

Objective-C 2.0 и программирование для Mac

ББК 32.973.26-018.2.75

Д15

УДК 681.3.07

Издательский дом "Вильямс".

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция канд. техн. наук И.В. Красикова

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

Далримпл, Марк, Кнастер, Скотт.

Д15 Objective-C 2.0 и программирование для Mac.: Пер. с англ. — М.: ООО "И.Д. Вильямс", 2010. — 320 с.: ил. — Парал. тит. англ.

ISBN 978-5-8459-1607-5 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Authorized translation from the English language edition published by APress, Inc., Copyright © 2009 by Mark Dalrymple and Scott Knaster.

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2010.

Научно-популярное издание
Марк Далримпл, Скотт Кнастер

Objective-C 2.0 и программирование для Mac

Литературный редактор

Верстка

Художественные редакторы

Корректор

Е.Д. Давидян

О.В. Мишуткина

Е.П. Дынник, Т.А. Тараброва

Л.А. Гордиенко

Подписано в печать 28.10.2009. Формат 70х100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 20,0. Уч.-изд. л. 17,9.

Тираж 1000 экз. Заказ № 19652.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

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

ISBN 978-5-8459-1607-5 (рус.)

ISBN 978-1430218159 (англ.)

© Издательский дом "Вильямс", 2010

© Copyright by Mark Dalrymple and Scott Knaster, 2009

Оглавление

Глава 1	Привет!.....	17
Глава 2	Расширения C	21
Глава 3	Введение в объектно-ориентированное программирование	35
Глава 4	Наследование	69
Глава 5	Композиция	85
Глава 6	Организация исходных файлов	97
Глава 7	Еще об Xcode	109
Глава 8	Краткий обзор Foundation Kit.....	139
Глава 9	Управление памятью	165
Глава 10	Инициализация объекта	181
Глава 11	Свойства	201
Глава 12	Категории	215
Глава 13	Протоколы	231
Глава 14	Введение в AppKit.....	243
Глава 15	Работа с файлами	259
Глава 16	Кодирование "ключ/значение"	271
Глава 17	NSPredicate	287
Приложение А	Переход к Objective-C с других языков	299
Предметный указатель		309

Содержание

	Об авторах	14
	О консультанте	14
	Благодарности	14
	Предисловие	15
Глава 1	Привет!.....	17
	Перед тем как начать	17
	Будущее делается вчера	18
	Что дальше	18
	Резюме	19
Глава 2	Расширения C	21
	Простейшая программа на Objective-C	21
	Создание программы "Hello Objective-C"	21
	Анализ программы "Hello Objective-C"	24
	Дурацкий #import	25
	NSLog() и @"строка"	26
	Логичен ли логический тип	29
	BOOL в действии	29
	Сравнение	32
	Резюме	33
Глава 3	Введение в объектно-ориентированное программирование	35
	Косвенность	36
	Косвенность и переменные	37
	Косвенность посредством имен файлов	39
	Косвенность в объектно-ориентированном программировании	45
	Процедурное программирование	45
	Реализация объектной ориентированности	51
	Перерыв на терминологию	56
	Объектно-ориентированное программирование в Objective-C	57

	Раздел @interface	57
	Раздел @implementation.....	61
	Инстанцирование объектов.....	63
	Расширение программы	65
	Резюме.....	67
Глава 4	Наследование	69
	Зачем нужно наследование.....	69
	Синтаксис наследования	73
	Терминология	75
	Как работает наследование.....	76
	Диспетчеризация методов	76
	Переменные экземпляра.....	78
	Перекрытие методов	80
	Это просто супер	81
	Резюме.....	83
Глава 5	Композиция	85
	Что такое композиция.....	85
	Модель автомобиля	86
	Настройка для NSLog().....	86
	Методы доступа	89
	Установка двигателя	91
	Настройка колес.....	92
	Обработка изменений в автомобиле.....	93
	Расширение составных частей.....	94
	Композиция или наследование?.....	95
	Резюме.....	96
Глава 6	Организация исходных файлов	97
	Разделение интерфейса и реализации	97
	Создание новых файлов в Xcode	98
	Как разбить автомобиль	100
	Зависимости между файлами	103
	Минимальная рекомпиляция.....	103
	Поехали.....	105
	Импорт и наследование.....	106
	Резюме.....	108

Глава 7	Еще об Xcode	109
	Изменение имени компании.....	109
	Советы по использованию редактора.....	110
	Помощь в написании кода	112
	Отступы.....	113
	Завершение кода.....	113
	Подсветка скобок.....	116
	Массовое редактирование.....	116
	Навигация по коду.....	120
	emacs — не Mac	120
	Поиск.....	121
	Сезам, откройся!.....	122
	Закладки.....	122
	Сфокусируйтесь.....	123
	Панель навигации.....	125
	Получение информации.....	127
	Research Assistance	128
	Документирование.....	129
	Отладка	130
	Проще некуда	130
	Отладчик Xcode.....	130
	Настройка.....	131
	Отладка	131
	Просмотр данных.....	135
	Шпаргалка.....	136
	Резюме.....	137
Глава 8	Краткий обзор Foundation Kit.....	139
	Некоторые полезные инструменты.....	140
	Диапазон	140
	Геометрические типы.....	141
	Строки.....	141
	Создание строки.....	142
	Методы класса.....	142
	Размер	143
	Сравнение строк	143
	Чувствительность к представлению данных.....	145

Кто внутри?.....	145
Изменчивость	146
Коллекция коллекций	148
NSArray	148
Изменяемые массивы	152
Перечисления.....	153
Быстрое перечисление.....	154
NSDictionary	155
Никаких наследников	156
Семейства значений	157
NSNumber	157
NSNumber	158
NSNumber	159
Пример: поиск файлов.....	160
Еще быстрее	163
Резюме.....	164

Глава 9 Управление памятью 165

Жизненный цикл объекта	166
Подсчет ссылок	166
Владение объектом.....	168
Владение и освобождение в методах доступа	169
Автоосвобождение.....	171
NSAutoreleasePool	171
О деструкции.....	172
Пулы в действии	173
Правила управления памятью Сосоа.....	174
Временные объекты	175
Захват объектов	176
Чисто там, где убирают	178
Резюме.....	179

Глава 10 Инициализация объекта 181

Выделение объектов	181
Инициализация объектов	182
Написание методов инициализации.....	182
Что делать при инициализации.....	184

Инициализаторы для удобства	185
Практический пример	185
Инициализация колес	186
Обновление функции main()	188
Автомойка	190
Использование сборки мусора	193
Инициализаторы для удобства	194
Стандартный инициализатор	195
Проблема наследования	196
Исправление инициализаторов Tire	198
Добавление инициализатора AllWeatherRadial	199
Правила инициализаторов	199
Резюме	199
Глава 11 Свойства	201
Упрощение кода при помощи свойств	201
Упрощение интерфейса	203
Упрощение реализации	204
...и точка!	206
Возражения против свойств	206
Разноименность	211
Защита от записи	212
Увы, свойства — не панацея	212
Резюме	213
Глава 12 Категории	215
Создание категорий	215
@interface	216
@implementation	216
Плохие категории	218
Хорошие категории	218
Разделение реализации по категориям	219
Использование категорий в нашем проекте	219
Опережающие ссылки и категории	223
Спасительные категории	224
Неформальные протоколы и категории делегирования	224
Проект iTunesFinder	225

Делегаты и категории.....	228
Ответы на селекторы.....	229
Другие применения селекторов.....	230
Резюме.....	230
Глава 13 Протоколы	231
Формальные протоколы.....	231
Объявление протоколов.....	232
Принятие протокола.....	232
Реализация протокола.....	233
Копирование.....	233
Копирование двигателей.....	234
Копирование колес.....	235
Копирование автомобиля.....	237
Протоколы и типы данных.....	240
Дополнения Objective-C 2.0.....	240
Резюме.....	241
Глава 14 Введение в AppKit.....	243
Создание проекта.....	243
Создание интерфейса класса.....	246
Interface Builder.....	247
Создание схемы пользовательского интерфейса.....	249
Установка соединений.....	251
Подключение выходов.....	251
Подключение действий.....	253
Реализация AppController.....	254
Резюме.....	257
Глава 15 Работа с файлами.....	259
Списки свойств.....	259
NSDate.....	259
NSData.....	260
Запись и чтение списков свойств.....	261
Кодирование объектов.....	263
Резюме.....	268

Глава 16 Кодирование "ключ/значение"	271
Исходный проект.....	271
Введение в KVC.....	273
В путь.....	275
Агрегаты	276
Пит-стоп.....	277
Операторы.....	281
Пакетные изменения	283
nil.....	284
Обработка необработанного.....	284
Резюме.....	286
Глава 17 NSPredicate.....	287
Создание предиката.....	288
Вычисление предиката.....	289
Фильтры.....	290
Спецификаторы формата	291
Операторы.....	293
Операторы сравнения и логические операторы	293
Массивы.....	294
SELF	295
Операции со строками	296
LIKE	297
Это все	298
Приложение А Переход к Objective-C с других языков	299
C.....	300
C++	301
Таблица виртуальных функций C++	
и динамическая диспетчеризация Objective-C.....	301
Objective-C++	304
Java.....	305
BASIC.....	307
Языки сценариев	307
Резюме.....	308
Предметный указатель	309

Джерри Шерцеру (Jerri Shertzer) — учителю и другу.

Марк

Об авторах

Марк Далримпл — программист на платформах Mac и Unix с большим стажем, работающий над кроссплатформенным инструментарием, инструментами для публикации в Интернете, высокопроизводительными веб-серверами и настольными приложениями. Он является также автором книги *Advanced Mac OS X Programming*. В свободное время Марк играет на тромбоне и фаготе и делает из надувных шариков забавных зверушек.

Скотт Кнастер — легендарный (т.е. очень старый) программист и автор таких бестселлеров, как *Take Control of Switching to the Mac* и *Macintosh Programming Secrets*. Его книга *How to Write Macintosh Software* не теряла своей популярности среди Mac-программистов более десятилетия. Он живет в доме со своими многочисленными домашними животными.

О консультанте

Джефф Ламарш (Jeff LaMarche) — разработчик программного обеспечения для Mac и сертифицированный разработчик Apple iPhone с более чем 20-летним стажем. Он пишет о Cocoa и Objective-C для *MacTech Magazine*, а также статьи для веб-сайта *Apple's Developer Technical Services*. Он также работал в PeopleSoft — сначала в качестве разработчика, а затем как независимый консультант.

Благодарности

Если вы не впервые сталкиваетесь с технической литературой, то встречались с разделом благодарностей и знаете, что, несмотря на наличие на обложке (в данном случае) только двух имен, над книгой работает масса народа.

В частности, нам бы хотелось отметить нашего основного критика — Дениз Санторо Линкольн (Denise Santoro Lincoln). Она постоянно кормила нас ассорти из проблем и вопросов, которое готовила с необычайным вкусом, изяществом и юмором. Спасибо также Клею Андресу (Clay Andres) и Джеффу Ламаршу, которые помогли убедиться, что в книге мы ни разу не соврали. Несчетное количество благодарностей мы шлем Лауре Истерман (Laura Esterman), нашему выпускающему редактору, и Хизеру Лэнгу (Heather Lang) за превращение кучи страниц с текстом в книгу, которую вы держите в руках.

Марк хотел бы поблагодарить Аарона Хайллегасса (Aaron Hillegass) за то, что он познакомил его не только с Objective-C и Cocoa, но и со Скоттом и Дэйвом. Без Аарона ничего этого не было бы. Кроме того, Марк хотел бы упомянуть Грегга Миллера (Greg Miller), который показал Марку всю крутизну кодирования "ключ/значение" и NSPredicate. Скотт же благодарит Марка за то, что он взял на себя основную массу работы.

Наконец, совершенно невозможные благодарности приносятся Дэйву Марку (Dave Mark). Без его пронизательности, настойчивости и... придирчивости эта книга никогда бы не вышла в свет.

Предисловие

Одна из опасностей, подстерегающих программиста с большим опытом работы, — потерять возможность получать наслаждение, которое приносит программирование. К счастью, постоянно появляются новые технологии, позволяющие возродить угасающий интерес, и Mac OS X не является исключением.

Objective-C — это язык программирования, который объединяет скорость и повсеместность C с элегантной объектно-ориентированной средой. Objective-C представляет собой почву, на которой выращены такие технологии Apple, как Cocoa или iPhone SDK. Овладев Objective-C, вы будете идеально вооружены для завоевания данной платформы. А затем можете попытаться овладеть и всем миром...

Привет!

Д

обро пожаловать в новую книгу, посвященную Objective-C для Mac! Назначение этой книги — обучить вас азам языка программирования Objective-C. Язык Objective-C представляет собой надмножество языка программирования C и используется для написания множества (если не большинства) приложений Mac OS X.

Эта книга обучит вас языку Objective-C и познакомит с его “компаньоном” — инструментарием Apple’s Cocoa. Cocoa написан на языке программирования Objective-C и содержит не только все элементы пользовательского интерфейса Mac OS X, но и еще очень много полезных для программиста вещей. После того как изучите Objective-C при помощи нашей книги, вы будете готовы поглубже познакомиться с Cocoa, что можно сделать с помощью книг Дэйва Марка (Dave Mark) и Джеффа Ламарша (Jeff LaMarche) *Learn Cocoa on the Mac* или *Beginning iPhone Development*.

В этой главе мы расскажем, что вам следует знать, перед тем как приступить к чтению основного материала книги. Мы также вкратце познакомим вас с историей Objective-C и бегло ознакомимся с содержанием прочих глав книги.

Перед тем как начать

Читателю крайне желательно иметь опыт программирования на C-подобном языке программирования, таком как C++, Java или, в конце концов, на самом C. Каким бы ни был язык программирования, читатель должен твердо знать основные его принципы. Вы должны знать, что такое переменные и функции, понимать, как управлять потоком выполнения программы при помощи условных переходов и циклов. Основным предметом нашего рассмотрения являются дополнительные возможности, вносимые Objective-C в базовый язык C, а также некоторые избранные вопросы Apple’s Cocoa.

Если вы перешли на Objective-C с языка программирования, не связанного с C, то и в этом случае вы можете приступить к изучению книги, но начать, пожалуй, лучше с приложения А, "Переход к Objective-C с других языков", или книги Дэйва Марка (Dave Mark) *Learn C*.

Будущее делается вчера

Сердцем операционной системы Apple's Mac OS X являются Cocoa и Objective-C. Хотя Mac OS X относительно новая операционная система, Objective-C и Cocoa гораздо старше. Брэд Кокс (Brad Cox) изобрел Objective-C в начале 1980-х годов при попытке объединить популярность и переносимость языка программирования C с элегантностью Smalltalk. В 1985 Стив Джобс (Steve Jobs) основал компанию NeXT, Inc., которая должна была создавать мощные и недорогие рабочие станции. В качестве операционной системы NeXT выбрала Unix и создала NextSTEP, мощный инструментальный пользовательского интерфейса, разработанный на Objective-C. Несмотря на широкие возможности, NextSTEP не имел большого коммерческого успеха.

Когда в 1996 году Apple приобрела NeXT (или наоборот?), NextSTEP был переименован в Cocoa и представлен широкому кругу программистов Macintosh. Компания Apple также бесплатно предоставила всем Mac-программистам свои средства разработки (включая Cocoa), так что теперь любой мог воспользоваться всей мощью средств разработки, доступной Apple. Все, что вам надо — это немного опыта в программировании, знания Objective-C и желание написать свой собственный шедевр.

Если Objective-C и Cocoa созданы еще в 1980-х годах, не говоря уж об Unix, — то не слишком ли они устарели для сегодняшних потребностей? Абсолютно нет! Objective-C и Cocoa — результат многолетних усилий команды выдающихся программистов, и они постоянно развиваются и улучшаются. С течением времени Objective-C и Cocoa превратились в чрезвычайно элегантный и мощный набор инструментов. Кроме того, Objective-C — ключ к написанию программного обеспечения для iPhone. Так что сейчас, спустя более чем 20 лет после того, как компания NeXT приняла Objective-C, его используют все настоящие программисты.

Что дальше

Objective-C представляет собой надмножество C. Objective-C начинался с C, к которому затем постепенно добавлялись небольшие, но очень важные дополнения. Если вы знакомы с C++ или Java, то удивитесь, насколько в действительности невелик Objective-C. Дополнения Objective-C к C будут детально рассмотрены в последующих главах.

- Глава 2, "Расширения C", рассматривает фундаментальные возможности Objective-C.
- В главе 3, "Введение в объектно-ориентированное программирование", обсудим основы объектно-ориентированного программирования.
- Глава 4, "Наследование", описывает создание классов, которые получают все возможности своих родителей.

- В главе 5, "Композиция", рассматриваются вопросы объединения объектов для совместной работы.
- В главе 6, "Организация исходных файлов", представлены реально используемые стратегии для работы с исходными текстами программ.
- В главе 7, "Еще об Xcode", приведены различные советы, которые помогут вам в повседневной работе программиста.
- Недолго отдохнуть от Objective-C вы сможете в главе 8, "Краткий обзор Foundation Kit", которая познакомит вас с возможностями Cocoa.
- В главе 9, "Управление памятью", вы все еще будете продолжать знакомство с Cocoa.
- Глава 10, "Инициализация объекта", посвящена таинству рождения объектов.
- В главе 11, "Свойства", вы узнаете о новой записи с точкой в Objective-C и о том, как легко и просто обращаться к объекту.
- В главе 12, "Категории", рассказывается об очень крутой возможности Objective-C, которая позволяет добавлять собственные методы в уже существующие классы — даже в те, которые создавали не вы.
- В главе 13, "Протоколы", рассказывается о форме наследования в Objective-C, которая позволяет классам реализовывать упакованные наборы возможностей.
- Глава 14, "Введение в AppKit", даст вам возможность создавать эффектные приложения с применением Cocoa.
- Из главы 15, "Работа с файлами", вы узнаете, как сохранять и считывать ваши данные.
- В главе 16, "Кодирование «ключ/значение»", описывается способ косвенного обращения к данным.
- И наконец, в главе 17, "NSPredicate", мы покажем вам, как работать с данными.

Если вы ранее работали с другими языками программирования наподобие Java или C++ или на других платформах типа Windows или Linux, то вам стоит взглянуть на приложение А, "Переход к Objective-C с других языков", которое поможет вам легче преодолеть сложности перехода к Objective-C.

Резюме

Программы для Mac OS X написаны на Objective-C с применением методик, берущих свое начало в далеких 1980-х годах и которые ныне превратились в мощный инструментальный программиста. Мы предполагаем, что читатель книги уже имеет некоторые знания о программировании на языке программирования C.

Надеемся, что книга принесет вам и пользу, и удовольствие!

Расширения C

Objective-C представляет собой не более чем язык программирования C с некоторыми дополнительными возможностями, надстроенными поверх него. В этой главе мы рассмотрим несколько ключевых добавлений в процессе создания вашей первой программы на Objective-C.

Простейшая программа на Objective-C

Вероятно, вы знакомы с C-версией классической программы "Hello, World", которая выводит этот (или некий другой) текст, и на этом ее работа завершается. Обычно это первая программа, с которой знакомятся неопытные. Не будем нарушать добрую традицию и начнем трудиться над программой, которую назовем "Hello Objective-C".

Создание программы "Hello Objective-C"

В этой книге мы считаем, что на вашем компьютере установлен инструментальный Apple's Xcode. Если Xcode у вас нет или вы никогда не использовали его ранее, то вам стоит обратиться к соответствующему разделу главы 2 книги Дэйва Марка (Dave Mark) *Learn C on the Mac*, где пошагово описан процесс получения, установки и создания программ с применением Xcode.

В этом разделе мы пошагово пройдем процесс использования Xcode для создания вашего первого проекта Objective-C. Если вы уже знакомы с Xcode, можете пропустить представленный далее материал, не боясь ущемить наши чувства. Перед тем как идти далее, распакуйте архив Learn ObjC Projects данной книги, который можно загрузить со страницы Source Code/Download веб-сайта Apress. Интересующий нас проект находится в папке 02.01 – Hello Objective-C.

Создание проекта начнем с запуска Xcode. Приложение Xcode находится в папке /Developer/Applications. Для упрощения запуска можете разместить пиктограмму Xcode в доке.

После того как Xcode запустится, выберите пункт меню **File**⇒**New Project**. Xcode выведет список различных типов проектов, которые вы можете создать. Пока что мы ограничимся выбором в левой части окна пункта **Command Line Utility**, а в правой — **Foundation Tool**, как показано на рис. 2.1. Затем щелкните на кнопке **Choose**.

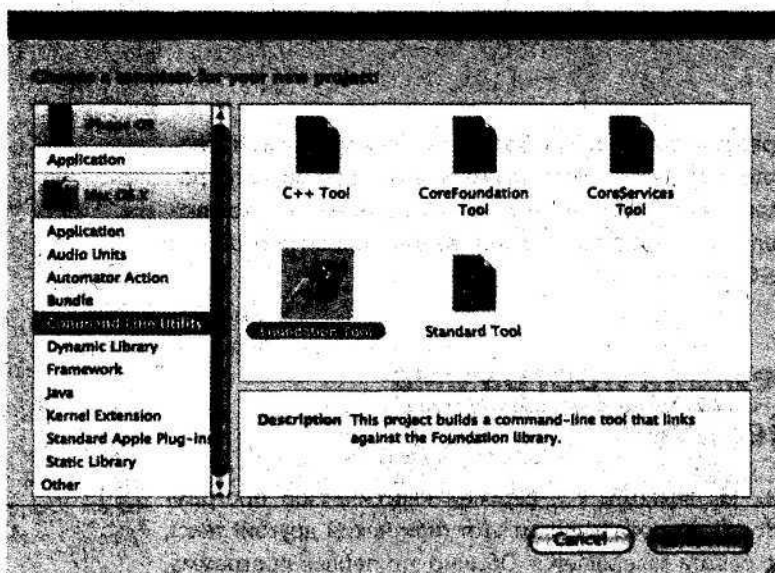


Рис. 2.1. Создание нового проекта *Foundation Tool*

Xcode запросит имя проекта. Вы можете дать ему любое имя, какое хотите, но, как видно из рис. 2.2, мы назвали его **Hello Objective-C**. Мы помещаем проект в один из наших каталогов **Projects**, чтобы обеспечить организованность данных, но вы можете разместить проект там, где это удобно вам.

После щелчка на кнопке **Save** Xcode откроет свое основное окно, которое называется окном проекта (рис. 2.3). Это окно наряду с окном редактора содержит составные части вашего проекта. Подсвеченный файл **Hello Objective-C.m** является исходным файлом, в котором содержится исходный текст приложения **Hello Objective-C**.

Файл **Hello Objective-C.m** содержит шаблонный код, любезно предоставляемый Xcode для каждого нового проекта. Но мы можем сделать наше приложение еще проще, чем предлагаемое Xcode. Удалите все из файла **Hello Objective-C.m** и введите в него следующий исходный текст.

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
    NSLog(@"Hello, Objective-C!");
    return (0);
} // main
```

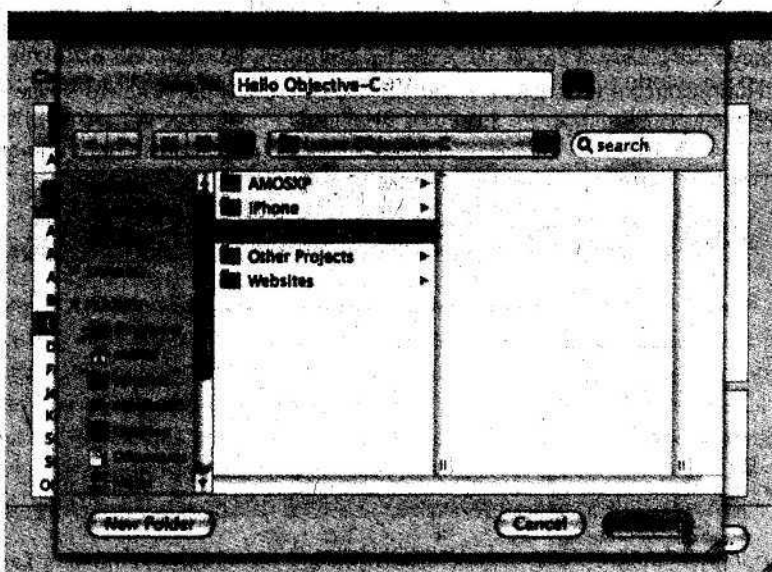


Рис. 2.2. Именованное нового проекта

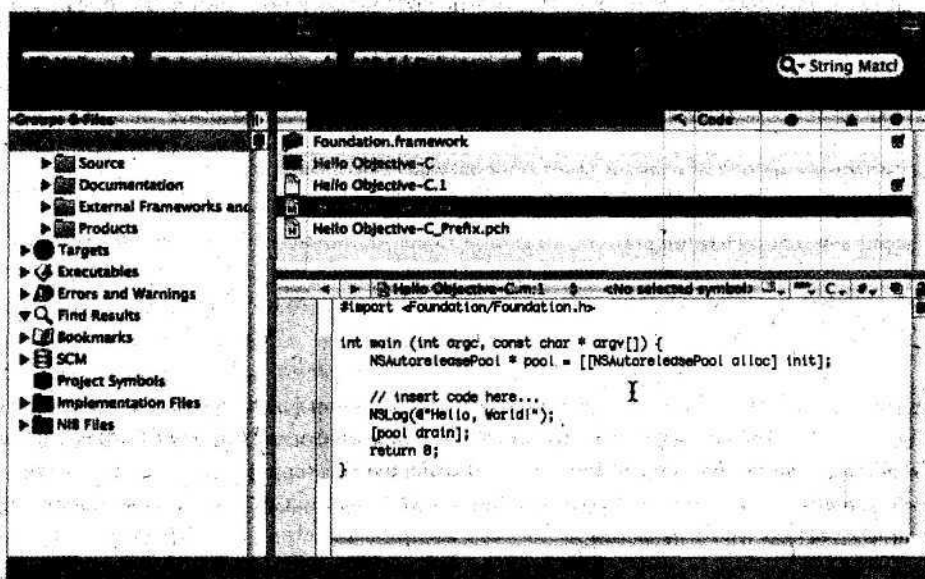


Рис. 2.3. Окно проекта Xcode

Если вы не понимаете, что означает приведенный код, не волнуйтесь. Вскоре мы рассмотрим эту программу детально, строка за строкой.

Исходный текст ничто, если вы не можете превратить его в работающую программу. Собрать и запустить программу можно при помощи щелчка на кнопке Build and Go или нажав

клавиши `<⌘+R>`. Если в исходном тексте нет синтаксических ошибок, Xcode скомпилирует и соберет вашу программу, а затем запустит ее. Откройте консольное окно Xcode (выбрав пункт меню `Run⇒Console` или нажав клавиши `<⌘+⇧+R>`), и вы увидите в нем вывод программы, показанный на рис. 2.4.

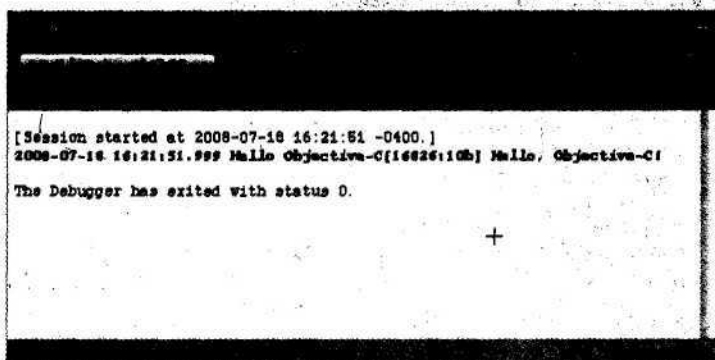


Рис. 2.4. Запуск программы *Hello Objective-C*

Итак, мы добились поставленной цели — первая работающая программа успешно создана и запущена. Поздравляем! Теперь разберем ее “по косточкам” и посмотрим, как она работает.

Анализ программы “Hello Objective-C”

Рассмотрим еще раз исходный текст программы *Hello Objective-C*.

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
    NSLog(@"Hello, Objective-C!");
    return (0);
} // main
```

Xcode использует расширение `.m` для указания того, что в файле хранится исходный текст на Objective-C, который будет обрабатываться компилятором Objective-C. Файлы, имена которых завершаются на `.c`, обрабатывает компилятор C, а `.cpp`-файлы — компилятор C++. (В Xcode все эти файлы компилирует GNU Compiler Collection (GCC), единый компилятор, работающий со всеми тремя языками.)

Файл `main.m` содержит две строки кода, которые должны быть знакомы любому, знающему язык программирования C: объявление `main()` и инструкция `return(0)` в конце. Поскольку в глубине души Objective-C — тот же C, синтаксис объявления функции `main()` и возврат из функции значения те же, что и в C. Остальной код несколько отличается от обычного C. Например, что за дурацкий `#import`? Чтобы узнать об этом — читайте дальше!

ПРИМЕЧАНИЕ

Расширение `.m` изначально, когда Objective-C только создавался, означало “messages” (сообщения), указывая на центральную особенность Objective-C, о которой мы поговорим попозже. Сейчас же мы просто говорим о таких файлах, как о “точка-m-файлах”.

Дурацкий `#import`

Так же, как и C, Objective-C использует заголовочные файлы для хранения объявлений таких элементов, как структуры, символьные константы и прототипы функций. В C для того, чтобы информировать компилятор о необходимости обратиться к заголовочному файлу за информацией, вы используете директиву `#include`. В программах на Objective-C вы также можете использовать `#include` для той же цели, но вряд ли будете это делать. Вместо этого вы будете применять директиву `#import`.

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

Директива `#import` представляет собой возможность, предоставляемую компилятором GCC, который Xcode использует для компиляции программ на Objective-C, C и C++. Эта директива гарантирует, что заголовочный файл будет включен в программу только один раз, независимо от того, сколько раз в файле будет обнаружена директива `#import`.

ПРИМЕЧАНИЕ

В C для избежания рекурсивного включения файлов, когда первый файл включает второй, который включает первый, программисты обычно применяют схему, основанную на директиве `#ifndef`.

В Objective-C для той же цели программисты используют директиву `#import`.

Инструкция `#import <Foundation/Foundation.h>` говорит компилятору о том, что ему надо просмотреть заголовочный файл `Foundation.h` в схеме (framework) `Foundation`.

Что такое схема? Рады ответить вам на этот вопрос. Это коллекция составных частей — заголовочных файлов, библиотек, изображений, звуков и прочего — собранных в единый модуль. Apple предоставляет такие технологии, как Cocoa, Carbon, QuickTime и OpenGL, в виде наборов схем. Cocoa состоит из пары схем, `Foundation` и `Application Kit` (известной также как `AppKit`), наряду со схемами поддержки, включающими, в частности, `Core Animation` и `Core Image`.

Схема `Foundation` обеспечивает возможности, лежащие в основе пользовательского интерфейса, такие как структуры данных и механизмы сообщений. Все программы в этой книге основаны на схеме `Foundation`.

ПРИМЕЧАНИЕ

Когда вы дочитаете эту книгу, следующим шагом по пути к вершинам программирования должно стать освоение `Cocoa's Application Kit`, в котором содержатся высокоуровневые возможности Cocoa: элементы пользовательского интерфейса, печать, управление цветом и звуком, поддержка `AppleScript` и т.д. В этом вам поможет книга Дэйва Марка (Dave Mark) и Джеффа Ламарша (Jeff LaMarche) *Learn Cocoa on the Mac*.

Каждая схема представляет собой большую коллекцию, зачастую состоящую из десятков или даже сотен заголовочных файлов. В каждой схеме есть основной заголовочный файл, который обеспечивает включение всех индивидуальных загрузочных файлов схемы. При помощи директивы `#import`, применяемой к основному заголовочному файлу, вы получаете доступ ко всем возможностям схемы.

Заголовочные файлы для схемы Foundation занимают около мегабайта дискового пространства и включают более 14 000 строк кода в более чем сотне файлов. При включении основного заголовочного файла конструкцией `#import <Foundation/Foundation.h>`, вы получаете доступ ко всей коллекции. Можно решить, что такое включение всего текста должно занять у компилятора слишком много времени, но Xcode достаточно разумен и использует предварительно скомпилированные заголовочные файлы (которые представляют собой сжатую и систематизированную информацию из заголовочных файлов) для ускорения работы.

Если вас интересуют файлы, включенные в схему Foundation, можете ознакомиться с ними в каталоге Headers (`/System/Library/Frameworks/Foundation.framework/Headers/`). Вы ни чему не навредите, просматривая эти файлы, — только не удаляйте и не изменяйте их!

NSLog() и @"строка"

Теперь, после того как воспользовались директивой `#import` для загрузки основного заголовочного файла схемы Foundation, мы готовы к написанию кода, использующего возможности Сосоа. Первая (и единственная) реальная строка кода в программе Hello Objective-C использует функцию `NSLog()`.

```
NSLog(@"Hello, Objective-C!");
```

Эта строка выводит "Hello, Objective-C!" на консоль. Если вы работали с C, то, наверняка, встречались с функцией `printf()`. Функция `NSLog()` представляет собой аналог `printf()` в Сосоа.

Так же как `printf()`, `NSLog()` в качестве первого аргумента принимает строку, которая может содержать спецификаторы формата (такие, как `%d`). Функция может принимать дополнительные аргументы, соответствующие спецификаторам форматов. Функция `printf()` подставляет эти дополнительные аргументы в строку перед ее выводом на консоль.

Как мы уже говорили ранее, Objective-C — это C, но под другим соусом, так что если вы захотите использовать `printf()` вместо `NSLog()` — нет вопросов! Но мы все же рекомендуем именно `NSLog()` из-за ее дополнительных возможностей, таких как указание даты и времени, а также автоматическое добавление символа новой строки (`"\n"`).

Вы можете решить, что `NSLog()` — странный выбор имени для функции. Что здесь делает "NS"? Оказывается, Сосоа добавляет ко всем именам своих функций, констант и типов префикс "NS". Этот префикс говорит о том, что функция взята именно из Сосоа, а не из некоторого другого набора.

Префиксы помогают предотвратить *коллизии имен*, возникающие при использовании одного и того же идентификатора для двух разных сущностей. Если бы в Сосоа эта функция носила имя `Log()`, то были бы достаточно велики шансы на столкновение этой функции с другой функцией `Log()`, созданной неким ничего не подозревающим программистом. При построении программы, содержащей такую функцию `Log()`, включение Сосоа приведет к тому,

что Xcode сообщит, что функция `Log()` многократно определена, так что построение работающей программы невозможно.

Теперь, когда вы понимаете, почему применение префиксов — такая хорошая идея, можете удивиться другому — почему именно "NS", а не, например, "Cocoa"? Дело в том, что префикс "NS" возник еще в те времена, когда инструментарий носил имя NextSTEP и был продуктом NeXT Software (бывш. NeXT, Inc.), приобретенной Apple в 1996 году. Чтобы не нарушать обратную совместимость кода, написанного для NextSTEP, Apple продолжила использовать префикс "NS". Сейчас — это просто исторический курьез.

Поскольку префикс NS "застолблен" Cocoa, очевидно, что вы не должны применять его для имен своих переменных или функций. Если вы так поступите, то, как минимум, введете в заблуждение читателей вашего исходного текста, которые будут считать, что эта функция или переменная относится к Cocoa. Кроме того, кто может поручиться, что Apple не добавит в Cocoa функцию с тем же именем, что использовали и вы? Централизованного реестра префиксов нет, так что вы вправе выбирать для себя любой собственный префикс. Многие используют в качестве префикса собственные инициалы или название компании. В данной книге для простоты мы не будем использовать префиксы вообще.

Давайте еще раз посмотрим на вызов `NSLog()`.

```
NSLog(@"Hello, Objective-C!");
```

Вы обратили внимание на знак перед строкой? Это не опечатка, а одна из особенностей, добавленных Objective-C к стандартному C. Строка в двойных кавычках с предваряющим символом @ должна рассматриваться как элемент Cocoa NSString.

А что такое элемент NSString? Убрав префикс "NS", мы увидим простое и понятное слово "String" (строка). Вы уже знаете, что строка — это последовательность символов, обычно "читабельных" человеком, так что вы, вероятно, вполне логично заключите, что NSString представляет собой последовательность символов в Cocoa.

Элементы NSString обладают огромным количеством возможностей и используются Cocoa везде, где требуются строки. Вот только малая часть того, на что способен элемент NSString:

- сообщить свою длину;
- сравнить себя с другой строкой;
- преобразовать себя в целое значение или в значение с плавающей точкой.

Уже из этого видно, что строки Cocoa обладают гораздо большими возможностями по сравнению со строками в стиле C. Более детально элементы NSString будут рассмотрены в главе 8, "Краткий обзор Foundation Kit".

