```
NSString *string = @"Some string";
NSLog(@"string retainCount = %lu", [string retainCount]);
NSNumber *numberI = @1;
NSLog(@"numberI retainCount = %lu", [number retainCount]);
NSNumber *numberF = @3.141f;
NSLog(@"numberF retainCount = %lu", [numberFloat retainCount]);
```

Вывод этого кода в Mac OS X 10.8.2 при 64-разрядной компиляции в Clang 4.1 выглядит так:

```
string retainCount = 18446744073709551615
numberI retainCount = 9223372036854775807
numberF retainCount = 1
```

Первое число равно $2^{64}-1$, а второе — $2^{63}-1$. Счетчики ссылок этих объектов очень велики, потому что они представляют синглетные объекты. `NSString` реализуется как синглетный объект, если это возможно — например, если строка является константой времени компиляции, как в приведенном примере. В этом случае компилятор создает специальный объект, помещает данные объекта `NSString`

в двоичный файл приложения и использует их вместо того, чтобы создавать объект NSString во время выполнения. С NSNumber компилятор действует аналогично, используя концепцию *помеченных указателей* (tagged pointers) для некоторых типов значений. В этой схеме объект NSNumber отсутствует; само значение указателя содержит всю информацию о числе. Исполнительная среда обнаруживает факт использования помеченного указателя в процессе диспетчеризации сообщений (см. подход 11) и выполняет соответствующие манипуляции со значением указателя, имитируя наличие полного объекта NSNumber. Впрочем, эта оптимизация выполняется только в некоторых ситуациях; именно поэтому для вещественного числа в этом примере выводится значение 1 (для него эта оптимизация не используется).

Кроме того, счетчики ссылок подобных синглетных объектов вообще не изменяются. Удержание и освобождение реализуются как пустые операции. Сама возможность такого поведения счетчиков (и даже то, что для двух синглетных объектов возвращаются разные значения счетчиков) снова показывает, что на это значение нельзя полагаться. Если ваш код рассчитан на то, что объекты NSNumber имеют увеличивающиеся и уменьшающиеся счетчики ссылок, а потом к ним будут применены помеченные указатели, код станет работать неверно.

Но что, если вы хотите использовать счетчики ссылок для отладочных целей? Даже в этом случае пользы от них не будет. Принадлежность объекта к пулу автоматического освобождения нарушает точность счетчика. Другие библиотеки могут повлиять на счетчик ссылок, удерживая и/или освобождая объект. При проверке значения счетчика вы можете ошибочно предположить, что он изменился из-за вашего кода, а не из-за причин, скрытых где-то в другой библиотеке. Для примера возьмем следующий код:

```
id object = [self createObject];
[opaqueObject doSomethingWithObject:object];
NSLog(@"retainCount = %lu", [object retainCount]);
```

Какое значение содержит счетчик ссылок? Какое угодно. Вызов doSomethingWithObject: мог добавить объект в несколько коллекций с удержанием его в процессе. Или же вызов мог несколько раз удержать объект и несколько раз автоматически освободить его, причем некоторые очистки пула автоматического освобождения еще не были выполнены. Скорее всего, конкретное значение счетчика окажется бесполезным для вас.

Когда использовать метод `retainCount`? Самый правильный ответ: никогда, особенно теперь, когда компания Apple официально объявила его устаревшим при использовании ARC.

УЗЕЛКИ НА ПАМЯТЬ

- Счетчик ссылок объекта может показаться полезным, но это впечатление обманчиво, потому что абсолютное значение счетчика в любой момент времени не дает полной картины жизненного цикла объекта.
- С появлением ARC метод `retainCount` официально объявлен устаревшим, а попытка его использования приводит к ошибке компиляции.

Глава 6
Блоки и Grand Central Dispatch

Многопоточное программирование — тема, актуальная для любого разработчика в контексте разработки современных приложений. Даже если вы не собирались делать свое приложение многопоточным, скорее всего, оно будет многопоточным, потому что системные библиотеки используют дополнительные потоки для выполнения работы вне потока пользовательского интерфейса. Нет ничего хуже приложения, парализованного из-за блокировки главного потока. В Mac OS X на экране появляется ненавистный вращающийся шарик; в iOS слишком долгая блокировка может привести к аварийному завершению приложения.

К счастью, компания Apple предложила совершенно новый подход к реализации многопоточности. Ключевыми особенностями современного многопоточного программирования являются блоки и Grand Central Dispatch (GCD). И хотя это совершенно разные технологии, появились они одновременно. Блоки предоставляют лексические замыкания (closures) в C, C++ и Objective-C; они чрезвычайно полезны — прежде всего тем, что они предоставляют механизм передачи фрагментов кода так, словно те являются объектами, для выполнения в другом контексте. При этом очень важно, что блоки могут использовать все что угодно из той области действия, в которой они определяются.

Сопутствующая технология GCD предоставляет абстракцию для многопоточного выполнения, базирующуюся на *очередях диспетчеризации* (dispatch queues). Блоки ставятся в эти очереди, а GCD обеспечивает все планирование: создает, повторно использует

и уничтожает фоновые потоки по своему усмотрению для обработки очередей с учетом системных ресурсов. Кроме того, GCD предоставляет простые решения стандартных задач программирования — например, потоково-безопасного выполнения кода и параллельного выполнения задач в зависимости от доступных системных ресурсов.

Блоки и GCD — одно из важнейших направлений современного программирования на Objective-C, поэтому вы должны понимать, как они работают и какие возможности предоставляют разработчику.

37 РАЗБЕРИТЕСЬ С БЛОКАМИ

Блоки предоставляют замыкания. Эта языковая функция, появившаяся как расширение компилятора GCC, доступна во всех современных версиях Clang (проект компилятора для разработки приложений Mac OS X и iOS). Компоненты исполнительной среды, необходимые для правильного функционирования блоков, доступны во всех версиях Mac OS X начиная с 10.4 и iOS начиная с 4.0. Технически этот механизм относится к уровню C, а следовательно, может использоваться в коде C, C++, Objective-C и Objective-C++, откомпилированном поддерживаемым компилятором и запущенном с наличием необходимых компонентов исполнительной среды.

ЗНАКОМСТВО С БЛОКАМИ

Блок отчасти напоминает функцию, но определяется внутри другой функции и использует область действия, в которой он был определен. Блоки обозначаются символом «крышка» (^), за которым следует реализация, заключенная в фигурные скобки. Простейший блок выглядит примерно так:

```
^{
    // Реализация блока
}
```

Блок представляет собой значение, и за ним закрепляется тип. Блок, как и значение `int`, `float` или объект Objective-C, можно присвоить переменной, а затем использовать, как любую другую переменную. Синтаксис типа блока напоминает синтаксис указателя на функцию. Простой пример блока, который не получает параметров и не возвращает значения:

```
void (^someBlock)() = ^{
    // Реализация блока
};
```

Блок определяет переменную с именем `someBlock`. На первый взгляд немного странно, что имя переменной находится в середине, но стоит разобраться в синтаксисе — и он легко читается. Синтаксис типа блока имеет следующую структуру:

```
возвращаемый_тип (^имя_блока)(параметры)
```

Например, определение блока, который возвращает `int` и получает два параметра `int`, выглядит так:

```
int (^addBlock)(int a, int b) = ^(int a, int b){
    return a + b;
};
```

Далее блок используется так, как если бы он был функцией. Блок `addBlock` из приведенного примера мог бы использоваться следующим образом:

```
int add = addBlock(2, 5); //< add = 7
```

У блоков есть одна выдающаяся особенность: они *захватывают* (capture) область видимости, в которой объявляются. Это означает, что все переменные, доступные для области действия, в которой объявляется блок, также доступны внутри блока. Например, можно определить блок, который использует другую переменную, следующим образом:

```
int additional = 5;
int (^addBlock)(int a, int b) = ^(int a, int b){
    return a + b + additional;
};
int add = addBlock(2, 5); //< add = 12
```

По умолчанию переменные, захваченные блоком, не могут им модифицироваться. Если в приведенном примере попытаться изменить переменную с именем `additional` в блоке, будет выдана ошибка. Однако переменные могут объявляться как изменяемые при помощи квалификатора `__block`. Например, блок может использоваться в перечислении массива (см. подход 48) для подсчета количества чисел в массиве, меньших 2:

```
NSArray *array = @[@0, @1, @2, @3, @4, @5];
__block NSInteger count = 0;
```

```
[array enumerateObjectsUsingBlock:
    ^(NSNumber *number, NSUInteger idx, BOOL *stop){
        if ([number compare:@2] == NSOrderedAscending) {
            count++;
        }
    }];
// count = 2
```

В этом примере также продемонстрировано использование встроенного (inline) блока. Блок, передаваемый методу `enumerateObjectsUsingBlock:`, не присваивается локальной переменной, а объявляется как встроенный в вызове метода. Этот распространенный паттерн программирования наглядно показывает, чем так удобны блоки. Прежде чем блоки стали частью языка, в приведенном фрагменте пришлось бы передать указатель на функцию или имя селектора, вызываемого методом перечисления. Состояние пришлось бы передавать туда и обратно вручную — обычно через непрозрачный указатель на `void`, что потребовало бы добавления нового кода и дробления метода. Объявление встроенного блока означает, что вся бизнес-логика сосредоточена в одном месте.

При захвате переменной объектного типа блок неявно удерживает ее. Переменная будет освобождена при освобождении самого блока. Это приводит к очень важной особенности блоков: сам блок может рассматриваться как объект. Собственно, блоки реагируют на многие селекторы, на которые реагируют другие объекты Objective-C. Но важнее всего понимать, что к блоку, как и ко всем остальным объектам, применяется подсчет ссылок. При исчезновении последней ссылки на блок он уничтожается. При этом все объекты, захваченные блоком, освобождаются, что компенсирует их удержание блоком.

Если блок определяется как метод экземпляра класса Objective-C, переменная `self` доступна наряду с переменными экземпляров класса. Переменные экземпляров всегда доступны для записи, и их не нужно явно объявлять с квалификатором `__block`. Но если переменная экземпляра захватывается посредством чтения или записи в нее, переменная `self` тоже неявно захватывается, потому что переменная экземпляра неявно относится к этому экземпляру. Для примера рассмотрим следующий блок в методе класса `EOCClass`:

```
@interface EOCClass

- (void)anInstanceMethod {
    // …
```

```
    void (^someBlock)() = ^{
        _anInstanceVariable = @"Something";
        NSLog(@"_anInstanceVariable = %@", _anInstanceVariable);
    };
    // …
}

@end
```

На конкретный экземпляр `EOCClass`, для которого выполняется метод `anInstanceMethod`, ссылается переменная `self`. Легко забыть, что `self` захватывается блоками такого рода, потому что в коде она явно не используется. Однако обращение к переменной экземпляра эквивалентно следующей команде:

```
self->_anInstanceVariable = @"Something";
```

Понятно, почему переменная `self` захватывается. Как правило, для обращения к переменным экземпляров будут использоваться свойства (см. подход 6), и тогда переменная `self` указывается явно:

```
self.aProperty = @"Something";
```

При этом важно помнить, что `self` является объектом, а следовательно, удерживается при захвате блоком. Эта ситуация часто приводит к возникновению циклов удержания, если сам блок удерживается тем же объектом, на который ссылается `self`. За дополнительной информацией обращайтесь к подходу 40.

ВНУТРЕННЕЕ СТРОЕНИЕ БЛОКА

Каждый объект в Objective-C занимает некоторую область памяти. Для всех объектов эти области имеют разные размеры, в зависимости от количества переменных экземпляров и содержащихся в них данных. Блок тоже является объектом, поскольку первая переменная в области памяти, в которой определяется блок, представляет собой указатель на объект `Class`, называемый указателем `isa` (см. подход 14). Остальная часть памяти блока содержит различные данные, необходимые для его правильного функционирования. На рис. 6.1 изображено строение блока в памяти.

Самое важное, на что следует обратить внимание в этой структуре, — переменная с именем `invoke`, содержащая указатель на функцию с реализацией блока. Прототип функции получает как минимум `void*`, то есть сам блок. Вспомните, что блоки представляют

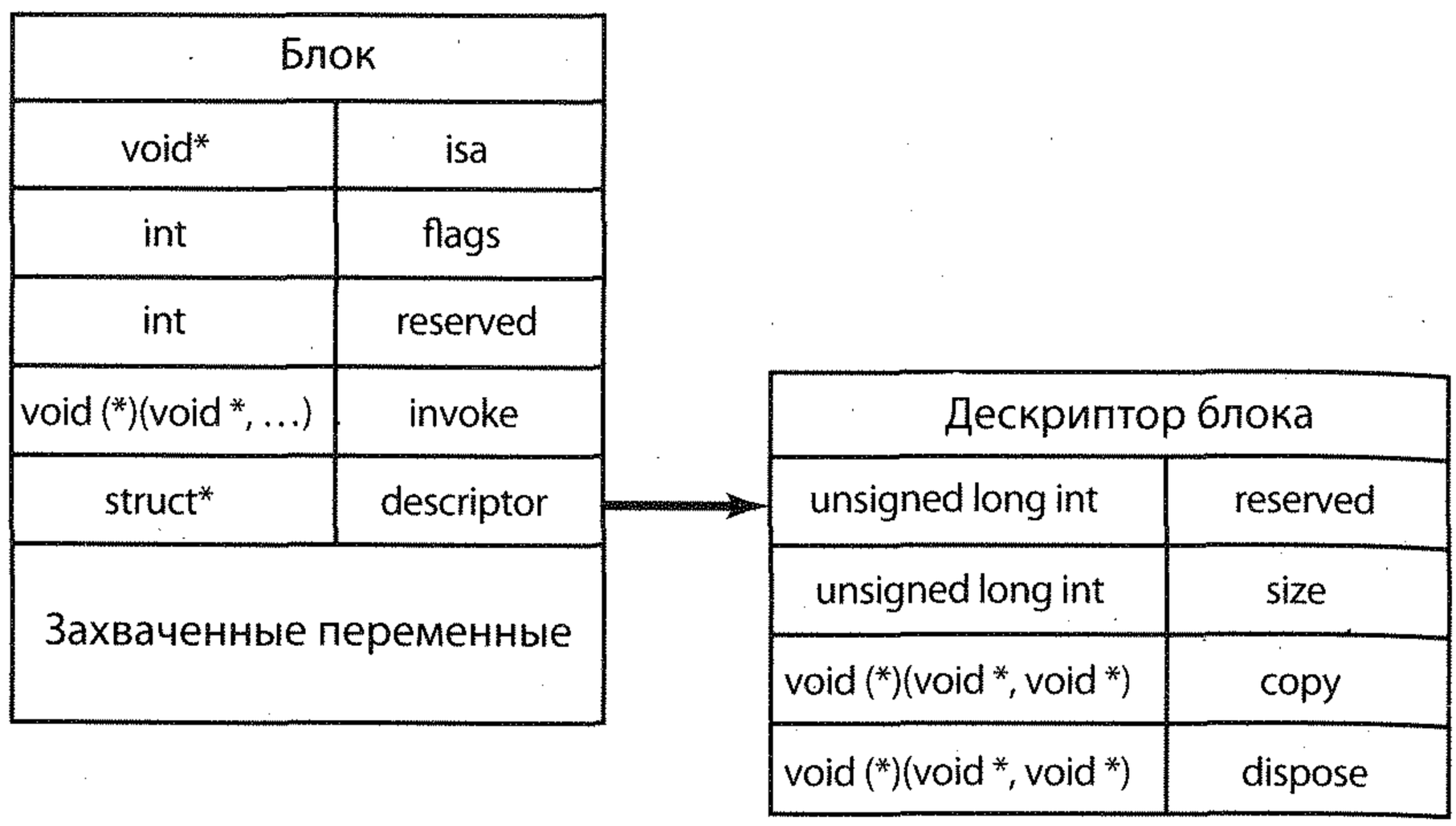


Рис. 6.1. Структура памяти объекта блока

собой простую замену для указателей на функции, с передачей состояния при помощи непрозрачного указателя на `void`. Блок упаковывает то, что раньше делалось стандартными языковыми средствами C, в компактный и простой в использовании интерфейс.

Переменная `descriptor` содержит указатель на дескриптор — структуру, связываемую с каждым блоком. В дескрипторе хранится общий размер объекта блока, а также указатели на функции для вспомогательных методов `copy` и `dispose`. Они выполняются при копировании и уничтожении блока — например, для выполнения соответственно удержания или освобождения захваченных объектов.

Наконец, в блоке содержатся копии всех захваченных им переменных. Эти копии следуют после переменной `descriptor` и занимают столько места, сколько необходимо для хранения всех захваченных переменных. Обратите внимание: копируются не сами объекты, а только переменные с указателями. При выполнении блока захваченные переменные читаются из этой области памяти; именно поэтому блок должен передаваться в параметре функции `invoke`.

ГЛОБАЛЬНЫЕ БЛОКИ, СТЕК И КУЧА

При определении блока занимаемая им область памяти выделяется из стека. Это означает, что блок действителен только в той области действия, в которой он определен. Например, следующий код представляет потенциальную опасность:

```
void (^block)();
if ( /* условие */ ) {
```

```
    block = ^{
        NSLog(@"Block A");
    };
} else {
    block = ^{
        NSLog(@"Block B");
    };
}
block();
```

Память для двух блоков, определяемых в `if` и `else`, выделяется в стеке. При этом компилятор может перезаписать эту память в конце области действия, в которой эта память была выделена. Таким образом, каждый блок гарантированно действителен только в соответствующей секции команды `if`. Код откомпилируется без ошибки, но во время выполнения может функционировать как правильно, так и неправильно. Если компилятор не сгенерировал код, который записывается на место выбранного блока, код выполняется без ошибки, но в противном случае неизбежен сбой.

Для решения этой проблемы блоки можно копировать, отправляя им сообщение `copy`. При этом блок копируется из стека в кучу. После копирования блок может использоваться за пределами области действия, в которой он был определен. Кроме того, после копирования в кучу блок становится объектом, к которому применяется подсчет ссылок. Все последующие операции копирования ограничиваются простым увеличением счетчика ссылок этого блока. Когда ссылок на блок в куче не остается, он должен освобождаться либо автоматически (при использовании ARC), либо явным вызовом `release` (с ручным подсчетом ссылок). При уменьшении счетчика ссылок до нуля блок в куче уничтожается, как и любой другой объект. Блок в стеке не нуждается в явном освобождении, поскольку память в стеке освобождается автоматически: поэтому предыдущий пример кода был потенциально опасным.

Учитывая все сказанное, для обеспечения безопасности этого кода достаточно применить пару вызовов `copy`:

```
void (^block)();
if ( /* some condition */ ) {
    block = [^{
        NSLog(@"Block A");
    } copy];
} else {
    block = [^{
```

```
        NSLog(@"Block B");
    } copy];
}
block();
```

Теперь этот код безопасен. Если бы в нем использовался ручной подсчет ссылок, блок также пришлось бы освободить после завершения его использования.

Глобальные блоки — еще одна разновидность блоков наряду с блоками в стеке и в куче. Блоки, не захватывающие никакого состояния (скажем, переменные из замыкающей области действия), не нуждаются в получении состояния для выполнения. Вся область памяти, используемая этими блоками, полностью известна во время компиляции; таким образом, глобальные блоки объявляются в глобальной памяти, вместо того чтобы создаваться в стеке при каждом использовании. Кроме того, копирование глобального блока является пустой операцией, а глобальные блоки никогда не уничтожаются. Фактически они представляют собой синглетные объекты. Пример глобального блока:

```
void (^block)() = ^{
    NSLog(@"This is a block");
};
```

Вся информация, необходимая для выполнения этого блока, известна во время компиляции, что позволяет сделать блок глобальным. Это всего лишь оптимизация для исключения лишней работы, которую пришлось бы выполнять для более сложных блоков, требующих особой реализации копирования и уничтожения.

УЗЕЛКИ НА ПАМЯТЬ

- Блоки представляют собой лексические замыкания в языках C, C++ и Objective-C.
- Блоки могут получать параметры и возвращать значения.
- Блоки могут создаваться в стеке, в куче или в глобальной памяти. Блок, созданный в стеке, может быть скопирован в кучу; с этого момента к нему применяется подсчет ссылок, как к любому стандартному объекту Objective-C.

38 СОЗДАЙТЕ TYPEDEF ДЛЯ ЧАСТО ИСПОЛЬЗУЕМЫХ ТИПОВ БЛОКОВ

Блоки имеют определенный тип; таким образом, они могут присваиваться переменным соответствующего типа. Тип определяется параметрами, получаемыми блоком, и возвращаемым типом. Рассмотрим следующий блок:

```
^(BOOL flag, int value){
    if (flag) {
        return value * 5;
    } else {
        return value * 10;
    }
}
```

Блок получает два параметра типа `BOOL` и `int` и возвращает значение типа `int`. Чтобы блок можно было присвоить переменной, он должен иметь соответствующий тип. Тип и операция присваивания выглядят так:

```
int (^variableName)(BOOL flag, int value) =
    ^(BOOL flag, int value){
        // Реализация
        return someInt;
    }
```

Синтаксис заметно отличается от обычного типа, но он покажется знакомым каждому, кто работал с указателями на функции. Тип имеет следующую структуру:

```
возвращаемый_тип (^имя_блока)(параметры)
```

Определение блочной переменной отличается от других типов тем, что имя переменной находится в середине типа, а не справа. Это усложняет как чтение, так и запоминание синтаксиса. По этим причинам рекомендуется создавать определения типов для часто используемых блочных типов, особенно если вы поставляете API, который будет использоваться другими. Типы блоков скрываются за удобочитаемым именем, которое дает представление о том, что должен делать блок.

Для сокрытия сложных типов блоков используется возможность языка C, называемая *определением типа*. Ключевое слово `typedef` позволяет определить удобочитаемое имя, которое становится

синонимом для другого типа. Например, чтобы определить новый тип для блока, возвращающего `int` и получающего два параметра — `BOOL` и `int`, — используется следующее определение:

```
typedef int(^EOCSomeBlock)(BOOL flag, int value);
```

Имя нового типа, как и имя блочной переменной при присваивании, находится в середине с префиксом `^`. Таким образом, в систему типов вводится новый тип с именем `EOCSomeBlock`. Итак, вместо создания переменной со сложным типом можно просто использовать новый тип:

```
EOCSomeBlock block = ^(BOOL flag, int value){
    // Реализация
};
```

Этот код читается намного проще, так как определение переменной находится слева, а имя переменной — справа, где мы их привыкли видеть для других переменных.

С определениями типов намного проще работать с API, в которых используются блоки. Любой класс, получающий блок в параметре метода (например, обработчик завершения асинхронной задачи), может использовать определения типов, чтобы упростить чтение кода. Для примера возьмем класс с методом `start`, которому передается блок-обработчик, выполняемый по завершении задачи. Без определения типа сигнатура метода выглядит примерно так:

```
- (void)startWithCompletionHandler:
            (void(^)(NSData *data, NSError *error))completion;
```

Обратите внимание: синтаксис типа блока снова отличается от синтаксиса определения переменной. Он намного проще читается, если тип в сигнатуре метода будет состоять из одного слова. Таким образом, мы можем создать определение типа и использовать его:

```
typedef void(^EOCCompletionHandler)
                            (NSData *data, NSError *error);
- (void)startWithCompletionHandler:
            (EOCCompletionHandler)completion;
```

Этот фрагмент проще читается и поясняет, что собой представляет параметр. Любая современная интегрированная среда разработки (IDE) автоматически расширяет определения типов для удобства разработчика.

Определения типов также пригодятся, если вам когда-нибудь потребуется изменить сигнатуру типа блока. Например, если вы решите, что обработчику завершения нужно передавать дополнительный параметр с временем, потраченным на выполнение задачи, достаточно изменить определение типа:

```
typedef void(^EOCCompletionHandler)
        (NSData *data, NSTimeInterval duration,
                                NSError *error);
```

Все фрагменты, в которых используется определение типа (например, сигнатуры методов), перестанут компилироваться; вы можете перебрать их и внести исправления. Без определения типа вам пришлось бы изменять множество типов в своем коде. При этом легко упустить одно-два вхождения, что приведет к трудноуловимым ошибкам.

Обычно рекомендуется приводить определения типов вместе с классом, в котором они используются. Также желательно снабдить имя нового типа префиксом класса, использующего определение типа; это сделает использование блока более очевидным. Если для одного типа сигнатуры блока создано несколько определений типов — ничего страшного. Избыток типов лучше их нехватки.

Пример такого рода встречается в библиотеке Accounts для Mac OS X и iOS. В ней среди прочего определяются следующие определения типов блоков:

```
typedef void(^ACAccountStoreSaveCompletionHandler)
                                (BOOL success, NSError *error);
typedef void(^ACAccountStoreRequestAccessCompletionHandler)
                                (BOOL granted, NSError *error);
```

Эти определения типов блоков имеют одинаковые сигнатуры, но используются в разных местах. Имя типа и имена параметров в сигнатуре помогают разработчику понять, как должен использоваться этот тип. Разработчик мог бы создать одно определение типа (например, `ACAccountStoreBooleanCompletionHandler`) и использовать его вместо этих двух определений. При этом будет потеряна наглядность использования блоков и параметров.

Аналогичным образом, если у вас имеются классы, выполняющие похожие, но не идентичные асинхронные задачи, которое не образуют иерархию классов, каждый класс должен иметь собственный тип обработчика завершения. Сигнатуры обработчиков могут полностью совпадать, но лучше использовать отдельные определе-

ния типов для каждого класса вместо одного общего определения. С другой стороны, если классы образуют иерархию, определение типа можно было бы включить в базовый класс, чтобы оно использовалось каждым субклассом.

УЗЕЛКИ НА ПАМЯТЬ

- Определения типов упрощают работу с блочными переменными.
- Всегда соблюдайте соглашения об именах при определениях новых типов, чтобы избежать конфликтов с другими типами.
- Не бойтесь определять несколько типов для одной сигнатуры блока. Возможно, вам потребуется внести изменения в одном месте, использующем конкретный тип блока, но оставить другие места без изменений.

39 ИСПОЛЬЗУЙТЕ БЛОКИ В ОБРАБОТЧИКАХ, ЧТОБЫ УМЕНЬШИТЬ ЛОГИЧЕСКОЕ РАЗБИЕНИЕ КОДА

Асинхронное выполнение задач стало стандартной парадигмой программирования пользовательского интерфейса. Поток, отвечающий за вывод пользовательского интерфейса и обработку касаний, не блокируется при выполнении продолжительных операций (например, ввода/вывода или сетевых операций). Этот поток часто называется *главным потоком* (main thread). Если бы такие методы выполнялись синхронно, пользовательский интерфейс переставал бы реагировать на происходящее во время выполнения задач. При определенных обстоятельствах приложение, не отвечающее на запросы в течение некоторого времени, автоматически завершается. В частности, это относится к приложениям iOS; системный сторож завершает приложение, главный поток которого остается заблокированным на определенный период времени.

Асинхронным методам необходим механизм оповещения заинтересованных сторон о завершении их работы. Существует несколько стандартных решений. Часто применяется протокол делегата (см. подход 23), который реализуется объектом. Объект-делегат оповещается о важных событиях — например, о завершении асинхронной задачи.

Рассмотрим класс для выборки данных по URL-адресу. При использовании паттерна «Делегат» класс может выглядеть примерно так:

```
#import <Foundation/Foundation.h>
@class EOCNetworkFetcher;
@protocol EOCNetworkFetcherDelegate <NSObject>
- (void)networkFetcher:(EOCNetworkFetcher*)networkFetcher
     didFinishWithData:(NSData*)data;
@end

@interface EOCNetworkFetcher : NSObject
@property (nonatomic, weak)
                        id <EOCNetworkFetcherDelegate> delegate;
- (id)initWithURL:(NSURL*)url;
- (void)start;
@end
```

Класс может использовать такие API следующим образом:

```
- (void)fetchFooData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/foo.dat"];
    EOCNetworkFetcher *fetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    fetcher.delegate = self;
    [fetcher start];
}

// …

- (void)networkFetcher:(EOCNetworkFetcher*)networkFetcher
     didFinishWithData:(NSData*)data
{
    _fetchedFooData = data;
}
```

Это решение работает, и неправильным его не назовешь. Однако блоки позволяют добиться того же результата существенно более удобным способом. API, использующий блоки, становится компактнее и чище. Разработчик определяет тип блока, который используется как обработчик завершения, передаваемый непосредственно методу `start`:

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

typedef void(^EOCNetworkFetcherCompletionHandler)(NSData *data);
```

```
@interface EOCNetworkFetcher : NSObject
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
            (EOCNetworkFetcherCompletionHandler)handler;
@end
```

Это решение очень похоже на использование протокола делегата, но у него есть дополнительное преимущество: обработчик завершения может определяться «на месте», прямо при вызове метода `start`, что значительно упрощает чтение кода. Возьмем класс, использующий API с блоком завершения:

```
- (void)fetchFooData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/foo.dat"];
    EOCNetworkFetcher *fetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [fetcher startWithCompletionHandler:^(NSData *data){
        _fetchedFooData = data;
    }];
}
```

По сравнению с кодом, использующим паттерн «Делегат», решение с блоком смотрится намного элегантнее. Бизнес-логика, выполняемая по завершении асинхронной задачи, располагается прямо рядом с кодом, запускающим эту задачу. Кроме того, поскольку блок объявляется в той же области действия, в которой создается объект загрузки данных, в нем доступны все переменные, доступные в этой области действия. В приведенном простом примере это не нужно, но в более сложной ситуации такая возможность может быть очень полезной.

У решения с паттерном «Делегат» имеется недостаток: если класс использует несколько объектов загрузки фрагментов данных, в методе делегата приходится использовать конструкцию выбора в зависимости от того, от какого объекта поступил обратный вызов. Код может выглядеть примерно так:

```
- (void)fetchFooData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/foo.dat"];
    _fooFetcher = [[EOCNetworkFetcher alloc] initWithURL:url];
    _fooFetcher.delegate = self;
    [_fooFetcher start];
}
```

```
- (void)fetchBarData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/bar.dat"];
    _barFetcher = [[EOCNetworkFetcher alloc] initWithURL:url];
    _barFetcher.delegate = self;
    [_barFetcher start];
}

- (void)networkFetcher:(EOCNetworkFetcher*)networkFetcher
     didFinishWithData:(NSData*)data
{
    if (networkFetcher == _fooFetcher) {
        _fetchedFooData = data;
        _fooFetcher = nil;
    } else if (networkFetcher == _barFetcher) {
        _fetchedBarData = data;
        _barFetcher = nil;
    }
    // И так далее.
}
```

Помимо удлинения обратного вызова делегата, этот код означает, что объекты загрузки данных придется сохранять в переменных экземпляра для проверки. Возможно, такое сохранение все равно необходимо по другим причинам (например, для последующий отмены при необходимости); и все же чаще оно оказывается побочным эффектом, приводящим к быстрому загромождению класса. Итак, в решении с блоками объекты загрузки данных не нужно сохранять, и оно не требует выбора. Вместо этого бизнес-логика каждого обработчика завершения определяется вместе с объектом загрузки данных:

```
- (void)fetchFooData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/foo.dat"];
    EOCNetworkFetcher *fetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [fetcher startWithCompletionHandler:^(NSData *data){
        _fetchedFooData = data;
    }];
}

- (void)fetchBarData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/bar.dat"];
```

```
    EOCNetworkFetcher *fetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [fetcher startWithCompletionHandler:^(NSData *data){
        _fetchedBarData = data;
    }];
}
```

Многие современные API также используют блоки для обработки ошибок; можно считать это расширением концепции. Возможны два подхода. Во-первых, для сбоев и успешных ситуаций могут использоваться разные обработчики. Во-вторых, сбойный случай может быть упакован в тот же блок завершения. Пример использования отдельного обработчика может выглядеть так:

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

@class EOCNetworkFetcher;
typedef void(^EOCNetworkFetcherCompletionHandler)(NSData
*data);
typedef void(^EOCNetworkFetcherErrorHandler)(NSError *error);

@interface EOCNetworkFetcher : NSObject
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
            (EOCNetworkFetcherCompletionHandler)completion
        failureHandler:
            (EOCNetworkFetcherErrorHandler)failure;
@end
```

Пример использования такого стиля API:

```
EOCNetworkFetcher *fetcher =
    [[EOCNetworkFetcher alloc] initWithURL:url];
[fetcher startWithCompletionHander:^(NSData *data){
    // Обработка успеха
}
                    failureHandler:^(NSError *error){
    // Обработка неудачи
}];
```

Этот стиль хорош тем, что код успеха в нем отделяется от кода неудачи; это означает, что код пользователя тоже будет логически разделен на ветви успеха и неудачи, благодаря чему такой код будет проще читаться. Кроме того, он позволяет игнорировать успех или неудачу в случае необходимости.

Другой стиль реализации, при котором код успеха и неудачи объединяется в одном блоке, выглядит так:

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

@class EOCNetworkFetcher;
typedef void(^EOCNetworkFetcherCompletionHandler)
                                (NSData *data, NSError *error);

@interface EOCNetworkFetcher : NSObject
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
                (EOCNetworkFetcherCompletionHandler)
completion;
@end
```

Пример кода, использующего этот стиль API:

```
EOCNetworkFetcher *fetcher =
    [[EOCNetworkFetcher alloc] initWithURL:url];
[fetcher startWithCompletionHander:
    ^(NSData *data, NSError *error){
        if (error) {
            // Обработка успеха
        } else {
            // Обработка неудачи
        }
    }];
```

В этом решении вся логика находится в одном месте, а ветвление осуществляется в зависимости от значения переменной `error`. Так как вся логика находится в одном месте, блок может стать длинным и сложным. С другой стороны, решение с одним блоком получается более гибким. Например, в нем можно передать как ошибку, так и данные. Представьте, что состояние сети позволило загрузить половину данных, после чего произошла ошибка. Возможно, в этом случае вы предпочтете вернуть загруженные данные и ошибку. Обработчик завершения выявляет проблему, обрабатывает ее, а затем делает что-то полезное с успешно загруженной частью данных.

Для включения кода успеха и неудачи в один блок есть и другая причина: иногда при обработке данных вроде бы успешного ответа пользователь обнаруживает ошибку (допустим, возвращенные данные оказались слишком короткими). Возможно, с точки зрения объекта загрузки данных ситуация будет обрабатываться так же,

как и неудачный случай. В таком случае наличие одного блока означает, что такая обработка возможна, а обнаруженная ошибка в данных может обрабатываться в ветви ошибки загрузки. Если успех и неудача будут разделены по разным обработчикам, то совместное использование кода обработки ошибок в таком сценарии становится невозможным без перемещения этого кода в отдельный метод, а это противоречит исходной цели использования блоков для размещения бизнес-логики в одном месте.

В общем случае я рекомендую использовать один блок обработчика для успеха и неудачи; компания Apple выбирает именно этот способ в своих API. Например, класс `TWRequest` из библиотеки Twitter и класс `MKLocalSearch` из библиотеки MapKit используют один блок обработчика.

Другая причина для использования блоков обработчиков — возможность обратного вызова в нужный момент времени. Например, пользователь объекта загрузки данных может захотеть, чтобы по мере загрузки данных ему отправлялись оповещения о ходе операции. Конечно, нужного результата можно добиться при помощи делегата, но в продолжение темы использования блоков обработчиков также можно добавить тип блока обработчика и свойство:

```
typedef void(^EOCNetworkFetcherCompletionHandler)
                                        (float progress);
@property (nonatomic, copy)
              EOCNetworkFetcherProgressHandler progressHandler;
```

Это удобная схема, которая, как и прежде, позволяет разместить всю бизнес-логику в одном месте: обработчик завершения определяется непосредственно при создании объекта загрузки данных.

Другой фактор, который необходимо принять во внимание при написании API с обработчиками, связан с необходимостью выполнения кода в определенном потоке. Например, все операции пользовательского интерфейса в Cocoa и Cocoa Touch должны выполняться в главном потоке (эквивалент главной очереди в контексте GCD). Таким образом, иногда требуется разрешить пользователю API с обработчиками решить, в какой очереди должен выполняться обработчик. Например, у класса `NSNotificationCenter` имеется метод для регистрации на оповещения о выполнении некоторого блока. Возможно (хотя и не обязательно) решить, в какой очереди должен планироваться блок. Если очередь не задана, то используется поведение по умолчанию и блок выполняется в потоке, отправившем оповещение. Метод добавления наблюдателя выглядит так:

```
- (id)addObserverForName:(NSString*)name
                  object:(id)object
                   queue:(NSOperationQueue*)queue
              usingBlock:(void(^)(NSNotification*))block
```

Очередь, в которой должен выполняться блок при выдаче оповещения, задается передаваемым объектом NSOperationQueue. При этом вместо низкоуровневых очередей GCD используются очереди операций, но семантика остается той же. (Отличия очередей GCD от других инструментов рассматриваются в подходе 43.)

Вы тоже можете спроектировать API с передачей очереди операций — или даже очереди GCD, если вы предпочитаете управлять работой API на этом уровне.

УЗЕЛКИ НА ПАМЯТЬ

- Используйте блок обработчика в тех случаях, когда бизнес-логику обработчика удобно объявить «на месте» при создании объекта.
- К преимуществам блоков обработчиков относится то, что они связываются непосредственно с объектом — в отличие от делегирования, при котором часто приходится использовать конструкцию выбора в зависимости от объекта (если отслеживаются несколько экземпляров).
- При проектировании API с блоками обработчиков можно передать в параметре очередь, в которую должен быть поставлен блок.

ИЗБЕГАЙТЕ ЦИКЛОВ УДЕРЖАНИЯ МЕЖДУ БЛОКАМИ И ОБЪЕКТАМИ, КОТОРЫМ ОНИ ПРИНАДЛЕЖАТ

Если разработчик недостаточно осмотрителен, использование блоков часто приводит к возникновению циклов удержания. Например, следующий класс предоставляет интерфейс для загрузки данных с некоторого URL-адреса. При запуске загрузки данных может быть указан блок обратного вызова, называемый *обработчиком завершения* (completion handler), который выполняется при завершении загрузки. Обработчик завершения должен храниться в переменной экземпляра, чтобы он был доступен при вызове метода, запрашивающего завершение.