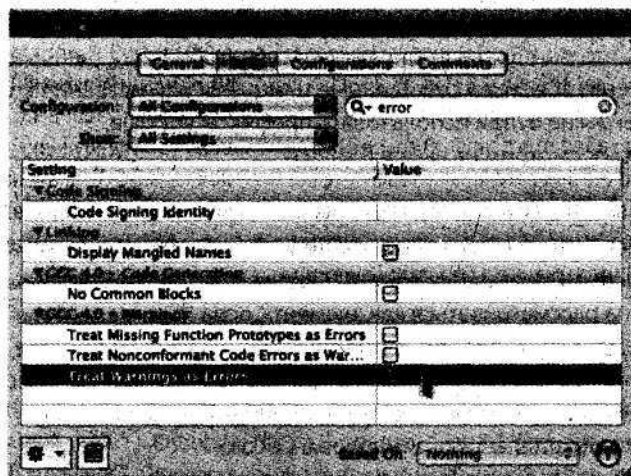
Вот еще одно замечание о NSString: самоописывающие имена — одна из приятных особенностей Cocoa. Большинство элементов Cocoa именовано так, что имена поясняют, о чем идет речь. Например, NSArray предоставляет возможность работы с массивами; NSDateFormatter помогает различным образом форматировать даты; NSThread предоставляет инструменты для многопоточного программирования; а NSSpeechSynthesizer позволяет прослушать речь.

СЛЕДИТЕ ЗА СТРОКАМИ

Легко допустить ошибку, которая заключается в передаче C-строки функции NSLog() вместо элемента NSString @"строка". Если вы сделаете такую ошибку, компилятор предупредит вас.

```
main.m:46: warning: passing arg 1 of 'NSLog'
from incompatible pointer type
```

При запуске такой программы может наступить ее аварийный останов. Чтобы отследить ошибки такого рода, следует указать Xcode рассматривать все предупреждения как ошибки. Чтобы это сделать, следует выбрать верхний пункт в списке Groups & Files, затем — пункт меню File⇒Get Info, выбрать вкладку Build, ввести в поле поиска error и установить переключатель Treat Warnings as Errors, как показано на приведенной далее копии экрана. Убедитесь также, что в выпадающем меню Configuration в верхней части окна указано All Configurations.



Вот еще одно замечание о NSString: самоописывающие имена — одна из приятных особенностей Сосоа. Большинство элементов Сосоа именовано так, что имена поясняют, о чем идет речь. Например, NSArray предоставляет возможность работы с массивами; NSDateFormatter помогает различным образом форматировать даты; NSThread предоставляет инструменты для многопоточного программирования; а NSSpeechSynthesizer позволяет прослушать речь.

Но вернемся к нашей программе. Последняя строка программы представляет собой инструкцию return, которая завершает выполнение функции main() и программы в целом.

```
return (0);
```

Нулевое возвращаемое значение говорит об успешном завершении программы. Это точно то же, что делает инструкция return в С.

Еще раз поздравляем! Вы только что написали, скомпилировали, запустили и проанализировали вашу первую программу на Objective-C.

Логичен ли логический тип

Многие языки имеют логический (булев) тип, который позволяет переменным содержать одно из двух значений — истинно или ложно. Objective-C исключением не является.

В C имеется тип данных `bool`, который может принимать значения `true` и `false`. Objective-C предоставляет аналогичный тип `BOOL`, который может иметь значения `YES` и `NO`. Так уж получилось, что в Objective-C тип `BOOL` появился более чем на десятилетие раньше типа `bool` в языке программирования C. Эти два логических типа могут сосуществовать в одной и той же программе, но при написании исходного текста Cocoa следует использовать тип `BOOL`.

ПРИМЕЧАНИЕ

`BOOL` в Objective-C в действительности представляет собой синоним типа (`typedef`) для знакового символьного типа (`signed char`), который использует 8 бит памяти. `YES` определено как 1, а `NO` — как 0 с применением директивы `#define`.

Objective-C не рассматривает `BOOL` как истинно логический тип, который в состоянии принимать только два значения — `YES` или `NO`. Компилятор рассматривает `BOOL` как 8-битовое число, а значения `YES` и `NO` представляют собой всего лишь соглашение. Это приводит к трудноуловимым ошибкам: если вы будете недостаточно внимательны и присвоите переменной типа `BOOL` значение размером более одного байта, такое как `short` или `int`, то для логического значения будет использован только младший его байт. Если этот байт случайно окажется нулевым (например, в случае 8960 (шестнадцатеричное значение 0x2300)), то значение `BOOL` тоже будет нулевым, т.е. значением `NO`.

BOOL в действии

Чтобы увидеть `BOOL` в действии, перейдем к следующему проекту — 02.02 – `BOOL Party`, в котором сравниваются пары целых чисел. Кроме `main()`, в программе определяются две функции. Первая из них, `areIntsDifferent()`, получает два целочисленных значения и возвращает значение типа `BOOL`: `YES`, если целые числа различны, и `NO`, если они одинаковы. Вторая функция, `boolString()`, получает параметр типа `BOOL` и возвращает строку `@YES`, если параметр равен `YES`, и `@NO`, если параметр равен `NO`. Это удобная функция, которая нужна для вывода удобочитаемого представления значения типа `BOOL`. Функция `main()` использует эти две функции для сравнения целочисленных значений и вывода результатов.

Создание проекта осуществляется точно так же, как и в случае предыдущего.

1. Запустите среду Xcode, если она еще не запущена.
2. Выберите пункт меню `File⇒New Project`.
3. Выберите в левой панели `Command Line Utility` и `Foundation Tool` — в правой.
4. Щелкните на кнопке `Choose`.
5. Введите `BOOL Party` в поле `Project Name` и щелкните на кнопке `Save`.

Отредактируйте файл `BOOL Party.m`, чтобы он имел следующий вид.

```

#import <Foundation/Foundation.h>

// Возвращает NO, если два целых числа имеют одно
// и то же значение, и YES в противном случае
BOOL areIntsDifferent (int thing1, int thing2)
{
    if (thing1 == thing2) {
        return (NO);
    } else {
        return (YES);
    }
} // areIntsDifferent

// В случае аргумента YES возвращает строку
// "YES". В противном случае возвращает "NO"
NSString *boolString (BOOL yesNo)
{
    if (yesNo == NO) {
        return (@ "NO");
    } else {
        return (@ "YES");
    }
} // boolString

int main (int argc, const char *argv[])
{
    BOOL areTheyDifferent;
    areTheyDifferent = areIntsDifferent (5, 5);
    NSLog (@ "Различны ли %d и %d? %@",
            5, 5, boolString(areTheyDifferent));
    areTheyDifferent = areIntsDifferent (23, 42);
    NSLog (@ "Различны ли %d и %d? %@",
            23, 42, boolString(areTheyDifferent));
    return (0);
} // main

```

Соберите и запустите программу. Чтобы увидеть результат работы программы, следует открыть окно консоли, выбрав пункт меню Run → Console или воспользовавшись клавишами <⌘+⌥+R>. В окне Run Debugger Console вы должны увидеть что-то наподобие следующего.

```

2008-07-20 16:47:09.528 02 B00L Party[16991:10b]
↳ различны ли 5 и 5? NO
2008-07-20 16:47:09.542 02 B00L Party[16991:10b]
↳ различны ли 23 и 42? YES

```

The Debugger has exited with status 0.

Давайте опять функция за функцией разберем программу и выясним, что она делает. Первая функция в нашем путешествии — areIntsDifferent().

```
// Возвращает NO, если два целых числа имеют одно
// и то же значение, и YES в противном случае
BOOL areIntsDifferent (int thing1, int thing2)
{
    if (thing1 == thing2) {
        return (NO);
    } else {
        return (YES);
    }
} // areIntsDifferent
```

Функция `areIntsDifferent()` получает два целочисленных параметра и возвращает значение типа `BOOL`. Синтаксис должен быть вам знаком из вашего опыта работы с C. Здесь вы можете видеть, как `thing1` сравнивается с `thing2`. Если они одинаковы, то возвращается значение `NO` (поскольку они НЕ различны). Если же они различны, возвращается значение `YES`. Все очень просто, правда?

БУДЬТЕ ОСТОРОЖНЫ

Опытные программисты на C могут захотеть переписать функцию `areIntsDifferent()` в виде одной инструкции.

```
BOOL areIntsDifferent_faulty (int thing1, int thing2)
{
    return (thing1 - thing2);
} // areIntsDifferent_faulty
```

Этот способ работает в предположении, что ненулевое значение — это то же самое, что и `YES`. Но это не так. Да, эта функция возвращает значение, которое C рассматривает как истинное или ложное, но вызывающая функция ожидает, что будет возвращено значение типа `BOOL` — `YES` или `NO`. Если программист попытается использовать функцию таким образом, как показано ниже, ничего не получится — поскольку 23 минус 5 равно 18.

```
if (areIntsDifferent_faulty(23, 5) == YES) {
    // ....
}
```

И хотя рассмотренный вариант функции возвращает истинное значение с точки зрения C, оно не равно `YES` (значение которого 1) в Objective-C. Здравая мысль заключается в том, чтобы никогда не сравнивать значение типа `BOOL` непосредственно с `YES`, поскольку слишком хитроумные программисты могут прибегнуть к штучкам наподобие показанной в функции `areIntsDifferent_faulty()`. Лучше переписать сравнение в следующем виде.

```
if (areIntsDifferent_faulty(5, 23)) {
    // ....
}
```

Непосредственное сравнение с `NO` всегда безопасно, так как ложь в C имеет единственное значение — ноль.

ГЛАВА 2. Расширения C

Вторая функция, `boolString()`, преобразует числовое значение `BOOL` в удобочитаемую строку.

```
// В случае аргумента YES возвращает строку
// "YES". В противном случае возвращает "NO"
NSString *boolString(BOOL yesNo)
{
    if (yesNo == NO) {
        return (@"NO");
    } else {
        return (@"YES");
    }
}
// boolString
```

Инструкция `if` в середине функции не должна преподносить никаких сюрпризов. Она просто сравнивает значение `yesNo` с константой `NO` и возвращает `@"NO"` в случае равенства. В противном случае `yesNo` должно быть истинным, так что возвращается строка `@"YES"`.

Обратите внимание, что возвращаемый тип функции `boolString()` — указатель на `NSString`. Это означает, что функция возвращает строку Cocoa, с которой вы уже знакомились при рассмотрении функции `NSLog()`. Если вы посмотрите на инструкцию `return`, то увидите символ `@` в начале возвращаемого значения, указывающий, что это элемент `NSString`.

Последняя функция — `main()`. После объявления возвращаемого типа и аргументов `main()` идет объявление локальной переменной типа `BOOL`.

```
int main (int argc, const char *argv[])
{
    BOOL areTheyDifferent;
```

Переменная `areTheyDifferent` хранит значение `YES` или `NO`, которое возвращает вызов `areIntsDifferent()`. Конечно, можно просто использовать возвращаемое функцией значение в инструкции `if`, не сохраняя его, но дополнительная переменная делает код более понятным. Глубоко вложенные конструкции сложны для чтения и понимания, а значит, представляют собой рассадники для всевозможных “жучков” в программах.

Сравнение

Следующие две строки кода сравнивают пару целочисленных значений при помощи вызова `areIntsDifferent()` и сохраняют возвращаемое значение в переменной `areTheyDifferent`. Функция `NSLog()` выводит числовые значения и строки, возвращаемые функцией `boolString()`.

```
BOOL areTheyDifferent;
areTheyDifferent = areIntsDifferent (5, 5);
NSLog (@"Различны ли %d и %d? %@",
    5, 5, boolString(areTheyDifferent));
```

Как вы видели ранее, `NSLog()` представляет собой Cocoa-аналог функции `printf()`, принимающей форматную строку и дополнительные параметры, которые используются ею для

вставки вместо соответствующих спецификаторов формата. Как видите в представленном во фрагменте **вызове** функции, две пятерки заменяют собой спецификаторы %d в форматной строке.

В конце переданной функции NSLog() строки имеется еще один символ @ — в этот раз это %@. Что бы это значило? Функция boolString() возвращает указатель на NSString. Функция printf() не знакома с NSString, так что и спецификатора для этого типа у нее нет. Но с функцией NSLog() дела обстоят иначе — на этот случай у нее имеется упомянутый спецификатор формата %@, который говорит функции, что соответствующий аргумент следует рассматривать как NSString.

ПРИМЕЧАНИЕ

Официально мы еще не знакомы с объектами, но скажем наперед: при выводе произвольного объекта функцией NSLog() используется спецификатор формата %@. При этом объект передает информацию о себе функции NSLog() при помощи метода description. Метод description у NSString просто выдает символы строки.

Две следующие строки очень похожи на только что рассмотренные.

```
areTheyDifferent = areIntsDifferent (23, 42);
NSLog (@ "Различны ли %d и %d? %@",
      23, 42, boolString(areTheyDifferent));
```

Функция сравнивает значения 23 и 42. В этот раз, поскольку они различны, функция areIntsDifferent() возвращает значение YES, и пользователь видит текст, гласящий, что 23 и 42 — разные числа.

И наконец, в программе идет инструкция return, завершающая наше приложение.

```
return (0);
} // main
```

В этой программе вы познакомились с типом BOOL Objective-C, а также с константами YES и NO, соответствующими истинным и ложным значениям. Вы можете использовать BOOL так же, как используете типы наподобие int и float: как переменные, аргументы функций или возвращаемые функциями значения.

Резюме

В этой главе вы написали первые две программы на Objective-C, и это было замечательно! Вы также познакомились с некоторыми расширениями Objective-C, такими как директива #import, которая заставляет компилятор включать заголовочные файлы, причем делать это однократно. Вы узнали о литералах NSString, которым предшествуют символы @, — как, например, в @"hello". Вы использовали Cocoa-функцию NSLog() для вывода текста на консоль, а также специальный спецификатор формата %@, который позволяет вставлять в вывод функции значения типа NSString.

Оставайтесь с нами — в следующей главе мы вступим в загадочный мир объектно-ориентированного программирования.

Введение в объектно- ориентированное программирование

Если бы хотя бы недолго занимались программированием, то, вероятно, не раз слышали термин "объектно-ориентированное программирование". *Объектно-ориентированное программирование* (часто сокращаемое как ООП) представляет собой технологию программирования, изначально разработанную для написания имитационных программ. Вскоре объектно-ориентированное программирование было принято на вооружение программистами, разрабатывающими и другие типы программ, например, использующие графический пользовательский интерфейс. Уже давно ООП стало расхожей аббревиатурой в программировании, и на нее смотрят как на некое магическое заклинание, делающее программирование простым и веселым занятием.

Это, конечно, преувеличение. Как и любая технология, объектно-ориентированное программирование требует упорного изучения и работы — но в этом случае оно действительно в состоянии облегчить работу, а иногда даже приносить удовольствие от нее. В этой книге мы будем обсуждать объектно-ориентированное программирование в первую очередь потому, что Cocoa базируется на его концепциях, а сам Objective-C создавался как объектно-ориентированный язык.

Так что же такое это объектно-ориентированное программирование? Это способ создания программного обеспечения, составленного из объектов. Объекты напоминают маленькие машины, которые живут внутри вашего компьютера и общаются друг с другом в процессе выполнения возложенных на них задач. В этой главе будут рассмотрены некоторые базовые концепции

объектно-ориентированного программирования. После этого мы изучим стиль программирования, ведущий к ООП, опишем мотивацию, лежащую в основе некоторых возможностей ООП. Закончим мы описанием механики объектно-ориентированного программирования.

ПРИМЕЧАНИЕ

Подобно многим "новым" технологиям, корни объектно-ориентированного программирования теряются в тумане времени. Объектно-ориентированное программирование прошло длинный путь, некоторыми из этапов которого являются Simula в 1960-х годах, Smalltalk в 1970-х, Cascal в 1980-х и многие другие языки программирования. Авторы современных языков программирования, таких как C++, Java, Python и, конечно, Objective-C, вдохновлялись в своей работе этими старыми языками.

Сейчас, когда мы собираемся погрузиться в объектно-ориентированное программирование, приготовьтесь услышать много новых странных слов. ООП отличается тем, что его терминология выглядит загадочнее и страшнее, чем то, что она обозначает. Можно подумать, что ученые-информатики специально придумывают новые слова, чтобы казаться умнее, но, в общем-то, это не так. Не беспокойтесь — каждый термин будет разъяснен.

Перед тем как приступить к самому объектно-ориентированному программированию, рассмотрим его ключевую концепцию — косвенность.

Косвенность

Старая программистская поговорка гласит: нет ничего, чего нельзя было бы достичь введением дополнительного уровня косвенности. *Косвенность* — это красивое слово с простым значением: вместо непосредственного использования значения в коде используется указатель на это значение. Вот пример из жизни: вы можете не знать телефонный номер любимой пиццерии, но вы знаете, что сможете найти его в телефонной книге. Такое использование телефонной книги — одна из форм косвенности.

Косвенность может также означать, что вместо того, чтобы сделать что-то самому, вы просите об этом другого человека. Пусть у вас есть книга, которую вы должны вернуть своему другу, живущему на другом конце города. Но тут вы узнаете, что ваш сосед собирается заехать к этому другу, и вместо того чтобы самому тащиться через весь город и обратно, вы просите отвезти книгу своего соседа. Это другая разновидность косвенности, когда, вместо того чтобы сделать что-то, вы выполняете некоторые иные действия.

В программировании косвенность может быть многоуровневой. Код может обращаться к другому коду, который, в свою очередь, выполняет обращение еще к одному коду. Вам не приходилось обращаться в службу технической поддержки? Вы излагаете свою проблему, и вас перенаправляют в отдел, в котором смогут решить вашу проблему. Здесь вас выслушивают еще раз и направляют к конкретному специалисту. (В этот момент выясняется, что вы, вообще-то, набрали неверный номер, вас не так поняли при первом звонке или произошло что-то подобное — и вам надо все начинать сначала.) Такое перебрасывание — тоже форма косвенности. К счастью, компьютеры терпеливы и безропотно следуют указаниям, пока не получат ответ.

Косвенность и переменные

Вас это может удивить, но вы, наверняка, имели дело с косвенностью в своих программах. Скромная переменная — это и есть реальное использование косвенности. Рассмотрим маленькую программу, выводящую числа от 1 до 5. Ее можно найти в папке Learn ObjC Projects, в подпапке 03.01 Count-1.

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
    NSLog(@"Числа от 1 до 5:");
    int i;
    for (i = 1; i <= 5; i++) {
        NSLog(@"%d\n", i);
    }
    return (0);
} // main
```

В программе имеется цикл `for`, выполняющий пять итераций и использующий функцию `NSLog()` для вывода значения переменной `i` на каждой итерации. При запуске программы вы увидите примерно следующее.

```
2008-07-20 11:54:20.463 03.01 Count-1[17985:10b] Числа от 1 до 5:
2008-07-20 11:54:20.466 03.01 Count-1[17985:10b] 1
2008-07-20 11:54:20.466 03.01 Count-1[17985:10b] 2
2008-07-20 11:54:20.466 03.01 Count-1[17985:10b] 3
2008-07-20 11:54:20.467 03.01 Count-1[17985:10b] 4
2008-07-20 11:54:20.467 03.01 Count-1[17985:10b] 5
```

Теперь предположим, что мы хотим обновить программу так, чтобы она выводила числа от 1 до 10. Вы должны исправить код в двух местах, выделенных полужирным шрифтом в приведенном ниже листинге, и заново собрать программу (эта версия находится в папке 03.02 Count-2).

```
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
    NSLog(@"Числа от 1 до 10:");
    int i;
    for (i = 1; i <= 10; i++) {
        NSLog(@"%d\n", i);
    }
    return (0);
} // main
```

Вот как выглядит вывод новой программы.

```
2008-07-20 11:55:35.909 03.02 Count-2[18001:10b] Числа от 1 до 10:
2008-07-20 11:55:35.926 03.02 Count-2[18001:10b] 1
2008-07-20 11:55:35.927 03.02 Count-2[18001:10b] 2
```

```
2008-07-20 11:55:35.928 03.02 Count-2[18001:10b] 3
2008-07-20 11:55:35.935 03.02 Count-2[18001:10b] 4
2008-07-20 11:55:35.936 03.02 Count-2[18001:10b] 5
2008-07-20 11:55:35.936 03.02 Count-2[18001:10b] 6
2008-07-20 11:55:35.939 03.02 Count-2[18001:10b] 7
2008-07-20 11:55:35.939 03.02 Count-2[18001:10b] 8
2008-07-20 11:55:35.940 03.02 Count-2[18001:10b] 9
2008-07-20 11:55:35.940 03.02 Count-2[18001:10b] 10
```

Конечно, нет никаких проблем во внесении в программу таких незначительных изменений — например, простой процедурой “найти и заменить”, тем более что в программе всего два таких места. А теперь представьте себе такой поиск с заменой в программе из нескольких тысяч строк. Учтите, что это может быть не просто замена всех пятерок на десятки — например, могут быть пятерки, которые не надо заменять на десятки, и наличествовать, скажем, четверки, которые надо заменить на девятки.

Решение этой проблемы — использование переменных. Вместо непосредственной вставки в код литерала 5 или 10, можно решить проблему, поместив число в переменную и добавив таким образом уровень косвенности. При добавлении переменной код уже не говорит “выполнить пять итераций цикла” — он говорит “посмотреть число, хранящееся в переменной count, и выполнить указанное количество итераций”. После введения переменной программа имеет следующий вид.

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

```
int main (int argc, const char * argv[])
{
    int count = 5;
    NSLog (@ "Числа от 1 до %d:", count);
    int i;
    for (i = 1; i <= count; i++) {
        NSLog (@ "%d\n", i);
    }
    return (0);
} // main
```

Вывод программы именно такой, как и ожидалось.

```
2008-07-20 11:58:12.135 03.03 Count-3[18034:10b] Числа от 1 до 5:
2008-07-20 11:58:12.144 03.03 Count-3[18034:10b] 1
2008-07-20 11:58:12.144 03.03 Count-3[18034:10b] 2
2008-07-20 11:58:12.145 03.03 Count-3[18034:10b] 3
2008-07-20 11:58:12.146 03.03 Count-3[18034:10b] 4
2008-07-20 11:58:12.151 03.03 Count-3[18034:10b] 5
```

ПРИМЕЧАНИЕ

Поскольку дополнительная информация, выводимая функцией `NSLog()`, занимает слишком много места и не несет полезной для наших целей изучения языка информации, в дальнейшем мы будем ее убирать.

Если мы захотим вывести числа от 1 до 100, то теперь нам потребуется внести изменения только в одном месте программы.

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

int main (int argc, const char * argv[])
{
    int count = 100;
    NSLog(@"Числа от 1 до %d:", count);
    int i;
    for (i = 1; i <= count; i++) {
        NSLog(@"%d\n", i);
    }
    return (0);
} // main
```

Введение в программу переменной сделало исходный текст понятнее и проще в работе, в особенности для внесения изменений в код. Для изменения количества итераций нам не надо заниматься расследованием функций, выполняемых каждой пятеркой в исходном тексте программы, чтобы выяснить, следует ли заменить ее на десятку. Мы изменяем значение переменной count и получаем желаемый результат.

Косвенность посредством имен файлов

Еще одним примером косвенности являются файлы. Рассмотрим программу, которая выводит список слов с указанием длины каждого слова (эта программа находится в папке 03.04 Word-Length-1). Вот ее листинг.

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

int main (int argc, const char * argv[])
{
    const char *words[4] = { "aardvark", "abacus",
                             "allude", "zygote" };
    int wordCount = 4;
    int i;

    for (i = 0; i < wordCount; i++) {
        NSLog(@"%s состоит из %d символов",
              words[i], strlen(words[i]));
    }
    return (0);
} // main
```

Цикл for определяет, какое слово в массиве words обрабатывается в данной итерации. Функция NSLog() в цикле выводит слово при помощи спецификатора формата %s. Мы использовали %s, так как words представляет собой массив C-строк, а не объектов @"NSString". Спецификатор формата %d получает целочисленное значение от функции strlen(), которая вычисляет длину переданной ей строки, и выводит его вместе со словом.

При запуске программы получим следующий результат.

```
aardvark состоит из 8 символов
abacus состоит из 6 символов
allude состоит из 6 символов
zygote состоит из 6 символов
```

ПРИМЕЧАНИЕ

Как и обещали, мы выбросили ненужную в настоящий момент информацию, выводимую функцией `NSLog()`.

Предположим теперь, что нам надо использовать различные наборы слов. Так как мы храним слова непосредственно в программе, вынуждены приступить к редактированию исходного текста программы, заменяя исходный список другим, с новыми именами. При редактировании следует быть очень внимательными, в особенности при использовании кавычек. Вот обновленная программа, которую вы можете найти в папке 03.05 Word-Length-2.

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

int main (int argc, const char * argv[])
{
    const char *words[4]
        = { "Joe-Bob \"Handyman\" Brown",
            "Jacksonville \"Sly\" Murphy",
            "Shinara Bain",
            "George \"Guitar\" Books" };
    int wordCount = 4;
    int i;

    for (i = 0; i < wordCount; i++) {
        NSLog (@"%s состоит из %d символов",
              words[i], strlen(words[i]));
    }
    return (0);
} // main
```

Поскольку мы были достаточно аккуратны, программа работает так, как от нее и ожидалось.

```
Joe-Bob "Handyman" Brown состоит из 24 символов
Jacksonville "Sly" Murphy состоит из 25 символов
Shinara Bain состоит из 12 символов
George "Guitar" Books состоит из 21 символов
```

Внесение таких изменений требует достаточно большого количества работы: мы должны отредактировать файл `Word-Length-2.m`, проверить его на наличие ошибок и опечаток и перестроить программу. Если такая программа работает, например, на веб-сайте, то мы должны дополнительно проверить ее и выложить на сервер.

Еще один способ состоит в отделении всех имен от кода и переносе их в отдельный текстовый файл, по одному имени в строке. Вы, наверняка, понимаете, что это и есть косвен-

ность. Вместо непосредственного размещения имен в исходном тексте, программа ищет их в другом месте. Программа считывает список обрабатываемых имен из текстового файла. Файлы проекта этой новой программы находятся в папке 03.06 Word-Length-3, а сам код имеет следующий вид.

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

int main (int argc, const char * argv[])
{
    FILE *wordFile = fopen ("/tmp/words.txt", "r");
    char word[100];

    while (fgets(word, 100, wordFile)) {
        // отбрасывание завершающего \n
        word[strlen(word) - 1] = '\0';
        NSLog (@"%s состоит из %d символов",
              word, strlen(word));
    }

    fclose (wordFile);

    return (0);
} // main
```

Давайте рассмотрим эту программу и разберемся, как она работает. Вначале функция `fopen()` открывает для чтения файл `words.txt`. Затем в цикле вызов `fgets()` считывает строку текста из файла и помещает ее в буфер `word`. Вызов `fgets()` сохраняет символ новой строки, отделяющий строки одну от другой, но этот символ нам как раз и не нужен: если оставить его как есть, он окажется учтен в общей длине строки. Поэтому мы заменяем символ новой строки нулевым символом, который указывает конец строки. И наконец, мы пользуемся помощью нашей старой знакомой — функции `NSLog()` — для вывода строки и ее длины.

ПРИМЕЧАНИЕ

Взгляните на имя файла, использованное функцией `fopen()`. Это — `/tmp/words.txt`. Это означает, что `words.txt` — файл в каталоге `/tmp`, временном каталоге Unix, который операционная система очищает при перезагрузке компьютера. Вы можете использовать каталог `/tmp` для хранения всяческих файлов, о сохранности которых вам не надо беспокоиться. В реальной работе, конечно, вы размещаете свои программы в таком месте, где им ничто не угрожает, — например, в своем домашнем каталоге.

Перед тем как запустить программу, воспользуйтесь текстовым редактором для создания файла `words.txt` в каталоге `/tmp`. Введите в этот файл следующие имена.

Joe-Bob "Handyman" Brown
 Jacksonville "Sly" Murphy
 Shinara Bain
 George "Guitar" Books

Чтобы сохранить файл в каталоге /tmp из текстового редактора, введите текст файла, выберите пункт Save, введите /tmp и нажмите клавишу <Enter>.

Вместо ввода имен вы можете просто скопировать файл words.txt из каталога 03.06 Word-Length-3 в каталог /tmp.

СОВЕТ

Если вы используете готовый проект Word-Length-3, то можете воспользоваться Xcode для копирования файла words.txt в каталог /tmp. Вот небольшая подсказка: обратите внимание на область Targets в панели Groups & Files.

При запуске программы вы получите тот же вывод программы, что и ранее.

Joe-Bob "Handyman" Brown состоит из 24 символов

Jacksonville "Sly" Murphy состоит из 25 символов

Shinara Bain состоит из 12 символов

George "Guitar" Books состоит из 21 символа

Рассмотренная программа представляет собой яркий пример косвенности. Вместо того чтобы жестко вносить слова непосредственно в программу, мы говорим "ищи слова в файле /tmp/words.txt". При использовании такой схемы можем использовать любой набор слов — просто отредактировав текстовый файл. Вы можете убедиться в этом самостоятельно, добавив в файл words.txt пару слов и перезапустив программу.

Такой подход лучше, поскольку отредактировать текстовый файл проще и менее опасно, чем изменять исходный текст программы. Это может сделать любой (не обязательно программист) с помощью текстового редактора. Трогать код не требуется вовсе.

Несмотря на существенный шаг в направлении косвенности, программа все еще далека от совершенства, поскольку включает полное имя файла с путем. Более того, файл находится в не самом приятном месте: при перезагрузке компьютера файл /tmp/words.txt исчезнет. Кроме того, если другие пользователи будут работать со своими программами, то они могут испортить и ваш файл /tmp/words.txt. Вам придется редактировать исходный текст программы всякий раз, когда потребуется использовать другой путь. Решение мы уже знаем — надо добавить еще один уровень косвенности.

Вместо того чтобы работать с файлом /tmp/words.txt, мы изменим программу и заставим ее узнавать о том, какой файл следует читать, из первого параметра, передаваемого программе при запуске. Вот очередная версия нашей программы, в которой аргумент командной строки указывает, из какого файла следует считывать информацию (эта программа может быть найдена в папке 03.07 Word-Length-4). Изменения, внесенные в исходный текст предыдущей программы, выделены полужирным шрифтом.

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

```
int main (int argc, const char * argv[])
```

```
{
    if (argc == 1) {
        NSLog (@ "Требуется имя файла");
        return (1);
    }
}
```

```

FILE *wordFile = fopen (argv[1], "r");
char word[100];

while (fgets(word, 100, wordFile)) {
    // отбрасывание завершающего \n
    word[strlen(word) - 1] = '\0';
    NSLog(@"%s состоит из %d символов",
          word, strlen(word));
}

fclose (wordFile);

return (0);
} // main

```

Цикл, обрабатывающий файл, тот же, что и в предыдущей версии программы, но код, открывающий этот файл, обновлен. Конструкция `if` проверяет, указал ли пользователь имя файла при запуске программы. Код проверяет значение аргумента `argc` функции `main()`, в котором хранится количество параметров, передаваемых программе при запуске. Поскольку имя самой программы всегда передается ей в качестве параметра запуска, значение `argc` всегда не меньше 1. Если пользователь не передает программе имя файла, значение `argc` равно 1, и мы не знаем, с каким файлом должны работать. Все, что нам остается — вывести сообщение об ошибке и завершить работу программы.

Если пользователь указывает имя файла, значение `argc` больше 1. Затем мы обращаемся к массиву `argv`, чтобы узнать значение переданного параметра. Элемент `argv[1]` содержит имя файла, переданное программе пользователем. (Если вас интересует, то аргумент `argv[0]` содержит имя программы.)

Если вы запускаете программу в терминале, то легко можете указать имя файла в командной строке.

```

$ ./Word-Length-4 /tmp/words.txt
Joe-Bob "Handyman" Brown состоит из 24 символов
Jacksonville "Sly" Murphy состоит из 25 символов
Shinara Bain состоит из 12 символов
George "Guitar" Books состоит из 21 символов

```

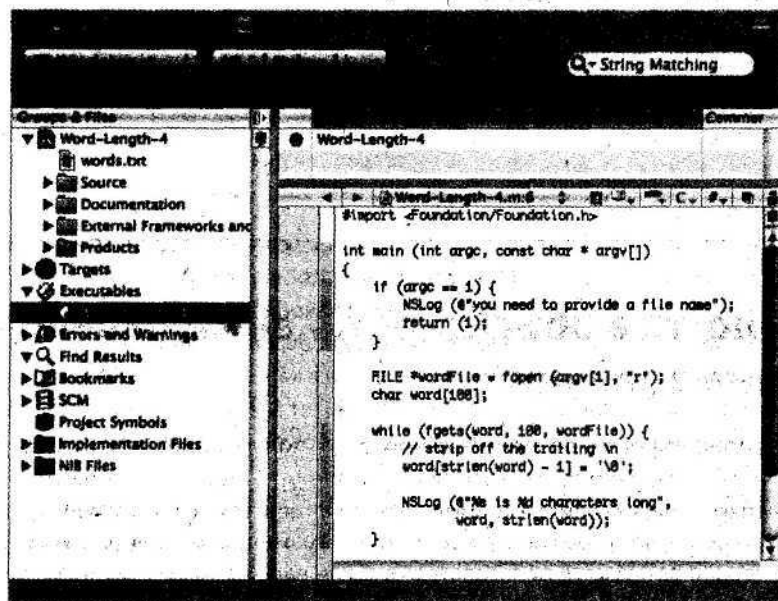
Просто ради интереса запустите эту программу, передав ей в качестве параметра файл `/usr/share/dict/words`, в котором содержится более 230 000 слов. Ваша программа в состоянии обработать это огромное количество данных! Когда вы устанете смотреть на все новые и новые строки в окне консоли Xcode, щелкните на красном значке останова для немедленного завершения работы программы.

Поскольку аргументы передаются во время выполнения программы, любой в состоянии воспользоваться ею для получения длин слов *любого* набора, даже аномально большого. Пользователи могут изменять данные без изменения кода. В этом вся суть косвенности: указать, где находятся необходимые данные.

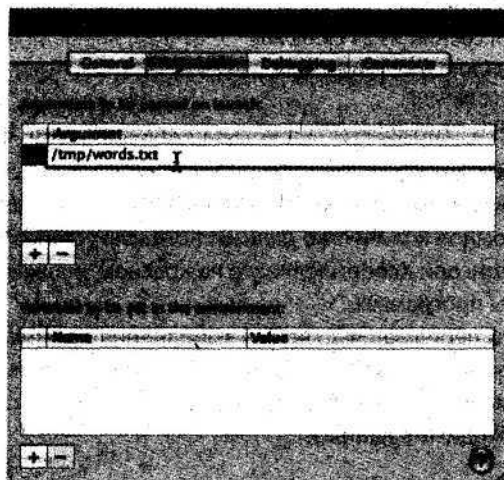
ПЕРЕДАЧА ИМЕНИ ФАЙЛА В XCODE

Если вы вместе с нами редактируете программу в Xcode, то передать параметр с именем файла несколько сложнее. Параметры запуска, именуемые также параметрами командной строки, указываются в Xcode немного сложнее, чем в терминале. Вот что для этого требуется.

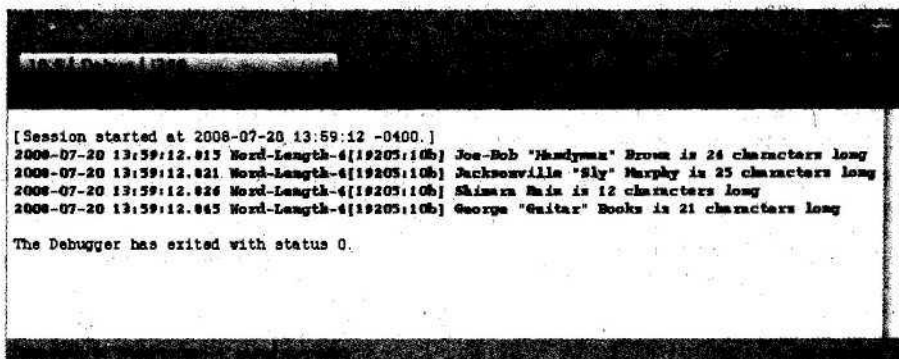
Сначала, в списке файлов Xcode, откройте список Executables и дважды щелкните на имени программы (Word-Length-4), как показано в приведенной далее копии экрана.



Далее, как показано на очередной копии экрана, щелкните на "+" в разделе Arguments и введите аргумент запуска (в нашем случае — путь к файлу words.txt).



Теперь, когда запустите программу, Xcode передаст аргумент запуска в массив argv программы. Вот что вы увидите в результате ее выполнения.



```

[Session started at 2008-07-20 13:59:12 -0400.]
2008-07-20 13:59:12.815 Word-Length-4[19205:10b] Joe-Bob "Handyman" Brown is 24 characters long
2008-07-20 13:59:12.821 Word-Length-4[19205:10b] Jacksonville "Sly" Murphy is 25 characters long
2008-07-20 13:59:12.826 Word-Length-4[19205:10b] Shimura Rain is 12 characters long
2008-07-20 13:59:12.845 Word-Length-4[19205:10b] George "Guitar" Books is 21 characters long

The Debugger has exited with status 0.
  
```

Косвенность в объектно-ориентированном программировании

Объектно-ориентированное программирование полно косвенности. Оно использует косвенность для обращения к данным, так же как в предыдущих примерах это делалось при помощи переменных, файлов и аргументов. Но действительно революция объектно-ориентированного программирования состоит в том, что оно использует косвенность при вызове кода. Функции не вызываются в объектно-ориентированном программировании непосредственно — это делается косвенно.

Теперь, когда вы знаете об этом факте, можете считать себя экспертом в объектно-ориентированном программировании. Все остальное — не более чем побочные эффекты этой косвенности.

Процедурное программирование

Чтобы завершить оценку гибкости объектно-ориентированного программирования, бросим беглый взгляд на процедурное программирование, чтобы вы получили представление о проблемах, которые призвано решать ООП. Процедурному программированию много, даже слишком много лет. При изучении программирования обычно начинают именно с него. Большинство программ на языках программирования типа BASIC, C, Tcl или Perl — процедурные.

В процедурных программах данные обычно хранятся в простых структурах, таких как элементы struct в языке программирования C. Имеются и более сложные структуры, такие как связанные списки и деревья. При вызове функции вы передаете в нее данные, а она их обрабатывает. Функции являются сердцем процедурного программирования: вы решаете, какие функции использовать, и вызываете их, передавая им необходимые данные.

Рассмотрим программу, выводящую на экран различные геометрические фигуры. Благодаря компьютеру вы можете не просто рассмотреть абстрактный пример, но и обратиться к конкретному исходному тексту программы в папке 03.08 Shapes-Procedural. Для простоты программа не рисует сами фигуры на экране, а выводит текст, соответствующий той или иной фигуре. Программа использует обычный C и стиль процедурного программирования. Исходный текст начинается с определения некоторых констант и структуры.

После обязательного включения заголовочных файлов идет перечисление различных видов фигур, которые могут выводиться программой: окружность, квадрат и нечто яйцеобразное.

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

```
typedef enum {
    kCircle,
    kRectangle,
    kOblateSpheroid
} ShapeType;
```

Затем очередное перечисление определяет цвета, используемые при выводе фигур.

```
typedef enum {
    kRedColor,
    kGreenColor,
    kBlueColor
} ShapeColor;
```

Затем приходит очередь структуры, описывающей прямоугольную область экрана, в которой выводится фигура.

```
typedef struct {
    int x, y, width, height;
} ShapeRect;
```

Наконец, у нас есть структура, которая объединяет все перечисленное для описания фигуры.

```
typedef struct {
    ShapeType type;
    ShapeColor fillColor;
    ShapeRect bounds;
} Shape;
```

После этого в нашем примере функция main() объявляет массив фигур, которые намеревается вывести. После объявления массива каждая фигура в нем инициализируется путем указания значений полей ее структуры. Приведенный далее код определяет красную окружность, зеленый прямоугольник и синий эллипс.

```
int main (int argc, const char * argv[])
{
    Shape shapes[3];
```

```

ShapeRect rect0 = { 0, 0, 10, 30 };
shapes[0].type = kCircle;
shapes[0].fillColor = kRedColor;
shapes[0].bounds = rect0;

ShapeRect rect1 = { 30, 40, 50, 60 };
shapes[1].type = kRectangle;
shapes[1].fillColor = kGreenColor;
shapes[1].bounds = rect1;

ShapeRect rect2 = { 15, 18, 37, 29 };
shapes[2].type = kOblateSpheroid;
shapes[2].fillColor = kBlueColor;
shapes[2].bounds = rect2;

drawShapes (shapes, 3);

return (0);
} // main

```

УДОБНОЕ СОКРАЩЕНИЕ

Прямоугольники в программе объявляются с применением небольшого трюка на языке программирования C: при объявлении переменной-структуры вы можете одновременно инициализировать все элементы структуры.

```
ShapeRect rect0 = { 0, 0, 10, 30 };
```

Элементы структуры получают значения в порядке объявления. Вспомним, что структура ShapeRect объявлена следующим образом.

```

typedef struct {
    int x, y, width, height;
} ShapeRect;

```

Приведенное выше присваивание переменной rect0 означает, что rect0.x и rect0.y получают значения 0; rect0.width становится равным 10, а rect0.height — 30.

Этот способ позволяет вам уменьшить количество нажатий клавиш при вводе программы без вреда для ее удобочитаемости.

После инициализации массива shapes функция main() вызывает функцию drawShapes() для вывода фигур.

Функция drawShapes() содержит цикл, обходящий все структуры Shape массива. Конструкция switch смотрит на значение поля type структуры и выбирает функцию, которая будет выводить данную фигуру. Программа вызывает соответствующую функцию для вывода фигуры, передавая ей область экрана и цвет.

```

void drawShapes (Shape shapes[], int count)
{
    int i;

```

```

for (i = 0; i < count; i++) {

    switch (shapes[i].type) {

    case kCircle:
        drawCircle (shapes[i].bounds,
                    shapes[i].fillColor);
        break;

    case kRectangle:
        drawRectangle (shapes[i].bounds,
                      shapes[i].fillColor);
        break;

    case kOblateSpheroid:
        drawEgg (shapes[i].bounds,
                shapes[i].fillColor);
        break;
    }
} // drawShapes

```

Вот как выглядит код функции drawCircle(), который просто выводит информацию об ограничивающем прямоугольнике и цвете, переданных ей.

```

void drawCircle(ShapeRect bounds,
                ShapeColor fillColor)
{
    NSLog (@ "Вывод окружности в (%d %d %d %d), цвет %@ ",
            bounds.x, bounds.y,
            bounds.width, bounds.height,
            colorName(fillColor));
} // drawCircle

```

Функция colorName(), вызываемая в NSLog(), просто возвращает литерал NSString, такой как @"красный" или @"синий", соответствующий переданному ей значению цвета.

```

NSString *colorName (ShapeColor colorName)
{
    switch (colorName) {

    case kRedColor:
        return @"красный";
        break;

    case kGreenColor:
        return @"зеленый";
        break;

    case kBlueColor:

```

```

    return @"синий";
    break;
}
return @"серобуромалиновый";
} // colorName

```

Остальные функции вывода фигур практически идентичны рассмотренной drawCircle(), за исключением того, что они выводят свои фигуры.

Вот вывод рассмотренной программы (с удаленной дополнительной информацией, выводимой функцией NSLog()).

```

Вывод окружности в (0 0 10 30), цвет красный
Вывод прямоугольника в (30 40 50 60), цвет зеленый
Вывод эллипса в (15 18 37 29), цвет синий

```

Все очень просто, правда? При использовании процедурного программирования вы тратите время на соединение данных и функций, созданных для работы с этими данными. Вы должны аккуратно использовать верную функцию для каждого типа данных: например, для фигуры kRectangle следует вызывать функцию drawRectangle(). Слишком уж легко передать прямоугольник функции, выводящей окружность.

Еще одна проблема при такого рода решении задачи заключается в том, что программу сложно расширять и поддерживать. Проиллюстрируем сказанное, добавив в программу еще одну фигуру — треугольник. Измененную таким образом программу вы найдете в папке 03.09 Shapes-Procedural-2. Мы должны внести изменения, как минимум, в четыре разных места.

Начнем с добавления константы kTriangle к перечислению ShapeType.

```

typedef enum {
    kCircle,
    kRectangle,
    kOblateSpheroid,
    kTriangle
} ShapeType;

```

Затем реализуем функцию drawTriangle(), имеющую тот же вид, что и прочие подобные функции.

```

void drawTriangle(ShapeRect bounds,
                  ShapeColor fillColor)
{
    NSLog(@"Вывод треугольника в (%d %d %d %d), цвет %@ ",
          bounds.x, bounds.y,
          bounds.width, bounds.height,
          colorName(fillColor));
} // drawTriangle

```

Затем добавляем новую часть case к конструкции switch в функции drawShapes(). В случае параметра kTriangle вызывается функция drawTriangle().

```

void drawShapes (Shape shapes[], int count)
{
    int i;

```

```

for (i = 0; i < count; i++) {
    switch (shapes[i].type) {
        case kCircle:
            drawCircle (shapes[i].bounds,
                        shapes[i].fillColor);
            break;
        case kRectangle:
            drawRectangle (shapes[i].bounds,
                           shapes[i].fillColor);
            break;
        case kOblateSpheroid:
            drawEgg (shapes[i].bounds,
                     shapes[i].fillColor);
            break;
        case kTriangle:
            drawTriangle (shapes[i].bounds,
                           shapes[i].fillColor);
            break;
    }
} // drawShapes

```

Наконец, добавляем треугольник в массив `shapes` (не забывая увеличить количество фигур в массиве `shapes`).

```
int main (int argc, const char * argv[])
```

```

{
    Shape shapes[4];

    ShapeRect rect0 = { 0, 0, 10, 30 };
    shapes[0].type = kCircle;
    shapes[0].fillColor = kRedColor;
    shapes[0].bounds = rect0;

    ShapeRect rect1 = { 30, 40, 50, 60 };
    shapes[1].type = kRectangle;
    shapes[1].fillColor = kGreenColor;
    shapes[1].bounds = rect1;

    ShapeRect rect2 = { 15, 18, 37, 29 };
    shapes[2].type = kOblateSpheroid;
    shapes[2].fillColor = kBlueColor;
    shapes[2].bounds = rect2;

    ShapeRect rect3 = { 47, 32, 80, 50 };

```

```

shapes[3].type = kTriangle;
shapes[3].fillColor = kRedColor;
shapes[3].bounds = rect3;

```

```

drawShapes (shapes, 4);

```

```

return (0);
} // main

```

Давайте рассмотрим программу в действии.

Вывод окружности в (0 0 10 30), цвет красный
 Вывод прямоугольника в (30 40 50 60), цвет зеленый
 Вывод эллипса в (15 18 37 29), цвет синий
 Вывод треугольника в (47 32 80 50), цвет красный

Добавление поддержки треугольников не очень сложное. Но учтите, что в нашей маленькой программе выполняется только одно действие — вывод фигур. Более сложную программу исправить гораздо сложнее. Допустим, программа выполняет более сложные действия — например, вычисляет площадь фигуры и выясняет, не находится ли указатель мыши в границах фигуры. В этом случае придется модифицировать каждую функцию, которая выполняет те или иные действия с фигурами, при этом внося изменения в отлаженный код и рискуя внести в него ошибки.

Вот еще один, не менее неприятный, сценарий: добавление новой фигуры, для которой требуется дополнительная описывающая информация. Например, прямоугольник с закругленными краями требует дополнительную информацию о радиусе закругления. Для этого надо добавить в структуру Shape поле радиуса, что приведет к излишнему расходу памяти (это поле не будет использоваться никакими другими фигурами). Кроме того, можно использовать объединение C, обеспечивающее перекрытие разных данных в одной и той же структуре, что только усложняет решение.

Объектно-ориентированное программирование элегантно решает поставленную задачу. Когда мы научим нашу программу использовать ООП, то увидим, как легко решается первая задача — модификация имеющегося кода для добавления новых фигур.

Реализация объектной ориентированности

Процедурные программы основаны на функциях, а данные, если можно так выразиться, вращаются вокруг функций. Объектная ориентированность меняет точку зрения, помещая данные в центр, а функции — на орбиту вокруг них. Вместо того чтобы фокусироваться на функциях, вы концентрируетесь на данных.

Это звучит интересно, но как это работает? В объектно-ориентированном программировании данные содержат ссылки на код, который работает с ними, используя концепцию косвенности. Вместо того чтобы говорить функции drawRectangle() "выведи прямоугольник с использованием переданной структуры данных", мы запрашиваем прямоугольник "выведи сам себя" (звучит грубо, но на самом деле это не так). Благодаря магии косвенности данные прямоугольника знают о том, как найти функцию, выполняющую его вывод.

Так что же такое объект? Это не более чем структура `struct` языка программирования C, которая в состоянии находить код, связанный с ней, обычно при помощи указателя на функцию. На рис. 3.1 показаны четыре объекта *Shape*: два квадрата, окружность и эллипс. Каждый объект способен найти функцию, которая выполняет его вывод.

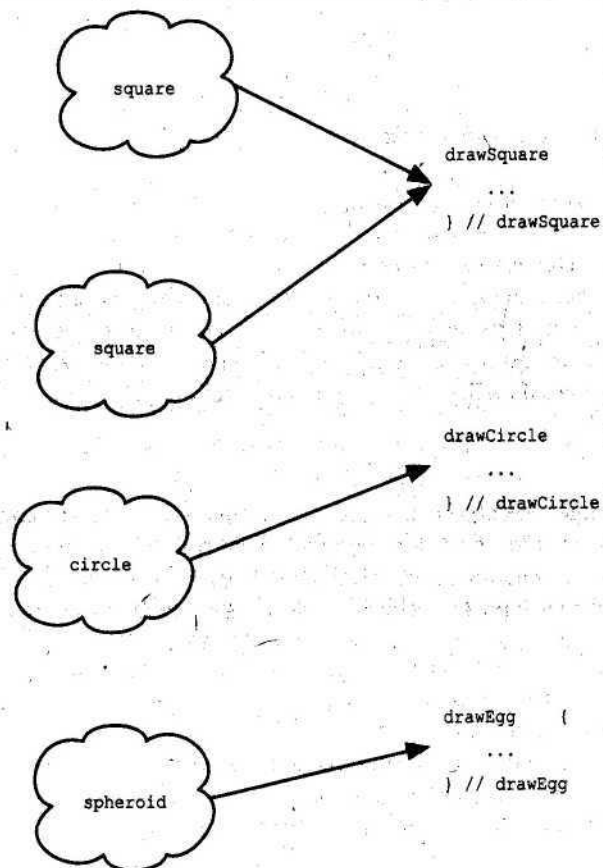


Рис. 3.1. Базовые объекты *Shape*

Каждый объект имеет собственную функцию `draw()`, которая знает, как выводить конкретную фигуру. Например, функция `draw()` объекта *Circle* знает, как вывести окружность. Функция `draw()` объекта *Rectangle* знает о том, как нарисовать четыре прямые линии, образующие прямоугольник.

Новая программа (03.10 *Shapes-Object*) выполняет те же действия, что и старая, но с применением объектно-ориентированных возможностей *Objective-C*. Вот как выглядит функция `drawShapes()` из новой программы.

```

void drawShapes (id shapes[], int count)
{
    int i;

```

```

for (i = 0; i < count; i++) {
    id shape = shapes[i];
    [shape draw];
}
} // drawShapes;

```

Эта функция содержит цикл, который просматривает каждую фигуру в массиве. В этом цикле программа просит каждую фигуру вывести саму себя.

Обратите внимание на разницу между этой версией функции drawShapes() и исходной. Как минимум, эта версия существенно короче! Коду не приходится спрашивать у каждой отдельной фигуры "а ты кто такой?".

Еще одно изменение заключается в shapes[], первом аргументе функции: теперь это массив объектов id. Что такое id? Это что, новая антикризисная политика правительства? Нет, это всего лишь обозначение для *идентификатора*. id — это обобщенный тип, используемый для ссылки на объект любого типа. Вспомним, что объект — это просто структура struct языка программирования C с некоторым прикрепленным кодом, так что id представляет собой указатель на одну из таких структур; в нашем случае структуры образуют различные типы фигур.

Следующее изменение в drawShapes() представляет собой тело цикла.

```

id shape = shapes[i];
[shape draw];

```

Первая строка выглядит как обычный код C. Код получает id — т.е. указатель на объект — из массива shapes и помещает его в переменную shape типа id. Это просто присваивание указателя; реального копирования всего содержимого фигуры не происходит. Взгляните на рис. 3.2, на котором показаны разные фигуры, доступные в программе. shapes[0] представляет собой указатель на красную окружность, shapes[1] — указатель на зеленый прямоугольник, а shapes[2] — на синий эллипс.

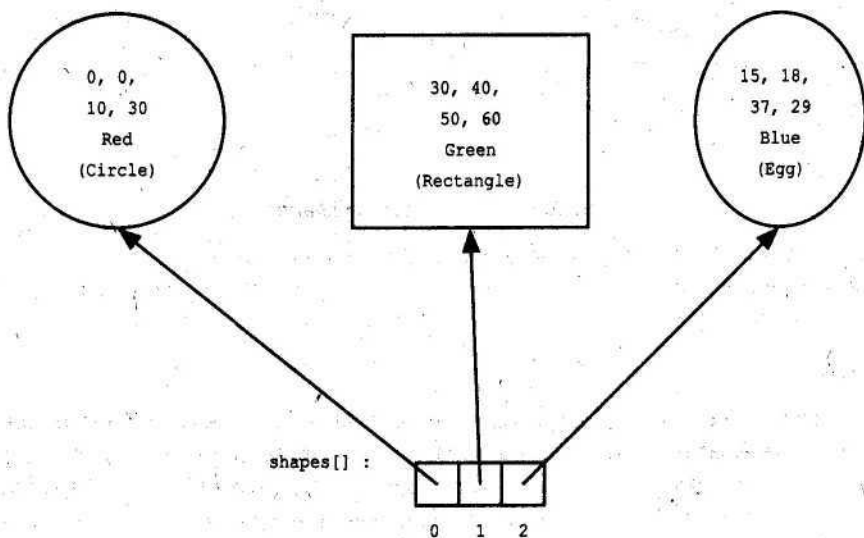


Рис. 3.2. Массив shapes

А вот последняя строка кода функции.

[shape draw];

Очень странная строка. Что бы это значило? Мы знаем, что квадратные скобки в С используются для обращения к элементам массива, но, похоже, здесь с массивом ничего не делается. В Objective-C квадратные скобки имеют дополнительный смысл: они используются для того, чтобы объяснить объекту, что он должен делать. Первый элемент в квадратных скобках — объект, а остальная часть представляет собой действие, которое должен выполнить объект. В данном случае мы говорим объекту по имени shape о необходимости выполнить действие draw. Если shape — окружность, будет выведена окружность. Если shape — прямоугольник, получим прямоугольник.

Когда Objective-C говорит объекту о необходимости выполнить некоторые действия, это называется *отправкой сообщения* (хотя многие говорят о “вызове метода”). Код [shape draw] отправляет сообщение draw объекту shape. (Таким образом, [shape draw] можно произносить как “отправить draw объекту shape.”) Как именно фигура выполняет вывод — зависит от реализации shape.

Каким образом вызывается необходимый код, когда вы посылаете сообщение объекту? Это происходит “за кулисами” с помощью классов.

Взгляните, пожалуйста, на рис. 3.3. В левой части рисунка показан объект circle, находящийся в нулевом элементе массива shapes, с которым мы последний раз встречались на рис. 3.2. Этот объект содержит указатель на свой класс, который представляет собой структуру, которая определяет вид объекта. На рис. 3.3 класс Circle содержит указатель на код, выводящий окружности, вычисляющий площадь круга, и прочий код, выполняющий все остальные действия, которые должен выполнять добропорядочный объект Circle.

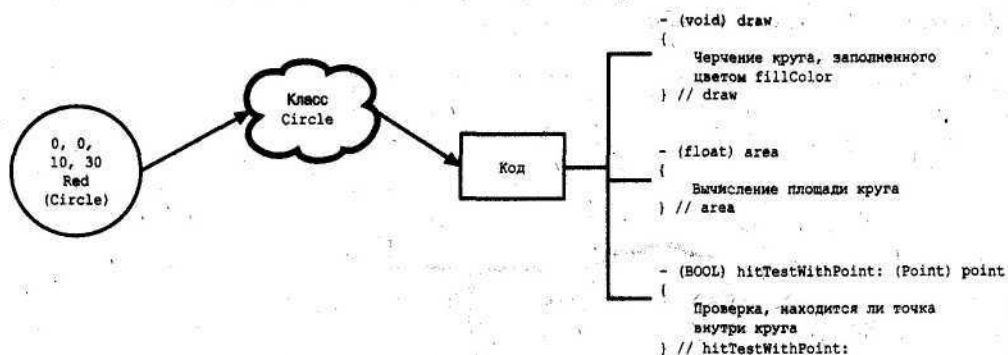


Рис. 3.3. Окружность и ее класс

Какова цель наличия класса объекта? Не проще ли, чтобы каждый объект просто непосредственно указывал на свой код? Да, это проще, и некоторые объектно-ориентированные системы именно так и поступают. Но наличие объектов класса имеет большое преимущество: если вы изменяете класс в процессе работы программы, все объекты класса автоматически получают это изменение (мы еще поговорим об этом в последующих главах).

На рис. 3.4 показано, как сообщение draw приводит к вызову корректной функции для объекта окружности.

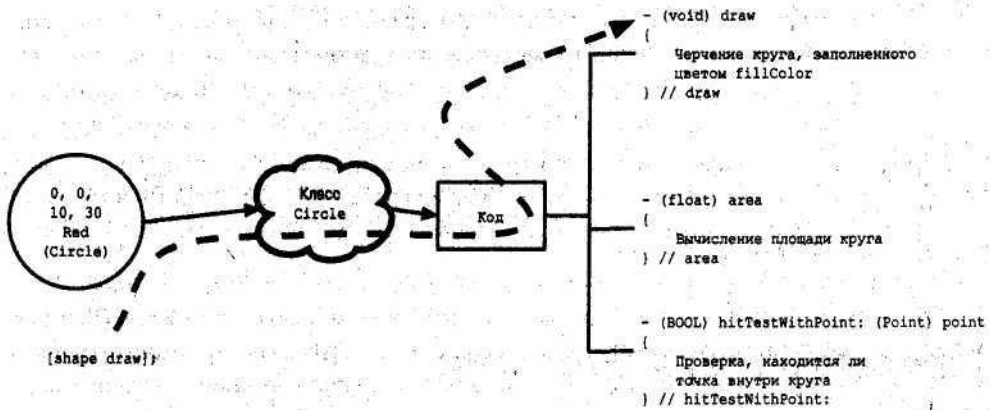


Рис. 3.4. Окружность ищет свой код вывода

Вот шаги, проиллюстрированные на рис. 3.4.

1. У целевого объекта сообщения (в данном случае — красной окружности) выясняется, к какому классу он относится.
2. Класс просматривается в поисках функции draw.
3. Будучи найденной, выполняется функция, выводящая окружности.

На рис. 3.5 показано, что происходит при вызове [shape draw] для второй фигуры массива (зеленого прямоугольника).

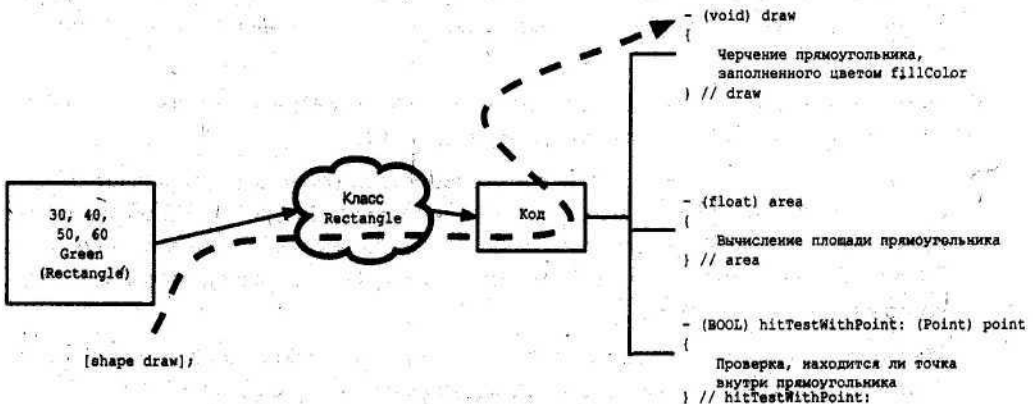


Рис. 3.5. Прямоугольник ищет свой код вывода

Шаги, показанные на рис. 3.5, практически идентичны шагам, проиллюстрированным на предыдущем рисунке.

1. У целевого объекта сообщения (в данном случае — зеленого прямоугольника) выясняется, к какому классу он относится.
2. Класс прямоугольника просматривается в поисках адреса функции draw.
3. Objective-C выполняет функцию, выводящую окружности.

Наша программа демонстрирует косвенность в действии! В процедурной версии программы мы должны были бы написать код, определяющий функцию, которую следует вызвать. Здесь же такое решение принимается “за кулисами” самим Objective-C, который сам опрашивает объекты на предмет принадлежности классам. Тем самым снижается вероятность вызова неверной функции, а сам код становится существенно проще с точки зрения поддержки.

Перерыв на терминологию

Перед тем как углубиться в дебри объектно-ориентированного программирования, давайте немного прервемся и познакомимся с объектно-ориентированными терминами. С некоторыми из них мы уже познакомились, другие же будут для нас совершенно новыми.

- **Класс** — структура, которая представляет тип объекта. Объект обращается к своему классу для получения информации о самом себе, в частности, о коде, который следует выполнить для осуществления тех или иных действий. В простых программах классы легко пересчитать по пальцам; в программах средней сложности счет идет на десятки. Стиль кодирования Objective-C поощряет использование имен классов, начинающихся с прописной буквы.
- **Объект** представляет собой структуру, содержащую значения и скрытый указатель на класс. Обычно в серьезных программах счет объектов идет на сотни и даже тысячи. Имена переменных Objective-C, хранящих объекты, обычно начинаются со строчной буквы.
- **Экземпляр** — еще одно слово для обозначения понятия “объект”. Например, объект circle можно также назвать экземпляром класса Circle.
- **Сообщение** — это действие, которое может выполнить объект. Это то, что вы посылаете объекту, чтобы побудить его сделать что-либо. В коде [shape draw] сообщение draw посылается объекту shape для того, чтобы заставить его вывести самого себя. Когда объект получает сообщение, происходит обращение к его классу, чтобы найти код, который следует выполнить.
- **Метод** — это код, который выполняется в ответ на сообщение. Сообщение, такое как draw, может вызывать различные методы, в зависимости от класса объекта.
- **Диспетчер методов** представляет собой механизм, используемый Objective-C для выяснения, какой метод должен быть выполнен в ответ на некоторое сообщение. Механизм диспетчеризации Objective-C будет более детально рассматриваться нами в следующей главе.

Это ключевые термины объектно-ориентированного программирования, которые будут нужны нам в дальнейшем. Кроме того, имеется пара терминов, связанных с обобщенным программированием, которые вскоре окажутся весьма важными.

- **Интерфейс** представляет собой описание возможностей, предоставляемых классом объектов. Например, интерфейс класса `Circle` объявляет, что его объекты могут получать сообщение `draw`.

ПРИМЕЧАНИЕ

Концепция интерфейсов не ограничивается объектно-ориентированным программированием. Например, заголовочные файлы в C предоставляют интерфейсы библиотек, таких как стандартная библиотека ввода-вывода (`#include <stdio.h>`) или математическая библиотека (`#include <math.h>`). Интерфейсы не раскрывают детали реализации, и основная идея в том, что они вовсе не должны вас волновать.

- **Реализация** представляет собой код, который выполняет работу интерфейса. В наших примерах реализация объекта окружности хранит код для вывода окружности на экран. Посылая сообщение `draw` объекту окружности, вы не должны знать о том, как работает эта функция. Все, что должно вас интересовать — это то, что данная функция выводит окружность на экран.

Объектно-ориентированное программирование в Objective-C

Если у вас начинает побаливать голова — это нормально. В нее набилось столько нового, что требуется время, чтобы все это в ней улеглось. Пока ваше подсознание “перемалывает” предыдущие разделы, давайте обратимся к оставшемуся не рассмотренным коду программы, включая новый синтаксис объявления классов.

Раздел @interface

Перед тем как создавать объекты определенного класса, компилятор Objective-C должен получить об этом классе определенную информацию. В частности, он должен знать о членах данных объекта (т.е. как выглядит структура C (`struct`) объекта) и о том, какие возможности он предоставляет. Для передачи всей необходимой информации компилятору используется директива `@interface`.

ПРИМЕЧАНИЕ

В рассматривавшейся программе `Shapes-Object` мы разместили все части программы в одном файле `Shapes-Object.m`. В больших программах обычно используется множество файлов, при этом каждый класс размещается в своем наборе файлов. Способы организации классов и файлов будут рассматриваться в главе 6, “Организация исходных файлов”.

Вот как выглядит интерфейс класса Circle.

```
@interface Circle : NSObject
{
    ShapeColor fillColor;
    ShapeRect bounds;
}

- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;
- (void) draw;
@end // Circle
```

Этот код использует пока неизвестный нам синтаксис, о котором мы и поговорим. В этом маленьком фрагменте скрыта большая информация. Давайте детально его разберем.

Первая строка имеет такой вид.

```
@interface Circle : NSObject
```

Как мы говорили в главе 2, "Расширения C", все в Objective-C, где используется символ @, представляет собой расширение языка программирования C. @interface Circle говорит компилятору "здесь начинается описание интерфейса нового класса с именем Circle".

ПРИМЕЧАНИЕ

NSObject в строке @interface говорит компилятору о том, что класс Circle базируется на классе NSObject. Это означает, что каждый Circle является также и NSObject и что каждый Circle наследует все поведение, определенное классом NSObject. Более детально наследование будет рассмотрено в следующей главе.

После начала объявления нового класса мы говорим компилятору о различных данных, необходимых объектам окружности.

```
{
    ShapeColor fillColor;
    ShapeRect bounds;
}
```

Все, что заключено в фигурные скобки, представляет собой шаблон, используемый для создания новых объектов Circle. Здесь сказано, что, когда создается новый объект Circle, он будет включать два элемента. Первый, fillColor (типа ShapeColor), представляет собой цвет, используемый при черчении окружности. Второй, bounds, представляет ограничивающий окружность прямоугольник. Его тип — ShapeRect. Этот прямоугольник указывает, где именно на экране должна быть выведена окружность.

Вы определяете fillColor и bounds в объявлении класса. Теперь всякий раз при создании объекта Circle он будет включать эти два элемента. Т.е. каждый объект класса Circle имеет собственную переменную fillColor и собственную переменную bounds. Значения fillColor и bounds называются *переменными экземпляра* объекта класса Circle.

Закрывающая фигурная скобка говорит компилятору, что мы завершили определение переменных экземпляра класса Circle.

Далее следуют несколько строк, выглядящих как прототипы функций в языке программирования C.

- (void) draw;
- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;

В Objective-C они называются *объявлениями методов*. Они очень похожи на прототипы функций в стиле старого доброго C и гласят "вот какие возможности поддерживает данный класс". Объявления метода указывают имя каждого метода, тип возвращаемого им значения и аргументы метода.

Начнем с простейшего объявления метода draw.

- (void) draw;

Ведущее тире указывает, что это объявление метода Objective-C. Это тире предназначено для того, чтобы отличить объявление метода от прототипа функции, который таким тире не предваряется. За тире следует тип возвращаемого значения метода, заключенный в круглые скобки. В нашем случае draw только выводит окружность на экран и ничего не возвращает. Для указания факта отсутствия возвращаемого значения Objective-C использует ключевое слово void.

Методы Objective-C могут возвращать те же типы, что и функции C: стандартные типы (int, float, char), указатели, ссылки на объекты или структуры.

Следующие объявления методов более интересны.

- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;

Каждый из этих методов получает единственный аргумент. setFillColor: в качестве аргумента получает цвет, который окружность использует для вывода самой себя. setBounds: получает прямоугольник, используемый для определения границ окружности.

ИНФИКСНАЯ ЗАПИСЬ

Objective-C использует синтаксис, именуемый *инфиксной записью*. Имя метода и его аргументы тесно переплетены. Например, вызвать метод с одним аргументом можно следующим образом.

```
[circle setFillColor: kRedColor];
```

Метод с двумя аргументами вызывается так.

```
[textThing setStringValue: @"hello there"
 color: kBlueColor];
```

setStringValue: и color: — это имена аргументов (которые в действительности являются частью имени метода, но об этом — немного позже), а @"hello there" и kBlueColor — это переданные аргументы.

Этот синтаксис отличается от синтаксиса C, в котором функция вызывается по имени, за которым следуют ее аргументы.

```
setTextThingValueColor (textThing, @"hello there",
 kBlueColor);
```

Хотя инфиксный синтаксис и выглядит на первый взгляд странновато, он нам нравится. Он делает код удобочитаемым; становится очень легко сопоставить аргументы с выполняемыми действиями. В исходных текстах C и C++ у функций не редкость наличие четырех или пяти аргументов, и разобраться, что именно представляет собой каждый из аргументов, без обращения к документации иногда достаточно сложно.

Объявление `setFillColor`: начинается с обычного ведущего типа и типа возвращаемого значения в скобках.

– (void)

Как и в случае метода `draw`, ведущее типе гласит “это объявление нового метода” (void) говорит о том, что этот метод не возвращает никакого значения. Продолжим наше рассмотрение.

`setFillColor`:

Имя метода — `setFillColor`.. Завершающее двоеточие представляет собой часть имени. Затем идет параметр.

(ShapeColor) `fillColor`;

Тип аргумента указан в скобках, и в данном случае это одно из наших значений `ShapeColor` (`kRedColor`, `kBlueColor` и т.д.). Следующее затем имя `fillColor` — это имя параметра. Это имя используется для обращения к параметру в теле метода. Исходный текст можно сделать более удобочитаемым, если выбирать значащие имена параметров вместо имен домашних животных или супергероев.

О ДВОЕТОЧИЯХ

Важно не забывать, что двоеточие представляет собой важную часть имени метода. Метод

– (void) `scratchTheCat`;

отличается от

– (void) `scratchTheCat`: (CatType) `critter`;

Распространенная ошибка многих начинающих программистов на Objective-C заключается в добавлении двоеточий в конце имени метода, не имеющего аргументов. Правило же таково: если у метода есть аргументы, то у него есть и двоеточие. Если аргументов нет — нет и двоеточия.

Объявление `setBounds`: в точности такое же, как и объявление `setFillColor`., за исключением типа аргумента, который представляет собой `ShapeRect`, а не `ShapeColor`.

Последняя строка сообщает компилятору об окончании объявления класса `Circle`.

@end // Circle

Хотя это и не обязательно, мы добавляем ко всем конструкциям `@end` комментарии с упоминанием имени класса. Это упрощает понимание, объявление какого класса вы читаете, в особенности когда вы находитесь вблизи конца файла, например.

Мы полностью рассмотрели интерфейс класса `Circle`. Теперь любой читающий код знает, что у данного класса две переменные экземпляра и три метода. Один метод определяет границы, второй — цвет, а третий выводит окружность на экран.

Теперь, когда мы разобрались с интерфейсом, самое время написать код, как именно работает данный класс. Вы же не думаете, что мы ограничимся одним только интерфейсом?

Раздел @implementation

Только что рассмотренный раздел @interface определяет открытый интерфейс класса. Интерфейс часто называют API (сокращение от "application programming interface", интерфейс прикладного программирования). Реальный код, который делает объекты работающими, находится в разделе @implementation.

Вот как выглядит раздел реализации класса Circle.