```
// EOCNetworkFetcher.h
#import <Foundation/Foundation.h>

typedef void(^EOCNetworkFetcherCompletionHandler)(NSData
*data);

@interface EOCNetworkFetcher : NSObject
@property (nonatomic, strong, readonly) NSURL *url;
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
            (EOCNetworkFetcherCompletionHandler)completion;
@end

// EOCNetworkFetcher.m
#import "EOCNetworkFetcher.h"

@interface EOCNetworkFetcher ()
@property (nonatomic, strong, readwrite) NSURL *url;
@property (nonatomic, copy)
          EOCNetworkFetcherCompletionHandler completionHandler;
@property (nonatomic, strong) NSData *downloadedData;
@end

@implementation EOCNetworkFetcher

- (id)initWithURL:(NSURL*)url {
    if ((self = [super init])) {
        _url = url;
    }
    return self;
}

- (void)startWithCompletionHandler:
        (EOCNetworkFetcherCompletionHandler)completion
{
    self.completionHandler = completion;
    // Запуск запроса
    // Запрос задает свойство downloadedData
    // После завершения запроса вызывается p_requestCompleted
}

- (void)p_requestCompleted {
    if (_completionHandler) {
        _completionHandler(_downloadedData);
    }
}

@end
```

Другой класс может создать такой объект загрузки данных и использовать его для загрузки данных по URL-адресу:

```
@implementation EOCClass {
    EOCNetworkFetcher *_networkFetcher;
    NSData *_fetchedData;
}

- (void)downloadData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/something.dat"];
    _networkFetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [_networkFetcher startWithCompletionHandler:^(NSData *data){
        NSLog(@"Request URL %@ finished", _networkFetcher.url);
        _fetchedData = data;
    }];
}

@end
```

В этом коде нет ничего необычного. Но, возможно, вы не заметили, что в нем возникает цикл удержания. Он обусловлен тем фактом, что блок обработчика завершения ссылается на переменную `self`, так как он задает переменную экземпляра `_fetchedData` (за информацией о захвате переменных обращайтесь к подходу 37). Это означает, что экземпляр `EOCClass`, который создает объект загрузки данных, удерживается блоком. Блок удерживается объектом загрузки данных, который в свою очередь удерживается тем же экземпляром `EOCClass`, потому что он хранится в сильной переменной экземпляра. Этот цикл удержания изображен на рис. 6.2.

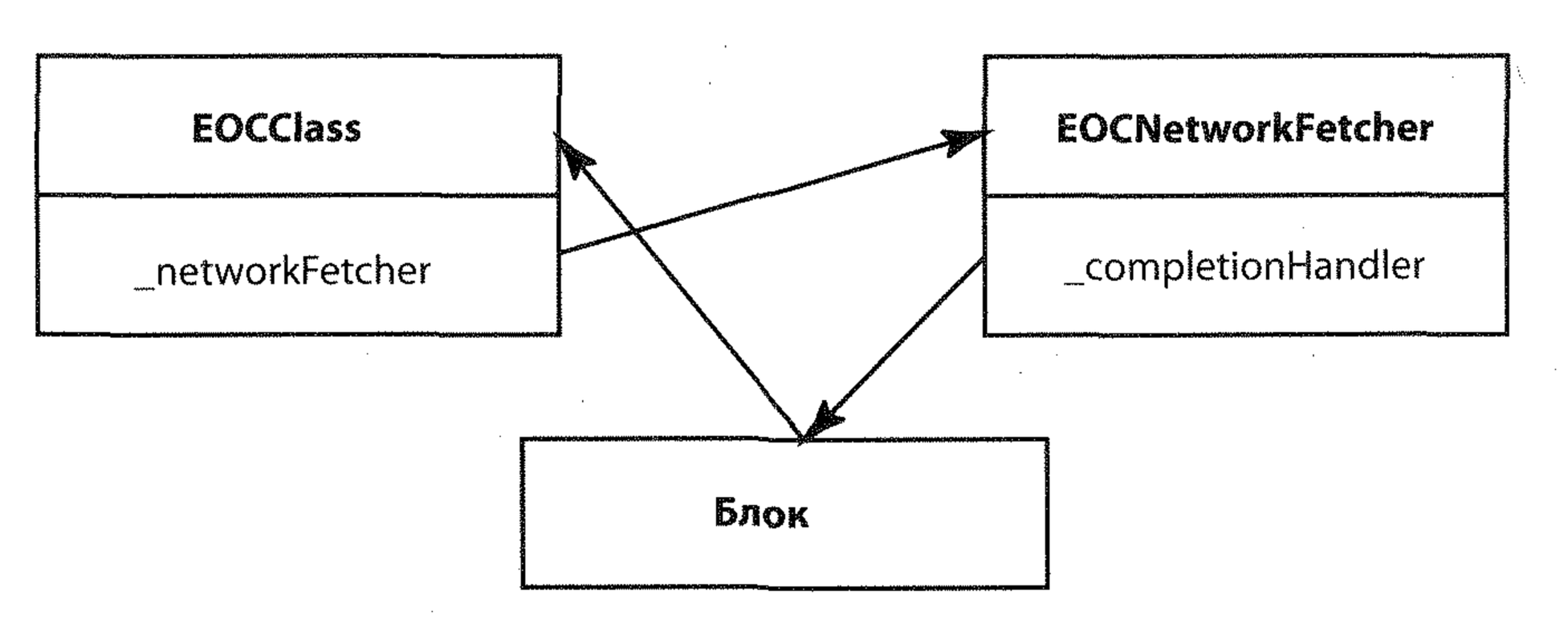


Рис. 6.2. Цикл удержания между объектом загрузки данных и классом, которому он принадлежит

Проблема с циклом удержания легко решается разрывом ссылки, хранящейся либо в переменной экземпляра `_networkFetcher`, либо в свойстве `completionHandler`. Это должно быть сделано после выполнения обработчика завершения, чтобы объект загрузки данных продолжал жить до того, как данные будут загружены. Например, блок обработчика завершения может быть приведен к следующему виду:

```
[_networkFetcher startWithCompletionHandler:^(NSData *data){
    NSLog(@"Request for URL %@ finished", _networkFetcher.url);
    _fetchedData = data;
    _networkFetcher = nil;
}
```

Проблема циклов удержания часто встречается в API, использующих блоки обратных вызовов завершения, поэтому вы должны понимать причины ее возникновения. Часто проблема может решаться очисткой одной из ссылок в подходящий момент; тем не менее не всегда можно гарантировать, что этот момент наступит. В приведенном примере цикл удержания разрывается только при выполнении обработчика завершения. Если обработчик завершения не будет выполняться, то цикл удержания не будет разорван и возникнет утечка памяти.

В решениях, использующих блоки обработчиков завершения, также встречается другая потенциальная разновидность циклов удержания. Эти циклы возникают тогда, когда блок обработчика завершения ссылается на объект, которому он в конечном итоге принадлежит. Расширим предыдущий пример: допустим, вместо хранения ссылки на объект загрузки данных во время его выполнения последний использует собственный механизм поддержания своего существования. Для этого объект загрузки данных может добавить себя в глобальную коллекцию (например, множество) при запуске и удалить себя по завершении. В этом случае код пользователя изменяется следующим образом:

```
- (void)downloadData {
    NSURL *url = [[NSURL alloc] initWithString:
                  @"http://www.example.com/something.dat"];
    EOCNetworkFetcher *networkFetcher =
        [[EOCNetworkFetcher alloc] initWithURL:url];
    [networkFetcher startWithCompletionHandler:^(NSData *data){
        NSLog(@"Request URL %@ finished", networkFetcher.url);
        _fetchedData = data;
    }];
}
```

Такой подход используется в большинстве сетевых библиотек, поскольку необходимость поддерживать существование объекта загрузки данных только раздражает пользователей. В качестве примера можно привести объект `TWRequest` из библиотеки Twitter. Однако в отношении кода `EOCNetworkFetcher` цикл удержания так и остается. Впрочем, он становится менее очевидным и возникает из-за того, что блок обработчика завершения ссылается на сам запрос. Таким образом, блок удерживает объект загрузки данных, который в свою очередь удерживает блок через свойство `completionHandler`. К счастью, у проблемы имеется простое решение. Вспомните, что обработчик завершения хранится в свойстве только для того, чтобы его можно было использовать позднее. После того как обработчик завершения был выполнен, удерживать блок ему уже незачем. Таким образом, проблема проще всего решается изменением следующего метода:

```
- (void)p_requestCompleted {
    if (_completionHandler) {
        _completionHandler(_downloadedData);
    }
    self.completionHandler = nil;
}
```

Цикл удержания разрывается по завершении запроса, и объект загрузки данных будет уничтожен при необходимости. Кстати, это убедительная причина для передачи обработчика завершения в методе запуска. Если бы вместо этого доступ к обработчику завершения предоставлялся через открытое свойство, мы бы не могли просто очистить его по завершении запроса, поскольку это привело бы к нарушению семантики инкапсуляции, которую мы предоставляем пользователю, объявляя обработчик завершения открытым. В этом случае возможен только один разумный способ разрыва цикла удержания: потребовать, чтобы сам пользователь сбросил свойство `completionHandler` в обработчике. Однако, скорее всего, пользователь этого не сделает и будет обвинять в утечке вас.

Обе ситуации встречаются достаточно часто. Подобные ошибки часто проникают в программы при использовании блоков; впрочем, при должной осторожности они так же легко искореняются. Главное — думать о том, какие объекты блок может захватить, а следовательно, и удержать. Если какой-либо из этих объектов может удержать блок (прямо или косвенно), решите, как разорвать цикл удержания в нужный момент.

УЗЕЛКИ НА ПАМЯТЬ

- Помните о потенциальной проблеме циклов удержания, которые возникают из-за захвата блоками объектов, прямо или косвенно удерживающих блоки.
- Проследите за тем, чтобы циклы удержания разрывались в подходящий момент. Никогда не возлагайте ответственность за разрыв циклов на пользователя вашего API.

41 ИСПОЛЬЗУЙТЕ ОЧЕРЕДИ ДИСПЕТЧЕРИЗАЦИИ ДЛЯ СИНХРОНИЗАЦИИ

Иногда в Objective-C встречается код, при выполнении которого из нескольких программных потоков возникают проблемы. В такой ситуации приложение обычно реализует некую разновидность синхронизации с использованием блокировок (locks). До появления GCD существовало два основных решения этой задачи; первое было основано на использовании встроенного блока синхронизации:

```
- (void)synchronizedMethod {
    @synchronized(self) {
        // Безопасное выполнение
    }
}
```

Эта конструкция автоматически создает блокировку для заданного объекта и ожидает получения этой блокировки для выполнения кода, содержащегося в блоке. В конце блока кода блокировка снимается. В приведенном примере синхронизация осуществляется по объекту `self`. Часто эта конструкция оказывается хорошим решением, так как она гарантирует, что каждый экземпляр объекта сможет независимо выполнять собственную копию `synchronizedMethod`. Однако злоупотребление `@synchronized(self)` приводит к неэффективности кода, так как все синхронизированные блоки будут выполняться последовательно. Чрезмерная синхронизация по `self` приводит к излишнему ожиданию кодом блокировки, удерживаемой другим, не относящимся к нему кодом.

Другое решение основано на непосредственном использовании объекта NSLock:

```
_lock = [[NSLock alloc] init];
- (void)synchronizedMethod {
    [_lock lock];
    // Безопасное выполнение
    [_lock unlock];
}
```

Также имеется объект NSRecursiveLock для рекурсивных блокировок, позволяющий одному потоку получить блокировку несколько раз без возникновения взаимной блокировки (deadlock).

Оба решения работают, но у них есть свои недостатки. Например, блоки синхронизации в особых ситуациях могут приводить к взаимным блокировкам, и они не всегда эффективны. Прямое использование блокировок может создавать проблемы с взаимными блокировками.

GCD реализует функциональность блокировок проще и эффективнее. Например, необходимость в синхронизации может возникнуть при доступе к свойствам (так называемая *атомарность свойств*). Нужного результата можно добиться при помощи атрибута свойств atomic (см. подход 6). Или, если методы доступа пишутся вручную, часто встречаются конструкции вида:

```
- (NSString*)someString {
    @synchronized(self) {
        return _someString;
    }
}

- (void)setSomeString:(NSString*)someString {
    @synchronized(self) {
        _someString = someString;
    }
}
```

Вспомните, что злоупотреблять конструкцией @synchronized(self) опасно, потому что такие блоки синхронизируются по отношению друг к другу. Если она применяется к нескольким свойствам, то каждое свойство будет синхронизироваться по отношению ко всем остальным — вряд ли это то, чего вы добивались. На самом деле доступ к каждому свойству должен синхронизироваться по отдельности.

Кстати сказать, хотя это решение в определенной степени улучшает потоковую безопасность, оно не гарантирует абсолютной потоковой безопасности объекта. Доступ к свойствам осуществляется в атомарном режиме, и вы гарантированно получите действительные результаты при использовании свойства, но при многократном вызове get-метода из одного потока результаты могут быть разными. Между обращениями свойство может быть изменено другими потоками.

Простая и эффективная альтернатива для блоков синхронизации и объектов блокировки заключается в использовании последовательной очереди синхронизации. Диспетчеризация операций чтения и записи в одну очередь обеспечивает синхронизацию доступа. Это выглядит так:

```
_syncQueue =
dispatch_queue_create("com.effectiveobjectivec.syncQueue",
NULL);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_sync(_syncQueue, ^{
        _someString = someString;
    });
}
```

В этом паттерне все обращения к свойству синхронизируются, потому что set- и get-методы выполняются в последовательной очереди GCD. Если не считать синтаксиса `__block` в get-методе, необходимого для того, чтобы блок мог обращаться к переменной (см. подход 37), такое решение выглядит намного элегантнее. Вся блокировка реализуется в GCD на очень низком уровне с множеством оптимизаций. Таким образом, разработчику не приходится беспокоиться об этом аспекте, и он направляет все усилия на написание методов доступа.

Однако можно сделать шаг вперед. Set-метод не обязан быть синхронным. Блок, в котором присваивается значение переменной

экземпляра, не должен ничего возвращать set-методу. Это означает, что set-метод можно привести к следующему виду:

```
- (void)setSomeString:(NSString*)someString {
    dispatch_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

В результате простого изменения — перехода с синхронной диспетчеризации на асинхронную — set-метод выполняется быстро с точки зрения вызывающей стороны, но операции чтения и записи выполняются последовательно по отношению друг к другу. Впрочем, если провести хронометраж, вы увидите, что это решение работает медленнее; асинхронная диспетчеризация требует копирования блока. Если время, затраченное на копирование, значительно по сравнению с временем выполнения блока, решение может работать медленнее — как это, скорее всего, будет в нашем простом примере. Но если блок, задействованный в диспетчеризации, выполняет более сложные задачи, вам стоит рассмотреть такое решение как потенциального кандидата.

Для ускорения этого процесса также можно воспользоваться тем фактом, что get-методы могут выполняться параллельно друг с другом, но не с set-методом. Именно здесь возможности GCD раскрываются в полной мере. Следующее решение не может быть легко реализовано с блоками синхронизации или блокировками. Представьте, что вместо последовательной очереди использовалась параллельная очередь:

```
_syncQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

В приведенном виде код не будет обеспечивать синхронизацию. Все операции чтения и записи выполняются в одной очереди, но, поскольку очередь является параллельной, чтение и запись могут выполняться одновременно. А ведь это именно то, что мы с самого начала пытались предотвратить! Однако в GCD существует простой механизм *барьеров* (barrier), способный решить эту проблему. Функция с барьерным блоком очереди выглядит так:

```
void dispatch_barrier_async(dispatch_queue_t queue,
                            dispatch_block_t block);
void dispatch_barrier_sync(dispatch_queue_t queue,
                           dispatch_block_t block);
```

Барьер выполняется монопольно по отношению ко всем остальным блокам этой очереди. Барьеры актуальны только для параллельных очередей, поскольку все блоки последовательной очереди всегда выполняются монопольно по отношению друг к другу. Если во время обработки очереди следующий блок является барьерным блоком, то очередь ожидает завершения всех текущих блоков, а затем выполняет барьерный блок. После того как выполнение барьерного блока завершится, обработка очереди продолжается в обычном режиме.

Барьеры можно использовать в set-методе примера со свойствами. Если set-метод использует барьерный блок, операции чтения свойства будут выполняться параллельно, а запись будет выполняться монопольно. На рис. 6.3 изображена очередь с несколькими операциями чтения и одной операцией записи.

Реализация такой очереди проста:

```
_syncQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_barrier_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

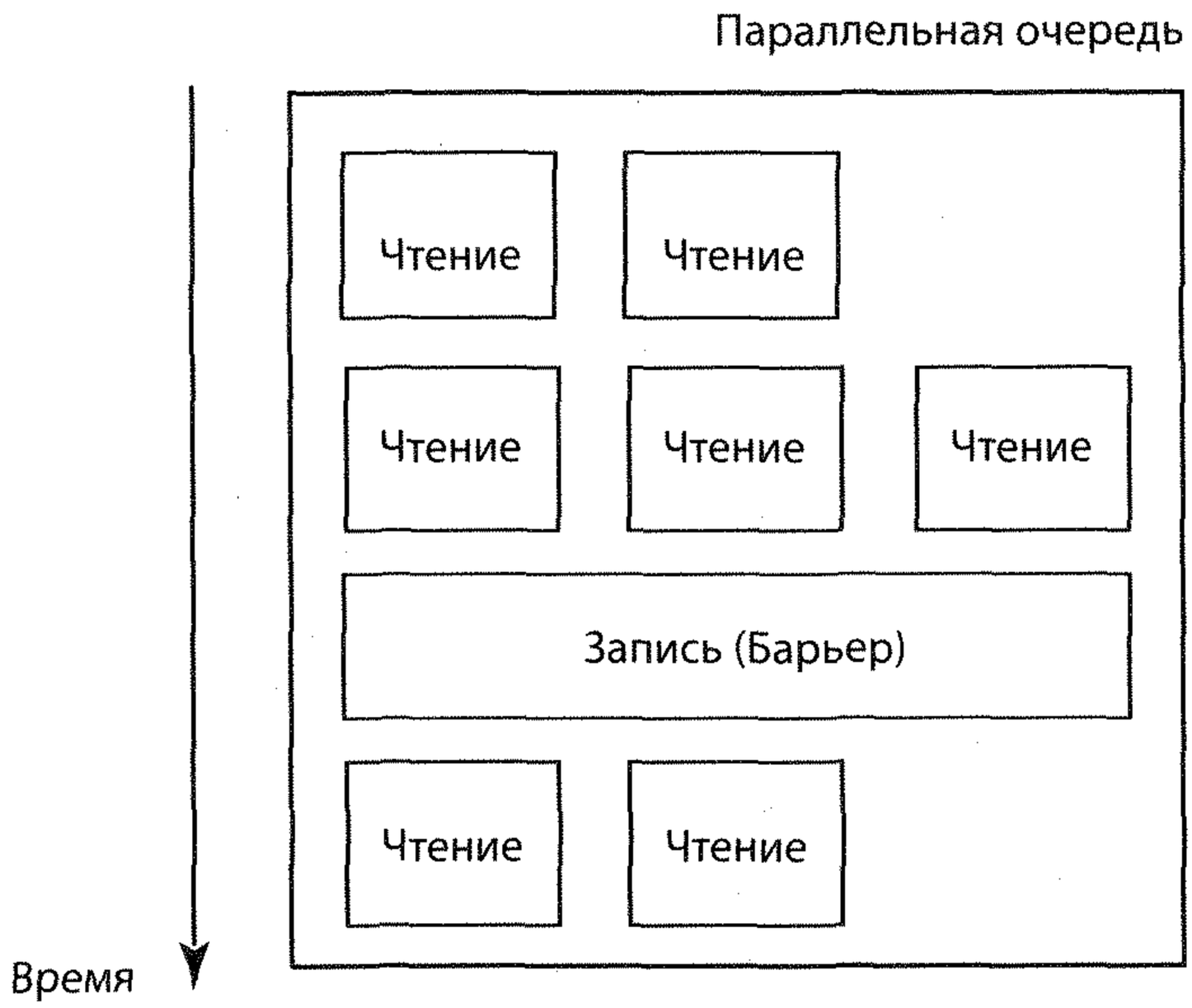


Рис. 6.3. Параллельная очередь с чтением в обычных блоках и записью в барьерном блоке. Операции чтения выполняются параллельно; операции записи выполняются монопольно

Если провести хронометраж этого решения, наверняка обнаружится, что оно работает быстрее решения с последовательной очередью. Кстати говоря, в set-методе также можно использовать синхронный барьер, что повысит его эффективность по тем же причинам. Разумно провести замеры по каждому решению и выбрать то, которое лучше подходит для вашей конкретной ситуации.

УЗЕЛКИ НА ПАМЯТЬ

- Очереди диспетчеризации предоставляют семантику синхронизации; это простая альтернатива для блоков `@synchronized` и объектов `NSLock`.
- Объединение синхронной и асинхронной диспетчеризации может реализовать такое же поведение синхронизации, как и обычные блокировки, но без приостановки вызывающего потока.
- Параллельные очереди и барьерные блоки используются для повышения эффективности синхронизации.

42 ИСПОЛЬЗУЙТЕ GCD ВМЕСТО МЕТОДА PERFORMSELECTOR И ЕГО СЕМЕЙСТВА

Благодаря чрезвычайно динамичной природе Objective-C (см. подход 11) некоторые методы, определенные в `NSObject`, позволяют вызвать любой метод по вашему усмотрению. В частности, они позволяют выполнить отложенный вызов методов или задать поток, в котором эти методы должны выполняться. Когда-то эта возможность была чрезвычайно полезна; однако сейчас, с появлением таких технологий, как Grand Central Dispatch и блоки, прежние методы уже не играют столь важную роль. И хотя код с использованием этих методов все еще встречается, я рекомендую держаться от них подальше.