```
@implementation Circle
```

```
- (void) setFillColor: (ShapeColor) c
{
    fillColor = c;
} // setFillColor

- (void) setBounds: (ShapeRect) b
{
    bounds = b;
} // setBounds

- (void) draw
{
    NSLog (@\"Вывод окружности в (%d %d %d %d), цвет %@\",
            bounds.x, bounds.y,
            bounds.width, bounds.height,
            colorName(fillColor));
} // draw
@end // Circle
```

Давайте теперь детально рассмотрим этот код. Реализация класса Circle начинается со строки

```
@implementation Circle
```

@implementation представляет собой директиву компилятора, которая говорит о том, что дальше будет представлен код "внутренностей" класса. Имя класса указывается после ключевого слова @implementation. В этой строке нет завершающего двоеточия, как после любой директивы компилятора Objective-C.

Затем идут определения отдельных методов. Они не обязаны следовать в том же порядке, что и в разделе @interface. В разделе @implementation вы даже можете определить методы, которые не имеют соответствующего объявления в разделе @interface. Их можно рассматривать как закрытые методы, которые используются только в реализации класса.

ПРИМЕЧАНИЕ

Вы можете решить, что определение метода только в разделе @implementation делает его недоступным вне реализации, но это не так. Objective-C в действительности не имеет закрытых методов. Нет никакого способа объявить метод закрытым и защитить его от вызова другим кодом. Это побочный эффект динамической природы Objective-C.

Первым определенным методом является `setFillColor`:

```
-(void) setFillColor: (ShapeColor) c
{
    fillColor = c;
} // setFillColor
```

Первая строка определения `setFillColor`: очень похожа на объявление в разделе `@interface`. Основное отличие заключается в том, что в этом случае в конце нет точки с запятой. Кроме того, как вы можете заметить, параметр здесь переименован просто в `c`. Это вполне допустимо; имена параметров в разделах `@interface` и `@implementation` могут отличаться. В нашем случае, если бы мы оставили имя параметра `fillColor`, то оно бы скрыло имя переменной экземпляра `fillColor` и привело бы к генерации предупреждения компилятором.

ПРИМЕЧАНИЕ

Почему мы вынуждены переименовать `fillColor`? Потому что у нас уже есть переменная экземпляра с именем `fillColor`, определенная классом. Мы можем обратиться к ней в нашем методе — она находится в области видимости. Если мы определим другую переменную с тем же именем, компилятор перекроет нам доступ к переменной экземпляра. Использование одного и того же имени скрывает исходную переменную. Мы можем избежать проблемы, используя другое имя параметра. Можно также переименовать переменную экземпляра в, например, `myFillColor` и оставить имя `fillColor` за параметром. Как вы увидите позже, в главе 16, "Кодирование «ключ/значение»", Cocoa может "разрулить" ситуацию, когда мы именуем переменные экземпляра так же, как и методы.

В разделе `@interface` мы использовали имя `fillColor` в объявлении метода, поскольку оно объясняет читателю кода назначение данного аргумента. В реализации нам следует обеспечить отсутствие коллизий между именем параметра и переменной экземпляра, и простейший способ достичь этого — переименовать параметр.

Тело метода состоит из одной строки.

```
fillColor = c;
```

Если вы очень любознательны, то вас может заинтересовать вопрос, где хранятся переменные экземпляра. Когда вы вызываете метод в Objective-C, получающему объекту передается секретный скрытый параметр `self`, являющийся ссылкой на объект-получатель. Например, в коде `[circle setFillColor: kRedColor]` метод в качестве параметра `self` получает `circle`. Поскольку `self` передается скрытно и автоматически, вы никак не вмешиваетесь в этот процесс. Код в методе, обращающийся к переменной экземпляра, работает следующим образом: `self->fillColor = c`.

Кстати, передача скрытых аргументов — еще один пример косвенности в действии. Поскольку Objective-C во время работы программы может передавать различные объекты как скрытый параметр `self`, он может определять, переменные экземпляра какого именно объекта должны быть изменены.

Второй метод, `setBounds`, такой же, как и метод `setFillColor`:

```
-(void) setBounds: (ShapeRect) b
{
```



```

bounds = b;
} // setBounds

```

Приведенный код определяет параметры ограничивающего прямоугольника.

Последний метод — draw. Обратите внимание на отсутствие двоеточия в конце имени метода, что говорит о том, что метод не принимает никаких аргументов.

```

- (void) draw
{
    NSLog(@"Вывод окружности в (%d %d %d %d), цвет %@",
          bounds.x, bounds.y,
          bounds.width, bounds.height,
          colorName(fillColor));
} // draw

```

Метод draw использует скрытый параметр self для поиска значений переменных экземпляра, так же как это делали методы setFillColor: и setBounds:. Затем этот метод вызывает функцию NSLog() для вывода текста с описанием окружности.

Разделы @interface и @implementation для других классов (Rectangle и OblateSpheroid) практически аналогичны таковым для класса Circle.

Инстанцирование объектов

Теперь мы созрели для последней части программы, в которой создаются объекты фигур, такие как красные окружности и зеленые прямоугольники. В этом процессе главное слово — *инстанцирование*. При инстанцировании объекта для него выделяется память, после чего она инициализируется некоторыми значениями по умолчанию — т.е. чем-то отличным от простого мусора в памяти в момент ее выделения. Когда выделение памяти и ее инициализация выполнены, мы говорим, что создан новый экземпляр.

ПРИМЕЧАНИЕ

Поскольку локальные переменные объекта для каждого экземпляра объекта свои, мы называем их *переменными экземпляра*.

Для создания нового объекта мы посылаем сообщение new интересующему нас классу. Когда класс получает и обрабатывает сообщение new, мы в ответ получаем новый экземпляр объекта.

Одна из приятных возможностей Objective-C заключается в том, что вы можете рассматривать класс как объект и отправлять ему сообщения. Наилучший пример сообщений такого рода — создание нового объекта. Когда вам нужна новая окружность, вы запрашиваете новый объект у класса Circle, вместо того чтобы отправлять сообщение существующей окружности.

Вот как выглядит функция main(), создающая окружность, прямоугольник и эллипс.

```

int main (int argc, const char * argv[])
{
    id shapes[3];

```



```

ShapeRect rect0 = { 0, 0, 10, 30 };
shapes[0] = [Circle new];
[shapes[0] setBounds: rect0];
[shapes[0] setFillColor: kRedColor];

ShapeRect rect1 = { 30, 40, 50, 60 };
shapes[1] = [Rectangle new];
[shapes[1] setBounds: rect1];
[shapes[1] setFillColor: kGreenColor];

ShapeRect rect2 = { 15, 19, 37, 29 };
shapes[2] = [OblateSpheroid new];
[shapes[2] setBounds: rect2];
[shapes[2] setFillColor: kBlueColor];

drawShapes (shapes, 3);
return (0);
} // main

```

Как видите, функция `main()` объектно-ориентированной программы очень похожа на соответствующую функцию процедурной версии. Хотя, конечно, имеется и пара различий. Вместо массива фигур у нас появляется массив элементов `id` (которые, как вы, вероятно, помните, представляют собой не что иное, как указатели на объекты любого вида). Отдельные объекты создаются путем отправления сообщений `new` классу объекта, который вы хотите создать.

```

...
shapes[0] = [Circle new];
...
shapes[1] = [Rectangle new];
...
shapes[2] = [OblateSpheroid new];
...

```

Еще одно отличие заключается в том, что в процедурной версии инициализация объектов выполнялась непосредственно, присвоением членов структуры. Объектно-ориентированная программа не работает с объектом непосредственно; она использует сообщения для того, чтобы каждый объект установил ограничивающий прямоугольник и цвет.

```

...
[shapes[0] setBounds: rect0];
[shapes[0] setFillColor: kRedColor];
...
[shapes[1] setBounds: rect1];
[shapes[1] setFillColor: kGreenColor];
...
[shapes[2] setBounds: rect2];
[shapes[2] setFillColor: kBlueColor];
...

```

После инициализации фигуры выводятся при помощи рассмотренной ранее функции `drawShapes()`.

```
drawShapes (shapes, 3);
```

Расширение программы

Помните, как мы добавляли в процедурную программу треугольники? Давайте сделаем то же самое и с объектно-ориентированной версией. Соответствующий проект находится в папке 03.11 Shapes-Object-2.

Нам пришлось повозиться, чтобы обучить процедурную программу работе с треугольниками: отредактировать перечисление `ShapeType`, добавить функцию `drawTriangle()`, добавить треугольник в список фигур и модифицировать функцию `drawShapes()`. В основном, это была скучная работа, в особенности в части модификации функции `drawShapes()`, где нам пришлось вносить изменения в цикл, который управлял выводом всех фигур, и где была наибольшая вероятность внести ошибку.

В объектно-ориентированной программе мы делаем следующее: создаем новый класс `Triangle`, а затем добавляем объект `Triangle` в список выводимых фигур.

Вот класс `Triangle`, который в точности совпадает с классом `Circle`, — только "Circle" в нем заменено на "Triangle".

```
@interface Triangle : NSObject
```

```
{
    ShapeColor fillColor;
    ShapeRect bounds;
}
```

```
-(void) setFillColor: (ShapeColor) fillColor;
```

```
-(void) setBounds: (ShapeRect) bounds;
```

```
-(void) draw;
```

```
@end // Triangle
```

```
@implementation Triangle
```

```
-(void) setFillColor: (ShapeColor) c
```

```
{
    fillColor = c;
} // setFillColor
```

```
-(void) setBounds: (ShapeRect) b
```

```
{
    bounds = b;
} // setBounds
```

```
-(void) draw
```

```
{
    NSLog(@"Вывод треугольника в (%d %d %d %d), цвет %@",
          bounds.x, bounds.y,
```

```

        bounds.width, bounds.height,
        colorName(fillColor));
} // draw
@end // Triangle

```

ПРИМЕЧАНИЕ

Недостатком программирования в стиле "копировать и вставить", как в случае класса `Triangle`, является большое количество дублирующегося кода, как, например, методы `setBounds:` и `setFillColor:`. Мы познакомим вас с наследованием в следующей главе, и вы узнаете, как избежать такой избыточности.

Далее нам надо отредактировать функцию `main()`, чтобы в ней создавался новый треугольник. Начнем с изменения размера массива `shapes` с 3 до 4, чтобы в нем было достаточно места для размещения нового объекта.

```
id shapes[4];
```

Затем добавим блок кода, создающего новый объект типа `Triangle`, точно так же как создается новый `Rectangle` или `Circle`.

```

ShapeRect rect3 = { 47, 32, 80, 50 };
shapes[3] = [Triangle new];
[shapes[3] setBounds: rect3];
[shapes[3] setFillColor: kRedColor];

```

И наконец, обновим вызов `drawShapes()`, передавая ему новый размер массива `shapes`.

```
drawShapes (shapes, 4);
```

Это все. Теперь наша программа понимает, что такое треугольники.

Вывод окружности в (0 0 10 30), цвет красный

Вывод прямоугольника в (30 40 50 60), цвет зеленый

Вывод эллипса в (15 19 37 29), цвет синий

Вывод треугольника в (47 32 80 50), цвет красный

Обратите внимание, что мы добавили новую функциональность в функцию `drawShapes()`, никак не трогая ни ее, ни какую-либо другую функцию для работы с фигурами. Это — объектно-ориентированное программирование в действии.

ПРИМЕЧАНИЕ

Код этой программы демонстрирует пример принципа открытости-закрытости гурзу объектно-ориентированного программирования Бертранда Мейера (Bertrand Meyer), который гласит, что объекты программ должны быть открыты для расширений, но закрыты для изменений. Функция `drawShapes()` открыта для расширений: мы просто добавили новый вид фигуры в массив для вывода. Функция `drawShapes()` при этом закрыта для изменений: мы расширили ее, не модифицируя ее код. Программы, придерживающиеся принципа открытости-закрытости, оказываются более надежными и стабильными перед лицом изменений, поскольку при этом не приходится трогать корректно работающий код.

Резюме

Это была большая глава, переполненная концепциями и идеями. Мы говорили в ней о мощной концепции косвенности и показали, что вы уже использовали эту концепцию в своих программах, — например, при работе с переменными и файлами. Затем мы рассмотрели процедурное программирование и показали ограниченность взгляда “функции первичны, данные вторичны”.

Мы познакомились с объектно-ориентированным программированием, которое использует косвенность для тесного связывания данных с кодом, который с ними работает. Оно позволяет программировать в стиле “данные первичны, функции вторичны”. Мы обсудили сообщения, которые посылаются объектам. Объекты обрабатывают эти сообщения путем выполнения методов, фрагментов кода, которые заставляют объект “петь и танцевать”. Кроме того, вы узнали, что каждый вызов метода включает скрытый параметр `self`, который представляет собой сам объект. Методы работают с данными с применением этого параметра `self`. Реализация методов и шаблона данных объектов определяется в классе объекта. Вы создаете новый объект путем отправления классу сообщения `new`.

В следующей главе вас ждет знакомство с наследованием, возможностью, которая позволяет вам использовать поведение уже имеющихся объектов и уменьшить количество кода, который приходится писать. Так что оставайтесь с нами и дальше!

Наследование

При написании реальной объектно-ориентированной программы — а мы надеемся, что вы собираетесь написать их много — классы и объекты в ней оказываются связаны друг с другом. Они сотрудничают, обеспечивая тем самым работоспособность программы.

При рассмотрении соотношений между классами и объектами наиболее важными оказываются два аспекта объектно-ориентированного программирования. Первый из них — *наследование*, являющееся темой данной главы. При создании нового класса зачастую полезно определить новый класс в терминах отличий от уже имеющегося. Используя наследование, вы можете определить класс, который обладает всеми возможностями родительского класса: он их просто *наследует*.

Второй аспект ООП — *композиция*, когда объекты содержат ссылки на другие объекты. Например, объект автомобиля в имитационной программе может содержать четыре объекта колеса. Когда ваш объект содержит ссылки на другие объекты, он может воспользоваться их функциональностью. В этом и заключается идея композиции, которая будет рассматриваться в следующей главе.

Зачем нужно наследование

Помните программу Shapes-Object из предыдущей главы? В ней имеется несколько классов с весьма схожими интерфейсами и реализацией. Они настолько схожи, что мы использовали при их написании метод "скопировать и вставить".

Давайте освежим память, взглянув на интерфейсы классов Circle и Rectangle.

```
@interface Circle : NSObject
{
    ShapeColor fillColor;
    ShapeRect bounds;
}
```

```

- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;
- (void) draw;
@end // Circle

```

```

@interface Rectangle : NSObject
{
    ShapeColor fillColor;
    ShapeRect bounds;
}

```

```

- (void) setFillColor: (ShapeColor) fillColor;
- (void) setBounds: (ShapeRect) bounds;
- (void) draw;
@end // Rectangle

```

Интерфейсы этих классов очень, *очень, очень* схожи. За исключением имени класса, они просто совпадают!

Реализации классов Circle и Rectangle также очень похожи. Вспомните из предыдущей главы, что в двух классах setFillColor: и setBounds: идентичны.

```

@implementation Circle
- (void) setFillColor: (ShapeColor) c
{
    fillColor = c;
} // setFillColor

- (void) setBounds: (ShapeRect) b
{
    bounds = b;
} // setBounds

// ...
@end // Circle

```

```

@implementation Rectangle
- (void) setFillColor: (ShapeColor) c
{
    fillColor = c;
} // setFillColor

- (void) setBounds: (ShapeRect) b
{
    bounds = b;
} // setBounds

// ...
@end // Rectangle

```


Эти методы выполняют одну и ту же работу — устанавливают значения переменных экземпляра `fillColor` и `bounds`. Однако реализации `Circle` и `Rectangle` не идентичны. Например, сигнатуры методов `draw` (т.е. имя метода и его параметры) одинаковы в обоих классах, но реализации методов различны.

```
@implementation Circle
// ...

- (void) draw
{
    NSLog(@"Вывод окружности в (%d %d %d %d), цвет %@",
          bounds.x, bounds.y,
          bounds.width, bounds.height,
          colorName(fillColor));
} // draw
@end // Circle

@implementation Rectangle
// ...

- (void) draw
{
    NSLog(@"Вывод прямоугольника в (%d %d %d %d), цвет %@",
          bounds.x, bounds.y,
          bounds.width, bounds.height,
          colorName(fillColor));
} // draw
@end // Rectangle
```

В программе `Shapes-Object` дублируется большое количество кода в классах `Circle` и `Rectangle`. На рис. 4.1 показана диаграмма этих классов.



Рис. 4.1. Архитектура без наследования

ПРИМЕЧАНИЕ

На рис. 4.1 имена классов приведены в верхней части каждого прямоугольника. В средней части указаны переменные экземпляра, а в нижней — методы класса. Такой тип диаграмм определен унифицированным языком моделирования (Unified Modeling Language, UML), который широко применяется для графического представления классов, их содержимого и связей между ними.

На рис. 4.1 имеется масса дублирующегося кода, и это, как минимум, приведет к снижению эффективности. Чем больше дублирующегося кода в программе, тем хуже была продумана ее архитектура. Как минимум, вам приходится поддерживать в два раза больше кода, искать и вносить изменения в два раза большее количество мест — все это существенно повышает шансы на появление новых ошибок. Если вы забудете внести изменения в одно из мест, ошибка может проявляться достаточно странно и быть весьма трудно обнаружимой.

Было бы неплохо, если бы было можно собрать весь дублирующийся код в одном месте. Еще лучше было бы, если бы мы могли поддерживать пользовательские методы там, где это нам надо, — например, методы для вывода окружностей и прямоугольников. Словом, нам нужна система, позволяющая сказать компилятору что-то вроде “класс Circle такой же, как и другие, с парочкой особенностей здесь и здесь”. Наверное, вы уже поняли, что все это достигается при помощи наследования.

На рис. 4.2 показано, как выглядит архитектура после того, как мы добавим в нее немного наследования. Мы создали новый класс Shape для хранения общих переменных экземпляра и объявлений методов. Класс Shape содержит реализацию методов setFillColor: и setBounds:. Класс Shape содержит реализацию методов setFillColor: и setBounds:.

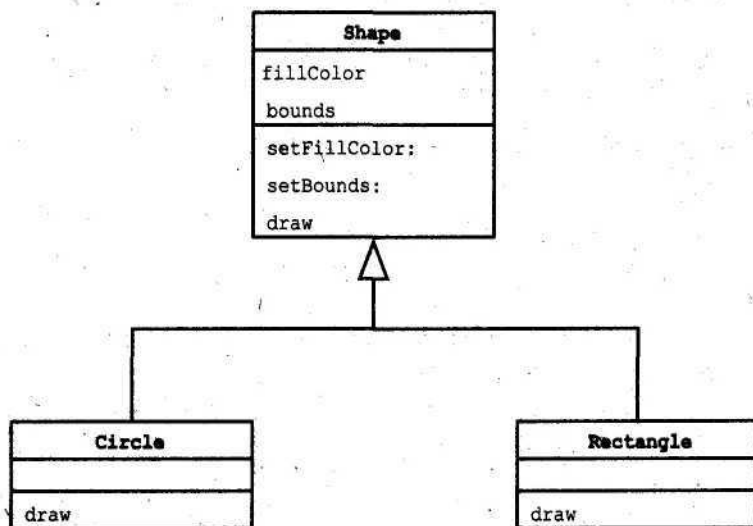


Рис. 4.2. Архитектура с применением наследования

Взгляните на новые классы Circle и Rectangle на рис. 4.2. Они существенно меньше, чем были до сих пор. Все их общие элементы вынесены в класс Shape. Единственное, что осталось в классах Circle и Rectangle, — это элементы, которые делают их уникальными, в частности, метод draw. Теперь мы говорим, что классы Circle и Rectangle *наследуют* класс Shape.

ПРИМЕЧАНИЕ

Линия со стрелкой на конце (см. рис. 4.2) представляет собой способ указания отношения наследования в UML. Эта линия указывает отношения наследования между классами Circle и Shape и классами Rectangle и Shape.

Так же как вы могли унаследовать от родителей такие особенности, как цвет волос, форма носа или ушей, или склонность к программированию, так и наследование в объектно-ориентированном программировании означает, что класс получает возможности от другого класса, его родителя, или *суперкласса*. Благодаря наследованию классов Circle и Rectangle от класса Shape, они получают две переменные экземпляра от класса Shape.

ПРИМЕЧАНИЕ

Непосредственное изменение значений переменных экземпляров считается дурным тоном. Используйте для их изменения методы.

В дополнение к переменным экземпляра, наследование предоставляет методы. Каждый класс Circle и Rectangle знает, как отвечать на сообщения setFillColor: и setBounds:. Они унаследовали эту возможность от класса Shape.

Синтаксис наследования

Давайте посмотрим на синтаксис, используемый при объявлении нового класса.

```
@interface Circle : NSObject
```

Идентификатор, следующий за двоеточием, представляет собой наследуемый класс. В Objective-C вы можете создавать классы, не унаследованные ни от чего, но при использовании Сосоа вы должны наследовать свои классы от NSObject, обладающего массой полезных возможностей (которые вы получаете и в том случае, когда наследуете класс, наследовавший NSObject). О многих возможностях NSObject вы узнаете, когда мы будем рассматривать управление памятью в главе 9, "Управление памятью".

ОДНОКРАТНОЕ НАСЛЕДОВАНИЕ

В некоторых языках, таких как C++, имеется возможность *множественного наследования*, когда класс может быть непосредственным наследником двух или более классов. Objective-C не поддерживает множественное наследование. Если вы попытаетесь использовать множественное наследование в Objective-C, которое показано далее, компьютер будет весьма недоволен.

```
@interface Circle : NSObject, PrintableObject
```

Многие преимущества множественного наследования можно получить при помощи других возможностей Objective-C, таких как категории (глава 12, "Категории") и протоколы (глава 13, "Протоколы").

Теперь, когда вы знаете о том, что такое наследование, и когда мы исправили архитектуру так, что теперь наши классы наследуют класс Shape, интерфейсы Circle и Rectangle выглядят так, как показано в следующем далее листинге (см. папку 04.01 Shapes-Inheritance):

```
@interface Circle : Shape
```

```
@end // Circle
```

```
@interface Rectangle : Shape
```

```
@end // Rectangle
```

Вы можете представить еще более простой код? Когда код прост, ошибкам просто негде спрятаться.

Обратите внимание на то, что мы не объявляем переменные экземпляра: мы получаем их из класса Shape как часть наследования. Мы не включаем фигурные скобки для отсутствующих переменных экземпляра: если переменных экземпляра нет, скобки можно опустить. Мы также не объявляем методы, которые получаем от класса Shape (setBounds: и setFillColor:).

Давайте теперь посмотрим на код класса Shape. Вот объявление класса.

```
@interface Shape : NSObject
```

```
{
```

```
    ShapeColor fillColor;
```

```
    ShapeRect bounds;
```

```
}
```

```
- (void) setFillColor: (ShapeColor) fillColor;
```

```
- (void) setBounds: (ShapeRect) bounds;
```

```
- (void) draw;
```

```
@end // Shape
```

Как видите, в классе Shape собрано в один пакет все, что ранее дублировалось в разных классах.

Реализация Shape проста и бесхитростна.

```
@implementation Shape
```

```
- (void) setFillColor: (ShapeColor) c
```

```
{
```

```
    fillColor = c;
```

```
} // setFillColor
```

```
- (void) setBounds: (ShapeRect) b
```

```
{
```

```
    bounds = b;
```

```
} // setBounds
```

```
- (void) draw
```

```
{
```

```
} // draw
```

```
@end // Shape
```

Хотя метод draw ничего не делает, мы все равно определяем его, так что все подклассы класса Shape могут реализовывать свои версии. Определение метода может иметь пустое тело или возвращать некоторое фиктивное значение.

Рассмотрим теперь реализацию класса Circle. Теперь она становится существенно проще.

```
@implementation Circle
```

```
- (void) draw
```

```
{
    NSLog(@"Вывод окружности в (%d %d %d %d), цвет %@",
          bounds.x, bounds.y,
          bounds.width, bounds.height,
          colorName(fillColor));
} // draw
@end // Circle
```

А вот как выглядит реализация класса Rectangle.

```
@implementation Rectangle
- (void) draw
{
    NSLog(@"Вывод прямоугольника в (%d %d %d %d), цвет %@",
          bounds.x, bounds.y,
          bounds.width, bounds.height,
          colorName(fillColor));
} // draw
@end // Rectangle
```

Классы Triangle и OblateSpheroid аналогично невелики. Подробнее познакомиться с программой можно в папке 04.01 Shapes-Inheritance.

Теперь вы можете запустить программу и убедиться, что она работает в точности так же, как и раньше. Обратите внимание на этот приятный факт: мы никоим образом не вмешивались в код функции main(), который создает и использует объекты. Дело в том, что мы никак не изменяли поведение объектов.

ПРИМЕЧАНИЕ

Перемещение и упрощение кода, которое мы выполнили только что, носит общее название **рефакторинг** — хорошо известное понятие в сообществе объектно-ориентированных программистов. При рефакторинге вы перемещаете код так, чтобы усовершенствовать архитектуру (в нашем случае — устранить дублирование) без изменения поведения программы и результатов ее работы. Типичный цикл разработки включает добавление некоторых возможностей в код с последующим рефакторингом для устранения дублирования.

Вас может удивить, что объектно-ориентированные программы могут стать проще после *добавления* новых возможностей, но такое случается, и примером тому — происшедшее с нашей программой при добавлении класса Shape.

Терминология

Разве бывает новая технология без новой терминологии? Вот несколько новых терминов, с которыми вам следует познакомиться.

- **Надкласс, суперкласс** — класс, от которого вы наследуете. Надклассом для Circle является класс Shape, а надклассом для Shape — класс NSObject.
- **Родительский класс** — еще одно название надкласса. Так, класс Shape является родительским по отношению к Rectangle.

- **Подкласс** — это наследующий класс. Класс Circle является подклассом класса Shape, а Shape — подклассом NSObject.
- **Дочерний класс** — еще одно название подкласса. Класс Circle является дочерним по отношению к классу Shape. Использовать ли термины “родительский/дочерний” или “надкласс/подкласс” — дело вкуса.
- Унаследованный метод **перекрывается** (override), если вы хотите изменить его реализацию. Класс Circle имеет свой собственный метод draw, так что мы говорим, что он перекрывает метод draw. Objective-C обеспечивает вызов реализации метода соответствующего класса при работе программы.

Как работает наследование

Мы выполнили крупную хирургическую операцию, перенесли код из Circle и Rectangle в Shape. Очень приятно, что остальная программа продолжает нормально работать без внесения каких-либо изменений. Создание и инициализация различных фигур в функции main() остались неизменными, функция drawShapes() осталась той же, и программа продолжает нормально работать:

-
- Вывод окружности в (0 0 10 30), цвет красный
 - Вывод прямоугольника в (30 40 50 60), цвет зеленый
 - Вывод эллипса в (15 19 37 29), цвет синий
 - Вывод треугольника в (47 32 80 50), цвет красный
-

Здесь вы видите еще один аспект мощи объектно-ориентированного программирования: можно внести радикальные изменения в программу, и если при этом быть осторожным, то по окончании работы все будет работать как ни в чем ни бывало. Конечно, тот же фокус можно провернуть и в процедурном программировании, но шансы на успех окажутся ниже.

Диспетчеризация методов

Как объекты узнают, какой метод им следует вызывать при получении сообщения? Например, код setFillColor: был перенесен из классов Circle и Rectangle. Каким же образом код класса Shape узнает, что вы послали сообщение setFillColor: объекту Circle? Вот в чем секрет: когда код отправляет сообщение, диспетчер методов Objective-C ищет метод в текущем классе. Если диспетчер не находит такой метод в классе объекта, получившего сообщение, он обращается к его надклассу.

На рис. 4.3 показана диспетчеризация в случае, когда код отправляет сообщение setFillColor: объекту Circle в старой версии программы, без класса Shape. Для обработки кода наподобие [shape setFillColor: kRedColor] диспетчер методов Objective-C просматривает объект, получающий сообщение, — в данном случае это объект класса Circle. Объект содержит указатель на свой класс, а класс — указатель на код. Диспетчер использует эти указатели для поиска кода, который должен быть выполнен.

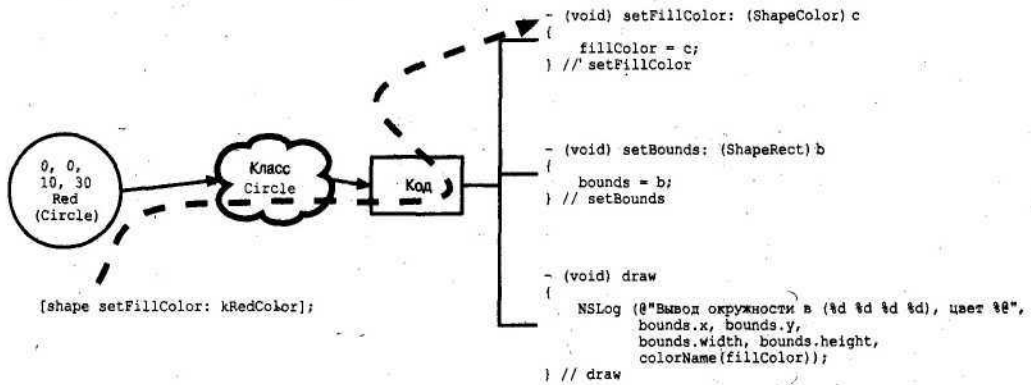


Рис. 4.3. Диспетчеризация без наследования

Теперь посмотрите на рис. 4.4, на котором показана новая структура с применением наследования. В этом коде класс Circle имеет ссылку на свой надкласс Shape. Диспетчер методов Objective-C использует эту информацию для того, чтобы найти корректную реализацию метода при получении сообщения.

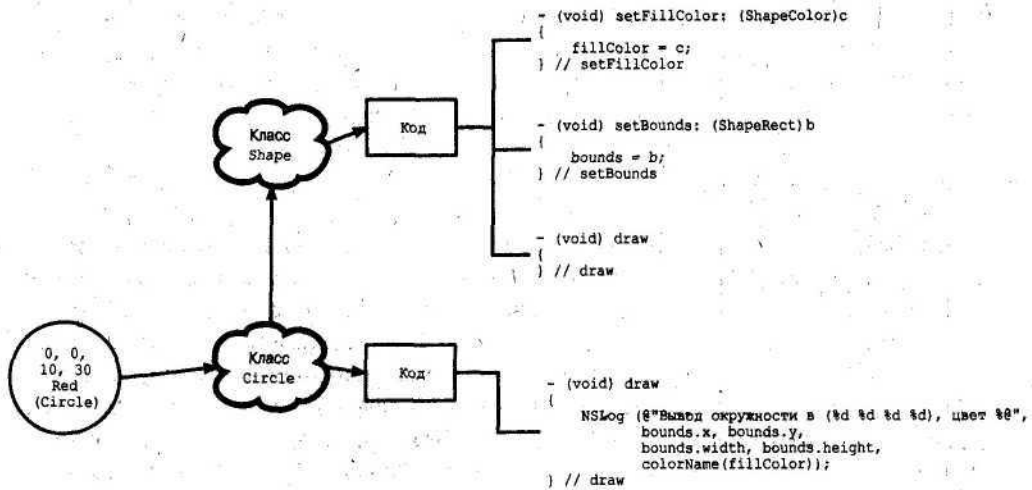


Рис. 4.4. Наследование и код класса

На рис. 4.5 показан процесс диспетчеризации метода при применении наследования. Когда вы отправляете сообщение setFillColor: объекту класса Circle, диспетчер сначала консультируется с классом Circle и выясняет, в состоянии ли он ответить на сообщение setFillColor: своим собственным кодом. В данном случае ответ отрицательный: диспетчер обнаруживает, что класс Circle не имеет определения для setFillColor:, так что следует обратиться к надклассу Shape. Затем диспетчер просматривает класс Shape, обнаруживает объявление setFillColor: и выполняет этот код.

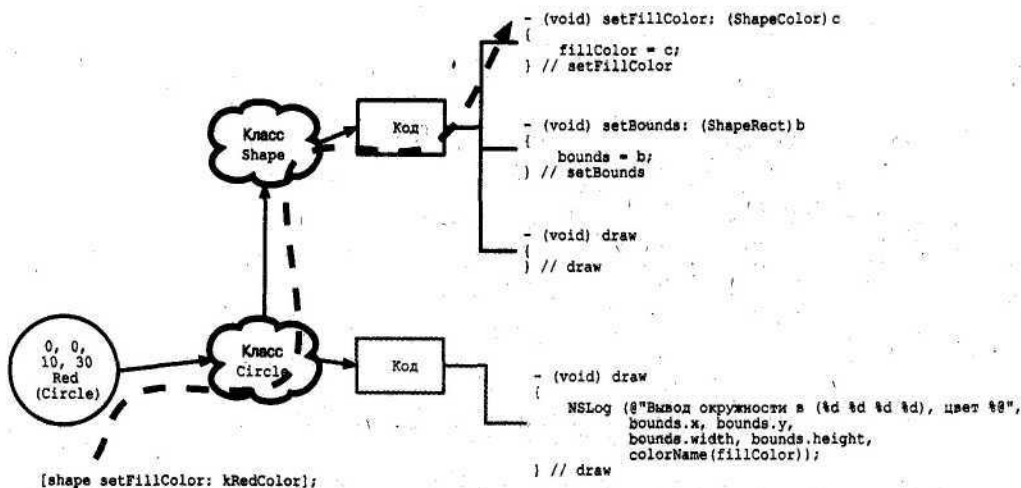


Рис. 4.5. Диспетчеризация метода при наследовании

Диспетчер действует по принципу "если я не могу найти нужное в классе, я поищу в надклассе" и при необходимости применяет этот принцип в каждом классе цепочки наследования. Если метод не будет обнаружен ни в классе Circle, ни в классе Shape, то диспетчер обратится к классу NSObject, поскольку это очередной надкласс в цепочке. Если метод не будет найден и в NSObject, выше которого в цепочке нет никого, вы получите ошибку времени выполнения (а кроме того, вы должны были получить предупреждение времени компиляции).

Переменные экземпляра

Итак, мы рассмотрели, как в ответ на сообщения вызываются соответствующие методы. Теперь давайте рассмотрим, как Objective-C обращается к переменным экземпляра. Как метод draw класса Circle находит переменные экземпляра bounds и fillColor, объявленные в классе Shape?

При создании нового класса его объекты наследуют переменные экземпляра от надклассов, после чего (необязательно) добавляют собственные переменные экземпляра. Чтобы увидеть, как работает наследование переменных экземпляра, давайте разработаем новую фигуру и добавим новую переменную экземпляра. Этому новому классу, RoundedRectangle, требуется переменная для хранения радиуса для рисования закругленных углов прямоугольника. Определение класса выглядит примерно так.

```

@interface RoundedRectangle : Shape
{
    int radius;
}
@end // RoundedRectangle

```

На рис. 4.6 показана схема размещения в памяти объекта класса RoundedRectangle. Класс NSObject объявляет одну переменную экземпляра isa, в которой хранится указатель на класс объекта. Затем идут две переменные экземпляра, объявленные в классе Shape: fillColor и bounds. И наконец, это переменная экземпляра radius, объявленная в классе RoundedRectangle.

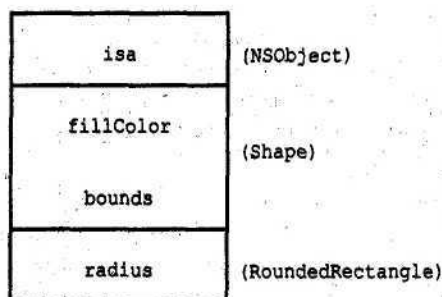


Рис. 4.6. Схема размещения переменных экземпляра

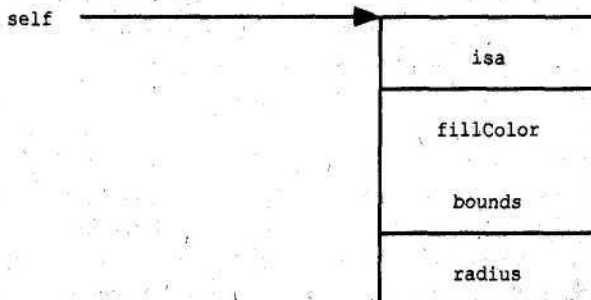
ПРИМЕЧАНИЕ

Переменная экземпляра NSObject носит имя *isa*, поскольку она фактически определяет отношение "is a" (является) между подклассом и надклассом, т.е. и Rectangle является Shape, и Circle является Shape. Код, который использует Shape, может не менее успешно использовать Rectangle или Circle.

Способность использовать более конкретный вид объекта (Rectangle или Circle) вместо обобщенного типа (Shape) называется *полиморфизмом* (греческое слово, означающее "много форм", достаточно хорошо описывает происходящее).

Вспомните, что каждому методу при вызове передается скрытый параметр *self*, который представляет собой указатель на объект, получающий сообщение. Методы используют этот параметр для поиска переменных экземпляра, с которыми они должны работать.

На рис. 4.7 показан параметр *self*, указывающий на объект класса RoundedRectangle. Параметр *self* указывает на первую переменную экземпляра первого класса в цепочке наследования. В случае RoundedRectangle цепочка наследования начинается с NSObject, затем продолжается классом Shape и заканчивается классом RoundedRectangle, так что *self* указывает на *isa*, первую переменную экземпляра. Компилятор Objective-C знает расположение переменных экземпляра благодаря объявлениям *@interface* каждого из этих классов. Обладая этой важной информацией, компилятор может генерировать код для поиска любой переменной экземпляра.

Рис. 4.7. Параметр *self*

ОСТОРОЖНО, СТЕКЛО!

При работе компилятор использует механизм “база плюс смещение”. Имея базовый адрес объекта — т.е. место расположения в памяти первого байта первой переменной экземпляра — компилятор может найти все остальные переменные экземпляра, добавляя к данному базовому адресу определенные смещения.

Например, если базовый адрес объекта класса `RoundedRectangle` равен `0x1000`, то переменная экземпляра `isa` располагается по адресу `0x1000+0`, который равен просто `0x1000`. Переменная `isa` представляет собой 4-байтовое значение, так что следующая переменная экземпляра, `fillColor`, начинается со смещением, равным 4, т.е. находится по адресу `0x1000+4=0x1004`. Каждая переменная экземпляра имеет свое смещение относительно базового адреса.

Когда вы обращаетесь в методе к переменной экземпляра `fillColor`, компилятор генерирует код, который получает значение из переменной `self` и добавляет к нему смещение (в нашем случае — 4) для получения указателя на место в памяти, где хранится значение переменной.

Это приводит к определенным проблемам. Такие смещения жестко закодированы компилятором в программе. Даже если программисты Apple захотят добавить еще одну переменную экземпляра в `NSObject`, они не смогут этого сделать, поскольку это внесло бы изменения во все смещения переменных экземпляра. Такая проблема называется *проблемой хрупкости базового класса*. Эта проблема решена в новой 64-битовой системе времени выполнения Objective-C, где для определения местоположения переменных экземпляра используется косвенность.

Перекрытие методов

Когда вы создаете собственные подклассы, зачастую добавляете к ним свои собственные методы. Иногда это новый метод, который добавляет совершенно новые возможности в класс, иногда вы заменяете или усовершенствуете существующий метод, определенный в одном из надклассов.

Например, вы можете начать с класса Cocoa `NSTableView`, который выводит прокручиваемый список, и добавить к нему новое поведение, например воспроизведение содержимого списка при помощи речевого синтезатора. Вы можете добавить новый метод `speakRows`, который передает содержимое таблицы синтезатору.

Вместо добавления совершенно новой возможности, вы можете создать подкласс, который подправляет существующее поведение, унаследованное от одного из надклассов. В программе `Shapes-Inheritance` класс `Shape` уже выполняет большую часть работы, которую мы требуем от фигуры, — устанавливает границы фигуры и ее цвет, но `Shape` не знает о том, как что-либо выводить. И не должен! `Shape` — это обобщенный, абстрактный класс, а каждая фигура выводится по-разному. Так что когда вы хотите создать класс `Circle`, наследуете класс `Shape` и пишете метод `draw`, который знает, как должна выводиться окружность.

Когда мы создавали `Shape`, то знали, что все его подклассы должны будут осуществлять вывод на экран, хотя и не знали точно, как именно будут реализованы эти варианты вывода. Поэтому мы дали классу `Shape` метод `draw`, но сделали его пустым, так что каждый подкласс может наполнить его своим содержанием. Когда классы типа `Circle` или `Rectangle` реализуют собственные методы `draw`, мы говорим, что они *перекрывают* метод `draw`.

Когда сообщение draw отправляется объекту типа Circle, диспетчер методов выполняет перекрытый метод — реализацию draw класса Circle. Любая реализация draw, определенная в надклассе, таком как Shape, полностью игнорируется. В нашем случае это хорошо — поскольку реализация этого метода в классе Shape не содержит никакого кода. Но бывает и так, что вы не хотите, чтобы версия метода из надкласса была проигнорирована. Чтобы узнать, что делать в этом случае, — читайте дальше!

Это просто супер

Objective-C предоставляет возможность как перекрытия метода, так и вызова реализации этого метода надклассом — что особенно полезно, когда требуется, чтобы надкласс выполнил свою долю работы до или после работы текущего класса. Для вызова реализации унаследованного метода в качестве целевого объекта вызова метода используется super.

Предположим, например, что мы узнали, что имеются страны, в которых власти ненавидят окружности красного цвета, так что для беспрепятственного распространения нашей программы мы должны выводить не красную окружность, а желтую или, например, синюю. Один из способов решения проблемы — модифицировать класс Circle так, чтобы все окружности выводились синим цветом. Почему бы нам не изменить для этого цвет в классе Circle непосредственно? Мы вполне можем это сделать. Но так везет не всегда — может оказаться, что у вас просто нет кода класса, который вы хотите модифицировать.

Вспомним, что метод setFillColor: определен в классе Shape. Таким образом, мы можем решить проблему, перекрывая setFillColor: в классе Circle. Мы смотрим на параметр цвета, и если он красный — меняем его на синий. Затем мы используем ключевое слово super, чтобы объяснить надклассу (Shape), что он должен сохранить измененный цвет в переменной экземпляра fillColor (полностью листинг можно найти в папке 04.02 Shapes-Green-Circles).

Раздел @interface класса Circle остается неизменным, так как мы не добавляем ни новые методы, ни переменные экземпляра. Нам надо только добавить код в раздел @implementation.

```
@implementation Circle
- (void) setFillColor: (ShapeColor) c
{
    if (c == kRedColor) {
        c = kBlueColor;
    }

    [super setFillColor: c];
} // setFillColor
```

```
// Остальная часть раздела @implementation
// остается неизменной
@end // Circle
```

В этой новой реализации setFillColor: мы проверяем параметр ShapeColor — не красный ли это цвет? Если да, то изменяем его на синий. Затем при помощи кода [super setFillColor: c] требуем от надкласса выполнить свою работу по сохранению цвета в переменной экземпляра.

Что такое `super`? Это не параметр и не переменная экземпляра, это — волшебное слово компилятора Objective-C. Когда вы посылаете сообщение для `super`, просите Objective-C отправить это сообщение надклассу текущего класса. Если соответствующий метод не определен, Objective-C продолжает обычный просмотр цепочки наследования.

На рис. 4.8 показан поток выполнения метода `setFillColor:` класса `Circle`. Объект отправляет сообщение `setFillColor:`. Диспетчер находит пользовательскую версию `setFillColor:`, реализованную классом `Circle`.

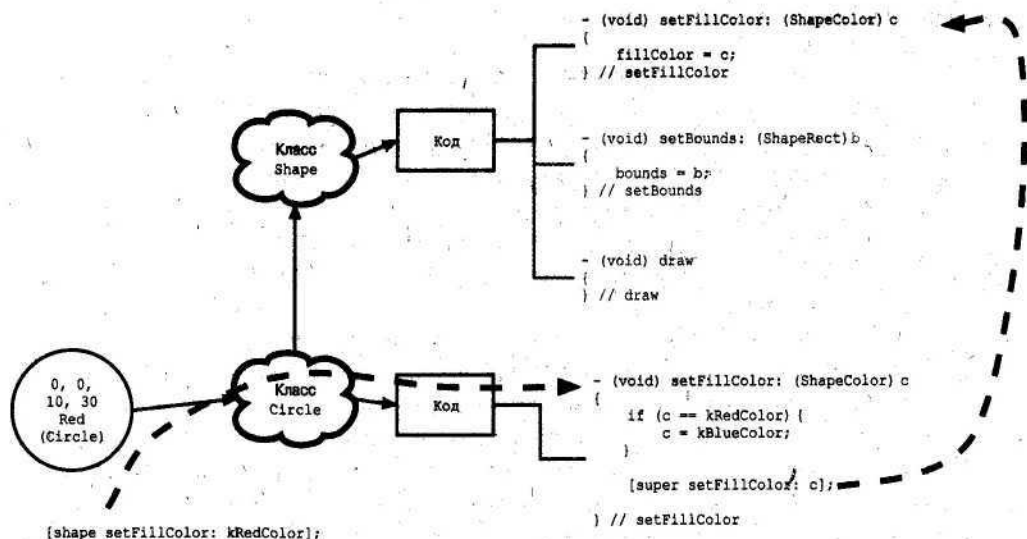


Рис. 4.8. Вызов метода надкласса

После того как версия `setFillColor:` класса `Circle` выполнила проверку на равенство параметра `kRedColor` и при необходимости изменила цвет, при помощи конструкции `[super setFillColor: c]` вызывается метод надкласса. Ключевое слово `super` приводит к тому, что вызывается метод `setFillColor:` класса `Shape`.

ПРИМЕЧАНИЕ

При перекрытии метода вызов метода надкласса почти всегда — хорошая идея, особенно если он выполняет работу, о которой вы не осведомлены. В нашем случае у нас есть доступ к исходному тексту класса `Shape`, так что мы знаем, что все, что делает класс `Shape` в методе `setFillColor:`, — это помещение параметра в переменную экземпляра. Но если бы у нас не было исходных текстов класса `Shape`, мы бы не могли уверенно утверждать, что этот метод не делает больше ничего. И даже если мы знаем, что он ничего не делает, — этот класс может быть изменен или усовершенствован в будущем. Вызывая унаследованный метод, мы гарантируем получение всех реализуемых им возможностей.

Резюме

Наследование является жизненно важной концепцией объектно-ориентированного программирования и используется многими другими технологиями ООП. В этой главе мы познакомились с наследованием и рассмотрели его применение для упрощения (и украшения) нашей конкретной программы. Мы выяснили, как можно создавать новые классы на базе существующих и как переменные экземпляра надкласса появляются в подклассах.

Мы познакомились с механизмом диспетчеризации методов в Objective-C и выяснили, что при этом осуществляется проход вверх по цепочке наследований, пока не будет найден ответ на конкретное сообщение. Наконец, мы узнали о существовании ключевого слова `super` и о том, как оно позволяет воспользоваться кодом надкласса в перекрытом методе.

Следующая глава посвящена композиции — другому способу сотрудничества объектов. Возможно, композиция не так впечатляюща, как наследование, но очень важна, так что не откладывайте книгу в сторону!

Композиция

В предыдущей главе вы познакомились с наследованием, отношением между двумя классами, которое позволяет существенно снизить дублирование кода. Мы также кратко упомянули, что имеется и другое соотношение — композиция, которая и представляет собой тему данной главы. Композицию можно использовать для совместной работы объектов. В типичной программе при создании пользовательских классов используются как наследование, так и композиция, так что следует хорошо разобраться с обеими концепциями.

Что такое композиция

Композиция в программировании подобна композиции в музыке: вы соединяете вместе отдельные компоненты, и в результате вместе они образуют нечто большее, чем порознь. В музыке при создании симфонии вы можете объединить партию фагота и партию гобоя. В программе вы можете скомбинировать объект педали и объект колеса и получить виртуальный моноцикл (однокольный велосипед).

В Objective-C такая комбинация создается при помощи включения указателей на объекты в виде переменных экземпляра. Так, наш моноцикл должен содержать указатель на объект Pedal и указатель на объект Tire и выглядеть примерно так.

```
@interface Unicycle : NSObject
{
    Pedal *pedal;
    Tire *tire;
}
@end // Unicycle
```

При композиции Unicycle состоит из Pedal и Tire.

ПРИМЕЧАНИЕ

Вы уже видели композицию в программе Shapes-Object: класс Shape использует прямоугольники (struct) и цвета (enum). Строго говоря, термин "композиция" относится только к объектам. Более примитивные типы наподобие int, float, enum или struct рассматриваются как части объекта.

Модель автомобиля

Давайте пока что отложим программу с фигурами и обратимся к моделированию автомобиля. В нашей упрощенной модели автомобиль имеет двигатель и четыре колеса. Вместо сложного физического моделирования настоящих колес и двигателя, мы используем пару классов, которые имеют единственный метод, выводящий сообщение о том, какую часть он представляет: колесо будет говорить о себе, что это колесо, а двигатель — что это двигатель. В реальной программе колеса имели бы такие атрибуты, как, например, давление в камере, а двигатель — мощность и расход горючего. Код этой программы можно найти в папке 05.01 CarParts.

Подобно программе для работы с фигурами, новая программа также хранит весь исходный текст в единственном файле `mainCarParts.m`. Программа начинается с импорта заголовочного файла схемы Foundation.

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

Далее следует класс `Tire`; он включает единственный метод `description`.

```
@interface Tire : NSObject
@end // Tire
```

```
@implementation Tire
- (NSString *) description
{
    return (@"Я — колесо.");
} // description
@end // Tire
```

ПРИМЕЧАНИЕ

В определении класса без переменных экземпляра фигурные скобки можно опустить.

Единственным методом класса `Tire` является `description`, и он не объявлен в интерфейсе. Откуда же он берется? Как пользователь может узнать о возможности использования метода `description` класса `Tire`, если таковой не включен в интерфейс? При помощи небольшой магии Cocoa.

Настройка для NSLog()

Вспомните, что функция `NSLog()` позволяет вам использовать спецификатор формата `%@` для вывода объектов. Когда функция `NSLog()` обрабатывает спецификатор `%@`, она запрашивает описание соответствующего объекта в списке параметров. Говоря технически, `NSLog()` отправляет сообщение `description` объекту, а метод `description` объекта строит строку `NSString` и возвращает ее. Функция `NSLog()` включает эту строку в свой вывод. Предоставляя метод `description` в вашем классе, вы можете настроить, каким образом ваш объект будет выводиться функцией `NSLog()`.

В методе `description` можно вернуть литерал `NSString`, такой как `@"Я — колесо"`, или создать строку, которая описывает различную информацию об объекте, такую как давление в камере или цена покрышки. Метод `description` класса `Cocoa NSArray` (который предназначен для работы с коллекциями объектов) предоставляет информацию о самом массиве, такую

как количество содержащихся в нем объектов и описание каждого из этих объектов. Такие описания, естественно, получаются путем отправления сообщения `description` каждому из содержащихся в массиве объектов.

Вернемся к нашей программе и рассмотрим класс `Engine`. Как и класс `Tire`, он имеет единственный метод `description`. В реальной программе ваш двигатель имел бы массу других методов, таких как `start` и `accelerate`, а также переменные экземпляра наподобие `RPM`. Но нам нужен простейший пример для демонстрации только одного — работы композиции, так что двигатель у нас не способен на большее, чем просто сообщить о своем существовании.

```
@interface Engine : NSObject
@end // Engine
```

```
@implementation Engine
- (NSString *) description
{
    return (@"Я — двигатель. P-p-p-p!");
} // description
@end // Engine
```

Последняя часть — сам автомобиль, у которого есть двигатель `engine` и C-массив с четырьмя колесами `tires`. Автомобиль использует композицию для сборки всего этого в единое целое. Класс `Car` имеет также метод `print`, который использует функцию `NSLog()` для вывода информации о колесах и двигателе.

```
@interface Car : NSObject
{
    Engine *engine;
    Tire *tires[4];
}

- (void) print;
@end // Car
```

Переменные `engine` и `tires` являются композицией, потому что `tires` и `engine` — переменные экземпляра класса `Car`. Можно сказать, что `Car` *составлен* из четырех колес и двигателя. Конечно, обычно так никто не говорит; вы можете сказать, что у автомобиля есть четыре колеса и двигатель.

Каждый объект автомобиля выделяет память для *указателей* на двигатель и колеса. Двигатель и колеса не встроены в автомобиль, это просто ссылки на другие объекты, располагающиеся где-то в памяти. Когда создается новый автомобиль, эти указатели инициализируются значениями `nil` (нулевое значение), указывающими, что у автомобиля нет ни двигателя, ни колес. Вы можете сами их дорисовать.

Давайте посмотрим на реализацию класса `Car`. Начнем с метода `init`, который инициализирует переменные экземпляра. Метод `init` создает двигатель и четыре колеса для оснащения автомобиля. Когда вы создаете новый объект с применением сообщения `new`, за кулисами в действительности выполняются два действия. Сначала происходит выделение памяти для объекта, т.е. для хранения его переменных экземпляра. Затем автоматически вызывается метод `init`, который приводит объект в работоспособное состояние.

```
@implementation Car
- (id) init
{
    if (self = [super init]) {
        engine = [Engine new];
        tires[0] = [Tire new];
        tires[1] = [Tire new];
        tires[2] = [Tire new];
        tires[3] = [Tire new];
    }
    return (self);
} // init
```

Метод `init` класса `Car` создает четыре новых колеса и присваивает их элементам массива `tires`. Затем метод `init` создает новый двигатель и присваивает его переменной экземпляра `engine`.

Следующим идет метод `print`.

```
- (void) print
{
    NSLog(@"%@", engine);
    NSLog(@"%@", tires[0]);
    NSLog(@"%@", tires[1]);
    NSLog(@"%@", tires[2]);
    NSLog(@"%@", tires[3]);
} // print
@end // Car
```

ЕСЛИ ЧТО?..

Эта строка кода в методе `init` выглядит немного странно:

```
if (self = [super init]) {
```

Что тут происходит? Вызов `[super init]` нужен потому, что надкласс (в данном случае — `NSObject`) тоже может выполнять некоторую инициализацию. Метод `init` возвращает значение (типа `id` — обобщенный указатель на объект), представляющее объект, который был инициализирован.

Присваивание результата `[super init]` переменной `self` представляет собой стандартное соглашение Objective-C. Мы делаем это в случае, когда надкласс, в качестве части работы по инициализации, возвращает другой объект, не тот, который был создан изначально. Этот вопрос мы рассмотрим более подробно попозже, при более детальном изучении метода `init`, так что сейчас просто кивните и отправимся дальше.

Метод `print` использует функцию `NSLog()` для вывода переменных экземпляра. Помните, что спецификатор `%@",` вызывает метод `description` каждого объекта и выводит результат. В реальной программе, конечно, вы бы использовали двигатель и колеса, чтобы отправить автомобиль по дороге.

Последняя часть `CarParts.m` представляет собой функцию `main()`, водителя нашей программы. Функция `main()` создает новый автомобиль, просит его рассказать о самом себе и завершает работу.

```
int main (int argc, const char * argv[])
{
    Car *car;

    car = [Car new];
    [car print];

    return (0);
} // main
```

Если мы соберем и запустим программу, то должны увидеть что-то похожее на это:

```
Я - двигатель. Р-р-р-р-р!
Я - колесо.
Я - колесо.
Я - колесо.
Я - колесо.
```

Вряд ли мы выиграем автомобильную гонку, но программа все же работает!

Методы доступа

Программисты редко довольны своими программами, так что работа над программами никогда не завершается. Всегда есть еще одна ошибка, которую надо исправить, еще одна функциональность, которую надо добавить, еще один способ сделать программу быстрее. Так что не удивительно, что наша программа не идеальна. Мы можем ее усовершенствовать и сделать код более гибким, воспользовавшись методами доступа. Код этой новой версии можно найти в папке 05.02 CarParts-Accessors.

Опытный программист, взглянув на метод `init` класса `Car`, может спросить: "Почему автомобиль создает свои собственные колеса и двигатель?" Программа будет существенно лучше, если вы сможете настраивать автомобиль для использования различных типов колес (например, с шипованной резиной для зимы) или двигателей (карбюраторных или дизельных).

Было бы хорошо, если бы мы могли указывать автомобилю, какие конкретно колеса или двигатель использовать. Тогда пользователи могли бы создавать собственные машины, соединяя в них разные колеса и двигатели.

Это можно сделать путем добавления методов доступа. Метод *доступа* (accessor) — это метод, предназначенный для чтения или изменения определенных атрибутов объекта. Например, метод `setFillColor` в программе `Shapes-Object` является методом доступа. Если мы добавим новый метод для изменения двигателя в объект `Car`, то это также будет методом доступа. Такой тип метода доступа называется устанавливающим (setter), так как он устанавливает значение объекта. Вы могли также слышать применяемый для обозначения таких методов, которые изменяют состояние объекта, термин мутатор (mutator).

Вероятно, вы уже осведомлены о другом типе методов доступа — получателях (getter). Метод-получатель предоставляет коду, использующему объект, возможность обращения к его атрибутам. В гоночной игре физическая модель будет обращаться к атрибутам колес и двигателя автомобиля для просчета поведения автомобиля на дороге.

ПРИМЕЧАНИЕ

При работе с атрибутами других объектов вы всегда должны использовать методы доступа и никогда не лезть внутрь объекта и пытаться изменить значения непосредственно. Например, функция `main()` не должна непосредственно обращаться к переменной экземпляра `engine` класса `Car` (при помощи `car->engine`), чтобы изменять ее. Вместо этого для внесения изменений ваш код должен использовать устанавливающий метод.

Методы доступа представляют собой еще один пример косвенности в действии. Косвенное обращение посредством методов доступа обеспечивает высокую гибкость реализации класса `Car`.

Давайте добавим методы доступа к классу `Car`, чтобы код, использующий этот класс, мог управлять типами используемых колес и двигателя. Вот новый интерфейс класса `Car`; его новые элементы выделены полужирным шрифтом.

```
@interface Car : NSObject
```

```
{
    Engine *engine;
    Tire *tires[4];
}
```

```
- (Engine *) engine;
- (void) setEngine: (Engine *) newEngine;
- (Tire *) tireAtIndex: (int) index;
- (void) setTire: (Tire *) tire
    atIndex: (int) index;

- (void) print;
@end // Car
```

Множество переменных экземпляра осталось неизменным, но добавились два новых метода: `engine` и `setEngine`: (для работы с атрибутами двигателя) и `tireAtIndex`: и `setTire:atIndex`: (для работы с колесами). Методы доступа почти всегда идут в паре: один для установки значения, а второй — для чтения. Иногда имеет смысл наличие одного только метода получения (для атрибутов только для чтения, таких как, например, размер файла на диске) или только метода установки (например, для установки секретного пароля), но чаще всего вам придется писать оба варианта метода доступа.

У Сосоа имеется соглашение по именованию методов доступа. Когда вы пишете методы доступа для ваших собственных классов, должны следовать этим соглашениям, чтобы и вы, и другие пользователи вашего кода не были сбиты с толку.

Методы установки именуются так же, как и сами атрибуты, с префиксом `set`. Вот примеры имен методов установки: `setEngine`:, `setStringValue`:, `setFont`:, `setFillColor`: или `setTextLineHeight`..

Методы получения получают просто имена атрибутов, значения которых они возвращают. Методы получения, соответствующие приведенным ранее методам установки, будут иметь имена `engine`, `stringValue`, `font`, `fillColor` и `textLineHeight`. Слово `get` в имени метода не требуется. Например, методы с именами `getStringValue` и `getFont` нарушают указанное соглашение. В некоторых языках программирования, таких как Java, имеются иные соглашения, которые используют префикс `get`. Но если вы пишете код Сосоа, забудьте об этом префиксе.

ПРИМЕЧАНИЕ

Слово "get" в Сосоа имеет особый смысл: в имени метода Сосоа оно означает, что метод возвращает значение посредством указателя, переданного в качестве параметра. Например, `NSData` (класс Сосоа для объектов, в которых хранится произвольная последовательность байтов) имеет метод `getBytes:`, который получает параметр, представляющий собой адрес буфера памяти, в котором хранятся данные. Класс `NSBezierPath` (используемый при черчении) имеет метод `getLineDash:count:phase:`, который получает указатель на массив чисел с плавающей точкой для пунктирной линии, указатель на целое число — для количества элементов в шаблоне и указатель на число с плавающей точкой — для определения места начала вывода.

Если вы используете в именах методов доступа префикс "get", то опытный программист Сосоа, использующий ваш код, будет считать, что этому методу следует передать в качестве аргументов указатели, и будет сбив с толку, обнаружив, что это простой метод доступа. С программистами лучше так не поступать.

Установка двигателя

Первая пара методов доступа работает с двигателем.

```
-(Engine *) engine;
-(void) setEngine: (Engine *) newEngine;
```

Код, который использует объекты `Car`, вызывает метод `engine` для доступа к двигателю и `setEngine:` — для его изменений. Вот как выглядит реализация этих методов.

```
-(Engine *) engine
{
    return (engine);
} // engine

-(void) setEngine: (Engine *) newEngine
{
    engine = newEngine;
} // setEngine
```

Метод получения `engine` возвращает текущее значение переменной экземпляра `engine`. Вспомните, что все взаимодействие объектов в Objective-C осуществляется посредством указателей, так что метод `engine` возвращает указатель на объект двигателя, содержащийся в `Car`.

Аналогично метод установки `setEngine:` устанавливает значение переменной экземпляра — указателя `engine`. Сам объект двигателя не копируется; копируется только значение указателя на него. Вот еще один способ сказать то же самое: после вызова `setEngine:` имеется только один двигатель, а не два.

ПРИМЕЧАНИЕ

Проведем "сеанс с разоблачением" и упомянем о наличии пары проблем, связанных с рассмотренными методами доступа к двигателю. Эти проблемы связаны с управлением памятью и владением объектами. Сейчас, однако, поднятие этой темы способно лишь сбить вас с толку, так что отложим рассмотрение способов написания абсолютно корректных методов доступа до главы 8, "Краткий обзор Foundation Kit".

Код использования методов доступа выглядит примерно так.

```
Engine *engine = [Engine new];
[car setEngine: engine];
NSLog (@\"Двигатель автомобиля %@\", [car engine]);
```

Настройка колес

Методы доступа у колес немного сложнее.

```
– (void) setTire: (Tire *) tire
    atIndex: (int) index;

– (Tire *) tireAtIndex: (int) index;
```

Поскольку у автомобиля несколько колес, объекты Car содержат для них массивы. Вместо того чтобы показывать всему миру массив tires, используется индексированный метод доступа. При установке колеса в автомобиль вы говорите не только, какого вида колесо установить, но и в какое место. Аналогично при обращении к колесу мы запрашиваем не просто колесо, но колесо, находящееся в определенном месте, а не колесо вообще..

Вот реализация методов доступа к колесам.

```
– (void) setTire: (Tire *) tire
    atIndex: (int) index
{
    if (index < 0 || index > 3) {
        NSLog (@\"Неверный индекс (%d) в setTire:atIndex:\",
            index);
        exit (1);
    }
    tires[index] = tire;
} // setTire:atIndex:

– (Tire *) tireAtIndex: (int) index
{
    if (index < 0 || index > 3) {
        NSLog (@\"Неверный индекс (%d) в tireAtIndex:\",
            index);
        exit (1);
    }
    return (tires[index]);
} // tireAtIndex:
```

Методы доступа колес содержат некоторый общий код, который проверяет корректность индекса массива для переменной экземпляра tires. Если индекс выходит за пределы диапазона от 0 до 3, программа выводит соответствующее сообщение и завершается. Этот код известен под названием *защитное программирование* и является хорошей практикой. Защитное программирование перехватывает ошибки, такие как использование неверного значения индекса, на ранних стадиях цикла разработки программы.

Мы должны проверить корректность индекса массива, поскольку `tires` представляет собой массив в стиле C и компилятор не выполняет никаких проверок при обращении к элементу массива по его индексу. Можно написать `tires[-5]` или `tires[23]`, и компилятор даже не подумает пожаловаться на нерадивого программиста. Конечно, массив содержит только четыре элемента, так что использование -5 или 23 в качестве индекса приведет к обращению к случайному месту в памяти, что закончится ошибками работы программы или, в лучшем случае, ее аварийным завершением.

После проверки индекса массив `tires` помещает колесо в нужное место. Вот пример кода, использующего рассматриваемые методы доступа.

```
Tire *tire = [Tire new];
```

```
[car setTire: tire
 atIndex: 2];
```

```
NSLog(@"Колесо номер 2 есть %@",
 [car tireAtIndex: 2]);
```

Обработка изменений в автомобиле

Имеется еще пара деталей, с которыми надо разобраться, прежде чем объявить о завершении работы.

Первая деталь — метод `init` класса `Car`. Поскольку теперь `Car` имеет методы доступа к двигателю и колесам, его метод `init` не обязан создавать их самостоятельно. Код, который создает автомобиль, отвечает за размещение двигателя и колес. Фактически можно полностью удалить метод `init`, поскольку в нем просто нет необходимости. Грубо говоря, можно купить автомобиль без двигателя и колес и поставить их самостоятельно (насколько все же программирование легче реальной жизни!).

Поскольку `Car` больше не создает собственные части, функцию `main()` следует обновить, с тем чтобы эти части создавались ею. Измените вашу функцию `main()` следующим образом.

```
int main (int argc, const char * argv[])
```

```
{
    Car *car = [Car new];
```

```
    Engine *engine = [Engine new];
    [car setEngine: engine];
```

```
    int i;
    for (i = 0; i < 4; i++) {
        Tire *tire = [Tire new];
        [car setTire: tire
         atIndex: i];
    }
```

```
    [car print];
    return (0);
} // main
```

Функция `main()`, как и ранее, создает новый автомобиль. Затем она создает новый двигатель и вставляет его в автомобиль. После этого выполняются четыре итерации цикла `for`. В каждой итерации создается новое колесо, после чего оно монтируется в автомобиль. Затем готовый автомобиль выводит информацию о себе, и программа завершается.

С точки зрения пользователя программа не изменилась.

Я — двигатель. P-p-p-p-p!

Я — колесо.

Я — колесо.

Я — колесо.

Я — колесо.

Как и ранее, мы выполнили рефакторинг программы, усовершенствовав ее внутреннюю структуру, но оставив прежней ее функциональность.

Расширение составных частей

Теперь, когда у класса `Car` имеются методы доступа, давайте воспользуемся предоставляемыми ими преимуществами. Вместо стандартных двигателя и колес реализуем вариации этих компонентов. При создании нового двигателя и колес мы воспользуемся наследованием, а для оснащения ими автомобиля — методами доступа. Код этой программы можно найти в папке 05.03 CarParts-2.

Начнем с нового типа двигателя — `Slant6` (если вам больше нравится V8 или `ThreeFiftyOneWindsor`, возьмите его).

```
@interface Slant6 : Engine
```

```
@end // Slant6
```

```
@implementation Slant6
```

```
- (NSString *) description
```

```
{
```

```
    return (@"Я — двигатель slant-6. P-p-p-p-p!");
```

```
} // description
```

```
@end // Slant6
```

Класс `Slant6` представляет собой разновидность двигателя, так что имеет смысл сделать его подклассом `Engine`. Вспомните, что наследование позволяет нам передавать подклассы (`Slant6`) туда, где ожидаются надклассы (`Engine`). Поскольку метод `setEngine:` класса `Car` получает аргумент типа `Engine`, мы можем безопасно передать аргумент типа `Slant6`.

Класс `Slant6` перекрывает метод `description`, чтобы выводить новое сообщение. Так как `Slant6` не вызывает метод `description` надкласса (т.е. не включает код `[super description]`), он полностью заменяет унаследованное описание.

Шаги по реализации нового класса колес `AllWeatherRadial` очень похожи на шаги реализации класса `Slant6`. Мы создаем подкласс существующего класса (`Tire`) и снабжаем его новым методом `description`.

```
@interface AllWeatherRadial : Tire
@end // AllWeatherRadial
```

```
@implementation AllWeatherRadial
- (NSString *) description
{
    return(@"Я – колесо, которому годится любая погода!");
} // description
@end // AllWeatherRadial
```

Наконец, мы изменим функцию main() так, чтобы она использовала новые виды двигателя и колес (измененный код выделен полужирным шрифтом).

```
int main (int argc, const char * argv[])
{
    Car *car = [Car new];

    int i;
    for (i = 0; i < 4; i++) {
        Tire *tire = [AllWeatherRadial new];
        [car setTire: tire
         atIndex: i];
    }

    Engine *engine = [Slant6 new];
    [car setEngine: engine];

    [car print];
    return (0);
} // main
```

Мы добавили два новых класса и немного изменили две строки кода, никак при этом не затрагивая класс Car. Наш неизменный Car вполне успешно воспользовался предложенными ему двигателем и колесами, и поведение программы радикально изменилось.