Центральное место в этом семействе занимает метод `performSelector:`. Он получает один аргумент (селектор) и имеет следующую сигнатуру:

```
- (id)performSelector:(SEL)selector
```

Вызов эквивалентен прямому вызову селектора. Таким образом, следующие две строки кода эквивалентны:

```
[object performSelector:@selector(selectorName)];
[object selectorName];
```

На первый взгляд, такая запись кажется избыточной. И это впечатление было бы истинным, если бы это был единственный способ использования данного метода. Однако его подлинная мощь связана с тем, что селектор может определяться во время выполнения. Такая «динамическая привязка поверх динамической привязки» означает, что в программе можно использовать фрагменты следующего вида:

```
SEL selector;
if ( /* некоторое условие */ ) {
    selector = @selector(foo);
} else if ( /* some other condition */ ) {
    selector = @selector(bar);
} else {
    selector = @selector(baz);
}
[object performSelector:selector];
```

Такие конструкции отличаются исключительной гибкостью и часто используются для упрощения сложного кода. Другой возможный вариант — сохранение селектора, который должен быть выполнен

после наступления некоторого события. В любом случае компилятор не знает до момента выполнения, какой селектор должен быть выполнен. Однако при попытке откомпилировать этот код с ARC компилятор выдает следующее предупреждение:

```
warning: performSelector may cause a leak because its selector
is unknown [-Warc-performSelector-leaks]
```

Вероятно, для вас это оказалось неожиданным? А если нет, то вы, возможно, знаете, почему с этими методами необходима осторожность. Сообщение выглядит странно — почему в нем упоминается какая-то утечка (leak)? В конце концов, вы просто пытаетесь вызвать метод. Дело в том, что компилятор не знает, какой селектор будет активизирован, и поэтому не знает ни сигнатуры метода, ни возвращаемого типа (и есть ли возвращаемый тип вообще). Кроме того, компилятор не знает имя метода, поэтому он не может применить правила управления памятью ARC и определить, нужно ли освобождать возвращаемое значение. По этой причине ARC действует «наверняка» и не добавляет операцию освобождения. Но в результате может возникнуть утечка памяти, так как объект может возвращаться как удерживаемый.

Рассмотрим следующий код:

```
SEL selector;
if ( /* некоторое условие */ ) {
    selector = @selector(newObject);
} else if ( /* другое условие */ ) {
    selector = @selector(copy);
} else {
    selector = @selector(someProperty);
}
id ret = [object performSelector:selector];
```

Эта небольшая вариация на тему предыдущего примера демонстрирует суть проблемы. Для первых двух селекторов объект `ret` должен освобождаться, а для третьего — нет. Это справедливо не только в мире ARC, но и при ручном подсчете ссылок с жестким соблюдением рекомендаций по выбору имен методов. Без ARC (а следовательно, и без предупреждений компилятора) объект `ret` должен освобождаться при истинности любого из первых двух условий, но не в других случаях. Даже статический анализатор не способен обнаружить последующую утечку памяти. Это одна из причин, по которым к методам семейства `performSelector` следует относиться с осторожностью.

Другая причина, по которой эти методы неидеальны, заключается в том, что возвращаемым типом может быть только `void` или объектный тип. Возвращаемым типом метода `performSelector` является тип `id`, хотя выполняемый селектор также может вернуть `void`. И хотя изощренные преобразования типов позволяют использовать селекторы с возвращением других значений (например, целых или вещественных), такой код может оказаться ненадежным. Технически возможно вернуть любой тип, размер которого совпадает с размером указателя, так как тип `id` является указателем на произвольный объект Objective-C: в 32-разрядных архитектурах это тип, размер которого составляет 32 бита, а в 64-разрядных — любой тип, размер которого составляет 64 бита. Если возвращаемым типом является структура (`struct`) языка C, метод `performSelector` использоваться не может.

Пара разновидностей `performSelector`, позволяющих передавать аргументы с сообщением, определяется следующим образом:

```
- (id)performSelector:(SEL)selector
          withObject:(id)object
- (id)performSelector:(SEL)selector
          withObject:(id)objectA
          withObject:(id)objectB
```

Например, эти методы могут использоваться для задания свойства с именем `value`:

```
id object = /* объект со свойством с именем 'value' */;
id newValue = /* новое значение свойства */;
[object performSelector:@selector(setValue:)
              withObject:newValue];
```

Методы на первый взгляд кажутся полезными, но у них имеются серьезные недостатки. Передаваемые объекты должны быть именно объектами, поскольку типом всегда является `id`. Таким образом, если селектор получает целое или вещественное число, методы использоваться не могут. Кроме того, селектор может получать не более двух параметров с использованием метода `performSelector:withObject:withObject:`. Эквивалентных методов для выполнения селекторов, получающих более двух параметров, не существует.

Из других особенностей методов семейства `performSelector` стоит выделить возможность выполнения селектора с задержкой или в другом потоке. Наиболее распространенные из этих методов:

```
- (void)performSelector:(SEL)selector
              withObject:(id)argument
```

```
                afterDelay:(NSTimeInterval)delay
- (void)performSelector:(SEL)selector
                 onThread:(NSThread*)thread
               withObject:(id)argument
            waitUntilDone:(BOOL)wait
- (void)performSelectorOnMainThread:(SEL)selector
                          withObject:(id)argument
                       waitUntilDone:(BOOL)wait
```

Тем не менее вскоре становится ясно, что с этими методами связано слишком много ограничений. Например, не существует метода для выполнения селектора с двумя аргументами после задержки. Методы потоков не отличаются универсальностью по той же причине. Код, в котором используются эти методы, часто упаковывает аргументы в словарь, который распаковывается в вызываемом методе. Все это требует дополнительных затрат ресурсов и повышает вероятность ошибок.

Чтобы справиться с этими ограничениями, стоит воспользоваться одной из альтернатив. Главная альтернатива — решение с блоками (см. подход 37). Кроме того, используя блоки с GCD, можно реализовать все причины для выбора методов `performSelector`. Выполнение после задержки достигается при помощи `dispatch_after`, а выполнение в другом потоке — при помощи `dispatch_sync` и `dispatch_async`.

Например, если вам потребуется выполнить задачу после задержки, то вместо первого фрагмента рекомендуется использовать второй:

```
// Использование performSelector:withObject:afterDelay:
[self performSelector:@selector(doSomething)
           withObject:nil
           afterDelay:5.0];

// Использование dispatch_after
dispatch_time_t time = dispatch_time(DISPATCH_TIME_NOW,
                                     (int64_t)(5.0 * NSEC_PER_SEC));
dispatch_after(time, dispatch_get_main_queue(), ^(void){
    [self doSomething];
});
```

Выполнение задачи в главном потоке:

```
// Использование performSelectorOnMainThread:withObject:
//                                          waitUntilDone:
```

```
[self performSelectorOnMainThread:@selector(doSomething)
                          withObject:nil
                       waitUntilDone:NO];

// Использование dispatch_async
// (или dispatch_sync, если waitUntilDone равно YES)
dispatch_async(dispatch_get_main_queue(), ^{
    [self doSomething];
});
```

УЗЕЛКИ НА ПАМЯТЬ

- Семейство методов `performSelector` потенциально небезопасно по отношению к управлению памятью. Если компилятор ARC не может определить, какой селектор должен быть выполнен, он не может вставить нужные вызовы управления памятью.
- Возможности методов `performSelector` очень ограничены в отношении типа возвращаемого значения и количеству параметров, передаваемых методу.
- Методы, выполняющие селектор в другом потоке, лучше заменить вызовами GCD с использованием блоков.

43 НАУЧИТЕСЬ ВЫБИРАТЬ: GCD ИЛИ ОЧЕРЕДИ ОПЕРАЦИЙ

GCD — замечательная технология, но в некоторых ситуациях лучше использовать другие технологии, являющиеся частью стандартных системных библиотек. Разработчик должен знать, когда лучше применять тот или иной инструмент; неправильный выбор затруднит сопровождение кода.

В общем случае у механизмов синхронизации GCD (см. подход 41) нет конкурентов. То же можно сказать об однократном выполнении кода с `dispatch_once` (см. подход 45). Однако GCD не всегда является лучшим способом выполнения задач в фоновом режиме. Другая, хотя и связанная с GCD, технология `NSOperationQueue` позволяет организовать очереди операций (субклассы `NSOperation`) с возможностью их параллельного выполнения. Сходство с очередями диспетчеризации GCD неслучайно. Очереди операций появились

раньше GCD; несомненно, GCD базируется на принципах, которые стали популярными благодаря очередям операций. Кстати говоря, в iOS 4 и выше, а также в Mac OS X 10.6 и выше GCD используется во внутренней реализации очередей операций.

Первое различие, о котором стоит сказать: GCD представляет собой API, написанный на «чистом» языке C, тогда как очереди операций представляют собой объекты Objective-C. В GCD задача, которая ставится в очередь, представляет собой блок — облегченную структуру данных (см. подход 37). С другой стороны, операции представлены более тяжеловесными объектами Objective-C. Таким образом, GCD не всегда оказывается однозначным решением. Иногда лишние затраты минимальны, а преимущества от использования полноценных объектов значительно перевешивают недостатки.

При использовании метода `addOperationWithBlock:` класса `NSOperationQueue` и `NSBlockOperation` синтаксис очередей операций может быть очень похож на синтаксис GCD. Некоторые преимущества `NSOperation` и `NSOperationQueue`:

- Отмена операций.

С очередями операций задача решается просто. Метод `cancel` класса `NSOperation` устанавливает внутренние флаги операции, предотвращающие ее запуск, хотя он и не может отменить уже запущенную операцию. С другой стороны, очереди GCD не могут отменить блок, уже запланированный к выполнению. Эта архитектура работает по принципу автономного выполнения. Реализация отмены на уровне приложения возможна, но для этого вам придется написать большой объем кода, уже написанного в форме операций.

- Зависимости операций.

Операция может зависеть от других операций. Это позволяет создать иерархию операций, при которой некоторые операции могут выполняться только после успешного выполнения других операций. Например, операции загрузки и обработки файлов с сервера могут начаться только после предварительной загрузки файла манифеста. Таким образом, от операции загрузки манифеста зависят последующие операций загрузки данных. Если очередь операций настроена для параллельного выполнения, то последующие загрузки могут выполняться параллельно — но только после завершения операции, от которой они зависят.

- Использование KVO со свойствами операций.

Многие свойства операций могут использоваться с KVO (Key Value Observing). Например, по свойству `isCancelled` можно определить, была ли операция отменена, а по свойству `isFinished` — узнать о ее завершении. KVO удобно использовать в коде, который должен знать об изменении состояния некоторой задачи. Этот механизм обеспечивает существенно более точное управление, нежели GCD со своими задачами.

- Приоритеты операций.

Операции обладают приоритетом, который определяет их важность относительно других операций в очереди. Высокоприоритетные операции выполняются ранее низкоприоритетных. Алгоритм планирования операций непрозрачен, но он наверняка был тщательно продуман. В GCD не существует прямых средств для достижения этой цели. Приоритеты очередей существуют, но они устанавливают приоритет для всей очереди, а не для отдельных блоков. Вряд ли кому-нибудь захочется писать собственный планировщик, так что приоритеты являются полезной возможностью операций.

С операциями также связывается приоритет потока, который определяет, с каким приоритетом будет выполняться поток при запуске операции. В GCD эту возможность можно реализовать самостоятельно, но с операциями она сводится к простому заданию свойства.

- Повторное использование операций.

Для представления операций разработчик создает собственный субкласс (если только он не использует один из встроенных конкретных субклассов `NSOperation` — например, `NSBlockOperation`). Поскольку этот класс является обычным объектом Objective-C, в нем можно хранить любую нужную информацию. При выполнении вся эта информация и любые методы, определенные для класса, находятся в вашем распоряжении. Таким образом, возможности такого объекта существенно шире, чем у простого блока, поставленного в очередь диспетчеризации. Вы можете повторно использовать классы операций в своем коде в соответствии с известным принципом разработки DRY (Don't Repeat Yourself, то есть «Не повторяйся»).

Как видите, существует немало веских причин для использования очередей операций вместо очередей диспетчеризации. Очереди операций предоставляют готовые решения для многих типичных ситуаций, возникающих при выполнении задач. Вместо самостоя-

тельного написания сложных планировщиков, семантики отмены или приоритетов вы получаете готовые решения.

В частности, в API `NSNotificationCenter` вместо очередей диспетчеризации используются очереди операций. В этом API имеется метод, позволяющий зарегистрироваться на оповещения через блок (вместо вызова селектора). Прототип метода выглядит так:

```
- (id)addObserverForName:(NSString*)name
                  object:(id)object
                   queue:(NSOperationQueue*)queue
              usingBlock:(void(^)(NSNotification*))block
```

Вместо очереди операций этот метод мог бы получать очередь диспетчеризации, в которую ставится блок обработки оповещений. Безусловно, проектировщики сознательно решили использовать высокоуровневый API Objective-C. В данном случае два решения почти не отличаются по эффективности. Возможно, такое решение было принято из-за того, что использование очереди диспетчеризации создает лишнюю зависимость от GCD; напомню, что блоки не относятся к GCD, поэтому блок сам по себе не создает такой зависимости. А может быть, разработчики хотели, чтобы весь код работал на уровне Objective-C.

Часто приходится слышать, что разработчик всегда должен использовать API самого высокого уровня, опускаясь на более низкий уровень только тогда, когда это неизбежно. Пожалуй, я поддерживаю эту мантру, но с одной оговоркой. Возможность реализации некоторого решения в высокоуровневом коде Objective-C не всегда означает, что это решение оптимально. Самый верный способ выбрать лучший вариант — хронометражные тесты.

УЗЕЛКИ НА ПАМЯТЬ

- Очереди диспетчеризации — не единственная технология организации многопоточного выполнения и управления задачами.
- Очереди операций предоставляют высокоуровневый API Objective-C, который реализует большую часть возможностей GCD. Кроме того, очереди позволяют решать более сложные задачи, для которых пришлось бы писать дополнительную прослойку поверх GCD.

44 ИСПОЛЬЗУЙТЕ ГРУППЫ ДИСПЕТЧЕРИЗАЦИИ ДЛЯ ПЛАТФОРМЕННОГО МАСШТАБИРОВАНИЯ

В GCD предусмотрена возможность создания *групп диспетчеризации* (dispatch groups) для простой группировки задач. Например, задача может ожидать завершения некоторого набора задач или же получать оповещения через механизм обратного вызова об их завершении. Эта возможность очень полезна во многих ситуациях, первая и самая интересная из которых — параллельное выполнение нескольких задач с выдачей оповещения о том, что все задачи завершились (например, при сжатии набора файлов).

Группа диспетчеризации создается следующей функцией:

```
dispatch_group_t dispatch_group_create();
```

Группа представляет собой простую структуру данных без каких-либо отличительных признаков — в отличие от очереди диспетчеризации, у которой имеется идентификатор. Существуют два способа включения задач в группу диспетчеризации. В первом используется следующая функция:

```
void dispatch_group_async(dispatch_group_t group,
                          dispatch_queue_t queue,
                          dispatch_block_t block);
```

Эта разновидность обычной функции `dispatch_async` получает дополнительный параметр `group`, определяющий гурппу, с которой связывается выполняемый блок. Во втором способе включения задачи в группу диспетчеризации используется пара функций:

```
void dispatch_group_enter(dispatch_group_t group);
void dispatch_group_leave(dispatch_group_t group);
```

Первая функция увеличивает количество задач в группе, а вторая — уменьшает его. Таким образом, у каждого вызова `dispatch_group_enter` должен быть парный вызов `dispatch_group_leave`. Этот механизм напоминает подсчет ссылок (см. подход 29), при котором удержания и освобождения должны быть сбалансированы для предотвращения утечек. В случае групп диспетчеризации, если вход в группу не сбалансирован с выходом, группа никогда не завершится.

Следующая функция ожидает завершения группы диспетчеризации:

```
long dispatch_group_wait(dispatch_group_t group,
                         dispatch_time_t timeout);
```

Функции передается группа, по которой ведется ожидание, и продолжительность тайм-аута. Второй параметр определяет, на какое время функция должна блокироваться в ожидании завершения группы. Если группа завершится до истечения тайм-аута, возвращается нуль; в противном случае возвращается ненулевое значение. Вместо конкретного значения тайм-аута может использоваться константа `DISPATCH_TIME_FOREVER`; она означает, что продолжительность ожидания не ограничена, а тайм-аут никогда не наступает.

Следующая функция предоставляет альтернативный способ блокирования текущего потока в ожидании завершения группы диспетчеризации:

```
void dispatch_group_notify(dispatch_group_t group,
                           dispatch_queue_t queue,
                           dispatch_block_t block);
```

Эта функция слегка отличается от функции `wait`; она позволяет задать блок, который будет выполнен в текущей очереди по завершении группы. Это может быть полезно в том случае, если текущий поток не должен блокироваться, но вам все равно необходимо знать о завершении задач. Например, и в Mac OS X, и в iOS главный поток приложения никогда не должен блокироваться, поскольку в нем выполняется вся прорисовка пользовательского интерфейса и обработка событий.

Использование этой возможности GCD можно продемонстрировать на примере выполнения задачи для массива объектов и последующего ожидания завершения всех задач. Вот как это делается:

```
dispatch_queue_t queue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_t dispatchGroup = dispatch_group_create();
for (id object in collection) {
    dispatch_group_async(dispatchGroup,
                         queue,
                         ^{ [object performTask]; });
}

dispatch_group_wait(dispatchGroup, DISPATCH_TIME_FOREVER);
// Продолжить выполнение после завершения задач
```

Если текущий поток не должен блокироваться, используйте функцию `notify` вместо ожидания:

```
dispatch_queue_t notifyQueue = dispatch_get_main_queue();
dispatch_group_notify(dispatchGroup,
                      notifyQueue,
                      ^{
                    // Продолжить выполнение после завершения
                    // задач
                    });
```

Очередь, в которую должен ставиться обратный вызов оповещения, полностью зависит от обстоятельств. В приведенном примере я использовал главную очередь — весьма типичная ситуация. Но с таким же успехом можно было использовать любую пользовательскую последовательную очередь или одну из глобальных параллельных очередей.

В этом примере для всех задач используется одна очередь, но это не обязательно. Допустим, некоторые задачи должны выполняться с более высоким приоритетом, но при этом все задачи должны быть объединены в одну группу диспетчеризации и вы должны получить оповещение об их общем завершении:

```
dispatch_queue_t lowPriorityQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0);
dispatch_queue_t highPriorityQueue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);
dispatch_group_t dispatchGroup = dispatch_group_create();

for (id object in lowPriorityObjects) {
    dispatch_group_async(dispatchGroup,
                         lowPriorityQueue,
                         ^{ [object performTask]; });
}

for (id object in highPriorityObjects) {
    dispatch_group_async(dispatchGroup,
                         highPriorityQueue,
                         ^{ [object performTask]; });
}

dispatch_queue_t notifyQueue = dispatch_get_main_queue();
dispatch_group_notify(dispatchGroup,
                      notifyQueue,
                      ^{
```

```
    // Продолжить выполнение после завершения
    // задач
    });
```

Помимо отправки задач в параллельные очереди, как в предыдущих примерах, можно использовать группы диспетчеризации для отслеживания нескольких задач в разных последовательных очередях. Впрочем, если все задачи находятся в одной последовательной очереди, группа особой пользы не принесет. Поскольку все задачи все равно выполняются последовательно, можно просто поставить в очередь еще один блок после задач, что эквивалентно блоку обратного вызова `notify` группы диспетчеризации:

```
dispatch_queue_t queue =
dispatch_queue_create("com.effectiveobjectivec.queue", NULL);

for (id object in collection) {
    dispatch_async(queue,
                   ^{ [object performTask]; });
}

dispatch_async(queue,
               ^{
                   // Продолжить выполнение после завершения
                   // задач
               });
```

Этот код показывает, что группы диспетчеризации нужны не всегда. В некоторых случаях желаемого эффекта можно добиться с одной очередью и стандартной асинхронной диспетчеризацией.

Почему я упомянул о выполнении задач в зависимости от доступных системных ресурсов? Если вернуться к примеру диспетчеризации в параллельной очереди, это станет ясно. GCD автоматически создает новые потоки или повторно использует уже существующие для обслуживания блоков в очереди. Для параллельных очередей это может означать многопоточное выполнение, то есть несколько блоков будут выполняться одновременно. GCD выбирает количество параллельных потоков, обрабатывающих заданную параллельную очередь, в зависимости от ряда факторов — прежде всего системных ресурсов. В системе с многоядерным процессором очереди, выполняющей большой объем работы, с большой вероятностью будут выделены множественные потоки. Группы диспетчеризации предоставляют простую возможность параллельного выполнения с оповещением о завершении группы задач. Вследствие природы

параллельных очередей GCD задачи будут выполняться параллельно и с учетом доступных системных ресурсов. Разработчику остается программировать бизнес-логику, не отвлекаясь на написание сложного планировщика для организации параллельного выполнения задач.