```
Я – двигатель slant-6. P-p-p-p-p!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
```

Композиция или наследование?

Программа CarParts-2 использует как наследование, так и композицию. Хороший вопрос... (нет, даже так — выдающийся вопрос) заключается в том, когда следует применять наследование, а когда — композицию?

Наследование определяет отношение **ЯВЛЯЕТСЯ**. Треугольник *является* фигурой. Slant6 *является* двигателем. AllWeatherRadial *является* колесом. Если мы можем сказать "X *является* Y", значит, мы можем использовать наследование.

Композиция же определяет отношение ИМЕЕТ. Фигура *имеет* цвет. Автомобиль *имеет* двигатель и *имеет* колеса. И напротив, автомобиль не является ни двигателем, ни колесом. Если вы можете сказать "X *имеет* Y", следует применять композицию.

Новички в объектно-ориентированном программировании часто допускают ошибку, стараясь везде и всюду использовать наследование, например порождать Car из Engine. Наследование — красивая игрушка, но не для того, чтобы играть ею везде и постоянно. Вы можете заставить такую программу заработать, так как из кода Car можете получить доступ к коду двигателя. Но такая программа оказывается бессмысленной как с точки зрения читающего ее код (как? автомобиль — это просто двигатель?), так и с точки зрения поддержки и развития имеющегося кода. Так что используйте наследование только там, где оно уместно.

Вот как вы можете поступать при создании своих структур данных. При создании новых объектов остановитесь и немного подумайте, следует ли использовать наследование или композицию. Например, при разработке автомобиля подумайте — "у автомобиля есть колеса, двигатель, трансмиссия. Значит, лучше использовать композицию и предоставить переменные экземпляра для всех этих частей".

В других обстоятельствах можно использовать наследование. Например, вам может понадобиться концепция транспортного средства, для управления которым требуется наличие прав. Автомобиль, мотоцикл, трактор — все они являются такими транспортными средствами. Автомобиль *является* транспортным средством, мотоцикл *является* им, и все это звучит неплохо для наследования. Поэтому, вероятно, вам нужен класс LicensedVehicle (в котором может храниться номер этого транспортного средства с использованием композиции, но не наследования!), от которого наследуются классы Automobile, Motorcycle и т.п.

Резюме

Композиция, которая представляет собой технологию создания объектов, содержащих ссылки на другие объекты, является фундаментальной концепцией объектно-ориентированного программирования. Например, объект-автомобиль содержит ссылки на объекты двигателя и колес. Рассматривая композицию, в данной главе мы познакомились с методами доступа, предоставляющими внешним объектам возможность изменять атрибуты, сохраняя при этом защищенность переменных экземпляров.

Методы доступа и композиция идут рука об руку, потому что обычно методы доступа создаются для каждого объекта, являющегося частью композиции. Вы познакомились с двумя типами методов доступа: устанавливающими, которые позволяют изменять атрибуты, и получающими, которые запрашивают у объекта значения его атрибутов.

В этой главе вы также познакомились с правилами Сосоа для именования методов доступа. В частности, вы узнали, что в именах методов доступа, которые возвращают значения атрибутов, не следует употреблять префикс "get".

В следующей главе вы увидите, как можно разделить классы по нескольким исходным файлам, а не держать их все в одном большом файле.

Организация исходных файлов

До сих пор все рассматриваемые нами проекты состояли из одного исходного файла `main.m`. Функция `main()` и все разделы `@interface` и `@implementation` наших классов располагались в одном файле. Такая структура пригодна для небольших программ, но не для больших проектов. С ростом программы файлы становятся слишком велики для просмотра или поиска. Возвращаясь к годам учебы — вы же не помещали все написанные вами рефераты в один большой файл? Наверняка, каждая работа располагалась в отдельном файле с именем, позволяющим судить о содержимом файла. Аналогично имеет смысл разделять исходный текст программы на несколько файлов, давая каждому из них свое (желательно — описывающее содержание файла) имя. Разделение вашей программы на небольшие файлы повышает шансы более быстро найти интересующий вас код и помогает другим быстрее разобраться в вашем проекте. Размещение кода в нескольких файлах облегчает задачу отправки исходного текста интересного класса друзьям: вам будет достаточно отправить только пару файлов, но не весь проект. В этой главе будут рассмотрены идеи и стратегии размещения частей программы в отдельных файлах.

Разделение интерфейса и реализации

Как вы уже знаете, исходный текст классов Objective-C разделяется на две части. Первая часть — интерфейс, предоставляющий вид класса "извне" и содержащий всю необходимую для использования класса информацию. Для того чтобы иметь возможность использовать объекты класса, вызывать его методы, вставлять объекты в другие классы или создавать подклассы, следует предоставить компилятору всю информацию, содержащуюся в разделе `@interface`.

Вторая часть исходного текста класса — его реализация. Раздел `@implementation` говорит компилятору Objective-C о том, как заставить класс работать. Здесь содержится код, который реализует методы, объявленные в интерфейсе.

В силу естественного разделения определения класса на интерфейс и реализацию, код класса часто точно так же разбивается на два файла. В первом файле хранятся компоненты интерфейса: директива `@interface`, открытые определения `struct`, константы `enum`, `#defines`, глобальные переменные `extern` и т.д. Поскольку Objective-C является наследником C, все это обычно располагается в заголовочном файле, который имеет то же имя, что и имя класса, и расширение `.h`. Например, у класса `Engine` заголовочный файл имеет имя `Engine.h`, а у класса `Circle` — `Circle.h`.

Все детали реализации, такие как директива `@implementation`, определения глобальных переменных, закрытые объявления `struct` и так далее, располагаются в другом файле — с тем же именем, что и имя класса, и расширением `.m`. Файлы `Engine.m` и `Circle.m` представляют собой файлы реализации соответствующих классов.

ПРИМЕЧАНИЕ

Если вы используете расширение `.mm`, то тем самым говорите компилятору, что ваш код написан на Objective-C++, который позволяет использовать вместе C++ и Objective-C.

Создание новых файлов в Xcode

При создании нового класса Xcode упрощает вашу жизнь, автоматически создавая для вас `.h`- и `.m`-файлы. Выбирая в Xcode пункт меню `File`⇒`New File`, вы получите окно наподобие показанного на рис. 6.1, в котором перечислены типы файлов, известные Xcode.

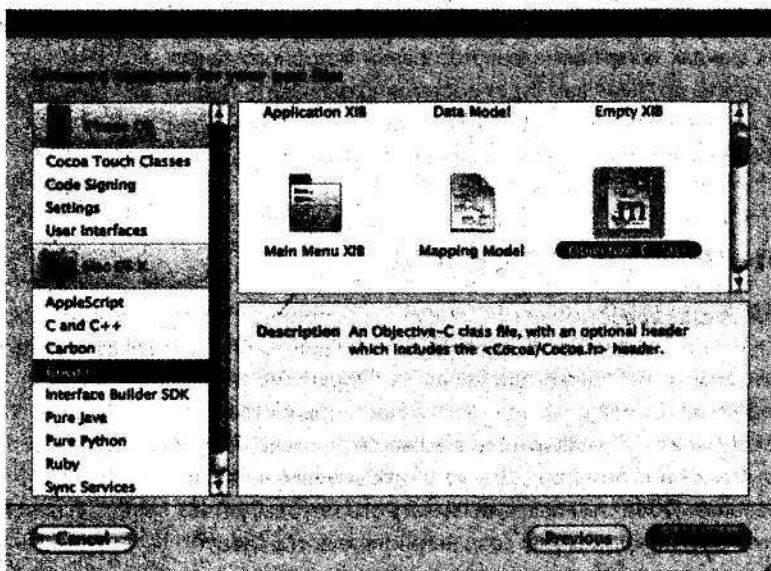


Рис. 6.1. Создание нового файла в Xcode

Выберите Objective-C class и щелкните на кнопке Next. Вы получите очередное окно, в котором у вас запросят имя файла (рис. 6.2).

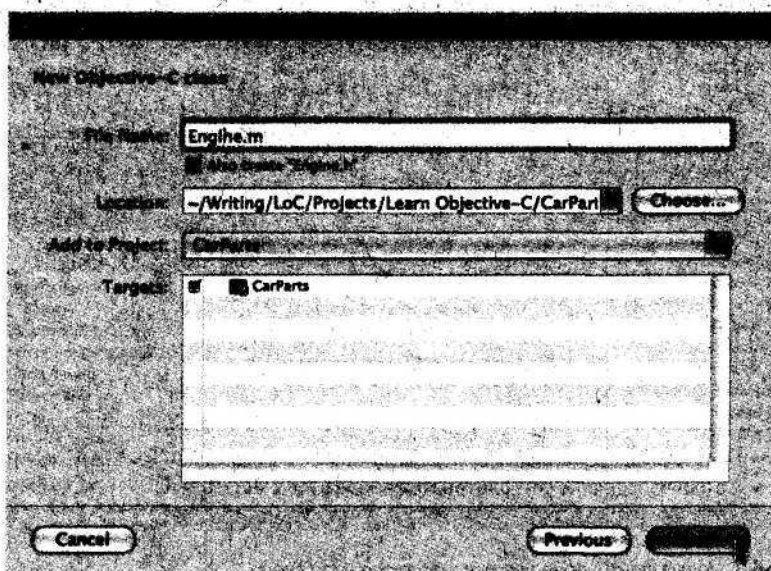


Рис. 6.2. Именованье нового файла

В этом окне не просто вводится имя файла. Здесь есть переключатель, которым можно воспользоваться, чтобы Xcode создал для вас файл Engine.h. Если у вас открыто несколько проектов, можно воспользоваться выпадающим списком Add to project для выбора проекта, к которому относятся вновь создаваемые файлы. Раздел Targets рассматривать не будем; скажем лишь, что сложные проекты могут иметь несколько конечных целей, каждая из которых имеет собственную конфигурацию исходных файлов и различные правила построения.

После щелчка на кнопке Finish Xcode добавит соответствующие файлы в проект и выведет результат в окне проекта, как показано на рис. 6.3.

Xcode помещает новые файлы в выбранную папку в панели Groups & Files (если перед созданием файлов вы выбрали Source, то файлы будут размещены в этой папке). Эти папки (именуемые Xcode Groups) предоставляют способ организации исходных файлов в вашем проекте. Например, вы можете сделать одну группу для классов вашего пользовательского интерфейса, другую — для классов, работающих с данными, что упростит работу с проектом. При создании групп Xcode в действительности не перемещает никакие файлы и не создает никакие каталоги на вашем жестком диске. Взаимосвязи файлов с группами — сугубо логические; если можно так выразиться, это просто красивые фантазии, поддерживаемые Xcode. Если вы хотите, то можете создать группу, которая будет указывать на определенное место в файловой системе, и Xcode будет помещать вновь создаваемые файлы в указанный вами каталог.

После создания файлов вы можете редактировать их — для этого следует выполнить двойной щелчок на имени интересующего вас файла в списке. Xcode любезно вставляет в файл стандартную заготовку исходного текста, такую как `#import <Cocoa/Cocoa.h>`, или пустые разделы `@interface` и `@implementation`, которые вам предстоит заполнить реальным кодом.

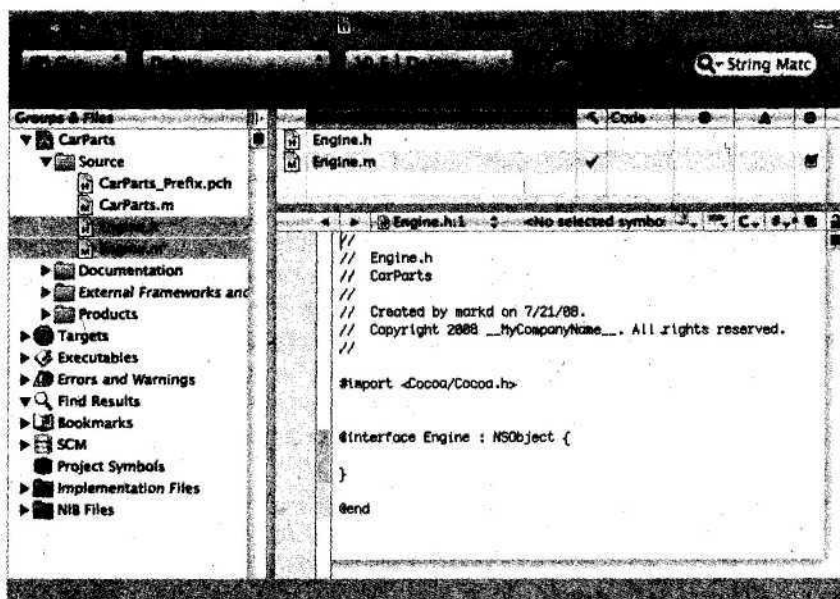


Рис. 6.3. Новые файлы появляются в окне проекта Xcode

ПРИМЕЧАНИЕ

До сих пор в данной книге мы вносили в исходный текст программ строку `#import <Foundation/Foundation.h>`, поскольку использовали только эту часть Cocoa. Но вместо нее можно использовать строку `#import <Cocoa/Cocoa.h>`. Такой импорт, среди прочего, подгрузит и заголовочные файлы схемы Foundation.

Как разбить автомобиль

В программе CarParts-Split (в папке проекта 06.01 CarParts-Split) все классы перемещены в собственные файлы. Каждый класс имеет собственный заголовочный файл (.h) и файл реализации (.m). Давайте рассмотрим, как создается такой проект. Начнем с двух классов, наследующих класс NSObject, — Tire и Engine. Воспользуйтесь командой меню New File и выберите Objective-C Class, после чего введите имя класса — Tire. Сделайте то же самое и для класса Engine. На рис. 6.4 показаны четыре новых файла в списке проекта.

Теперь перенесем раздел `@interface` класса Tire из файла CarParts-Split.m в файл Tire.h. Этот файл принимает следующий вид.

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

```
@interface Tire : NSObject
@end // Tire
```

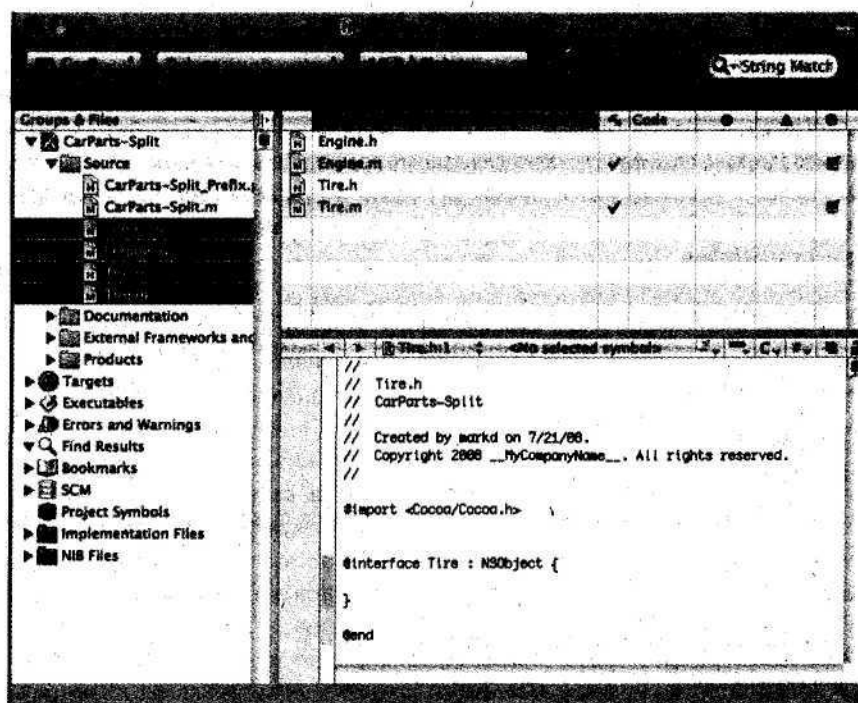



Рис. 6.4. Добавление в проект Tire и Engine

Затем из файла CarParts-Split.m в файл Tire.m переносим раздел @implementation класса Tire. Кроме того, в начало файла следует добавить строку #import "Tire.h". Теперь файл Tire.m должен выглядеть следующим образом.

```
#import "Tire.h"
```

```
@implementation Tire
- (NSString *) description
{
    return (@"Я – колесо.");
} // description
@end // Tire
```

Первая директива #import представляет определенный интерес. Это не импорт заголовочных файлов Cocoa.h или Foundation.h, который осуществлялся ранее. Здесь импортируется заголовочный файл класса. Это стандартная процедура, и вы должны выполнять ее практически в каждом создаваемом проекте. Компилятору для генерации корректного кода требуется информация о схеме размещения переменных экземпляра в памяти, но автоматической загрузки заголовочных файлов компилятор сам не выполняет. Ему следует явно указать подгружаемые заголовочные файлы при помощи директивы #import. Если при компиляции вы получаете сообщение об ошибке наподобие "Cannot find interface definition for Tire" (не могу найти определение интерфейса для Tire), то это обычно означает, что вы забыли директиву #import с заголовочным файлом класса.

ПРИМЕЧАНИЕ

Обратите внимание на наличие двух различных способов импорта: с применением кавычек и угловых скобок — например, сравните директивы `#import <Cocoa/Cocoa.h>` и `#import "Tire.h"`. Версия с угловыми скобками используется для импорта системных заголовочных файлов. Версия с кавычками указывает, что заголовочный файл локален для данного проекта. Если вы видите заголовочный файл в угловых скобках — он предназначен только для чтения в рамках вашего проекта, поскольку им владеет система. Если же имя файла взято в кавычки, то вы (или иной программист проекта) можете вносить в него изменения.

Теперь выполним те же действия для класса `Engine`. Перенесем раздел `@interface` этого класса из файла `CarParts-Split.m` в `Engine.h`, который принимает следующий вид.

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

```
@interface Engine : NSObject
@end // Engine
```

Затем перенесем раздел `@implementation` в файл `Engine.m`, который после этой операции выйдет так.

```
#import "Engine.h"
```

```
@implementation Engine
- (NSString *) description
{
    return (@"Я – двигатель. P-p-p-p-p!");
} // description
@end // Engine
```

Если теперь попробовать скомпилировать программу `CarParts-Split.m`, то вы получите сообщение об ошибке из-за отсутствия объявлений классов `Tire` и `Engine`. Исправить эту ошибку очень легко. Достаточно добавить две следующие строки в начало файла `CarParts-Split.m`, сразу после строки `#import <Foundation/Foundation.h>`.

```
#import "Tire.h"
#import "Engine.h"
```

ПРИМЕЧАНИЕ

Помните, что директива `#import` подобна директиве `#include`, которая обрабатывается препроцессором C. В данном случае препроцессор C, по сути, выполняет добавление содержимого файлов `Tire.h` и `Engine.h` в файл `CarParts-Split.m` перед началом компиляции.

Вы можете собрать и запустить программу, получив при этом ту же функциональность программы, что и ранее, при использовании классов `AllWeatherRadials` и `Slant6`.

```
Я – двигатель slant-6. P-p-p-p-p!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
Я – колесо, которому годится любая погода!
```

Зависимости между файлами

Зависимость представляет собой взаимоотношение между двумя сущностями. Вопросы зависимостей часто возникают в процессе разработки и написания программ. Зависимости могут иметься между двумя классами: например, `Slant6` зависит от `Engine` из-за отношения наследования. При изменении `Engine`, таком как добавление новой переменной экземпляра, `Slant6` должен быть перекомпилирован, чтобы воспринять это изменение.

Зависимости могут иметься и между двумя и более файлами. `CarParts-Split.m` зависит от `Tire.h` и `Engine.h`. Если какой-то из этих файлов изменится, файл `CarParts-Split.m` должен быть перекомпилирован, чтобы воспринять эти изменения. Например, `Tire.h` может содержать константу `kDefaultTirePressure`, равную 2 атм. Допустим, программист, который работает над программой, решает, что давление по умолчанию следует изменить на 3 атм. и вносит соответствующее изменение в файл `Tire.h`. Теперь необходимо перекомпилировать файл `CarParts-Split.m`, чтобы старое значение было заменено новым.

Импорт заголовочного файла устанавливает строгую зависимость между заголовочным файлом и импортирующим его файлом с исходным текстом. Если заголовочный файл изменяется, то все файлы, зависящие от него, требуют перекомпиляции. Представим, что у нас есть сотня `.m`-файлов, каждый из которых включает некоторый (один и тот же) заголовочный файл — скажем, `UserDefaultsConstants.h`. Если вы внесете изменение в `UserDefaultsConstants.h`, все сто `.m`-файлов должны быть перекомпилированы, что может потребовать большого количества времени, какая бы мощная машина не имелась в вашем распоряжении.

Вопрос рекомпиляции может быть еще более неприятным в силу транзитивности зависимости. Например, если `Thing1.h` импортирует `Thing2.h`, который в свою очередь импортирует `Thing3.h`, то любые изменения в `Thing3.h` приводят к необходимости рекомпиляции файлов, импортирующих `Thing1.h`. Xcode отслеживает все зависимости файлов за вас.

Минимальная рекомпиляция

А вот хорошая новость: Objective-C предоставляет способ минимизировать перекомпиляцию, вызванную зависимостями. Вопросы зависимостей появляются из-за того, что компилятор Objective-C для успешной работы нуждается в определенной информации. Временами ему требуется полная информация о классе, такая как схема размещения переменных экземпляра и цепочка наследования, а иногда достаточно только имени класса без его полного определения.

Например, при композиции (с которой вы познакомились в предыдущей главе) применяются указатели на объекты, поскольку все объекты Objective-C используют динамически выделяемую память. Компилятору достаточно знать о том, что некоторый элемент является классом, — при этом ему известно, что переменная экземпляра имеет размер указателя (который одинаков во всей программе).

В Objective-C вводится ключевое слово `@class`, назначение которого — сказать компилятору “это — класс, и я собираюсь обращаться к нему только через указатели”. Компилятор может быть спокоен: ему ничего не надо знать об этом классе, кроме того, что это нечто в памяти, доступ к чему осуществляется посредством указателя.

Мы воспользуемся ключевым словом `@class` при перемещении класса `Car` в собственный класс. Создадим файлы `Car.h` и `Car.m` при помощи Xcode, так же как мы делали это с классами `Tire` и `Engine`. Переместим раздел `@interface` класса `Car` в файл `Car.h`, который теперь принимает следующий вид.

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

```
@interface Car : NSObject
```

```
{
    Tire *tires[4];
    Engine *engine;
}
```

```
- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;
- (void) setTire: (Tire *) tire
    atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
- (void) print;
@end // Car
```

Если мы попытаемся воспользоваться этим заголовочным файлом, то получим от компилятора сообщение об ошибке, говорящее о том, что он не понимает, что такое `Tire` или `Engine`. Сообщение, скорее всего, будет иметь вид `error: parse error before "Tire"`, что в переводе с компиляторного на русский означает "я этого не понимаю".

У нас есть два варианта исправления этой ошибки. Первый заключается в импорте `Tire.h` и `Engine.h`, которые снабжают компилятор всей имеющейся информацией об указанных классах.

Но есть способ и получше. Если внимательно посмотреть на интерфейс класса `Car`, то вы увидите, что обращения к классам `Tire` и `Engine` осуществляются через указатели, т.е. это как раз тот случай, когда можно применить ключевое слово `@class`. Вот как выглядит файл `Car.h` с добавленными строками `@class`.

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

```
@class Tire;
```

```
@class Engine;
```

```
@interface Car : NSObject
```

```
{
    Tire *tires[4];
    Engine *engine;
}
```

```
- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;
- (void) setTire: (Tire *) tire
    atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
- (void) print;
@end // Car
```

Здесь достаточно информации для того, чтобы компилятор мог работать с интерфейсом класса Car.

ПРИМЕЧАНИЕ

Ключевое слово `@class` является *опережающей ссылкой* (forward reference). Это способ сообщить компилятору "доверься мне; ты знаешь, что такой класс есть, а большего тебе пока знать и не надо".

Ключевое слово `@class` также полезно в ситуации с *циклическими зависимостями*, т.е. когда класс A использует класс B, а класс B — класс A. Если вы попытаетесь импортировать каждый класс из заголовочного файла другого класса, то получите ошибку компиляции. Но если применить `@class B` в A.h, а `@class A` — в B.h, то проблема будет успешно разрешена.

Поехали

Итак, мы сумели исправить заголовочный файл класса Car. Но файлу Car.m для компиляции требуется больше информации о классах Tire и Engine. Компилятор должен знать, от каких классов унаследованы Tire и Engine, чтобы убедиться, что объекты в состоянии отвечать на посылаемые им сообщения. Для этого мы импортируем Tire.h и Engine.h в Car.m. Нам также требуется удалить раздел `@implementation` класса Car из файла CarParts-Split.m. Теперь файл Car.m имеет следующий вид.

```
#import "Car.h"
#import "Tire.h"
#import "Engine.h"

@implementation Car
- (void) setEngine: (Engine *) newEngine
{
    engine = newEngine;
} // setEngine

- (Engine *) engine
{
    return (engine);
} // engine

- (void) setTire: (Tire *) tire
    atIndex: (int) index
{
    if (index < 0 || index > 3) {
        NSLog (@ "Неверный индекс (%d) в setTire:atIndex:",
            index);
        exit (1);
    }
    tires[index] = tire;
} // setTire:atIndex:

- (Tire *) tireAtIndex: (int) index
{

```

```

if (index < 0 || index > 3) {
    NSLog(@"Неверный индекс (%d) в setTire:atIndex:",
          index);
    exit(1);
}
return (tires[index]);
} // tireAtIndex:

- (void) print
{
    NSLog(@"%@ ", engine);
    NSLog(@"%@ ", tires[0]);
    NSLog(@"%@ ", tires[1]);
    NSLog(@"%@ ", tires[2]);
    NSLog(@"%@ ", tires[3]);
} // print
@end // Car

```

Теперь можно в очередной раз скомпилировать и запустить программу и получить тот же вывод, что и ранее. Похоже, мы выполнили очередной рефакторинг (но об этом не рекомендуется говорить вслух). Мы улучшили внутреннюю структуру нашей программы, сохранив ее поведение неизменным.

Импорт и наследование

Нам надо освободить из заточения в CarParts-Split.m еще два класса: Slant6 и AllWeatherRadial. Данная задача немного сложнее, поскольку эти классы унаследованы от созданных нами классов: Slant6 наследует Engine, а AllWeatherRadial — Tire. Поскольку мы не просто применяем указатели на классы, а выполняем наследование, не можем воспользоваться трюком с ключевым словом @class в заголовочных файлах. Мы должны вставить #import "Engine.h" в файл Slant6.h и #import "Tire.h" — в файл AllWeatherRadial.h.

И все же, давайте уточним, почему именно мы не можем применить ключевое слово @class в этой ситуации. Дело в том, что компилятор должен все знать о надклассе, чтобы успешно скомпилировать раздел @interface для его подкласса. Компилятор должен знать схему размещения (типы, размеры и порядок) переменных экземпляра в надклассе. Вспомните, что когда вы добавляете переменные экземпляра в подкласс, они размещаются после переменных экземпляра надкласса. Затем компилятор использует эту информацию для определения расположения переменных экземпляра в памяти, начиная со скрытого указателя self, передаваемого при вызове каждого метода. Для корректного вычисления местоположения переменных экземпляра в памяти компилятору требуется полная информация о классе.

Теперь приступим к классу Slant6. Создадим новые файлы Slant6.m и Slant6.h в Xcode, затем вынесем раздел @interface этого класса из CarParts-Split.m. Если вы все сделаете аккуратно, то файл Slant6.h должен иметь следующий вид.

```

#import "Engine.h"

@interface Slant6 : Engine
@end // Slant6

```

Этот файл импортирует только Engine.h, но не <Cocoa/Cocoa.h>. Почему? Мы знаем, что Engine.h импортирует <Cocoa/Cocoa.h>, так что нам не надо импортировать его здесь еще раз самостоятельно. Но если вы поместите в этот файл директиву #import <Cocoa/Cocoa.h>, ничего страшного не произойдет, так как директива #import достаточно интеллектуальна, чтобы избежать многократного включения.

Файл Slant6.m содержит перенесенный из файла CarParts-Split.m раздел @implementation класса Slant6 с директивой импорта заголовочного файла Slant6.h.

```
#import "Slant6.h"
```

```
@implementation Slant6
```

```
- (NSString *) description
```

```
{
```

```
    return (@"Я — двигатель slant-6. P-p-p-p!");
```

```
} // description
```

```
@end // Slant6
```

Выполните аналогичные шаги по переносу класса AllWeatherRadial в собственную пару файлов. Вот как после этого должен выглядеть файл AllWeatherRadial.h.

```
#import "Tire.h"
```

```
@interface AllWeatherRadial : Tire
```

```
@end // AllWeatherRadial
```

А вот вид файла AllWeatherRadial.m.

```
#import "AllWeatherRadial.h"
```

```
@implementation AllWeatherRadial
```

```
- (NSString *) description
```

```
{
```

```
    return (@"Я — колесо, которому годится любая погода!");
```

```
} // description
```

```
@end // AllWeatherRadial
```

От файла CarParts-Split.m теперь осталась лишь жалкая оболочка. Теперь в нем нет ничего, кроме директив #import и единственной функции.

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

```
#import "Tire.h"
```

```
#import "Engine.h"
```

```
#import "Car.h"
```

```
#import "Slant6.h"
```

```
#import "AllWeatherRadial.h"
```

```
int main (int argc, const char * argv[])
```

```
{
```

```
    Car *car = [Car new];
```

```
    int i;
```



```

for (i = 0; i < 4; i++) {
    Tire *tire = [AllWeatherRadial new];
    [car setTire: tire
     atIndex: i];
}

Engine *engine = [Slant6 new];
[car setEngine: engine];

[car print];

return (0);
} // main

```

Если мы скомпилируем и запустим проект, то вновь получим тот же вывод на экран, что и до начала процесса "распыления" проекта по множеству файлов.

Резюме

В этой главе вы познакомились с применением множества файлов для организации вашего кода. Обычно каждому классу отводится два файла: заголовочный файл, содержащий интерфейс класса, и .m-файл с его реализацией. Пользователи класса импортируют посредством директивы `#import` заголовочный файл класса и получают доступ ко всем его возможностям.

Вы познакомились также с зависимостями между файлами, когда некоторый заголовочный или исходный файл нуждается в информации из другого заголовочного файла. Паутина зависимостей может привести к резкому увеличению времени компиляции и к излишней перекомпиляции — там, где без нее можно обойтись. Аккуратное применение директивы `@class` может снизить количество импортируемых заголовочных файлов, а значит, и время компиляции.

Далее мы познакомим вас с некоторыми интересными возможностями Xcode, так что не откладывайте книгу в сторону!

Еще об Xcode

Многие программисты большую часть своего рабочего времени тратят на написание кода и работу с ним в среде Xcode, обладающей массой чудесных (но зачастую не очевидных) возможностей. Если вы планируете всерьез и надолго применять Xcode, вам стоит изучить этот инструмент так тщательно, как у вас получится. В этой главе мы познакомим вас с некоторыми хитростями при работе с редактором, которые помогут вам эффективнее писать код, ориентироваться в нем и быстро находить нужную информацию, а также с тем, как Xcode может помочь вам в отладке ваших программ.

Xcode — очень большое приложение, с огромными возможностями по настройке под конкретного пользователя — иногда доходящими до смешного. Об Xcode можно написать целые тома (и они действительно написаны), но наша книга все же посвящена несколько иной теме. Наша основная рекомендация — по крайней мере, первое время использовать настройки Xcode по умолчанию. Если вам что-то будет не нравиться — скорее всего, вы обнаружите, как изменить настройки так, чтобы поведение Xcode изменилось в нужном для вас направлении.

При встрече такого огромного приложения, как Xcode, наилучшая стратегия состоит в том, чтоб бегло ознакомиться с документацией и приступить к практической работе с приложением. Затем, приобретя определенный опыт работы, вновь вернуться к документации. На каждой итерации все больший и больший объем материала будет становиться вам понятен.

Мы рассказываем об Xcode 3.1 — текущей версии на момент написания книги. Apple любит добавлять все новые и новые рюшечки и завитушечки к программам, так что если вы используете, скажем, Xcode 42.0, то копии экранов, вероятно, покажутся вам устаревшими.

Изменение имени компании

При создании нового исходного файла в Objective-C вы должны были заметить блок комментариев, генерируемый для вас Xcode.

```
//
// TapDance.h
// Groovilicious
//
// Created by markd on 7/25/08.
// Copyright 2008 __MyCompanyName__. All rights reserved.
//
```

Xcode включает в блок имя файла и имя проекта, а также имя пользователя, создавшего файл, и время создания — вся эта информация может позже помочь при поиске нужного файла. Конечно, имя компании, используемое по умолчанию — `__MyCompanyName__` — вам вряд ли понравится.

По необъяснимым причинам в Xcode 3.1 отсутствует пользовательский интерфейс, который мог бы помочь вам изменить это имя — вам придется перейти для этого в терминал. Но вам стоит не пожалеть времени для смены имени, в особенности если вы планируете создавать много новых файлов. Это может быть ваше собственное имя, имя вашей компании или что-то иное.

Сначала откройте в Finder папку Utilities. Чтобы перейти к ней, можно также воспользоваться комбинацией клавиш `<⌘+⇧+U>`. Найдите приложение Terminal и запустите его. В терминале введите следующую команду — в точности такую, как в книге, но в одну строку, и вместо `"Length-O-Words.com"` вставьте то имя, которое вам надо.

```
defaults write com.apple.Xcode
PBXCustomTemplateMacroDefinitions
'{"ORGANIZATIONNAME" = "Length-O-Words.com";}'
```

После того как введете весь текст в одну строку, нажмите клавишу `<Enter>`. Если все работает верно, вы не увидите никакого сообщения. К счастью, такую длинную команду вам надо ввести только один раз. Выйдите и перезапустите Xcode, и теперь генерируемый блок комментариев будет иметь существенно более привлекательный вид.

```
//
// HulaDance.h
// Untitled3
//
// Created by markd on 7/25/08.
// Copyright 2008 Length-O-Words.com. All rights reserved.
//
```

Больше до конца этой главы вам не придется пользоваться терминалом.

Советы по использованию редактора

Xcode предоставляет несколько основных способов организации проекта и редакторов исходного текста. До сих пор мы использовали интерфейс по умолчанию, который, в основном, представляет собой единое окно для выполнения всех задач, связанных с работой над проектом. Имеются и вспомогательные окна, подобно окну журнала выполнения, показан-

ному на рис. 7.1. Для редактирования всех исходных файлов используется одна панель, а содержимое редактора изменяется в соответствии с тем, какой исходный файл выбран в левой панели Groups & Files.

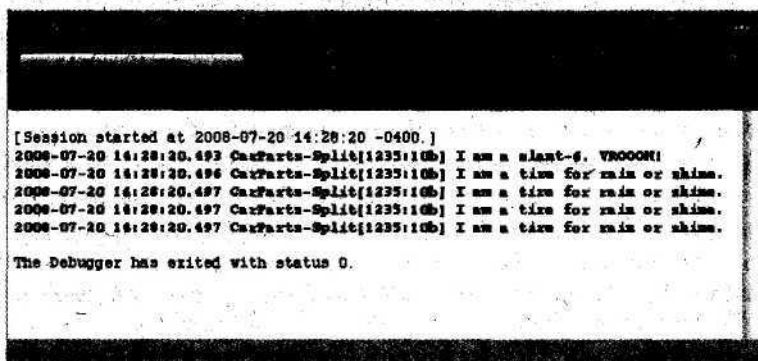
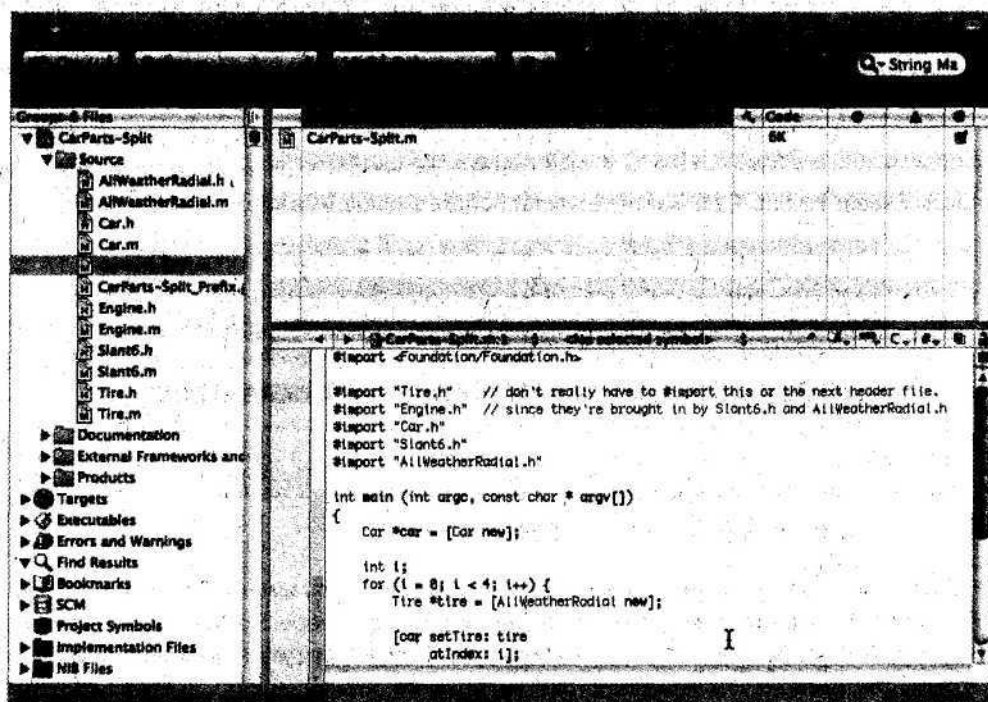


Рис. 7.1. Пользовательский интерфейс Xcode по умолчанию: исходный текст и отладчик

Xcode имеет также режим работы, при котором каждый исходный файл открывается для редактирования в отдельном окне. Если у вас экран огромного размера и вас не смущает наличие множества окон, — этот режим работы может оказаться именно тем, что вам надо.

Далее же мы полагаем, что вы все же выбрали режим работы с редактором кода, встроенным в окно Project, хотя бы по той причине, что так проще делать копии экрана.

В левой части окна находится список Groups & Files, в котором перечислены все части вашего проекта — исходные файлы, схемы, а также пункт Targets, в котором описывается, как именно должны строиться ваши программы. Кроме того, вы найдете некоторые утилиты, такие как закладки, репозиторий кода (особенно удобный при совместной работе над проектом группы программистов) и некоторые другие.

В верхней части окна, ниже панели инструментов, находится браузер, который показывает файлы, выбранные из списка Groups & Files. Для поиска нужных файлов можно воспользоваться полем Search. На рис. 7.2 показан результат поиска файлов, имя которых содержит букву *n*.

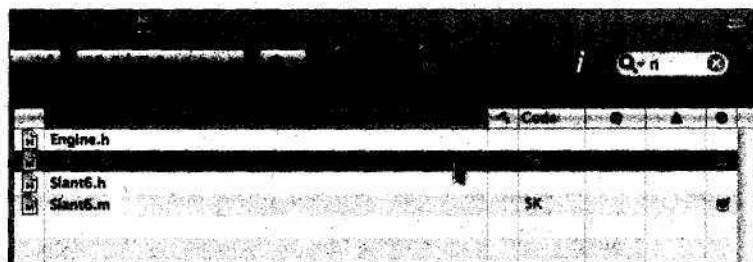


Рис. 7.2. Сокращение списка файлов при поиске

Браузер показывает все исходные файлы с буквой *n* в его имени. Для того чтобы этот файл был открыт в редакторе, щелкните на его имени в браузере. Поскольку большие проекты могут состоять более чем из сотни исходных файлов, браузер оказывается удобным инструментом навигации (о чем мы поговорим немного позже).

При работе с исходным текстом вы можете захотеть скрыть браузер, чтобы увеличить окно редактирования кода. Одна из пиктограмм на панели инструментов по умолчанию, расположенная далеко справа (и не видимая на представленных копиях экрана), называется Editor; она предназначена для включения и отключения браузера. Это же переключение состояния браузера можно выполнить при помощи комбинации клавиш $\text{⌘} + \text{⌥} + \text{E}$.

Даже при работе в однооконном режиме может оказаться полезным размещение одного или двух исходных файлов в собственных окнах, в особенности если вы сравниваете два разных файла. Дважды щелкните на исходном файле в панели Groups & Files, чтобы открыть его в новом окне. Один и тот же файл может быть открыт в двух разных окнах, однако учтите, что эти два окна могут оказаться в рассинхронизированном состоянии, пока вы не щелкнете на каждом из них.

Помощь в написании кода

Одни программисты работают над кодом целыми днями. Другие захватывают еще и ночи. Xcode предлагает всем им некоторые возможности, делающие написание исходного текста не только более простым, но и более веселым делом.

Отступы

Вы, вероятно, обратили внимание на аккуратные отступы в исходных текстах, приведенных в этой книге, когда тела таких конструкций, как `if` или `for`, смещены относительно окружающего их кода. Objective-C не требует наличия отступов, но это повышает удобочитаемость текста и подчеркивает его логическую структуру. Xcode автоматически создает отступы при вводе вами исходного текста.

Иногда при редактировании кода отступы сбиваются. Xcode может помочь и в этом случае. Щелкните правой кнопкой мыши для вывода контекстного меню и выберите в нем пункт `Re-indent selection`. Это приведет к тому, что Xcode пройдет по выделенному фрагменту текста, корректно применяя к нему отступы. Для этого действия нет комбинации клавиш по умолчанию, но вы можете добавить ее в Xcode при помощи панели `Key Bindings`.

Комбинации клавиш `<⌘+[>` и `<⌘+>` сдвигают выделенный код влево или вправо, что очень удобно, если вы хотите разместить конструкцию типа `if` вокруг некоторого фрагмента кода.

Скажем, у вас в редакторе имеется следующий фрагмент.

```
Engine *engine = [Slant6 new];
[car setEngine: engine];
```

Позже вы решили создавать новый двигатель только по желанию пользователя.

```
if (userWantsANewEngine) {
Engine *engine = [Slant6 new];
[car setEngine: engine];
}
```

Вы можете выделить две средние строки и нажать комбинацию клавиш `<⌘+>` для их смещения.

Можно до бесконечности настраивать механизм отступа Xcode. Вы можете предпочесть использовать пробелы или символы табуляции. Можете предпочесть размещать фигурные скобки в отдельных строках, а не в той же строке, где находится инструкция `if`. Вы можете настроить Xcode так, чтобы он следовал Вашему Единственно Правильному Стилю. Кстати, вот еще один вредный совет: если вы хотите развязать драку среди программистов — начните разговор о том, как следует правильно форматировать исходные тексты программ.

Завершение кода

При работе в редакторе Xcode иногда можно заметить, что при вводе кода вам делаются определенные предложения по дальнейшему вводу. Это так называемая возможность `Code Sense`, часто именуемая просто *завершением кода*, или *автозавершением*. Пока вы пишете свою программу, Xcode строит индекс всех введенных вами слов, включая имена переменных и методов из вашего проекта и включенных схем. При вашем вводе Xcode постоянно сравнивает уже введенную вами часть слова со своим индексом и при соответствии делает вам предложение по автоматическому завершению вводимого, как показано на рис. 7.3.

Здесь мы ввели `[All`, и Xcode решил, что, возможно, мы хотим послать сообщение классу `AllWeatherRadial`. Xcode предположил совершенно верно, так что можно нажать клавишу табуляции, чтобы принять автоматическое завершение слова `AllWeatherRadial`.

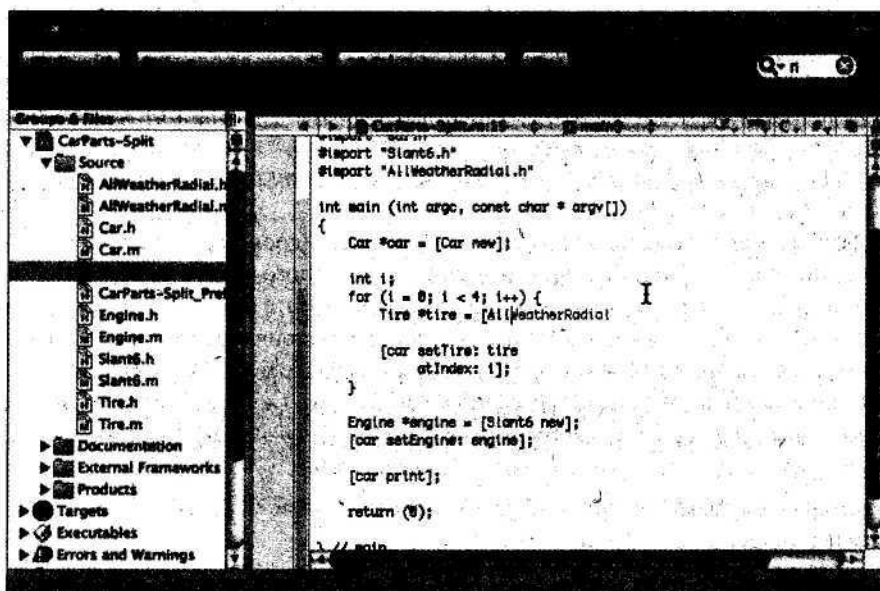


Рис. 7.3. Завершение кода в Xcode

Вы, наверное, скажете: "А, это слишком просто! У нас всего лишь один класс, начинающийся на 'All!'. Да, это так, но Xcode при наличии нескольких возможностей услужливо предлагает целое меню, и в любом случае вы можете нажать клавишу <Esc> для того, чтобы заставить Xcode открыть меню со всеми возможными вариантами завершения, как показано на рис. 7.4.

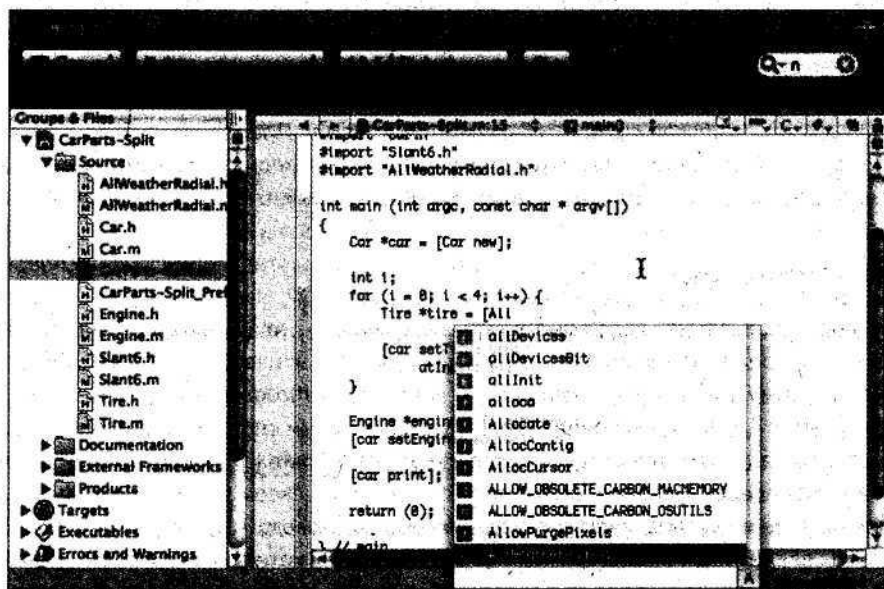


Рис. 7.4. Варианты завершения для текста "all"

Вы можете увидеть несколько вариантов, начинающихся на "all". Xcode разумно предполагает, что в первую очередь следует предлагать варианты из текущего проекта. Чтобы вам было легче ориентироваться в предложениях, рядом с именем располагаются цветные квадратики, указывающие, чем является предлагаемое слово: E указывает на символы перечислений, f — на функции, # — на макроопределения `#define`, m — на методы, C — на классы и т.д.

Если вы не хотите делать выбор в меню, воспользуйтесь комбинацией клавиш `<Control+>` для циклического прохода по вариантам в прямом направлении и `<Shift+Control+>` — для прохода в обратном направлении.

Меню завершения можно использовать как краткий справочник по API класса. Рассмотрим класс `NSDictionary`, у которого есть метод, позволяющий вам указать список аргументов, представляющих собой ключи и объекты для построения словаря. Это `+dictionaryWithKeysAndObjects` или `+dictionaryWithObjectsAndKeys`? Кто может запомнить все эти имена? Простейший способ выяснить, как называется нужный метод, — это ввести `[NSDictionary`, затем ввести пробел для указания, что вы завершили ввод имени класса, и нажать клавишу `<Esc>`. Xcode понимает, что теперь вы намерены ввести имя метода, и выведет все имена методов класса `NSDictionary`, и среди прочих — `dictionaryWithObjectsAndKeys`, показанный вблизи начала меню на рис. 7.5.

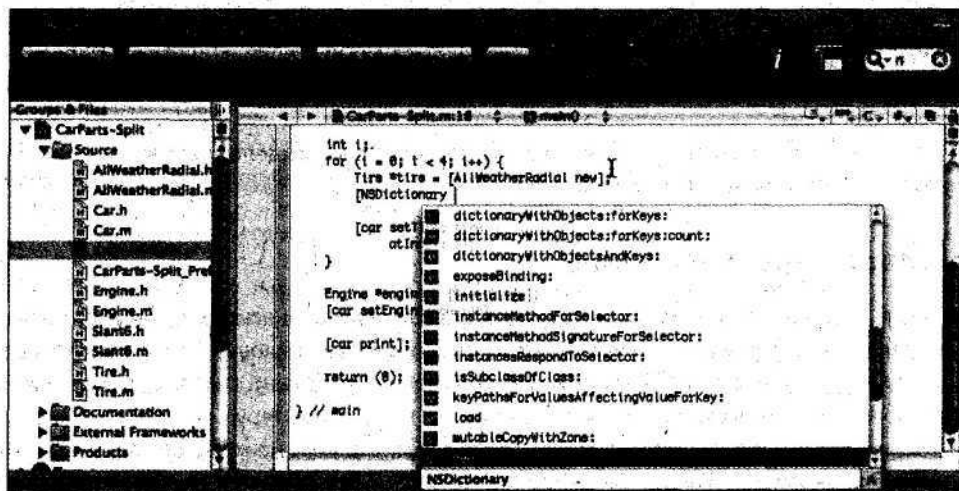


Рис. 7.5. Изучение класса при помощи Code Sense

Иногда при использовании возможности завершения кода вы видите странные небольшие прямоугольники вокруг частей завершающего текста, показанные на рис. 7.6. Что бы это значило?

Как видите, Xcode предлагает для завершения `-setTire:atIndex:`, метод, имеющий два параметра. Code Sense идет дальше, чем простое завершение имен. Показанные два параметра на самом деле представляют собой заполнители. Если вы нажмете клавишу табуляции, метод будет завершён как `setTire` (рис. 7.7).

Первый заполнитель подсвечен. Введите текст, заменяя заполнитель реальным аргументом. Вы можете щелкнуть на втором заполнителе и заменить и его. Причем это можно сделать, и не отрывая рук от клавиатуры, — перейти к следующему заполнителю можно при помощи комбинации клавиш `<Control+>/>`.

```
Engine *engine = [Slant6 new];
[car setTire:[Tire *tire, atIndex:(int)index]
```

Рис. 7.6. Заполнители в завершающем коде

```
Engine *engine = [Slant6 new];
[car setTire:[Tire *tire, atIndex:(int)index]
```

Рис. 7.7. Выбор заполнителя

Подсветка скобок

При вводе исходного текста вы можете заметить небольшую подсветку на экране при вводе определенных символов, таких как `,` `]` или `}`. Это Xcode указывает вам, чему соответствует вводимая закрывающая скобка, как показано на рис. 7.8.