Пример с перебором коллекции и выполнением задачи для каждого ее элемента также можно реализовать при помощи другой функции GCD:

```
void dispatch_apply(size_t iterations,
                    dispatch_queue_t queue,
                    void(^block)(size_t));
```

Эта функция выполняет заданное количество итераций блока с передачей значения, последовательно увеличиваемого от нуля до количества итераций минус 1. Пример использования:

```
dispatch_queue_t queue =
  dispatch_queue_create("com.effectiveobjectivec.queue", NULL);
dispatch_apply(10, queue, ^(size_t i){
    // Выполнение задачи
});
```

По сути происходящее эквивалентно простому циклу `for` от 0 до 9:

```
for (int i = 0; i < 10; i++) {
    // Perform task
}
```

Важно заметить, что очередь может быть параллельной. В этом случае блоки будут выполняться параллельно в отношении системных ресурсов, как и в примере с группами диспетчеризации. Если бы коллекция в этом примере была массивом, то пример можно было бы переписать с использованием `dispatch_apply` в следующем виде:

```
dispatch_queue_t queue =
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_apply(array.count, queue, ^(size_t i){
    id object = array[i];
    [object performTask];
});
```

Этот пример в очередной раз показывает, что без групп диспетчеризации часто можно обойтись. Однако вызов `dispatch_apply` блокирует выполнение вплоть до завершения всех итераций. По этой

причине при попытке выполнения блоков в текущей очереди (или последовательной очереди, находящейся над текущей очередью в иерархии) произойдет взаимная блокировка. Если вы хотите, чтобы задачи выполнялись в фоновом режиме, придется использовать группы диспетчеризации.

УЗЕЛКИ НА ПАМЯТЬ

- Группы диспетчеризации используются для группировки задач. Также возможно получать оповещения о завершении выполнения группы.
- Группы диспетчеризации могут использоваться для одновременного выполнения нескольких задач через параллельную очередь диспетчеризации. В этом случае GCD планирует выполнение задач в зависимости от доступности системных ресурсов. Самостоятельная реализация этой функциональности потребует большого объема кода.

ИСПОЛЬЗУЙТЕ DISPATCH_ONCE ДЛЯ ПОТОКОВО-БЕЗОПАСНОГО ОДНОРАЗОВОГО ВЫПОЛНЕНИЯ КОДА

Паттерн «Одиночка» («Синглет») достаточно хорошо известен в мире Objective-C. Обычно он реализуется методом класса, который называется, например, `sharedInstance`; метод возвращает синглетный экземпляр класса, вместо того чтобы создавать новый экземпляр при каждом вызове. Стандартная реализация метода синглетного экземпляра для класса `EOCClass` выглядит примерно так:

```
@implementation EOCClass

+ (id)sharedInstance {
    static EOCClass *sharedInstance = nil;
    @synchronized(self) {
        if (!sharedInstance) {
            sharedInstance = [[self alloc] init];
        }
    }
    return sharedInstance;
}

@end
```

По поводу паттерна «Одиночка» идут ожесточенные споры, особенно в Objective-C. Главная обсуждаемая тема — потоковая безопасность этого паттерна. Приведенный код создает синглетный экземпляр, заключенный в блок синхронизации для обеспечения потоковой безопасности. Паттерн применяется достаточно широко, и такой код встречается сплошь и рядом.

Однако в GCD появилась одна функция, которая значительно упрощает реализацию синглетных экземпляров. Эта функция выглядит так:

```
void dispatch_once(dispatch_once_t *token,
                   dispatch_block_t block);
```

В параметрах передается специальный тип `dispatch_once_t`, который я далее буду называть «маркером» (token), и блок. Функция гарантирует, что для заданного маркера блок выполняется один и только один раз. Блок всегда выполняется по первому запросу и, что самое важное, — с полной потоковой безопасностью. Обратите внимание: передаваемый маркер должен быть одинаковым для каждого блока, который должен быть выполнен ровно один раз. Обычно это означает объявление переменной маркера со статической или глобальной областью действия.

Синглетный метод создания общего экземпляра, переписанный с использованием этой функции, выглядит так:

```
+ (id)sharedInstance {
    static EOCClass *sharedInstance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}
```

Использование `dispatch_once` упрощает код и полностью гарантирует потоковую безопасность, так что вам даже не нужно думать о блокировках или синхронизации. Все это делается автоматически во внутренней реализации GCD. Маркер объявлен статическим, потому что он должен полностью совпадать при каждом вызове. Определение переменной в статической области действия означает, что вместо создания новой переменной при каждом вызове `sharedInstance` компилятор многократно использует одну переменную.

Кроме того, решение с `dispatch_once` работает эффективнее. Вместо тяжеловесного механизма синхронизации, который получает блокировку при каждом выполнении кода, в нем используется атомарный доступ к маркеру диспетчеризации. Простые хронометражные тесты на моем 64-разрядном компьютере с Mac OS X 10.8.2 показали, что решение с `dispatch_once` почти вдвое превосходит по скорости решение с `@synchronized`.

УЗЕЛКИ НА ПАМЯТЬ

- Потоково-безопасное однократное выполнение кода — весьма распространенная задача. GCD предоставляет для этой цели простой и удобный инструмент — функцию `dispatch_once`.
- Маркер следует объявлять со статической или глобальной областью действия, чтобы для каждого блока, который должен быть выполнен однократно, передавался один и тот же маркер.

46 ОСТЕРЕГАЙТЕСЬ ФУНКЦИИ DISPATCH_GET_CURRENT_QUEUE

При использовании GCD — и особенно при диспетчеризации с различными очередями — часто бывает нужно определить, в какой очереди в настоящее время выполняется запрос. Например, операции пользовательского интерфейса в Mac OS X и iOS должны выполняться в главном потоке, что эквивалентно главной очереди в GCD. Иногда кажется необходимым определить, выполняется ли текущий код в главном потоке. В документации встречается следующая функция:

```
dispatch_queue_t dispatch_get_current_queue()
```

В документации указано, что функция возвращает текущую очередь. Описание полностью соответствует действительности, но при использовании этой функции необходима осторожность. Более того, в iOS версии 6.0 она была официально признана устаревшей. Впрочем, в Mac OS X версии 10.8 функция устаревшей пока не считается, и все же в Mac OS X ее лучше избегать.

Типичный антипаттерн с использованием этого метода — сравнение текущей очереди с конкретной очередью для разрешения взаимной

блокировки, которая может возникнуть при синхронной диспетчеризации. Рассмотрим следующие методы доступа, использующие очередь для синхронизации доступа к переменной экземпляра (см. подход 41):

```
- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

Проблема, которая может возникнуть с этим паттерном, — взаимная блокировка в get-методе, если get-метод вызывается из очереди, используемой для синхронизации (`_syncQueue` в данном примере), потому что `dispatch_sync` не возвращает управление до полного выполнения блока. Но если целевая очередь блока является текущей, то у блока не будет возможности выполниться, потому что `dispatch_sync` будет продолжать блокироваться, ожидая, пока очередь станет доступной для выполнения целевого блока. Это пример метода, не допускающего повторное вхождение, или *реентерабельность* (not reentrant).

Прочитав документацию по `dispatch_get_current_queue`, можно подумать, что метод легко сделать реентерабельным — достаточно проверить, является ли текущая очередь очередью синхронизации, и если является — просто выполнить блок вместо диспетчеризации:

```
- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_block_t accessorBlock = ^{
        localSomeString = _someString;
    };
    if (dispatch_get_current_queue() == _syncQueue) {
        accessorBlock();
    } else {
        dispatch_sync(_syncQueue, accessorBlock);
```

```
    }

    return localSomeString;
}
```

Вероятно, в простой ситуации это решение сработает. Тем не менее оно слишком опасно и может привести к взаимной блокировке. Чтобы понять, почему это происходит, рассмотрим следующую ситуацию с двумя последовательными очередями диспетчеризации:

```
dispatch_queue_t queueA =
  dispatch_queue_create("com.effectiveobjectivec.queueA",
NULL);
dispatch_queue_t queueB =
  dispatch_queue_create("com.effectiveobjectivec.queueB",
NULL);

dispatch_sync(queueA, ^{
    dispatch_sync(queueB, ^{
        dispatch_sync(queueA, ^{
            // Взаимная блокировка
        });
    });
});
```

Во внутреннем вызове `dispatch_sync` для `_queueA` всегда будет возникать взаимная блокировка, потому что он будет дожидаться завершения внешнего вызова `dispatch_sync`, который не завершится без внутреннего вызова до завершения `dispatch_sync`. Теперь попробуем добавить ту же проверку с использованием `dispatch_get_current_queue`:

```
dispatch_sync(queueA, ^{
    dispatch_sync(queueB, ^{
        dispatch_block_t block = ^{ /* … */ };
        if (dispatch_get_current_queue() == queueA) {
            block();
        } else {
            dispatch_sync(queueA, block);
        }
    });
});
```

Но и в этом случае произойдет взаимная блокировка, поскольку `dispatch_get_current_queue` возвращает текущую очередь, которой в предыдущем примере будет `_queueB`. Таким образом,

синхронная диспетчеризация по `_queueA` все равно будет выполняться, а это, как и прежде, приведет к взаимной блокировке.

В этой ситуации не нужно стараться сделать метод доступа реентерабельным. Вместо этого следует позаботиться о том, чтобы очередь, используемая для синхронизации, никогда не пыталась обратиться к свойству; никогда не вызывайте `someString`. Очередь должна использоваться только для синхронизации свойства. Очереди диспетчеризации относительно легковесны, поэтому можно создать несколько очередей, чтобы очередь синхронизации использовалась исключительно для синхронизации одного конкретного свойства.

Приведенный пример выглядит немного искусственным, но из-за другого аспекта, связанного с очередями, эта проблема может возникнуть там, где вы ее вряд ли ожидали увидеть. Очереди образуют иерархию, то есть блоки, поставленные в одну очередь, выполняются в родительской очереди. Последней очередью в иерархии всегда является одна из глобальных параллельных очередей. Простая иерархия изображена на рис. 6.4.

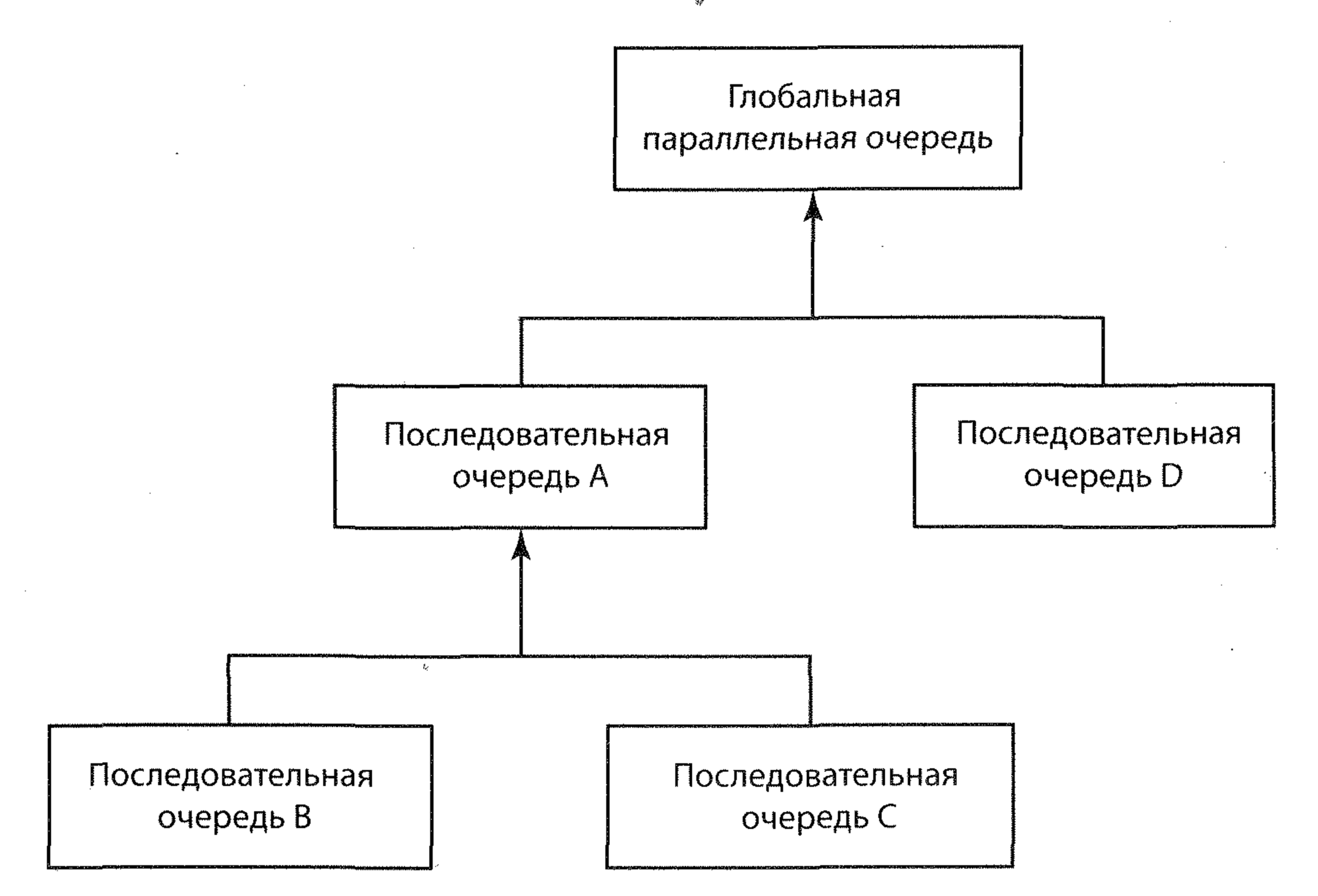


Рис. 6.4. Иерархия очередей диспетчеризации

Блоки, поставленные в очередь B или C, в дальнейшем планируются для выполнения в последовательной очереди A. Таким образом, блоки в очередях A, B и C будут выполняться монопольно по отно-

шению друг к другу. Однако блоки, поставленные в очередь D, будут выполняться параллельно с блоками, поставленными в очередь A (а следовательно, B и C), потому что целевая очередь для A и D является параллельной. Параллельные очереди выполняют блоки в нескольких потоках в зависимости от доступных системных ресурсов (например, количества ядер у процессора).

Из-за иерархии очередей проверка текущей очереди на равенство с очередью синхронной диспетчеризации может не сработать. Например, блок, поставленный в очередь C, вернет в качестве текущей очередь C, поэтому код может решить, что он имеет право безопасно осуществить синхронную постановку в очередь A. Однако это приведет к взаимной блокировке, как и прежде. Проблема возникает в том случае, если API позволяет указать очередь для планирования блоков обратного вызова, а во внутренней реализации используется последовательная очередь, для которой в качестве целевой указана очередь обратного вызова. Код, использующий этот API, ошибочно предполагает, что текущая очередь, возвращаемая вызовом `dispatch_get_current_queue` в блоках обратного вызова, всегда равна указанной. В действительности вместо нее будет возвращаться внутренняя очередь синхронизации.

Данная проблема лучше всего решается использованием функций GCD, позволяющих связать с очередью произвольные данные в виде пары «ключ-значение». Если при выборке данных с ключом не связано никакое значение, система проходит вверх по иерархии, пока не найдет данные или не достигнет корня. Чтобы вы лучше поняли, как использовать эту особенность, рассмотрим пример:

```
dispatch_queue_t queueA =
  dispatch_queue_create("com.effectiveobjectivec.queueA",
NULL);
dispatch_queue_t queueB =
  dispatch_queue_create("com.effectiveobjectivec.queueB",
NULL);
dispatch_set_target_queue(queueB, queueA);

static int kQueueSpecific;
CFStringRef queueSpecificValue = CFSTR("queueA");
dispatch_queue_set_specific(queueA,
                            &kQueueSpecific,
                            (void*)queueSpecificValue,
                            (dispatch_function_t)CFRelease);

dispatch_sync(queueB, ^{
```

```
    dispatch_block_t block = ^{ NSLog(@"No deadlock!"); };

    CFStringRef retrievedValue =
                    dispatch_get_specific(&kQueueSpecific);
    if (retrievedValue) {
        block();
    } else {
        dispatch_sync(queueA, block);
    }
});
```

В этом примере создаются две очереди. Целевой очередью для очереди B назначена очередь A, у которой целевой очередью остается используемая по умолчанию глобальная параллельная очередь. Для очереди A задается отличительный признак с использованием следующей функции:

```
void dispatch_queue_set_specific(dispatch_queue_t queue,
                                 const void *key,
                                 void *context,
                                 dispatch_function_t
destructor);
```

Функции передается очередь, для которой задается значение, за которым следуют ключ и значение. И ключ, и значение представляют собой непрозрачные `void`-указатели. Важно помнить, что ключи сравниваются по значению указателя, а не по содержимому. Таким образом, поведение данных, связанных с очередью, отличается от поведения объектов `NSDictionary`, сравнивающих ключи по равенству объектов. Данные, связанные с очередью, больше напоминают ассоциированные ссылки (см. подход 10). Значения (`context` в прототипе функции) также представляют собой непрозрачные `void`-указатели, поэтому в них может храниться все что угодно. Впрочем, вам придется выполнять с объектом все необходимые операции управления памятью, а это сильно усложняет использование объектов Objective-C в качестве значений в ARC. В приведенном примере в качестве значения используется строка CoreFoundation, потому что ARC не пытается управлять памятью объектов CoreFoundation. Такие объекты хорошо подходят для данных, связываемых с очередями, потому что при необходимости они легко преобразуются в соответствующие классы Objective-C из Foundation (см. подход 49).

В последнем аргументе передается функция-деструктор, выполняемая при удалении объекта для заданного ключа, — либо из-за

уничтожения очереди, либо из-за задания нового значения для этого ключа. Тип `dispatch_function_t` определяется следующим образом:

```
typedef void (*dispatch_function_t)(void*)
```

Таким образом, деструктором должна быть функция, которая получает единственный аргумент с указателем и возвращает `void`. В нашем примере передается функция `CFRelease`, хотя с таким же успехом это может быть пользовательская функция, которая в свою очередь вызывает `CFRelease` для выполнения всей необходимой зачистки.

Таким образом, связывание данных с очередью является простым и удобным механизмом для решения одной из основных проблем `dispatch_get_current_queue`. Также функция `dispatch_get_current_queue` часто применяется для отладки, когда вы можете спокойно использовать устаревший метод — код все равно не будет откомпилирован в окончательной версии. Если ваши конкретные потребности для обращения к текущей очереди не обеспечиваются другими функциями, отправьте в Apple запрос на расширение функциональности.

УЗЕЛКИ НА ПАМЯТЬ

- Функция `dispatch_get_current_queue` в целом работает не так, как можно было бы ожидать. Она официально признана устаревшей, и использовать ее следует только в целях отладки.
- Очереди диспетчеризации объединены в иерархию; следовательно, текущую очередь нельзя полностью описать одним объектом.
- Основная причина для использования `dispatch_get_current_queue` — предотвращение взаимных блокировок из-за нереентерабельности кода — может быть реализована посредством связывания данных с очередями.

Глава 7
Системные фреймворки

Теоретически на Objective-C можно программировать без использования системных фреймворков, но такой подход встречается крайне редко. Даже стандартный корневой класс `NSObject` является частью фреймворка Foundation, а не самого языка. Если вы не хотите использовать Foundation, вам придется написать собственный корневой класс, а также собственные коллекции, циклы событий и другие полезные классы. Более того, Objective-C без системных фреймворков не может использоваться для разработки приложений Mac OS X и iOS. Системные фреймворки предоставляют разработчику много полезных возможностей, но к своему сегодняшнему состоянию они пришли через многие годы разработки. Некоторые их компоненты могут показаться морально устаревшими и неудобными, однако порой встречаются и настоящие шедевры.

47 Познакомьтесь поближе с системными фреймворками

При написании приложений на Objective-C почти всегда используются системные фреймворки, которые предоставляют многие классы, необходимые для разработки приложений (например, коллекции). Если разработчик недостаточно хорошо понимает, какая именно функциональность предоставляется системными фреймворками, он может заново написать уже готовый код. При обновлении операционной системы пользователи приложения получают новейшие версии системных фреймворков. Таким образом, при использовании классов из этих фреймворков вы автоматически получаете доступ ко всем улучшениям без обновления самого приложения.

Фреймворк (framework) представляет собой совокупность кода, оформленного в динамическую библиотеку, и заголовочных файлов с описанием его интерфейса. Иногда в сторонних фреймворках для iOS используются статические библиотеки, так как распространение динамических библиотек с приложениями iOS запрещено. Такие фреймворки не соответствуют определению «настоящего» фреймворка, хотя часто обозначаются этим термином. Все системные фреймворки используют динамические библиотеки и для iOS.

Каждому разработчику графических приложений для Mac OS X или iOS наверняка знаком фреймворк Cocoa (или Cocoa Touch для iOS). Cocoa не является фреймворком в общепринятом смысле; это набор других фреймворков, часто используемых при создании приложений.

Важнейший фреймворк, с которым вам придется иметь дело, называется Foundation; в нем находятся такие классы, как `NSObject`, `NSArray` и `NSDictionary`. Классы фреймворка Foundation снабжаются префиксом `NS`, который появился еще в те времена, когда Objective-C использовался для работы над операционной системой NeXTSTEP. Фреймворк Foundation лежит в основе всех приложений Objective-C; без него большая часть материала этой книги потеряла-бы актуальность.

Foundation предоставляет не только основные возможности (например, коллекции), но и более сложные функции — скажем, обработку строк. Например, класс `NSLinguisticTagger` обеспечивает разбор строк с выделением всех существительных, глаголов и т. д. Короче говоря, возможности Foundation выходят далеко за рамки базовой функциональности.

Помимо Foundation, хорошо известен другой фреймворк: CoreFoundation. Хотя формально CoreFoundation представляет собой API языка C, воспроизводящий большую часть функциональности Foundation, этот фреймворк также играет важную роль при написании приложений Objective-C. Фреймворки CoreFoundation и Foundation объединяют не только похожие названия. Механизм упрощенного преобразования (toll-free bridging) позволяет легко переходить от структур данных C, используемых в CoreFoundation, к объектам Objective-C из Foundation, и наоборот. Например, строка в Foundation представлена классом `NSString`, который преобразуется в эквивалент из CoreFoundation: `CFString`. Работа упрощенного преобразования основана на довольно сложном коде, благодаря которому объекты CoreFoundation выглядят для исполнительной среды так, как если бы они были объектами Objective-C. К сожалению, реализация упрощенного преобразования очень сложна, так что воспроизвести ее в вашем коде очень трудно. Этот механизм

стоит использовать, но копировать его можно только в том случае, если вы действительно хорошо понимаете, что делаете.

Наряду с Foundation и CoreFoundation существует много других системных библиотек. Ниже перечислены лишь некоторые из них:

- CFNetwork — сетевые средства уровня C для взаимодействия с сетями через простую и удобную абстракцию на базе сокетов BSD. Foundation инкапсулирует компоненты этого фреймворка для предоставления сетевого интерфейса Objective-C (например, `NSURLConnection` для загрузки данных по URL-адресу).
- CoreAudio — API уровня C для взаимодействия со звуковым оборудованием устройства. С этим фреймворком относительно трудно работать из-за сложной природы обработки звука. К счастью, абстракции Objective-C несколько упрощают работу со звуком.
- AVFoundation — объекты Objective-C для воспроизведения и записи аудио- и видеоданных (например, классы представлений пользовательского интерфейса для отображения видео).
- CoreData — интерфейсы Objective-C для сохранения объектов в базе данных. CoreData обеспечивает загрузку и сохранение данных и может использоваться для межплатформенной передачи между Mac OS X и iOS.
- CoreText — интерфейс C для высокопроизводительного форматирования и вывода текста.

Также существуют и другие фреймворки, но даже этот короткий список демонстрирует важную особенность программирования Objective-C: разработчику часто приходится опускаться до API уровня C. API, написанные на C, отличаются более высоким быстродействием, так как они работают в обход исполнительной среды Objective-C. Конечно, при работе с такими API необходимо уделять больше внимания управлению памятью, поскольку механизм ARC (см. подход 30) доступен только для объектов Objective-C. Для работы с такими фреймворками необходимы хотя бы минимальные знания C.

Скорее всего, вы будете писать приложения Mac OS X или iOS, использующие фреймворки пользовательского интерфейса. Базовые фреймворки пользовательского интерфейса, AppKit и UIKit, предоставляют классы Objective-C, построенные на базе Foundation и CoreFoundation. Они предоставляют элементы пользовательского интерфейса и механизмы связи для объединения элементов в при-

ложение. В основе этих основных фреймворков пользовательского интерфейса лежат фреймворки CoreAnimation и CoreGraphics.

Фреймворк CoreAnimation написан на Objective-C; он предоставляет инструменты, которые используются фреймворками пользовательского интерфейса для вывода графики и выполнения анимаций. Вам никогда не придется опускаться до этого уровня, но о его существовании следует знать. Формально CoreAnimation является не самостоятельным фреймворком, а частью фреймворка QuartzCore. Тем не менее CoreAnimation следует рассматривать как полноправного участника семейства фреймворков.

Фреймворк CoreGraphics написан на C; он предоставляет структуры данных и функции, необходимые для вывода двумерной графики. Например, в нем определяются структуры данных `CGPoint`, `CGSize` и `CGRect`, используемые классом `UIView` из фреймворка UIKit для обозначения относительной позиции представлений относительно друг друга.

На базе фреймворков пользовательского интерфейса построено много других фреймворков — например, MapKit для работы с картами в iOS или Social для работы с социальными сетями в Mac OS X и iOS. Обычно разработчик имеет дело с ними и с базовым фреймворком пользовательского интерфейса для той платформы, на которой он работает.

Многие фреймворки включаются в стандартную установку Mac OS X и iOS. Таким образом, если вам понадобилось написать новый вспомогательный класс — попробуйте сначала поискать его в системных фреймворках. Часто выясняется, что он уже был написан за вас.

УЗЕЛКИ НА ПАМЯТЬ

- Разработчику доступны многочисленные системные фреймворки. Самые важные из них — Foundation и CoreFoundation — предоставляют базовую функциональность, на которой строится большая часть функциональности приложения.
- Фреймворки существуют для многих стандартных задач — обработки аудио- и видеоинформации, работы с Сетью и управления данными.
- Помните, что фреймворки, написанные на языке C (а не на Objective-C), тоже играют важную роль в вашей работе, поэтому хороший разработчик Objective-C должен понимать основные концепции C.

48 ИСПОЛЬЗУЙТЕ ПЕРЕБОР С ВЫПОЛНЕНИЕМ БЛОКОВ ВМЕСТО ЦИКЛОВ FOR

Перебор элементов коллекции является очень распространенной задачей в программировании. В современном языке Objective-C существует много способов ее решения, от стандартных циклов C до `NSEnumerator` в Objective-C 1.0 и быстрого перебора в Objective-C 2.0. С добавлением в язык блоков (см. подход 37) появились новые методы, о которых разработчики иногда забывают. Эти методы перебирают элементы коллекции и выполняют для каждого элемента переданный блок. Как вы вскоре убедитесь, обычно с ними намного проще работать.

При таком переборе чаще всего используются коллекции, представленные в этом подходе: `NSArray`, `NSDictionary` и `NSSet`. Также можно реализовать описанные приемы перебора в пользовательских коллекциях, хотя подробное описание того, как это делается, выходит за рамки подхода.

ЦИКЛЫ FOR

Первый метод перебора коллекции — старый добрый цикл `for` — напоминает о том, что Objective-C уходит корнями к языку C. Это простейший способ — а следовательно, его возможности ограничены. Обычно это делается примерно так:

```
NSArray *anArray = /* … */;
for (int i = 0; i < anArray.count; i++) {
    id object = anArray[i];
    // Какие-то действия с 'object'
}
```

Такое решение приемлемо, но для словарей и множеств оно усложняется:

```
// Dictionary
NSDictionary *aDictionary = /* … */;
NSArray *keys = [aDictionary allKeys];
for (int i = 0; i < keys.count; i++) {
    id key = keys[i];
    id value = aDictionary[key];
    // Какие-то действия с 'key' и 'value'
}
```

```
// Множество
NSSet *aSet = /* … */;
NSArray *objects = [aSet allObjects];
for (int i = 0; i < objects.count; i++) {
    id object = objects[i];
    // Какие-то действия с 'object'
}
```

Словари и множества по определению не упорядочены, поэтому обратиться напрямую к значению с известным целочисленным индексом не удастся. Следовательно, приходится запрашивать все ключи словаря или все объекты множества; в обоих случаях для обращения к значениям выполняется перебор элементов упорядоченного массива. Создание дополнительного массива требует лишней работы, а созданный дополнительный объект удерживает объекты в коллекции. Конечно, эти объекты будут освобождены при освобождении массива, но это приводит к лишним вызовам методов. Все остальные способы перебора обходятся без создания промежуточного массива.

Перебор в обратном направлении реализуется циклом `for` от количества объектов без единицы с уменьшением счетчика при каждой итерации и завершением цикла при достижении счетчиком нуля. Этот вариант намного проще.

ПЕРЕБОР С ИСПОЛЬЗОВАНИЕМ NSENUMERATOR В OBJECTIVE-C 1.0

Объект `NSEnumerator` — абстрактный базовый класс, который определяет только два метода, реализуемых конкретными субклассами:

```
- (NSArray*)allObjects
- (id)nextObject
```

Ключевой метод `nextObject` возвращает следующий объект при переборе. При каждом вызове этого метода происходит обновление внутренних структур данных, чтобы его следующий вызов вернул следующий объект. После того как перебор пройдет все объекты, возвращается `nil` — признак конца перебора.

Во всех встроенных классах коллекций из фреймворка Foundation перебор реализован именно этим способом. Например, перебор массива выполняется так:

```
NSArray *anArray = /* … */;
NSEnumerator *enumerator = [anArray objectEnumerator];
```

```
id object;
while ((object = [enumerator nextObject]) != nil) {
    // Какие-то действия с 'object'
}
```

Происходящее напоминает стандартный цикл `for`, но требует дополнительной работы. Единственное реальное преимущество заключается в том, что перебор любой коллекции осуществляется с похожим синтаксисом. Для примера рассмотрим эквиваленты для словаря и множества:

```
// Словарь
NSDictionary *aDictionary = /* … */;
NSEnumerator *enumerator = [aDictionary keyEnumerator];
id key;
while ((key = [enumerator nextObject]) != nil) {
    id value = aDictionary[key];
    // Какие-то действия с 'key' и 'value'
}

// Множество
NSSet *aSet = /* … */;
NSEnumerator *enumerator = [aSet objectEnumerator];
id object;
while ((object = [enumerator nextObject]) != nil) {
    // Какие-то действия с 'object'
}
```

Перебор в словаре немного отличается. Так как в словаре хранятся ключи и значения, значение приходится извлекать из словаря по заданному ключу. У `NSEnumerator` также есть другое преимущество: часто доступно несколько способов перебора. Например, для массива существует возможность перебора элементов в обратном направлении:

```
NSArray *anArray = /* … */;
NSEnumerator *enumerator = [anArray reverseObjectEnumerator];
id object;
while ((object = [enumerator nextObject]) != nil) {
    // Какие-то действия с 'object'
}
```

Такой код читается намного проще, чем эквивалентный синтаксис обратного перебора с циклом `for`.

БЫСТРЫЙ ПЕРЕБОР

Быстрый перебор появился в Objective-C 2.0. Механизм быстрого перебора напоминает перебор с `NSEnumerator`, но отличается большей лаконичностью синтаксиса: в цикл `for` добавляется ключевое слово `in`. В результате синтаксис перебора коллекции становится чрезвычайно компактным:

```
NSArray *anArray = /* … */;
for (id object in anArray) {
    // Какие-то действия с 'object'
}
```

Чтобы обозначить поддержку быстрого перебора, объект реализует протокол с именем `NSFastEnumeration`. Протокол определяет простой метод:

```
- (NSUInteger)countByEnumeratingWithState:
                              (NSFastEnumerationState*)state
                                    objects:(id*)stackbuffer
                                      count:(NSUInteger)length
```

Полное рассмотрение механизма быстрого перебора выходит за рамки данного подхода. Впрочем, в Интернете хватает учебных руководств, которые хорошо разъясняют эту тему. Важно заметить, что этот метод позволяет классу возвращать несколько объектов одновременно, что повышает эффективность цикла перебора.

Перебор в словарях и множествах выполняется так же просто:

```
// Словарь
NSDictionary *aDictionary = /* … */;
for (id key in aDictionary) {
    id value = aDictionary[key];
    // Какие-то действия с 'key' и 'value'
}

// Множество
NSSet *aSet = /* … */;
for (id object in aSet) {
    // Какие-то действия с 'object'
}
```

Обратный перебор также можно реализовать другим способом: обратите внимание на то, что объекты `NSEnumerator` также реали-

зуют NSFastEnumeration. Таким образом, обратное перемещение по элементам массива может выглядеть так:

```
NSArray *anArray = /* … */;
for (id object in [anArray reverseObjectEnumerator]) {
    // Какие-то действия с 'object'
}
```

Этот метод перебора является лучшим по синтаксису и эффективности, но для перебора в словаре все равно потребуется дополнительный шаг, если вам нужны как ключ, так и значение. Кроме того, в отличие от традиционного цикла for, для получения индекса текущей итерации придется дополнительно потрудиться. Знать текущий индекс часто бывает полезно, поскольку он используется многими алгоритмами.

ПЕРЕБОР С ВЫПОЛНЕНИЕМ БЛОКА

Блочные методы — последняя разновидность перебора, поддерживаемая в современном Objective-C. Простейший метод перебора массива, определенный для NSArray, выглядит так:

```
- (void)enumerateObjectsUsingBlock:
          (void(^)(id object, NSUInteger idx, BOOL *stop))
block
```

Другие методы этого семейства могут получать параметры для управления перебором; они будут рассмотрены ниже.

Для массива и множества блоку, выполняемому при каждой итерации, передается текущий объект, индекс итерации и указатель на логический признак. Первые два параметра не требуют пояснений, а последний предоставляет механизм прерывания перебора.

Например, перебор массива с использованием этого метода может выглядеть так:

```
NSArray *anArray = /* … */;
[anArray enumerateObjectsUsingBlock:
    ^(id object, NSUInteger idx, BOOL *stop){
        // Какие-то действия с 'object'
        if (shouldStop) {
            *stop = YES;
        }
    }];
```

Этот синтаксис чуть объемнее синтаксиса быстрого перебора, но он достаточно элегантен, и в нем доступен как индекс итерации, так и объект. Метод также позволяет легко прервать перебор при помощи переменной `stop`, хотя эту функциональность можно также чисто реализовать в других способах перебора.

Представленный способ перебора подходит не только для массивов. Аналогичный метод перебора с выполнением блока существует и в `NSSet`, а в слегка измененном виде — и в `NSDictionary`:

```
- (void)enumerateKeysAndObjectsUsingBlock:
                    (void(^)(id key, id object, BOOL *stop))block
```

Таким образом, перебор в словарях и множествах выполняется ничуть не сложнее:

```
// Словарь
NSDictionary *aDictionary = /* … */;
[aDictionary enumerateKeysAndObjectsUsingBlock:
    ^(id key, id object, BOOL *stop){
        // Какие-то действия с 'key' и 'object'
        if (shouldStop) {
            *stop = YES;
        }
    }];

// Множество
NSSet *aSet = /* … */;
[aSet enumerateObjectsUsingBlock:
    ^(id object, BOOL *stop){
        // Какие-то действия с 'object'
        if (shouldStop) {
            *stop = YES;
        }
    }];
```

Главное преимущество заключается в том, что непосредственно в блоке доступно больше информации. Для массивов вы получаете индекс текущей итерации; то же относится и к упорядоченным множествам (`NSOrderedSet`). Для словаря вы получаете как ключ, так и значение без какой-либо дополнительной работы; тем самым достигается экономия вычислительных ресурсов, необходимых для выборки значения для заданного ключа. Словарь может предоставить оба значения одновременно, что с большой вероятностью будет происходить более эффективно, поскольку ключи и значения хранятся рядом во внутренних структурах данных словаря.

К преимуществам также относится возможность изменения сигнатуры блока для снижения необходимости в преобразовании; фактически преобразование заносится в сигнатуру блока. Возьмем код быстрого перебора словаря. Если вы знаете, что объекты в словаре представляют собой строки, можно поступить так:

```
for (NSString *key in aDictionary) {
    NSString *object = (NSString*)aDictionary[key];
    // Какие-то действия с 'key' и 'object'
}
```

При использовании перебора на базе блоков можно выполнить преобразование в сигнатуре блока:

```
NSDictionary *aDictionary = /* … */;
[aDictionary enumerateKeysAndObjectsUsingBlock:
    ^(NSString *key, NSString *obj, BOOL *stop){
        // Какие-то действия с 'key' и 'obj'
    }];
```

Такое решение работает, потому что тип `id` весьма специфичен и может переопределяться подобным образом. Если бы в исходной сигнатуре блока ключ и объект были определены с типом `NSObject*`, то этот фокус не прошел бы. Этот прием полезнее, чем может показаться на первый взгляд. Передача точного типа объекта позволяет компилятору лишний раз помочь вам, выдав сообщение об ошибке, если вызываемый для объекта метод не существует. Если вы можете гарантировать тип объектов, хранящихся в коллекции, всегда используйте такое обозначение типа.

Возможность обратного перебора при этом не теряется. Массивы, словари и множества реализуют разновидность приведенного метода с возможностью передачи управляющей маски `options`:

```
- (void)enumerateObjectsWithOptions:
                (NSEnumerationOptions)options
                             usingBlock:
                (void(^)(id obj, NSUInteger idx, BOOL *stop))block
- (void)enumerateKeysAndObjectsWithOptions:
                (NSEnumerationOptions)options
                                usingBlock:
                (void(^)(id key, id obj, BOOL *stop))block
```

Тип `NSEnumerationOptions` представляет собой перечисление (`enum`), значения которого объединяются поразрядной операцией ИЛИ в маску, определяющую поведение перебора. Например, можно

потребовать, чтобы итерации происходили параллельно (то есть блоки выполнялись одновременно, если это возможно при текущих системных ресурсах); для этого в маску включается флаг `NSEnumerationConcurrent`. Во внутренней реализации параллельное выполнение осуществляется средствами GCD, вероятнее всего, с использованием групп диспетчеризации (см. подход 44). Впрочем, реализация нас сейчас не интересует. Обратный перебор включается флагом `NSEnumerationReverse`. Учтите, что флаг доступен только в тех случаях, когда он имеет смысл (например, для массивов или упорядоченных множеств).

В целом перебор с выполнением блоков обладает всеми возможностями других способов — и не только. По компактности он слегка уступает быстрому перебору, но дополнительные преимущества: доступность индекса, доступность ключа и значения при переборе в словаре, возможность параллельного выполнения итераций — компенсируют некоторое возрастание объема кода.

УЗЕЛКИ НА ПАМЯТЬ

- Перебор содержимого коллекций может быть реализован четырьмя способами. Простейший вариант — цикл `for`, за ним следует перебор с `NSEnumerator` и быстрый перебор. Самый современный и передовой способ — методы перебора с выполнением блоков.
- Перебор с выполнением блоков может выполняться параллельно без какого-либо дополнительного кода, за счет использования GCD. С другими способами перебора реализовать такую возможность не удастся.
- Измените сигнатуру блока и включите в нее точные типы объектов, если они вам известны.

ИСПОЛЬЗУЙТЕ УПРОЩЕННОЕ ПРЕОБРАЗОВАНИЕ ДЛЯ КОЛЛЕКЦИЙ С НЕСТАНДАРТНОЙ СЕМАНТИКОЙ УПРАВЛЕНИЯ ПАМЯТЬЮ

Выбор классов коллекций из системных библиотек Objective-C достаточно широк: массивы, словари и множества. Фреймворк Foundation определяет классы Objective-C для этих и других типов

коллекций. Аналогичным образом фреймворк CoreFoundation определяет API уровня C для манипуляций со структурами данных, представляющими эти и другие типы коллекций. Например, в Foundation массив представляется классом Objective-C `NSArray`, а эквивалентной ему структурой данных CoreFoundation является `CFArray`. Казалось бы, эти два способа создания массива далеки друг от друга, но мощная возможность *упрощенного преобразования* (toll-free bridging) позволяет с минимальными усилиями переходить от класса Objective-C, определенного в Foundation, к структурам данных C из CoreFoundation — и наоборот, конечно. Я говорю о «структурах данных» в API уровня C, потому что они не совпадают с классами или объектами в Objective-C. Например, на `CFArray` ссылается `CFArrayRef` — указатель на структуру `struct __CFArray`. Для работы со структурой используются различные функции — такие, как функция `CFArrayGetCount` для получения размера массива. Этим они отличаются от своих аналогов из Objective-C, где вы создаете объект `NSArray` и вызываете методы для объекта (например, метод `count` для получения размера массива).

Простой пример упрощенного преобразования выглядит так:

```
NSArray *anNSArray = @[@1, @2, @3, @4, @5];
CFArrayRef aCFArray = (__bridge CFArrayRef)anNSArray;
NSLog(@"Size of array = %li", CFArrayGetCount(aCFArray));
// Вывод: Size of array = 5
```

Конструкция `__bridge` в преобразовании сообщает ARC (см. подход 30), что нужно делать с объектом Objective-C, составляющим часть преобразования. Сама по себе она означает, что ARC является владельцем объекта Objective-C. И наоборот, конструкция `__bridge_retained` означает, что ARC передает принадлежность объекта. С таким использованием, как в предыдущем примере, мы бы отвечали за добавление `CFRelease(aCFArray)` после завершения работы с массивом. С `__bridge_transfer` ситуация противоположная. Например, если `CFArrayRef` преобразуется в `NSArray*` и вы хотите, чтобы объект принадлежал ARC, используйте этот тип преобразования.

Резонно спросить, для чего использовать эту возможность в «чистом» приложении Objective-C? Дело в том, что классы Objective-C из Foundation способны делать то, что не могут делать структуры данных C из CoreFoundation, и наоборот. Например, при использовании словарей Foundation встречается одна распространенная проблема: ключи копируются, а значения удерживаются. Эту

семантику невозможно изменить без использования упрощенного преобразования.

Тип словаря из фреймворка CoreFoundation называется `CFDictionary`. Изменяемый аналог называется `CFMutableDictionary`. При создании `CFMutableDictionary` можно задать нестандартную семантику управления памятью, применяемую к ключам и значениям; для этого используется следующий метод:

```
CFMutableDictionaryRef CFDictionaryCreateMutable(
    CFAllocatorRef allocator,
    CFIndex capacity,
    const CFDictionaryKeyCallBacks *keyCallBacks,
    const CFDictionaryValueCallBacks *valueCallBacks
)
```

Первый параметр определяет используемый распределитель памяти. Если вы провели большую часть своей жизни в мире Objective-C, этот раздел CoreFoundation может показаться инородным. Распределитель отвечает за выделение и освобождение памяти, необходимой для хранения структур данных объектов CoreFoundation. Обычно в этом параметре передается `NULL`, чтобы использовать распределитель по умолчанию.

Второй параметр просто определяет исходный размер словаря. Он не ограничивает максимальный размер, а всего лишь рекомендует распределителю, сколько памяти следует выделить с самого начала. Если вы знаете, что создаете словарь для десяти объектов, передайте десять.

Наибольший интерес представляют последние параметры. Они определяют обратные вызовы, которые будут выполняться при различных операциях с ключами и значениями, хранящимися в словаре. В обоих параметрах передаются указатели на структуры:

```
struct CFDictionaryKeyCallBacks {
    CFIndex version;
    CFDictionaryRetainCallBack retain;
    CFDictionaryReleaseCallBack release;
    CFDictionaryCopyDescriptionCallBack copyDescription;
    CFDictionaryEqualCallBack equal;
    CFDictionaryHashCallBack hash;
};

struct CFDictionaryValueCallBacks {
    CFIndex version;
```

```
    CFDictionaryRetainCallBack retain;
    CFDictionaryReleaseCallBack release;
    CFDictionaryCopyDescriptionCallBack copyDescription;
    CFDictionaryEqualCallBack equal;
};
```

В параметре `version` в настоящее время следует передавать 0. Это значение зарезервировано на случай, если Apple в будущем решит изменить структуру. Этот параметр может использоваться для проверки совместимости старой и новой версий. Остальные структуры представляют собой указатели на функции, которые должны выполняться при выполнении различных операций. Например, функция `retain` вызывается для каждого ключа и значения, добавленного в словарь. Тип этого параметра определяется следующим образом:

```
typedef const void* (*CFDictionaryRetainCallBack) (
    CFAllocatorRef allocator,
    const void *value
);
```

Итак, это указатель на функцию, получающую `CFAllocatorRef` и `const void*`. Передаваемый функции параметр `value` содержит ключ или значение, добавляемое в словарь. Функция возвращает `void*` — значение, которое в итоге будет добавлено в словарь. Вы можете написать собственный метод обратного вызова :

```
const void* CustomCallback(CFAllocatorRef allocator,
                           const void *value)
{
    return value;
}
```

Метод просто возвращает значение без изменений. Таким образом, если словарь создается с передачей этой функции в качестве обратного вызова `retain`, ключи и значения удерживаться не будут. В сочетании с упрощенным преобразованием это позволяет создать объект `NSDictionary`, поведение которого отличается от поведения объекта, созданного в Objective-C.

Полный пример того, как можно эффективно использовать эту возможность:

```
#import <Foundation/Foundation.h>
#import <CoreFoundation/CoreFoundation.h>

const void* EOCRetainCallback(CFAllocatorRef allocator,
```

```
                                const void *value)
{
    return CFRetain(value);
}

void EOCReleaseCallback(CFAllocatorRef allocator,
                        const void *value)
{
    CFRelease(value);
}

CFDictionaryKeyCallBacks keyCallbacks = {
    0,
    EOCRetainCallback,
    EOCReleaseCallback,
    NULL,
    CFEqual,
    CFHash
};

CFDictionaryValueCallBacks valueCallbacks = {
    0,
    EOCRetainCallback,
    EOCReleaseCallback,
    NULL,
    CFEqual
};

CFMutableDictionaryRef aCFDictionary =
    CFDictionaryCreateMutable(NULL,
                              0,
                              &keyCallbacks,
                              &valueCallbacks);

NSMutableDictionary *anNSDictionary =
    (__bridge_transfer NSMutableDictionary*)aCFDictionary;
```

Вместо обратного вызова `copyDescription` передается `NULL`, потому что значения по умолчанию достаточно. Вместо обратных вызовов `equal` и `hash` передаются `CFEqual` и `CFHash` соответственно, потому что они используют тот же метод, что и реализация по умолчанию `NSMutableDictionary`. В результате `CFEqual` вызывает метод `isEqual:` класса `NSObject`, а `CFHash` вызывает метод `hash` класса `NSObject` — очередной пример мощи упрощенного преобразования.

Обратным вызовам `retain` и `release` ключей и значений задаются функции `EOCRetainCallback` и `EOCReleaseCallback` соответственно. Для чего это делается? Вспомните, что `NSMutableDictionary` копирует свои ключи и удерживает значения по умолчанию. Что, если объекты, которые вы хотите использовать в качестве ключей, не могут копироваться? В этом случае их нельзя использовать с обычным объектом `NSMutableDictionary`, потому что это приведет к ошибке времени выполнения:

```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '-[EOCClass
copyWithZone:]: unrecognized selector sent to instance
0x7fd069c080b0'
```

Из сообщения об ошибке следует, что класс не поддерживает протокол `NSCopying`, потому что в нем не реализован метод `copyWithZone:`. Опускаясь на уровень CoreFoundation и создавая словарь на этом уровне, вы изменяете семантику управления памятью и создаете словарь, который удерживает, а не копирует ключи.

Аналогичный подход может использоваться для создания массива или множества, не удерживающего содержащиеся в нем объекты. Например, это может быть полезно, если при создании массива, удерживающего некоторые объекты, создается цикл удержания. Однако в данной ситуации лучше поискать другое решение. Создание массива, не удерживающего свои объекты, небезопасно. Если один из объектов будет уничтожен, но при этом останется в массиве, при обращении к этому объекту с большой вероятностью произойдет сбой приложения.

УЗЕЛКИ НА ПАМЯТЬ

- Механизм упрощенного преобразования позволяет осуществлять преобразования между объектами Objective-C из Foundation и структурами данных C из Core Foundation.
- Создание коллекций на уровне CoreFoundation позволяет задать различные методы обратного вызова, используемые при работе с содержимым коллекции. Благодаря применению упрощенного преобразования вы в конечном итоге получаете коллекцию Objective-C с нестандартной семантикой управления памятью.

50 ИСПОЛЬЗУЙТЕ NSCACHE ВМЕСТО NSDICTIONARY ДЛЯ КЭША

При разработке приложений Mac OS X и iOS, загружающих графику из Интернета, часто приходится решать проблему организации их кэширования. Первое неплохое решение — использование словаря для хранения в памяти загруженных изображений, чтобы их не приходилось загружать заново при последующих запросах. Наивный разработчик просто возьмет класс `NSDictionary` (или, скорее, его изменяемую версию), потому что этот класс часто используется при разработке. Однако в фреймворке Foundation также присутствует еще более подходящий класс `NSCache`, предназначенный специально для таких задач.

Преимущество `NSCache` перед `NSDictionary` заключается в том, что при заполнении системной памяти из кэша автоматически вытесняется часть содержимого. При использовании словаря разработчику обычно приходится писать код вытеснения самостоятельно, перехватывая системные оповещения о недостатке памяти. `NSCache` предоставляет эту функциональность автоматически; этот класс, являющийся частью фреймворка Foundation, может подключаться к системе на более глубоком уровне, чем ваша реализация. `NSCache` также начинает с удаления тех объектов, которые наиболее давно не использовались. Написать код самостоятельной реализации этой функциональности для словарей — задача не из простых.

Кроме того, `NSCache` не копирует ключи, а удерживает их. Этим поведением можно управлять и из `NSDictionary`, но для этого потребуется более сложный код (см. подход 49). Обычно кэш по возможности должен обходиться без копирования ключей, потому что в качестве ключа часто используется объект, не поддерживающий копирование. Так как `NSCache` не выполняет копирование по умолчанию, с этим классом проще работать в подобных ситуациях. Кроме того, класс `NSCache`, в отличие от `NSDictionary`, обладает потоковой безопасностью, то есть к `NSCache` можно одновременно обращаться из нескольких потоков без необходимости введения собственных блокировок. Обычно потоковая безопасность полезна для кэшей, потому что вы можете выполнить чтение в одном потоке, и если некоторый ключ не существует — загрузить данные этого ключа. Обратные вызовы загрузки могут находиться в фоновом потоке, в котором в конечном итоге будет выполняться добавление данных в кэш.

Разработчик может управлять тем, когда происходит вытеснение объектов из кэша. Наряду с системными ресурсами существуют две метрики, находящихся под контролем пользователя: ограничение количества объектов в кэше и общей «стоимости» объектов. Каждому объекту при добавлении в кэш может быть назначена условная стоимость. Когда общее количество объектов превышает порог количества или суммарная стоимость превышает порог стоимости, из кэша могут быть вытеснены некоторые объекты (как это происходит при нехватке системной памяти). Однако следует заметить, что вытеснение объектов *возможно*, но не обязательно. Порядок вытеснения объектов зависит от реализации. В частности, это означает, что манипуляции с метрикой стоимости для активизации вытеснения в определенном порядке обычно не приводят к желаемому эффекту.

Метрика стоимости должна использоваться только в том случае, если вычисление стоимости при добавлении объекта в кэш выполняется с минимальными затратами. Если вычисления обходятся дорого, может оказаться, что кэш не оптимален, так как каждое кэширование объекта сопряжено с дополнительными затратами. В конце концов, кэш предназначен для того, чтобы повысить скорость отклика приложения. Например, обращение к диску для определения размера файла или к базе данных было бы нежелательно. Хороший пример метрики стоимости встречается при добавлении в кэш объектов `NSData`; в этом случае в качестве стоимости можно использовать размер данных. Они уже известны объекту `NSData`, так что вычисление сводится к простому чтению свойства.

Пример использования кэша:

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

// Класс загрузки данных из сети
typedef void(^EOCNetworkFetcherCompletionHandler)(NSData
*data);
@interface EOCNetworkFetcher : NSObject
- (id)initWithURL:(NSURL*)url;
- (void)startWithCompletionHandler:
                (EOCNetworkFetcherCompletionHandler)handler;
@end

// Класс, использующий загрузку данных и кэширующий результаты
@interface EOCClass : NSObject
@end
```

```
@implementation EOCClass {
    NSCache *_cache;
}

- (id)init {
    if ((self = [super init])) {
        _cache = [NSCache new];

        // Кэширование до 100 URL-адресов
        _cache.countLimit = 100;

        /**
         * Размер данных в байтах используется
         * как метрика стоимости,
         * следующая команда устанавливает порог стоимости
         * 5 Мбайт.
         */
        _cache.totalCostLimit = 5 * 1024 * 1024;
    }
    return self;
}

- (void)downloadDataForURL:(NSURL*)url {
    NSData *cachedData = [_cache objectForKey:url];
    if (cachedData) {
        // Попадания
        [self useData:cachedData];
    } else {
        // Промахи
        EOCNetworkFetcher *fetcher =
            [[EOCNetworkFetcher alloc] initWithURL:url];
        [fetcher startWithCompletionHandler:^(NSData *data){
            [_cache setObject:data forKey:url
                              cost:data.length];
            [self useData:data];
        }];
    }
}

@end
```

В этом примере URL-адрес, с которого загружаются данные, используется как ключ кэширования. При кэш-промахах данные загружаются и добавляются в кэш. Стоимость вычисляется по длине данных. В момент создания кэша максимальное количество

кэшируемых объектов задается равным 100, а общая стоимость — 5 Мбайт, потому что стоимость определяется размером в байтах.

Также в сочетании с NSCache часто используется класс NSPurgeableData — субкласс NSMutableData, реализующий протокол NSDiscardableContent. Этот протокол определяет интерфейс для объектов, память которых может очищаться в случае необходимости. Таким образом, память данных NSPurgeableData автоматически очищается при нехватке системных ресурсов. Информацию о том, была ли память очищена, можно получить при помощи метода isContentDiscarded, входящего в протокол NSDiscardableContent.

Если программе потребуется обратиться к объекту NSPurgeableData, она вызывает метод beginContentAccess, чтобы указать, что память объекта с этого момента не должна очищаться. Завершив работу с объектом, программа вызывает метод endContentAccess, который сообщает, что память объекта может быть очищена при необходимости. Вызовы могут быть вложенными, так что их можно рассматривать как аналогию для увеличения и уменьшения счетчика ссылок. Только когда счетчик ссылок уменьшается до нуля, память объекта может быть очищена.

Если объекты NSPurgeableData добавляются в NSCache, при очистке объект автоматически удаляется из кэша. Автоматическое удаление можно включать и выключать при помощи свойства evictsObjectsWithDiscardedContent кэша.

Таким образом, для использования NSPurgeableData в предыдущем примере необходимо внести некоторые изменения:

```
- (void)downloadDataForURL:(NSURL*)url {
    NSPurgeableData *cachedData = [_cache objectForKey:url];
    if (cachedData) {
        // Запрет возможного уничтожения данных
        [cacheData beginContentAccess];

        // Использование кэшированных данных
        [self useData:cachedData];

        // Повторное включение возможности уничтожения
        [cacheData endContentAccess];
    } else {
        // Кэш-промах
        EOCNetworkFetcher *fetcher =
            [[EOCNetworkFetcher alloc] initWithURL:url];
        [fetcher startWithCompletionHandler:^(NSData *data){
```

```
        NSPurgeableData *purgeableData =
            [NSPurgeableData dataWithData:data];
        [_cache setObject:purgeableData
                  forKey:url
                    cost:purgeableData.length];

        // Вызывать beginContentAccess не нужно,
        // изначально режим обращения к объекту уже включен

        // Использование загруженных данных
        [self useData:data];

        // Данные снова могут быть удалены
        [purgeableData endContentAccess];
    }];
  }
}
```

Учтите, что при создании удаляемый объект данных возвращается со счетчиком ссылок удаления +1, так что вам не нужно явно вызывать для него `beginContentAccess`, но исходное состояние должно быть скомпенсировано вызовом `endContentAccess`.

УЗЕЛКИ НА ПАМЯТЬ

- Используйте объекты `NSCache` вместо `NSDictionary` для кэширования. Класс `NSCache` предоставляет оптимальное поведение вытеснения, обеспечивает потоковую безопасность и не копирует ключи, в отличие от словаря.
- Используйте ограничения количества объектов и стоимости для управления вытеснением объектов из кэша. Не относитесь к этим метрикам как к жестким лимитам; это всего лишь рекомендации для кэша.
- Используйте объекты `NSPurgeableData` для создания автоматически очищаемых данных, которые также автоматически удаляются из кэша при очистке.
- Правильное использование кэша повысит скорость отклика вашего приложения. Кэшируйте только те данные, повторное получение которых обходится относительно дорого — например, если данные загружаются из Сети или читаются с диска.

51 ПРИДЕРЖИВАЙТЕСЬ КОМПАКТНЫХ РЕАЛИЗАЦИЙ INITIALIZE И LOAD

Иногда для успешного использования с объектом необходимо выполнить некоторую инициализацию. В Objective-C классы, наследующие от корневого класса `NSObject` (к этой категории относится подавляющее большинство классов), содержат пару методов для выполнения этой задачи.

Первый из этих методов называется `load`, а его прототип выглядит так:

```
+ (void)load
```

Он вызывается один и только один раз для каждого класса и категории, добавляемых в исполнительную среду. Это происходит при загрузке библиотеки, содержащей класс или категорию, — обычно при запуске приложения (для любого кода, написанного для iOS, дело всегда обстоит именно так). Приложения Mac OS X обычно обладают большей свободой в реализации таких возможностей, как динамическая загрузка, поэтому библиотека может загружаться после запуска приложения. Метод `load` для категории всегда вызывается после метода класса, к которому относится категория.

Проблема с методом `load` заключается в том, что на момент его выполнения исполнительная среда находится в неустойчивом состоянии. Методы `load` всех суперклассов гарантированно выполняются до методов любого класса; кроме того, методы `load` классов зависимых библиотек гарантированно будут выполнены первыми. Однако в пределах любой конкретной библиотеки порядок загрузки классов не детерминирован. Следовательно, использование других классов в методе `load` небезопасно. Для примера рассмотрим следующий код:

```
#import <Foundation/Foundation.h>
#import "EOCClassA.h" //< Из той же библиотеки

@interface EOCClassB : NSObject
@end

@implementation EOCClassB
+ (void)load {
    NSLog(@"Loading EOCClassB");
    EOCClassA *object = [EOCClassA new];
    // Использование 'object'
}
@end
```

Использование NSLog и класса NSString, задействованного при выводе, безопасно, поскольку мы знаем, что к моменту выполнения метода load фреймворк Foundation уже загрузился. Однако использование EOCClassA в методе load класса EOCClassB уже небезопасно, потому что невозможно детерминированно предсказать, будет ли класс EOCClassA загружен к моменту вызова метода load класса EOCClassB. Не исключено, что в своем методе load класс EOCClassA выполняет какую-то важную работу, которая должна быть завершена перед тем, как экземпляр можно будет использовать.

Также важно заметить, что load не подчиняется обычным правилам наследования для методов. Если класс не реализует load, он не вызывается, что бы ни делали его суперклассы. Кроме того, load может присутствовать как в категории, так и в самом классе. В этом случае будут вызваны обе реализации, причем реализация класса будет вызвана раньше реализации категории.

Проследите за тем, чтобы реализация load была по возможности компактной, то есть в ней выполнялась минимальная необходимая работа, потому что на время ее выполнения будет заблокировано все приложение. Если метод load выполняет какие-то серьезные вычисления, приложение перестанет реагировать на указанный период. Не пытайтесь выполнять ожидание по любым блокировкам или вызывать методы, которые могут зависеть от блокировок. В сущности, метод должен делать как можно меньше. Метод load почти никогда не является правильным решением для выполнения операций, которые должны быть завершены до использования класса. Его единственным реальным применением является отладка — например, в категории, если вы хотите проверить, правильно ли она загружается. Возможно, когда-то этот метод был полезен, но сейчас можно с уверенностью сказать, что в современном коде Objective-C на него можно не обращать внимания.

Другой способ выполнения инициализации класса основан на переопределении следующего метода:

```
+ (void)initialize
```

Этот метод вызывается для каждого класса — один и только один раз, перед использованием класса. Метод вызывается исполнительной средой и никогда не должен вызываться напрямую. Он похож на load, но отличается от него в нескольких важных аспектах. Во-первых, метод initialize вызывается в отложенном (lazy) режиме, то есть в первый раз он будет вызван перед первым использованием

класса. Таким образом, если класс ни разу не используется, его метод `initialize` вызываться не будет. Впрочем, это означает, что в работе приложения нет периода, в котором выполняются все реализации `initialize`, — в отличие от реализаций `load`, блокирующих приложение до момента их завершения.

Второе отличие от `load` заключается в том, что исполнительная среда во время выполнения находится в нормальном состоянии, поэтому с точки зрения целостности исполнительной среды использование любых классов и вызовы их методов полностью безопасны. Кроме того, исполнительная среда следит за тем, чтобы выполнение `initialize` происходило в потоково-безопасной среде; это означает, что взаимодействие с классом или экземплярами класса разрешается только потоку, выполняющему `initialize`. Другие потоки блокируются до завершения `initialize`.

И последнее отличие: сообщение `initialize` передается так же, как любое другое сообщение. Если оно не поддерживается классом, но реализовано в его суперклассе, будет выполнена эта реализация. На первый взгляд это утверждение очевидно, но о нем часто забывают. Рассмотрим следующие два класса:

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

@interface EOCBaseClass : NSObject
@end

@implementation EOCBaseClass
+ (void)initialize {
    NSLog(@"%@ initialize", self);
}
@end

@interface EOCSubClass : EOCBaseClass
@end

@implementation EOCSubClass
@end
```

Хотя класс `EOCSubClass` не реализует `initialize`, ему все равно будет отправлено сообщение. Кроме того, сначала вызываются реализации `initialize` суперклассов. Итак, при первом использовании `EOCSubClass` будет получен следующий вывод:

```
EOCBaseClass initialize
EOCSubClass initialize
```

Возможно, вас это удивит, но результат абсолютно разумный. К `initialize`, как и к другим методам (кроме `load`!), применяются нормальные правила наследования, так что реализация из `EOCBaseClass` выполняется один раз при инициализации `EOCBaseClass`, а потом снова при инициализации `EOCSubClass`, поскольку она не была переопределена в `EOCSubClass`. Из-за этого очень часто встречаются реализации `initialize` следующего вида:

```
+ (void)initialize {
    if (self == [EOCBaseClass class]) {
        NSLog(@"%@ initialized", self);
    }
}
```

С такой проверкой инициализация выполняется только при инициализации нужного класса. В предыдущем примере вместо двух строк будет выведена только одна:

```
EOCBaseClass initialize
```

Все это приводит нас к главной рекомендации по поводу `load` и `initialize`, приводившейся ранее. Реализации обоих методов должны быть компактными. Они должны ограничиваться настройкой состояния, необходимого для правильного функционирования класса, но без выполнения сколько-нибудь продолжительных операций или ожидания блокировок. В случае `load` объяснение было приведено ранее; для `initialize` действуют аналогичные причины. Во-первых, никому не нравится, когда приложение зависает. Класс инициализируется при первом использовании, а это может происходить в любом потоке. Если инициализация выполняется в потоке пользовательского интерфейса, то поток блокируется на время инициализации, а приложение перестает реагировать на действия пользователя. Иногда трудно предсказать, какой поток впервые использует класс, и, конечно, нежелательно заставлять класс инициализироваться в конкретном потоке.

Во-вторых, вы не управляете тем, когда происходит инициализация класса. Она заведомо будет выполнена до первого использования класса, но полагаться на то, что инициализация будет выполнена в какой-то конкретный момент, слишком рискованно. В будущем какие-то аспекты инициализации классов в исполнительной среде могут измениться и ваши предположения относительно конкретного момента инициализации могут стать недействительными.

Наконец, при сложной реализации вы можете использовать (прямо или косвенно) другие классы из своего класса. Если эти классы

еще не были инициализированы, они тоже могут быть вынуждены инициализироваться. Однако инициализатор первого класса к этому моменту еще не отработал. Если другие классы рассчитывают на то, что некие данные первого класса уже инициализированы, это предположение может быть нарушено в момент выполнения `initialize` другого класса. Пример:

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

static id EOCClassAInternalData;
@interface EOCClassA : NSObject
@end

static id EOCClassBInternalData;
@interface EOCClassB : NSObject
@end

@implementation EOCClassA

+ (void)initialize {
    if (self == [EOCClassA class]) {
        [EOCClassB doSomethingThatUsesItsInternalData];
        EOCClassAInternalData = [self setupInternalData];
    }
}

@end

@implementation EOCClassB

+ (void)initialize {
    if (self == [EOCClassB class]) {
        [EOCClassA doSomethingThatUsesItsInternalData];
        EOCClassBInternalData = [self setupInternalData];
    }
}

@end
```

Если класс `EOCClassA` инициализируется первым, то его внутренние данные не будут готовы к тому моменту, когда класс `EOCClassB` вызовет `doSomethingThatUsesItsInternalData` для `EOCClassA`. На практике проблема может быть не столь очевидной, поскольку в ней может быть задействовано более двух классов. Соответственно, разработчику будет сложнее определить, почему что-то работает не так, как задумано.

Итак, метод `initialize` предназначен для инициализации внутренних данных класса. В нем не следует вызывать какие-либо методы, даже методы самого класса. Если позднее в вызываемый метод будет добавлена новая функциональность, у вас могут возникнуть описанные выше проблемы. Ограничьте инициализаторы настройкой глобального состояния, которое не может быть инициализировано во время компиляции. Следующий пример демонстрирует это:

```
// EOCClass.h
#import <Foundation/Foundation.h>

@interface EOCClass : NSObject
@end

// EOCClass.m
#import "EOCClass.h"

static const int kInterval = 10;
static NSMutableArray *kSomeObjects;

@implementation EOCClass

+ (void)initialize {
    if (self == [EOCClass class]) {
        kSomeObjects = [NSMutableArray new];
    }
}

@end
```

На стадии компиляции может быть определено целое число, но не массив, потому что это объект Objective-C и для создания его экземпляра необходима активная исполнительная среда. Обратите внимание: некоторые объекты Objective-C (например, экземпляры `NSString`) могут создаваться во время компиляции. Но при попытке откомпилировать следующий фрагмент вы получите сообщение об ошибке:

```
static NSMutableArray *kSomeObjects = [NSMutableArray new];
```

Помните об этом, если вам когда-либо придется писать метод `load` или `initialize`. Компактная реализация избавит вас от многочасовой отладки. А если простой инициализации глобального состояния окажется недостаточно, создайте метод для ее выполнения и потребуйте, чтобы пользователи вызывали его перед использованием

класса. В качестве примера можно привести синглетные классы, которые выполняют дополнительную работу при первом обращении.

УЗЕЛКИ НА ПАМЯТЬ

- Классы проходят фазу загрузки, в которой для них вызывается метод `load`, если он был реализован. Этот метод также может присутствовать в категориях, причем метод `load` класса всегда вызывается до метода `load` категории. В отличие от других методов, метод `load` не участвует в переопределении.
- Перед первым использованием классу отправляется сообщение `initialize`. Этот метод участвует в переопределении, так что обычно в нем стоит проверить, какой именно класс инициализируется.
- Реализации `load` и `initialize` должны быть как можно более компактными. Это обеспечивает быстрый отклик приложения и сокращает вероятность появления циклов взаимозависимостей.
- Оставьте методы `initialize` для настройки глобального состояния, которая не может быть выполнена во время компиляции.

52 ЗАПОМНИТЕ, ЧТО NSTIMER УДЕРЖИВАЕТ ПРИЕМНИК

Таймер — полезный объект в арсенале разработчика. Фреймворк Foundation содержит класс с именем `NSTimer`, который может планироваться для выполнения либо по абсолютной дате/времени, либо по истечении заданной задержки. Таймеры также могут срабатывать периодически; для этого с каждым таймером связывается интервал, определяющий частоту их срабатывания. Например, таймер может срабатывать каждые 5 секунд для опроса состояния некоторого ресурса.

С таймером связывается цикл выполнения (run loop), который управляет срабатыванием таймера. При создании таймер либо предварительно планируется в текущем цикле выполнения, либо разработчик самостоятельно обеспечивает его создание и планирование. В любом случае таймер срабатывает только в том случае, если он запланирован в цикле выполнения. Например, метод для создания таймера с предварительным планированием выглядит так:

```
+ (NSTimer *)scheduledTimerWithTimeInterval:
                    (NSTimeInterval)seconds
                              target:(id)target
                            selector:(SEL)selector
                            userInfo:(id)userInfo
                             repeats:(BOOL)repeats
```

Метод может использоваться для создания таймера, который срабатывает по истечении заданного интервала времени. Также возможно повторение с ручной остановкой таймера в будущем. Приемник (`target`) и селектор (`selector`) указывают, какой селектор и для какого объекта будет вызываться при срабатывании таймера. Таймер удерживает свой приемник и освобождает его, когда таймер становится недействительным, что происходит при вызове `invalidate` или при срабатывании. Если для таймера включено повторение, вызовите для таймера `invalidate`, когда захотите остановить его.

Так как таймер удерживает свой приемник, таймеры с повторением часто создают проблемы в приложениях. Повторение часто создает в приложении ситуацию с циклом удержания. Чтобы понять, как это происходит, рассмотрим пример:

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

@interface EOCClass : NSObject
- (void)startPolling;
- (void)stopPolling;
@end

@implementation EOCClass {
    NSTimer *_pollTimer;
}

- (id)init {
    return [super init];
}

- (void)dealloc {
    [_pollTimer invalidate];
}

- (void)stopPolling {
    [_pollTimer invalidate];
    _pollTimer = nil;
}
```

```
- (void)startPolling {
    _pollTimer =
    [NSTimer scheduledTimerWithTimeInterval:5.0
                                     target:self
                                   selector:@selector(p_doPoll)
                                   userInfo:nil
                                    repeats:YES];
}

- (void)p_doPoll {
    // Опрос ресурса
}
@end
```

А вы заметили проблему? Подумайте, что произойдет при создании экземпляра класса и запуске опроса. Созданный таймер удерживает экземпляр, потому что приемником является `self`. Однако таймер также удерживается экземпляром, потому что он присваивается переменной экземпляра (см. подход 30). В результате возникает цикл удержания; это было бы приемлемо, если бы цикл в какой-то момент разрывался. Разрыв возможен только при вызове `stopPolling` или уничтожении экземпляра. Без контроля над кодом, использующим класс, мы не можем быть уверены в том, что метод `stopPolling` будет вызван. Но даже в этом случае нежелательно требовать, чтобы для предотвращения утечки был вызван этот метод. Кроме того, с недействительностью таймера посредством уничтожения экземпляра возникает «порочный круг»: экземпляр не будет уничтожен, потому что его счетчик ссылок никогда не уменьшается до нуля, пока таймер остается действительным. А таймер остается действительным, пока он не станет недействительным. Ситуация изображена на рис. 7.1.

После того как последняя ссылка на экземпляр `EOCClass` будет освобождена, экземпляр продолжит существовать благодаря таймеру, который его удерживает. Таймер никогда не будет освобожден, потому что экземпляр содержит сильную ссылку на него. Что еще хуже, экземпляр будет потерян навсегда, потому что ссылка на таймер является единственной ссылкой на него, — а на таймер нет других ссылок, кроме ссылки через экземпляр. Возникает утечка. В данном случае она особенно неприятна, потому что опрос будет продолжаться бесконечно. Если опрос приводит к загрузке данных из Сети, данные будут загружаться и загружаться, усиливая потенциальную утечку.

Для решения этой проблемы мало что можно сделать на уровне использования таймеров. Можно потребовать, чтобы перед освобождением экземпляра всеми остальными объектами вызывался

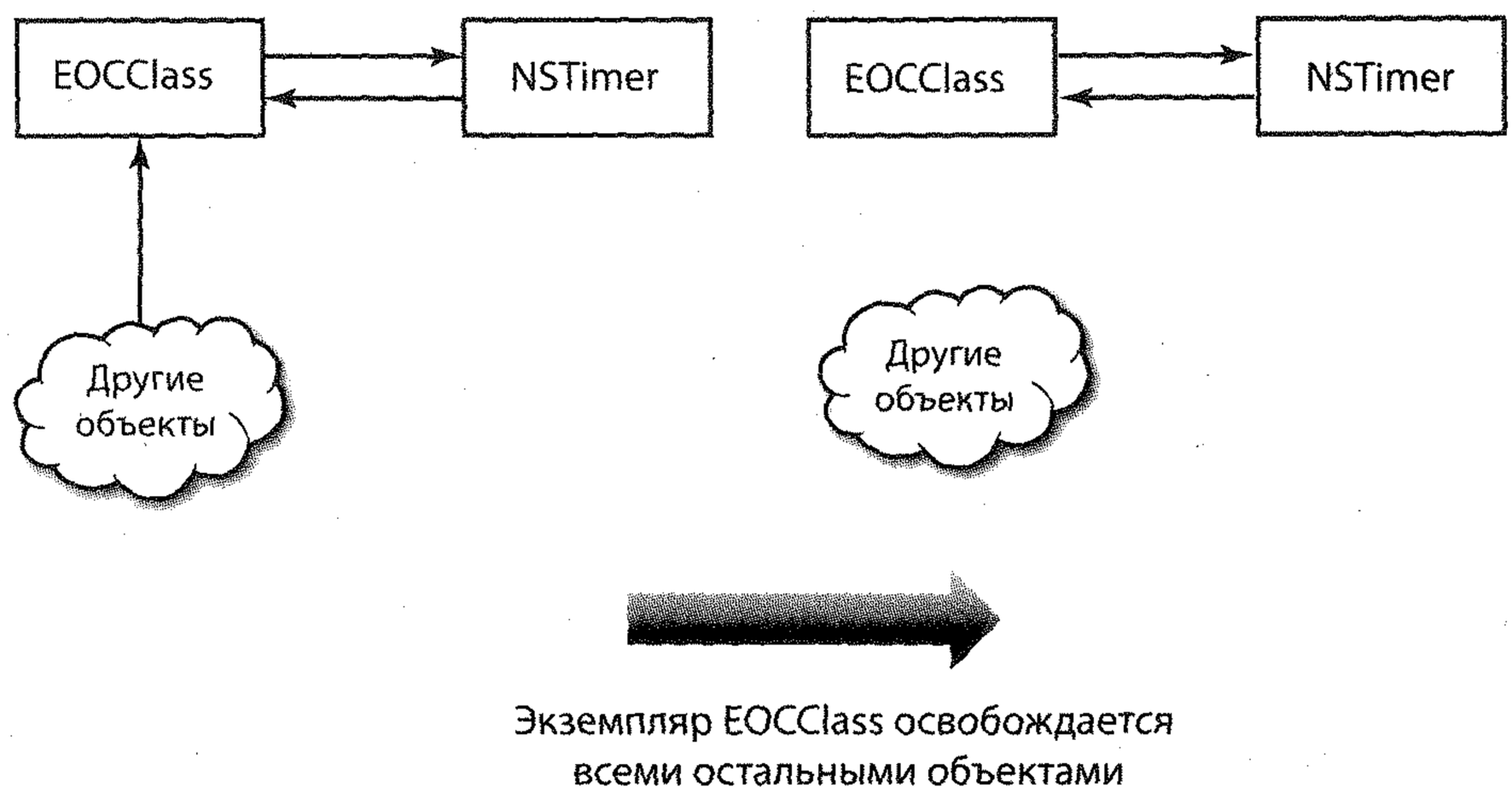


Рис. 7.1. Цикл удержания возникает из-за того, что таймер удерживает свой приемник, который в свою очередь удерживает таймер

метод `stopPolling`. Однако проконтролировать выполнение этого требования невозможно, и, если класс образует часть открытого API, предоставляемого другим разработчикам, вы не можете быть уверены в том, что метод будет вызван.

Один из способов решения проблемы основан на использовании блоков. Хотя таймеры в настоящее время не поддерживают блоки напрямую, необходимая функциональность может быть добавлена косвенно:

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

@interface NSTimer (EOCBlocksSupport)

+ (NSTimer*)eoc_scheduledTimerWithTimeInterval:
                            (NSTimeInterval)interval
                                     block:(void(^)())block
                                   repeats:(BOOL)repeats

@end

@implementation NSTimer (EOCBlocksSupport)

+ (NSTimer*)eoc_scheduledTimerWithTimeInterval:
                            (NSTimeInterval)interval
                                     block:(void(^)())block
                                   repeats:(BOOL)repeats
{
```

```
    return [self scheduledTimerWithTimeInterval:interval
                                         target:self
                                       selector:@selector(eoc_blockInvoke:)
                                       userInfo:[block copy]
                                        repeats:repeats];
}

+ (void)eoc_blockInvoke:(NSTimer*)timer {
    void (^block)() = timer.userInfo;
    if (block) {
        block();
    }
}

@end
```

Причины для такого решения проблемы цикла удержания вскоре станут более понятны. Блок, который должен выполняться при срабатывании таймера, задается в параметре `userInfo` таймера. Это непрозрачное значение, которое удерживается таймером, пока тот остается действительным. Копирование блока необходимо для того, чтобы он гарантированно был создан в куче (см. подход 37); в противном случае блок может оказаться недействительным при его последующем выполнении. Приемником таймера теперь является объект класса `NSTimer`, то есть синглетный объект, поэтому уже неважно, удерживается ли он таймером. Цикл удержания здесь остается, но, поскольку необходимость в уничтожении объекта класса никогда не возникает, это уже неважно.

Само по себе это «решение» не исключает проблемы, оно всего лишь предоставляет средства для ее решения. Представьте, что проблемный код был изменен и в нем используется новая категория:

```
- (void)startPolling {
    _pollTimer =
    [NSTimer eoc_scheduledTimerWithTimeInterval:5.0
                                          block:^{
                                              [self p_doPoll];
                                              }
                                        repeats:YES];
}
```

Хорошенько поразмыслив над этим кодом, вы заметите, что цикл удержания в нем так и остался. Блок удерживает экземпляр, потому что он захватывает `self`. В свою очередь, таймер удерживает блок через параметр `userInfo`. Наконец, таймер удерживается экземпля-

ром. Однако цикл удержания может быть разорван посредством использования слабых ссылок (см. подход 33):

```
- (void)startPolling {
    __weak EOCClass *weakSelf = self;
    _pollTimer =
    [NSTimer eoc_scheduledTimerWithTimeInterval:5.0
                                          block:^{
                                    EOCClass *strongSelf = weakSelf;
                                    [strongSelf p_doPoll];
                                                }
                                       repeats:YES];
}
```

В коде используется полезный паттерн определения слабой переменной `self`, которая захватывается блоком вместо обычной переменной `self`. Это означает, что `self` не будет удерживаться. Однако при выполнении блока немедленно генерируется сильная ссылка, которая гарантирует, что экземпляр заведомо будет существовать на время выполнения блока.

В этом паттерне при освобождении последней внешней ссылки экземпляр `EOCClass` будет уничтожен. При уничтожении таймер становится недействительным (см. исходный пример); это гарантирует, что таймер не будет выполняться повторно. Слабая ссылка повышает надежность решения; если таймер будет снова запущен по какой-либо причине (например, если вы забыли объявить его недействительным при уничтожении), `weakSelf` в блоке будет содержать `nil`.

УЗЕЛКИ НА ПАМЯТЬ

- Объект `NSTimer` удерживает свой приемник до того момента, когда таймер станет недействительным — либо из-за срабатывания, либо из-за явного вызова `invalidate`.
- Циклы удержания часто возникают при использовании таймеров с повтором и при удержании таймера его приемником. Цикл может создаваться прямо или косвенно через другие объекты в графе объектов.
- Для разрыва циклов удержания можно использовать расширение `NSTimer` с блоками. До того момента, когда эта функциональность станет частью открытого интерфейса `NSTimer`, ее приходится добавлять при помощи категории.

М. Гэлловей

Сила Objective-C 2.0.
Эффективное программирование для iOS и OS X

Перевел с английского Е. Матвеев

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Художник	*Л. Адуевская*
Корректоры	*С. Беляева, Л. Галаганова*
Верстка	*Л. Егорова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;
95 3005 — литература учебная.

Подписано в печать 25.11.13. Формат 70×100/16. Усл. п. л. 24,510. Тираж 1500. Заказ № 874.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.