```
for (i = 0; i < 4; i++) {
    Tire *tire = [AllWeatherRadial new];
    [car setTire: tire atIndex: i]
}
```

Рис. 7.8. Подсветка скобок

Эта возможность иногда называется “поцелуем скобок” (kissing the parentheses) и может оказаться реально полезной, когда вы завершаете сложное выражение, переполненное скобками. Вы можете убедиться, что ввели все закрывающие скобки и ничего не перепутали (если вы попытаетесь ввести `]` там, где на самом деле требуется `,`), Xcode предупредит вас звуковым сигналом.

Если дважды щелкнуть на одном из таких символов, Xcode выделит весь код между ним и соответствующим ему парным символом.

Массовое редактирование

Иногда имеется изменение кода, которое надо выполнить в нескольких местах, но вы не хотите делать его вручную. В общем-то, это верно, поскольку человек плохо приспособлен для выполнения скучной, надоедливой, повторяющейся работы. К счастью, такую скучную работу можно передать отлично справляющемуся с ней компьютеру.

Первая возможность Xcode не выполняет действий с кодом, но помогает в работе с ним. Воспользуйтесь командой меню `File⇒Make Snapshot`, и Xcode запомнит состояние вашего проекта. Теперь вы можете как угодно редактировать исходные файлы и папки, словом, делать что угодно — вы всегда сможете вернуться к старому состоянию, воспользовавшись окном снимков при помощи команды меню `File⇒Snapshots`. Хорошая практика заключается в создании снимка перед тем, как приступать к каким-то рискованным действиям с проектом.

ПРИМЕЧАНИЕ

Снимки хранятся в образе диска, располагающемся в файле `~/Library/Application Support/Developer/Shared/SnapshotRepository.sparseimage`. Бывает, что образ диска повреждается (может, от излишней жесткости диска?), и Xcode сообщает о загадочной ошибке “Snapshot Failed: A project snapshot cannot be created”. Если вы получили это сообщение, попробуйте удалить указанный файл и перегрузить компьютер.

И, конечно же, Xcode обладает функциональностью для выполнения поиска и замены, доступной через подменю **Edit**⇒**Find** с рядом других пунктов. Пункт **Find in Project** позволяет выполнять поиск и замену во всех файлах проекта. Соответствующее окно показано на рис. 7.9,

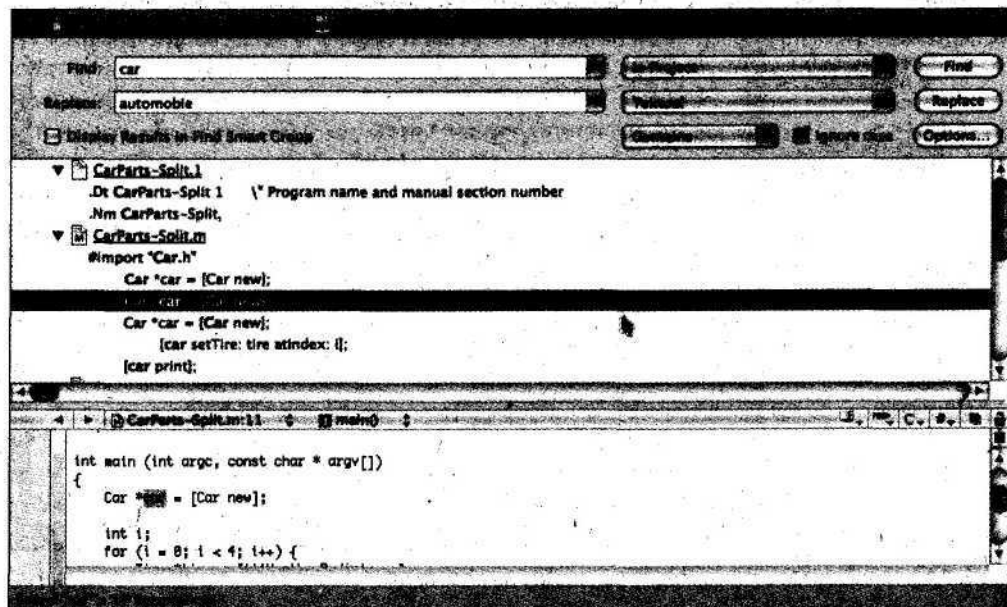


Рис. 7.9. Поиск и замена уровня проекта

Допустим, мы хотим заменить "car" на "automobile". После заполнения соответствующих полей и щелчка на кнопке **Find** вы можете увидеть ссылки на класс **Car** и на локальную переменную **car**. Вы должны снять флажок **Ignore case** и изменить только локальную переменную функции **main()**. Вы можете щелкнуть на кнопке **Replace All**, чтобы выполнить глобальную замену.

Однако инструмент поиска и замены несколько туповат. Если вы пытаетесь переименовать переменную в пределах функции, то он сделает слишком много — постарается выполнить переименование во всем файле. Если же вы попытаетесь переименовать класс — он сделает слишком мало; в частности, даже не попытается переименовать файлы.

Xcode обладает двумя возможностями для заполнения промежутка. Первая называется "редактировать все в области видимости" (**Edit all in Scope**). Вы можете выбрать символ, типа локальной переменной или параметра, и выбрать пункт меню **Edit**⇒**Edit all in Scope**. После этого при вводе будут корректно обновлены все вхождения этого символа. Это не просто быстро — это очень красиво выглядит!

На рис. 7.10 показано редактирование "car" в области видимости. Обратите внимание, что все локальные переменные **car** находятся в прямоугольниках. Теперь, если вы начнете вводить слово **Automobile**, будут одновременно изменяться все прямоугольники, как показано на рис. 7.11.

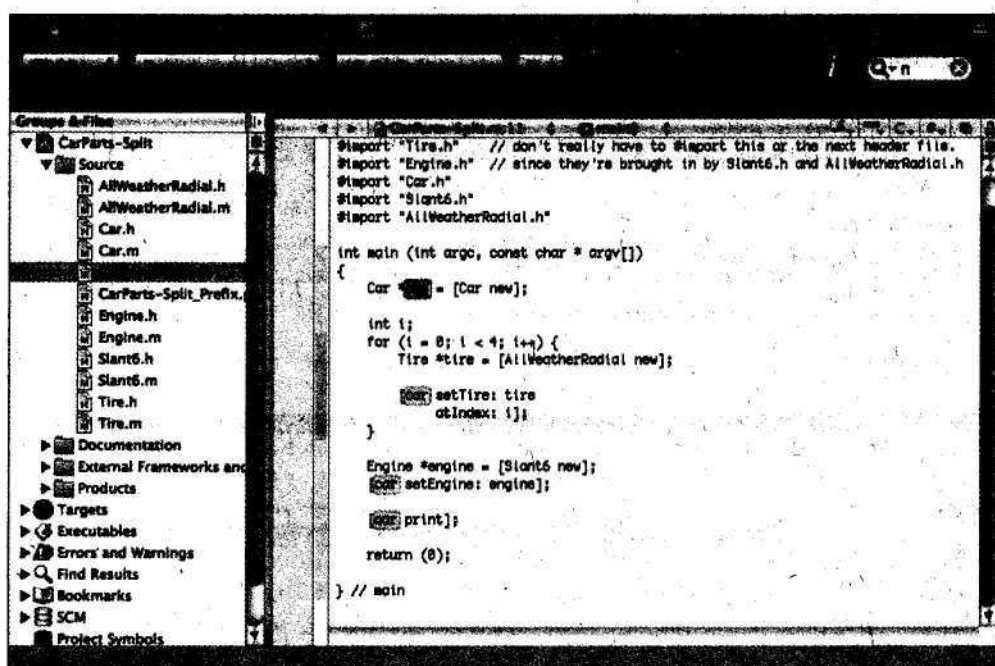


Рис. 7.10. Начало редактирования — замена car на automobile

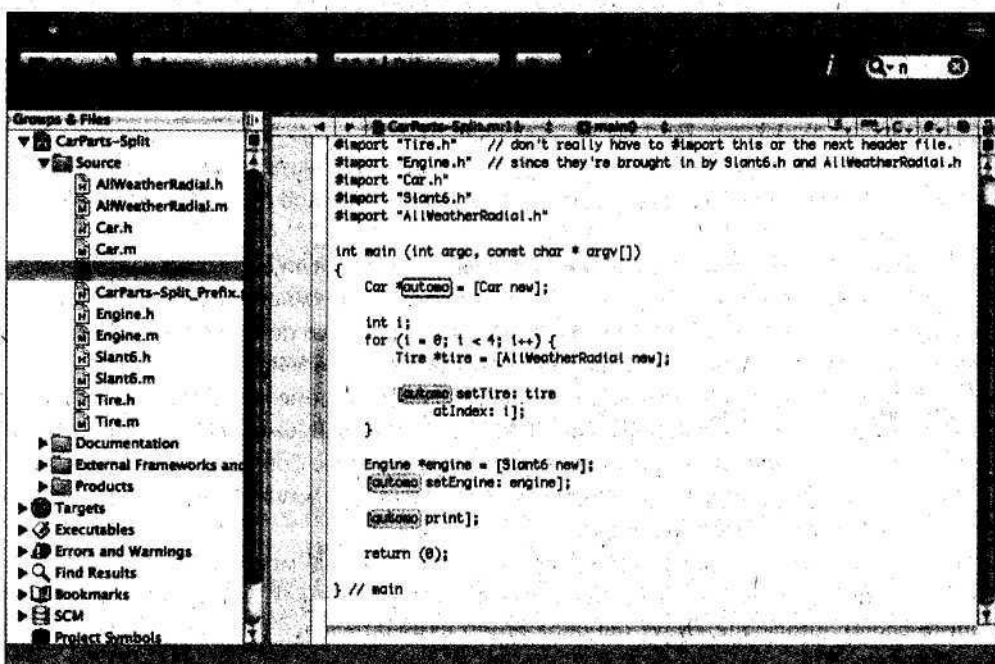


Рис. 7.11. Одновременное редактирование всего в области видимости

Когда вы закончите ввод, щелкните в любом месте окна редактирования, и вы выйдете из режима Edit all in Scope.

Иногда вам надо выполнить изменения, подобные описанным, но пункт меню Edit all in Scope оказывается отключен. Эта возможность тесно связана с выделением синтаксиса цветом в редакторе Xcode, так что если последнее отключено или не работает, то и Edit all in Scope может отказаться работать. Для исправления ситуации вернитесь к настройкам и включите выделение синтаксиса цветом.

Помните, как в прошлой главе мы постоянно упоминали термин “рефакторинг”? Это не просто загадочно звучащее слово — некоторый инструментарий рефакторинга встроен в Xcode. Одно из средств рефакторинга облегчает переименование класса — оно переименовывает не только сам класс, но и исходные файлы.

Давайте попробуем изменить все упоминания класса Car на Automobile. Откроем в редакторе файл Car.h и поместим точку ввода в слове Car. После этого следует выбрать пункт меню Edit⇒Refactor. Вы увидите диалоговое окно, подобное показанному на рис. 7.12, в котором вы вводите Automobile в качестве замены для Car.

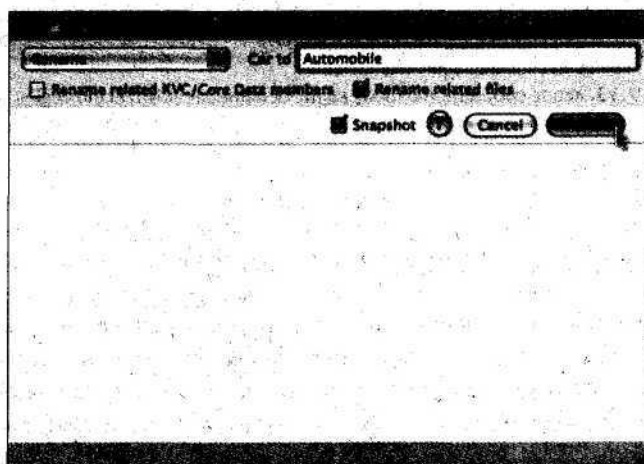


Рис. 7.12. Начало рефакторинга

Для безопасности следует отметить флажок Snapshot. Xcode определяет, что именно следует сделать, и после щелчка на кнопке Preview представляет вам план работы, как показано на рис. 7.13.

Как видите, Xcode планирует переименовать Car.h и Car.m в файлы с теми же расширениями, но именем Automobile. Вы можете щелкнуть на исходном файле, чтобы увидеть, какие изменения Xcode планирует внести в исходный текст. На рис. 7.13 видно, что Xcode собирается заменить Car на Automobile в директиве #import, а также в других местах, где использовано данное имя класса.

Печально, но при рефакторинге не переименовываются имена в комментариях, так что в комментариях в конце класса и в заголовочных файлах, генерируемых Xcode, а также в документирующих комментариях, которые вы могли написать самостоятельно, имена останутся теми же, и их придется исправлять вручную. Чтобы облегчить выполнение этой задачи, можно воспользоваться средством поиска и замены.

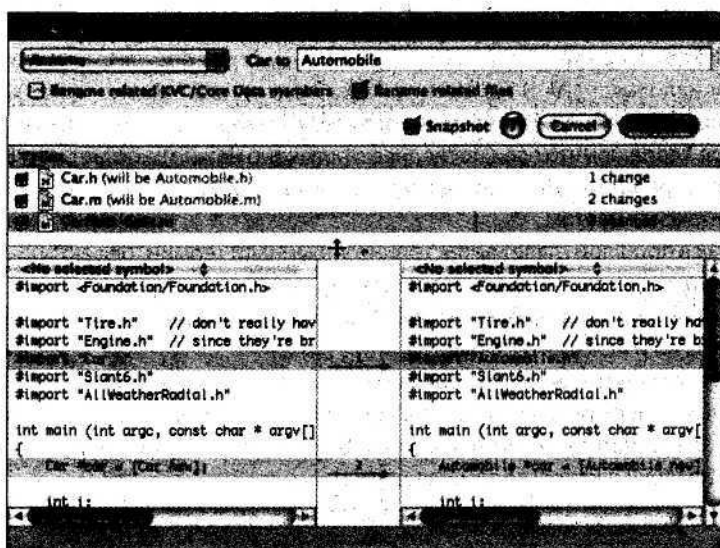


Рис. 7.13. Xcode сообщает о том, какие действия будут выполнены

Навигация по коду

Большинство исходных файлов имеет привычный жизненный цикл. Они создаются, в них добавляется большое количество кода. Затем они переходят в режим, когда количество дополнений и изменений примерно одинаково, а потом — в режим сопровождения, когда вы вынуждены читать большое количество кода перед тем, как внести небольшое дополнение или изменение. Наконец, когда класс переходит в стабильное состояние, вам приходится постоянно обращаться к его исходному тексту перед тем, как использовать его где-то в вашей программе. В этом разделе мы рассмотрим различные пути навигации по коду в процессе его жизненного цикла.

emacs — не Mac

На современных Mac доступен древний, известный еще с 1970-х годов, редактор emacs. Некоторые (включая одного из авторов этой книги, Марка Далримпла) пользуются им ежедневно. Здесь мы не будем подробно рассматривать emacs, опишем только некоторые из его комбинаций клавиш.

Комбинации клавиш emacs позволяют работать, не убирая руки от основной части клавиатуры. Так же, как одни предпочитают перемещать курсор при помощи клавиш со стрелками, чтобы не тянуться к мыши, так и другие пользователи emacs предпочитают использовать его комбинации клавиш для перемещения курсора, чтобы не тянуться к клавишам со стрелками. А вот и самое интересное: те же комбинации клавиш работают во всех текстовых полях Cocoa, включая не только Xcode, но и TextEdit, поле URL в Safari и т.д. Вот эти комбинации клавиш.

- <Control+F> — перемещение вперед (вправо) (то же, что и стрелка вправо).
- <Control+B> — перемещение назад (влево) (то же, что и стрелка влево).
- <Control+P> — перемещение на предыдущую строку (то же, что и стрелка вверх).
- <Control+N> — перемещение на следующую строку (то же, что и стрелка вниз).
- <Control+A> — перемещение в начало строки.
- <Control+E> — перемещение в конец строки.
- <Control+T> — обмен символов с двух сторон от курсора.
- <Control+D> — удаление символа справа от курсора.
- <Control+K> — удаление остатка строки.
- <Control+L> — центрирование точки ввода в окне (удобно при потере текстового курсора).

Если вы запишете эти комбинации не только в голову, но и в пальцы, — то сможете работать существенно быстрее (не только в Xcode).

Поиск

В правом верхнем углу окна проекта Xcode находится поле поиска, Xcode фильтрует содержимое браузера на основе того, что вы вводите в это поле. Для использования поля поиска выберите что-то в списке Groups & Files, например папку Source. Браузер покажет все исходные файлы. Теперь введите в поле, например, Car, и вы увидите список с отфильтрованными именами, как показано на рис. 7.14.

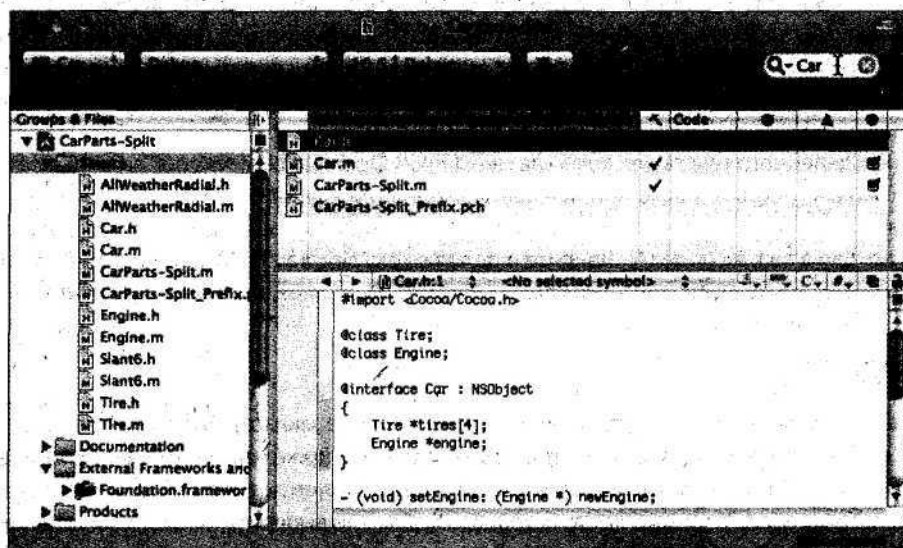


Рис. 7.14. Фильтрация файлов с car

Вы можете также просматривать используемую схему. Выберите Foundation Framework из папки External Frameworks and Libraries, и вы увидите список заголовочных файлов, предоставляемых схемой. Вы можете сократить его, вводя текст в поле поиска; например, на рис. 7.15 показан результат фильтрации для введенного в поле поиска слова `array`.

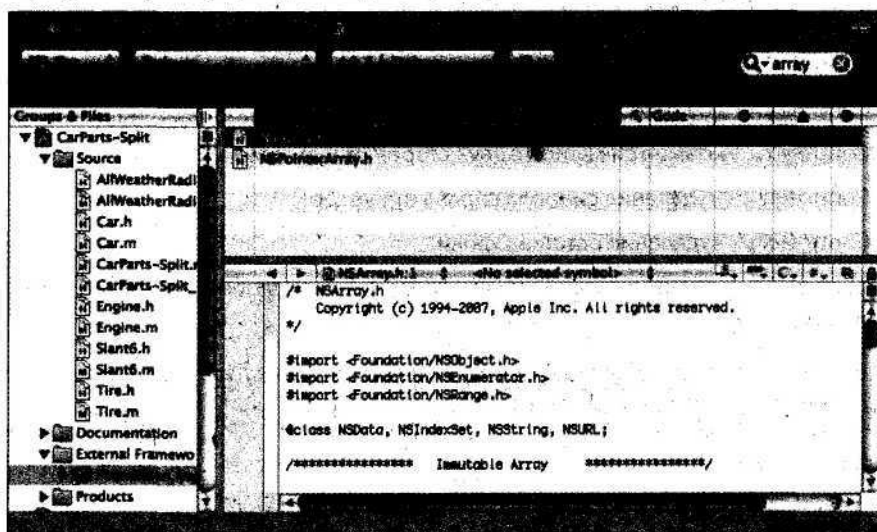


Рис. 7.15. Фильтрация файлов схемы

Сезам, откройся!

Вы просматриваете один из исходных файлов и в его начале видите директиву `#import`. Правда, было бы здорово, если бы вы могли быстро открыть соответствующий заголовочный файл без особой полировки стола мышью? И вы можете это сделать! Просто выделите имя файла (можете даже не выделять `.h`) и выберите команду меню `File → Open Quickly`. Xcode откроет для вас указанный заголовочный файл.

Если выделенного текста нет, команда меню `File → Open Quickly` откроет диалоговое окно, которое можно открыть также при помощи комбинации клавиш `<⌘+⌥+D>`. Это очень простое окно, содержащее поле поиска и таблицу и предоставляющее возможность очень быстрого поиска в вашем проекте. Введите в поле поиска `tire`, как показано на рис. 7.16, чтобы увидеть все файлы, связанные с колесами.

Закладки

Xcode позволяет вам размещать в вашем исходном тексте закладки. В вашем коде могут быть разные интересные места, например строка, в которую вы позже намерены внести изменения, или, быть может, вы хотите пометить определение очень важного класса. Для создания закладки поместите точку ввода в нужное место исходного файла или выберите область текста. Затем выберите пункт меню `Edit → Add to Bookmarks` или воспользуйтесь комбинацией клавиш по умолчанию `<⌘+D>` (такой же, как в Safari) и, после появления приглашения, введите имя закладки.

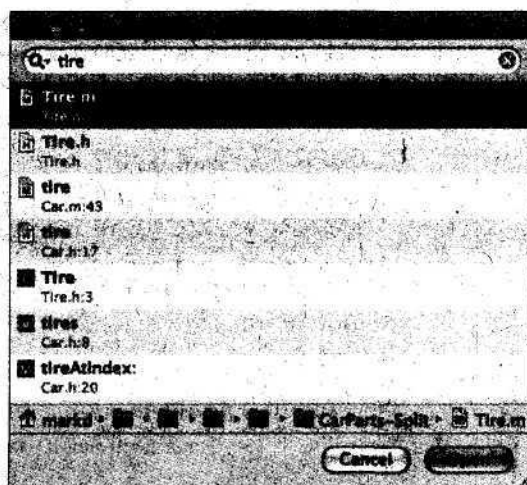


Рис. 7.16. Диалоговое окно Open Quickly

Ваши закладки вы можете увидеть в разделе Bookmarks в панели Groups & Files, как показано на рис. 7.17.

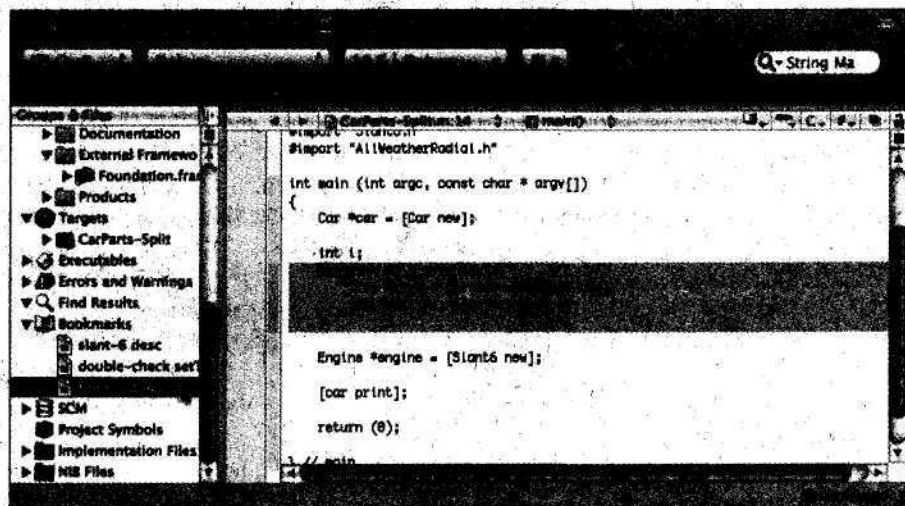


Рис. 7.17. Закладки Xcode

Сфокусируйтесь

Вы могли заметить два пустых столбца слева от вашего исходного текста. Более широкий столбец называется *отладочной полосой* (gutter), и о нем мы поговорим позже, когда будем рассматривать отладку. Более узкий столбец называется *фокусирующей лентой* (focus ribbon), и как следует из его имени, эта лента позволяет фокусировать внимание на разных частях вашего кода.

Обратите внимание на цвет ленты: чем глубже вложена часть кода, тем темнее остальные части ленты. Такая раскраска помогает легче справляться со сложным кодом. Вы можете провести указателем мыши над разными серыми частями ленты, чтобы выделить соответствующий блок кода, как показано на рис. 7.18.

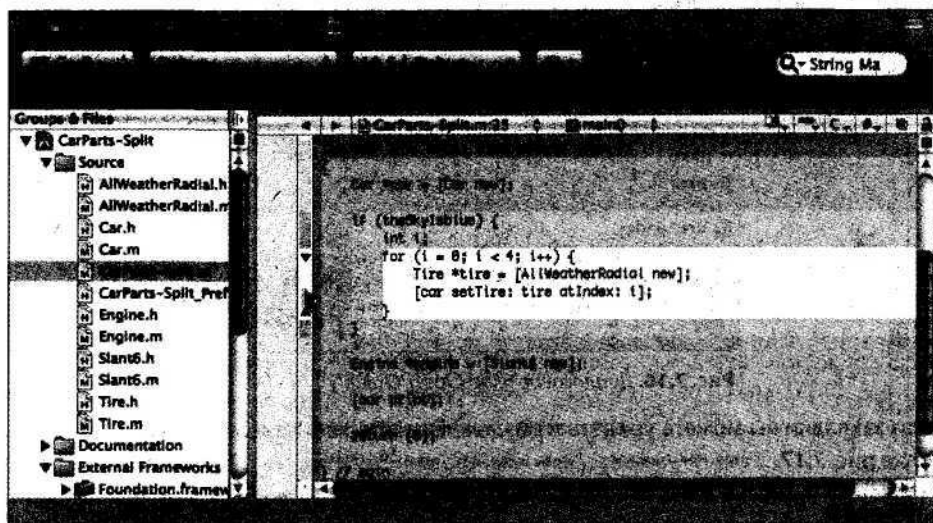


Рис. 7.18. Подсветка кода при помощи фокусирующей ленты

Вы можете щелкнуть на фокусирующей ленте для свертывания блока кода. Скажем, вы убеждены в корректности конструкции `if` и цикла `for`, показанных на рис. 7.18, и не хотите постоянно об них «спотыкаться» взглядом, а хотите, напротив, сфокусироваться на остальной части кода функции. Щелкните слева от конструкции `if`, и ее тело свернется, как показано на рис. 7.19.

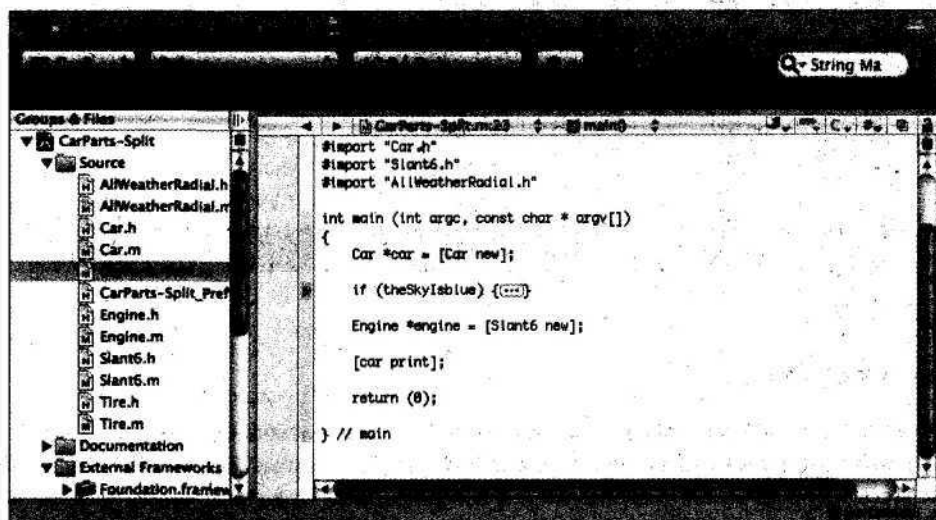


Рис. 7.19. Свернутый код

Панель навигации

В верхней части редактора кода находится небольшая лента с управляющими элементами (рис. 7.20), известная как *панель навигации* (navigation bar). Многие управляющие элементы, располагающиеся на ней, обеспечивают быстрый доступ к разным частям исходных файлов проекта.

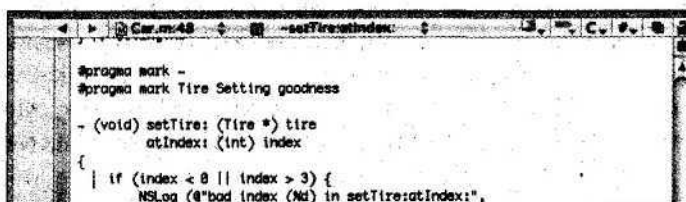


Рис. 7.20. Панель навигации

На панели (слева направо) располагаются кнопки перемещения вперед и назад по файлам, которые были открыты на протяжении текущей сессии редактирования (они работают подобно стрелкам в Safari). Затем идет выпадающее меню, которое показывает текущий файл (Car.m) и номер строки в файле, где находится точка ввода (48). Щелкнув на этом меню, вы увидите историю открытия ваших файлов и при желании можете открыть один из них в редакторе.

Затем идет меню функций. Оно указывает, что в настоящий момент точка ввода находится в методе -setTire:atIndex:. Если щелкнуть на меню, вы увидите все интересующие вас идентификаторы, как показано на рис. 7.21.

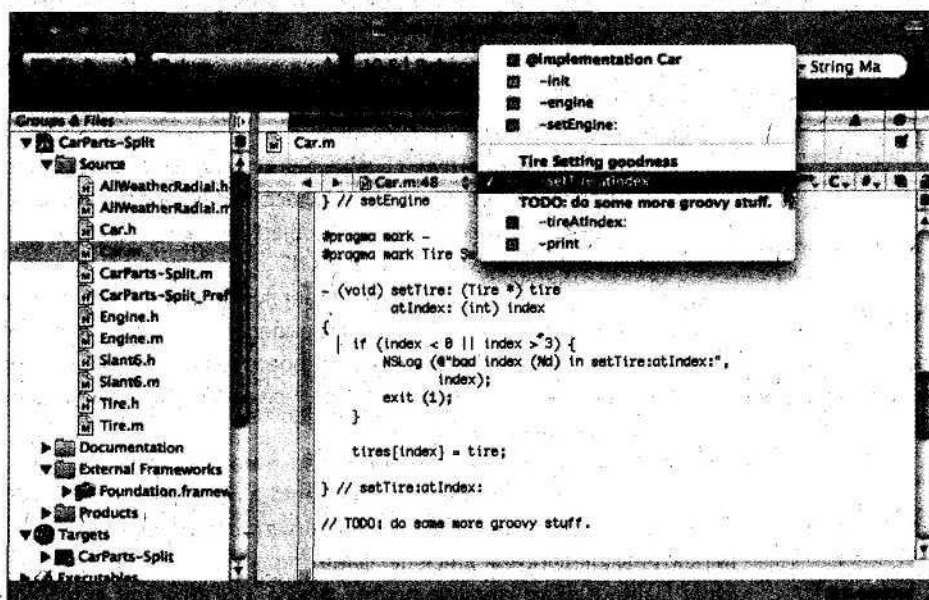


Рис. 7.21. Идентификаторы файла

Метод `-setTireAtIndex:` выделен, поскольку именно в нем находится курсор. Вы можете видеть другие методы выше и ниже, отсортированные в том порядке, в котором они встречаются в файле (однако их можно отсортировать и в алфавитном порядке). Обратите внимание на некоторые дополнительные пункты, такие как `Tire Setting goodness` и `TODD`. Откуда они взялись?

Вносить свои пункты в это меню можно несколькими способами. Один способ состоит в использовании строки `#pragma mark stuff` для размещения в меню строки `stuff`. Это очень удобно для добавления пунктов меню, которые могут читать другие программисты. Если внести строку `#pragma mark -` (знак минус), то это приведет к появлению в меню горизонтальной разделяющей линии. Кроме того, Xcode ищет комментарии, начинающиеся со слов `MARK:` (ведет себя так же, как и `#pragma mark`), `TODD:`, `FIXME:`, `!!!:` и `???:`, и также вносит их текст в меню функций. Все это — программистские сигналы типа “обрати на это внимание, перед тем как сдавать работу!”

ПРИМЕЧАНИЕ

Слово “прагма” с греческого обозначает “действие”. Директива `#pragma` представляет собой способ передачи информации или инструкций компилятору и редактору кода вне обычных строк кода Objective-C. Обычно прагмы игнорируются, но они могут иметь смысл для инструментария, применяющегося при разработке программного обеспечения. Если некий инструмент не понимает прагму, он должен смущенно улыбнуться и проигнорировать ее наличие, не генерируя никаких предупреждений или, тем более, сообщений об ошибках.

Очередной управляющий элемент, похожий на маленькую книгу, показывает все закладки текущего файла. Вы можете увидеть, что эта пиктограмма такая же, как и пиктограмма Bookmarks в панели Groups & Files.

Следующий управляющий элемент содержит все точки останова файла. О точках останова мы поговорим позже, при обсуждении отладки.

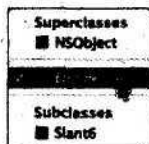


Рис. 7.22. Меню классов панели навигации

Меню `C` обеспечивает навигацию в иерархии классов. Если в редакторе кода выводится `Engine.m` и вы вызовете меню классов, то увидите что-то наподобие рис. 7.22.

На рис. 7.22 выделен пункт `Engine`. Как видите, Xcode знает о том, что `Slant6` — подкласс, а `NSObject` — надкласс. Выбор `Slant6` откроет файл `Slant6.m`, а выбор `NSObject` — файл `NSObject.h` из заголовочных файлов схемы Foundation. У нас нет доступа к `NSObject.m`, поскольку этот файл принадлежит Apple, так что Xcode делает лучшее, на что способен, и загружает заголовочный файл. Если вы выберете `NSObject`, что приводит к открытию заголовочного файла `NSObject.h`, а затем вновь откроете меню классов, то увидите подклассы `NSObject`. Их — неимоверное количество! Как видите, меню классов очень удобно для навигации по дереву наследования вашего кода.



Рис. 7.23. Меню включаемых файлов

Очередное меню, помеченное как `#`, — меню включаемых файлов; это меню в случае просмотра файла `Slant6.h` показано на рис. 7.23.

Из него видно, что `Slant6.h` импортирует или включает (с помощью директивы `#import` или `#include`) файл `Engine.h`. Xcode также знает, что файл `Slant6.h` включает два других файла, `CarParts-Split.m` и `Slant6.m`. Это меню представляет собой способ быстрого перемещения по зависимостям вашего проекта.

Перекрывающиеся квадраты приводят к открытию файла, парного текущему.

Последняя пиктограмма строки, имеющая вид замка, позволяет открыть файл в режиме только для чтения. Если вы просто просматриваете файл, его можно маркировать как файл только для чтения, чтобы случайно не внести в него изменения — например, если в этот момент на клавиатуру запрыгнет ваш кот.

Под замком, на вершине полосы прокрутки, находится кнопка разделения окна. При щелчке на ней окно делится пополам, позволяя одновременно увидеть два места в вашем файле, как показано на рис. 7.24. Вы можете изменять размеры частей окна, перетягивая разделяющую их полосу.

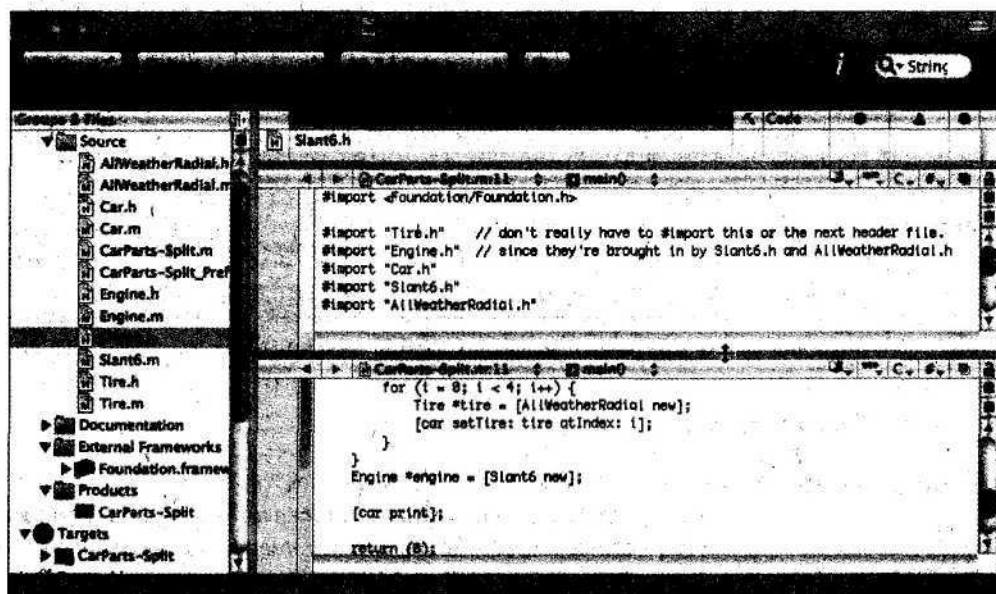


Рис. 7.24. Разделение редактора кода

У каждой части имеется своя панель навигации, которую можно использовать для назначения панелям разных файлов. Щелчок на разделителе при нажатой клавише <Option> переключает горизонтальное разделение на вертикальное и наоборот. Такое разделение можно выполнить в каждой части отдельно и в конечном счете довести ситуацию до абсурдного количества окон редактирования, как показано на рис. 7.25. Избавиться от такого мини-окна можно при помощи щелчка на прямоугольной кнопке под кнопкой разделения.

Получение информации

Переключения между вашим кодом и заголовочными файлами Cocoa — это, конечно, приятная возможность, но иногда требуется информация извне. К счастью, Xcode оснащен неисчерпаемыми залежами информации в документации и справочных материалах.

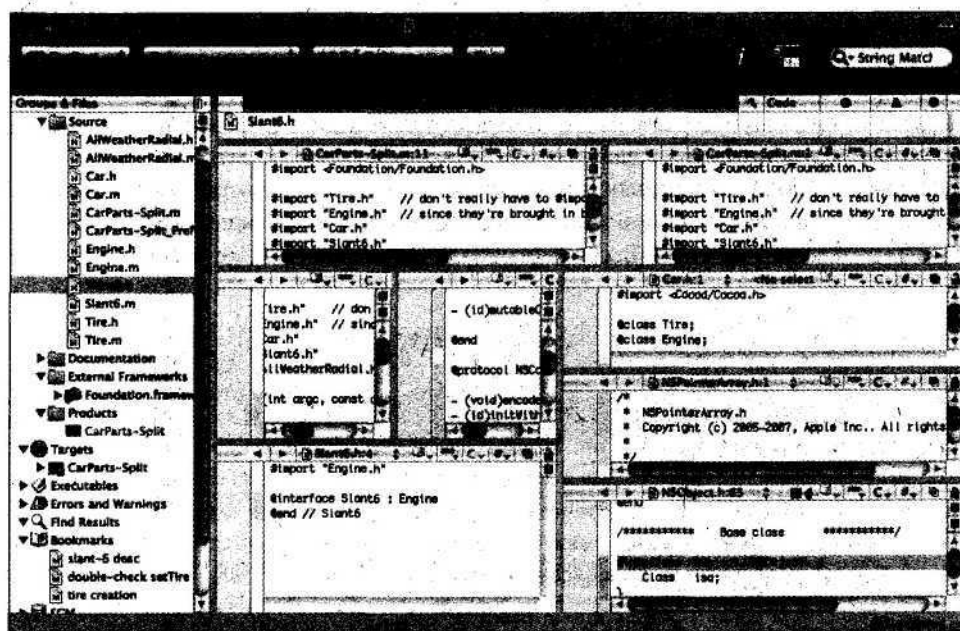


Рис. 7.25. Разделение одного окна редактирования на множество мини-окон

Research Assistance

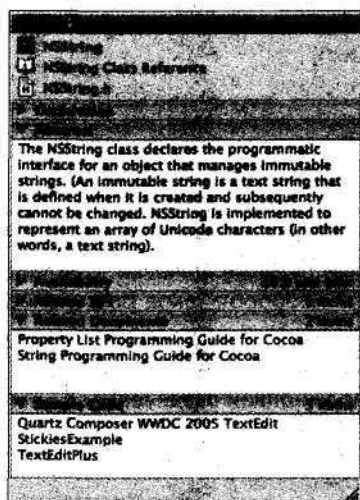


Рис. 7.26. Окно Research Assistant

Маленькое незакрепленное окно Research Assistant содержит информацию, которая автоматически обновляется в соответствии с тем, что именно вы делаете в Xcode. Открыть это окно можно при помощи команды меню Help → Show Research Assistant.

Пусть, например, ваша точка ввода находится в слове `NSString`. Окно `Research Assistant` при этом будет иметь вид, показанный на рис. 7.26.

Имеется огромное количество доступной информации. Первые два пункта позволяют вывести информацию о классе `NSString` в окне документации. Пункт `NSString.h` открывает указанный заголовочный файл в окне редактора.

Панель Abstract описывает класс. Если ваш курсор находится в середине вызова метода, то в панели будет описываться метод и связанные с ним вызовы. Имеются и указатели на документацию более высокого уровня и примеры кода, использующие NSString.

Research Assistant особенно полезен при установке настроек построения проекта. Выберите в панели информации о проекте пункт настроек и прочтите подробную информацию о них в окне Research Assistant.

Документирование

Если вас интересует официальная документация Apple по API, то очень быстрый способ добраться до нее — это при нажатой клавише <Option> дважды щелкнуть на интересующем вас идентификаторе (это сокращенный вариант вызова поиска данного идентификатора в документации).

Пусть у вас есть код типа `[someString UTF8String]`, который конвертирует `someString` (типа `NSString`) в строку C в Unicode. Если вы дважды щелкнете при нажатой клавише <Option> на `UTF8String`, браузер документации предоставит соответствующую информацию, как показано на рис. 7.27.

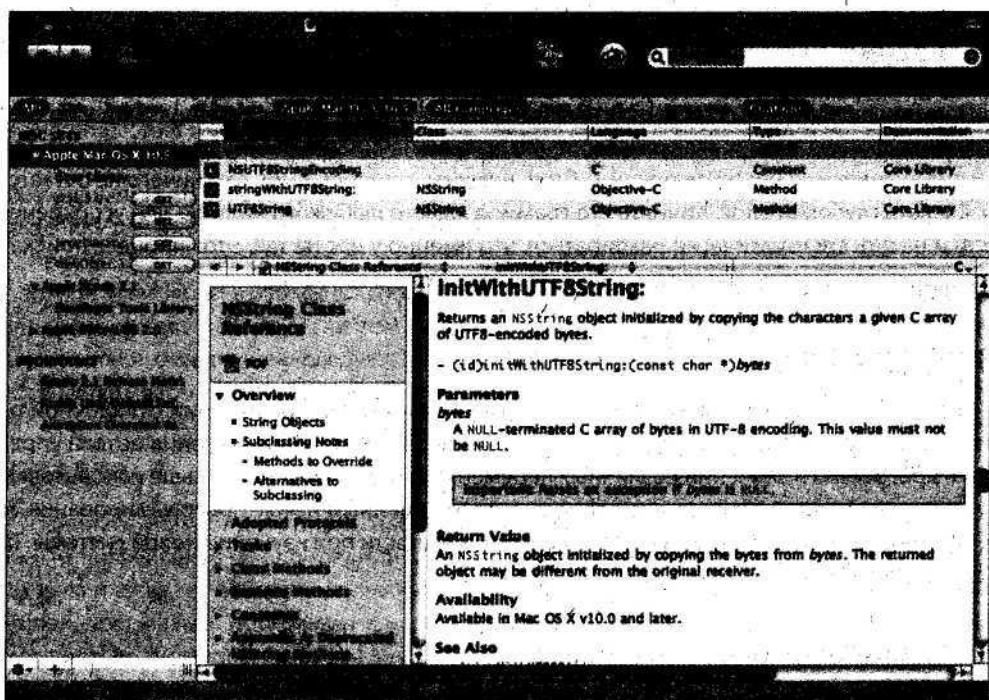


Рис. 7.27. Окно документации Xcode

В этом окне упаковано огромное количество информации. В верхнем правом углу находится поле поиска, в которое введено слово `UTF8String`. Лента с кнопками под панелью инструментов позволяет вам указать, в каком наборе документации проводится поиск. Вы можете просмотреть всю документацию или только документацию по Mac OS X или iPhone. Вы можете ограничить поиск языком программирования — например, если вы редактируете сценарий JavaScript, то вам нет смысла выполнять поиск в информации о C++.

Ниже ленты с кнопками располагается панель с наборами документации и закладок. Вы можете добавить закладки к определенным блокам документации при помощи комбинации клавиш <⌘+D>. Эти закладки отличаются от закладок, которые вы можете разместить в ва-

шем коде. Закладки в документации глобальны для Xcode и не ограничиваются определенным проектом.

В правой части окна сверху браузер показывает результаты поиска. На рис. 7.27 показаны три метода и константа. Наиболее интересен последний пункт, точно соответствующий нашему вызову.

Ниже браузер выводит документы. Это окно слишком мало для чтения документов, но очень легко открыть документацию в отдельном окне Xcode или даже в веб-браузере. Если вы хотите открыть документ в новом окне, щелкните правой кнопкой мыши (или левой при нажатой клавише <Control>) на области документации, чтобы открыть соответствующее меню.

Отладка

Произошла ошибка. Это обычное явление жизни — когда ошибки расползаются по всем щелям программы, как тараканы — особенно если вы начинаете работу на новом языке программирования и на новой платформе. Столкнувшись с проблемами, вам ничего не остается, как сделать глубокий вдох, принять сто граммов вашего любимого напитка для успокоения, сесть и начать систематически разбираться, что именно у вас не так. Этот процесс поиска и устранения программных ошибок именуется *отладкой* (debugging).

Проще некуда

Простейший и древнейший способ отладки заключается в применении отладочной печати, т.е. в использовании инструкций вывода (типа NSLog) для отслеживания потока управления программы и значений различных переменных. Вероятно, вы и раньше использовали эту методику отладки. Многие считают этот способ каменным веком программирования, но он может быть, на удивление, эффективным, в особенности при изучении новой системы. Так что не обращайтесь внимания на этих злопыхателей.

Отладчик Xcode

В дополнение ко всему остальному, в Xcode имеется отладчик. *Отладчик* — это программа, которая располагается между вашей программой и операционной системой и может прерывать вашу программу, позволяя при этом просматривать данные программы и даже изменять их. По окончании работы с данными вы можете возобновить выполнение и посмотреть, что получилось. Вы можете также пошагово проходить код, выполняя строку программы за строкой, и точно видеть, что делает с данными ваша программа.

В Xcode имеется несколько мест, где вы можете использовать отладчик. Первое из них — прямо в вашем текстовом редакторе: щелкнув на отладочной полосе, вы можете установить точку останова. *Точка останова* — это место, где отладчик должен остановить вашу программу и посмотреть, что получилось.

В Xcode имеется мини-отладчик, который представляет собой незакрепленное окно с некоторыми основными управляющими элементами, позволяющими решать простейшие от-

ладочные задачи. В Xcode есть также окно отладки, в котором выводится масса информации, и консоль отладки, в которой вы можете вводить команды для отладчика.

ПРИМЕЧАНИЕ

Xcode использует отладчик GDB, который является частью проекта GNU. GDB доступен для множества разных платформ; при желании вы можете запустить его из командной строки. GDB очень хорошо документирован, а в Интернете можно найти несколько учебников по работе с ним.

Мы рассмотрим отладку в текстовом редакторе Xcode. Но если вы хотите всерьез научиться работе с отладчиком, то должны поработать и в других режимах отладки.

Настройка

Если вы планируете отлаживать программу, ее следует собирать с включенной конфигурацией сборки Debug. Проверить конфигурацию можно в выпадающем меню Active Build Configuration в панели инструментов Xcode. Конфигурация Debug указывает компилятору на необходимость использования дополнительной отладочной информации, которая нужна отладчику для определения местоположения в программе.

Кроме того, убедитесь, что вы выполняете программу в отладчике. Имеется несколько способов запуска программы в Xcode. Воспользуйтесь командой меню Run⇒Run или комбинацией клавиш `<⌘+R>`, если вам надо запустить программу без отладчика. Чтобы воспользоваться отладчиком, следует выбрать либо команду меню Run⇒Go (Debug), либо Run⇒Debug, либо комбинацию клавиш `<⌘+Y>`.

Отладка

Начинать учиться работать с отладчиком лучше с версии с графическим пользовательским интерфейсом, а не с командной строки. Графическая программа большей частью сидит без дела и ожидает действий пользователя, что дает массу времени на поиск необходимых кнопок, прерывания программы и исследования данных. Программа командной строки не дает таких возможностей, как графическая. Так что приступим к изучению графического отладчика, и начнем с установки точки останова в функции `main()`. Мы будем учиться отладке на примере программы 06.01 CarParts-split из предыдущей главы.

Откройте файл CarParts-Split.m и щелкните на отладочной полосе, которая, как вы уже видели ранее, представляет собой широкую полосу слева от фокусирующей ленты. Вы должны получить голубую стрелку, указывающую на новую точку останова (рис. 7.28).

Удалить точку останова можно перетягиванием ее с отладочной полосы, а отключить — при помощи щелчка на ней. Контекстное меню точки останова содержит массу пунктов, которыми можно воспользоваться для настройки.

Теперь воспользуйтесь командой меню Run⇒Debug для запуска вашей программы. Ваша программа должна остановиться в точке останова, как показано на рис. 7.29. Обратите внимание на красную стрелку, указывающую на строку кода. Это знак, который гласит: "Вы — здесь".

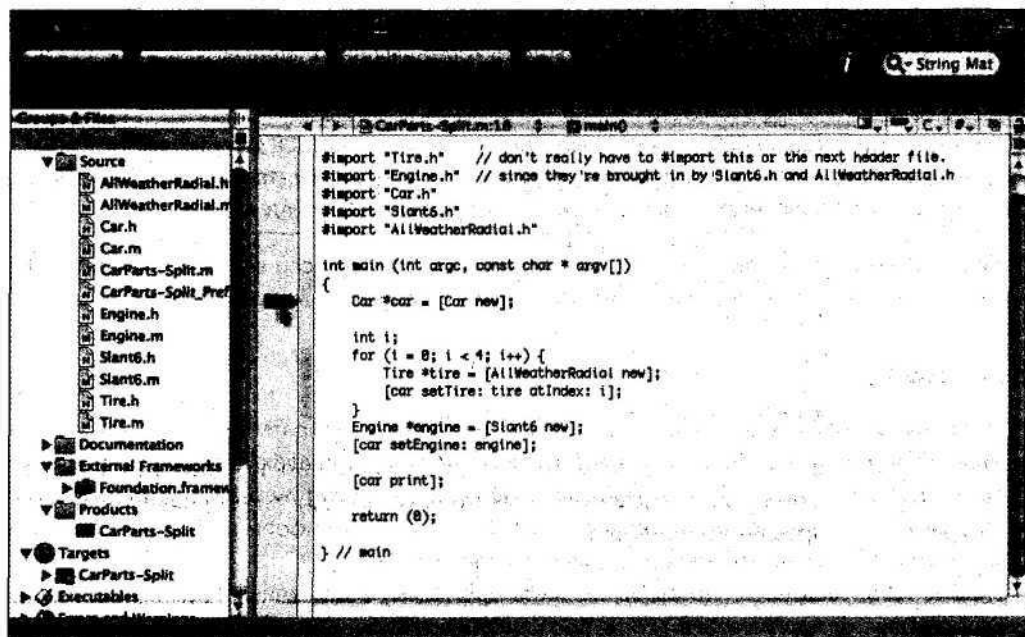


Рис. 7.28. Установка точки останова

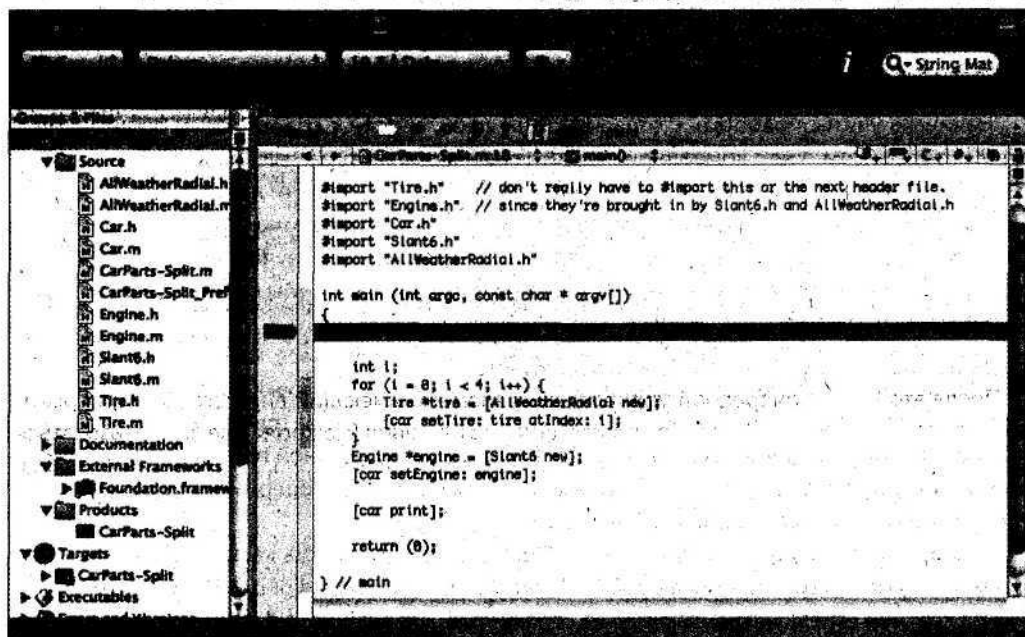


Рис. 7.29. Вы — здесь

Строка состояния в нижней части окна Xcode гласит: GDB: Stopped at breakpoint... При этом над полосой навигации появляется новая полоса с управляющими элементами, показанная на рис. 7.30.

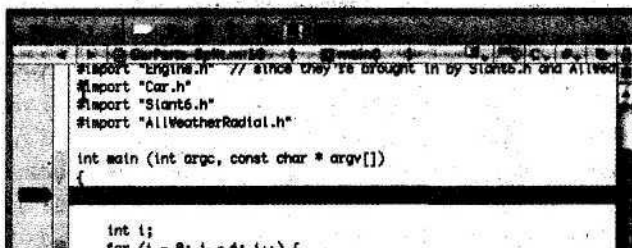


Рис. 7.30. Управляющие элементы отладки

Первым слева идет выпадающий список, позволяющий выбрать интересующий вас поток. Мы пока что не хотим забивать вам голову многопоточным программированием, так что игнорируйте этот управляющий элемент.

ПРИМЕЧАНИЕ

Многопоточное программирование — это программирование с использованием нескольких одновременно работающих потоков выполнения; написание корректно работающей многопоточной программы — очень сложное дело. При этом не редкость ошибки, которые крайне сложно обнаружить. Если кто-то скажет вам, что многопоточное программирование — простое занятие, этот человек либо лжец, либо пытается устроиться к вам на работу.

Следующий управляющий элемент выглядит так же, как стрелка точки останова, и используется для включения и отключения всех точек останова. В какой-то момент вы можете решить «ну, кажется, все ошибки найдены и исправлены», но вместо удаления всех точек останова вы можете просто деактивировать их и выполнить программу. Если выяснится, что ошибки все же остались, вы сможете просто включить их снова и продолжить отладку программы.

Затем идет четыре управляющих элемента, позволяющих управлять тем, что будет дальше, после останова. Первый, напоминающий кнопку проигрывания на CD-плеере, является кнопкой продолжения (можно также воспользоваться комбинацией клавиш <⌘+↵+P>). После щелчка на этой кнопке программа продолжает работу — до новой точки останова или завершения программы (штатного или аварийного).

После этого идет очередной управляющий элемент, имеющий вид чего-то, перепрыгивающего через точку, — это кнопка пошагового выполнения (можно также воспользоваться комбинацией клавиш <⌘+↵+O>). При щелчке на этой кнопке выполняется одна строка кода, после чего управление вновь возвращается вам. Если вы щелкнете на этой кнопке три раза, то стрелка «Вы — здесь» переместится к вызову `-setTire:atIndex`, как показано на рис. 7.31.

Очередная кнопка, имеющая вид стрелки, указывающей вниз на точку, выполняет функцию пошагового выполнения с заходом в функции (можно также воспользоваться комбинацией клавиш <⌘+↵+I>). Если у вас есть исходный текст текущих функции или метода, то Xcode войдет в эту функцию, выведет ее код и установит стрелку «Вы — здесь» на начало кода функции, как показано на рис. 7.32.

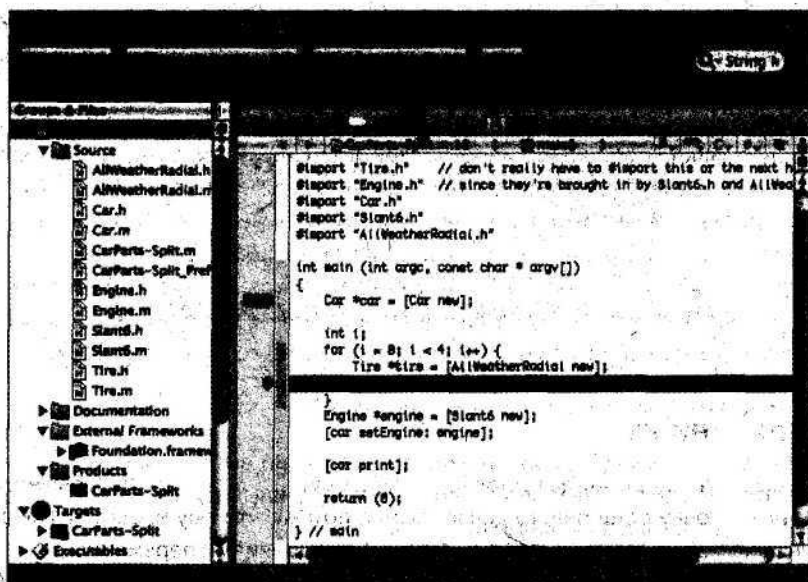


Рис. 7.31. Пошаговое выполнение

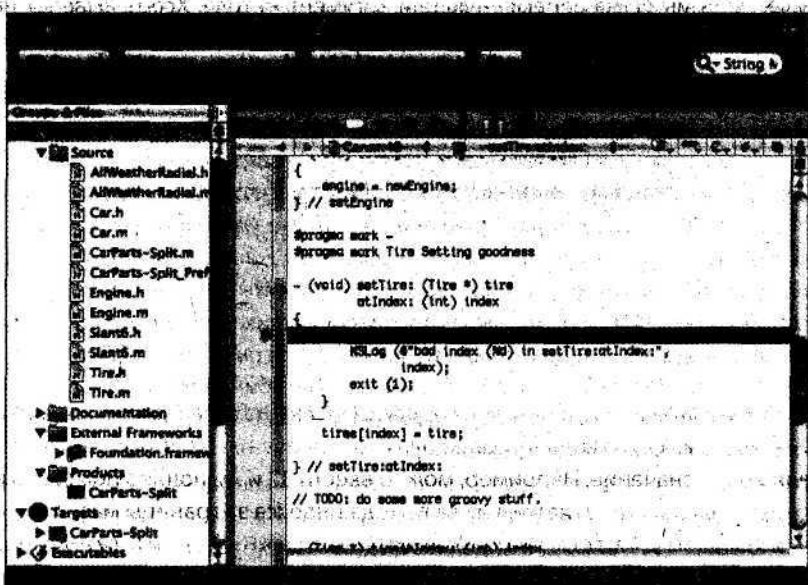


Рис. 7.32. Пошаговый вход в метод

Последняя кнопка — выход из функции (можно также воспользоваться комбинацией клавиш $\text{⌘} + \text{⏏} + \text{T}$) — приводит к завершению текущей функции и передаче вам управления на следующей за вызовом строке вызывающей функции. Если вы намерены продолжать работу с нами, пока что не используйте эту кнопку.

Завершим нашу экскурсию рассмотрением следующей кнопки (на которой нарисован баллончик с аэрозолем), которая вызывает окно отладки Xcode, и кнопки, вызывающей консоль GDB, где вы можете обратиться к отладчику непосредственно.

Последний управляющий элемент — выпадающее меню, показывающее *стек вызовов*, который представляет собой не что иное, как текущее множество активных функций. Если A вызывает B, а B вызывает C, C рассматривается как вершина стека, а B и A располагаются под C. Если вы откроете стек вызовов в настоящий момент, то увидите — [Car setTire:atIndex:], за которым идет main. Это означает, что main вызывает setTire:atIndex:. В более сложных программах стек вызовов может содержать десятки элементов. Иногда при отладке требуется ответить на вопрос «как этот код оказался вызван?». Просматривая стек вызовов, вы можете увидеть, кто кого вызывал и как программа дошла до нынешнего состояния.

Просмотр данных

Чем бы заняться теперь, когда выполнение программы остановилось? Обычно, когда кто-то устанавливает точку останова, то делает это не потому, что ему нравится, как это звучит, а чтобы ознакомиться с *состоянием программы*, т.е. значениями переменных.

У Xcode имеются *подсказки о данных* (datatips), аналогичные подсказкам, сообщающим о том, что делает та или иная кнопка на панели инструментов. Если в редакторе Xcode вы наведете указатель мыши на переменную или аргумент метода, Xcode выведет небольшое окно, в котором будет показано значение переменной (рис. 7.33).

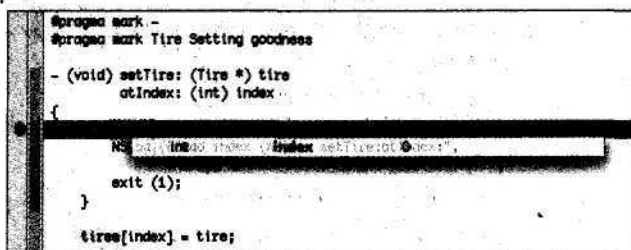


Рис. 7.33. Подсказки о данных

На рис. 7.33 указатель мыши наведен на index. Подсказка о данных показывает, то значение переменной, как мы и ожидали, равно нулю. Это значение можно изменить, щелкнув на нуле и введя новое значение. Например, можно ввести 37 и, выполнив пошагово несколько команд, увидеть выход из программы из-за выхода индекса за границы массива.

Пока вы все еще находитесь в цикле, наведите указатель на tires, и вы увидите массив. Переместите мышь вниз, на стрелку, и подсказка раскроется, показывая вам все четыре колеса. Теперь переместите указатель на первое колесо, и Xcode покажет вам его «внутренности». В наших колесах нет переменных экземпляра, так что вы увидите не очень много. Но если у класса есть переменные экземпляра, они будут выведены и их можно будет редактировать. Результаты описанных перемещений указателя мыши показаны на рис. 7.34.

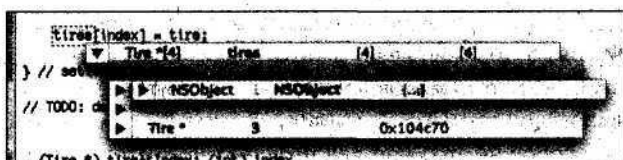


Рис. 7.34. Погружение в данные программы

На этом завершим беглое знакомство с отладчиком Xcode. Этой информации, а также вашего труда и большого количества времени должно быть достаточно для того, чтобы помочь вам отладить все, что требует отладки, и, тем более, то, что и так отлично работает!

Шпаргалка

В этой главе упомянута масса комбинаций клавиш. Для удобства мы собрали их все в одну таблицу — табл. 7.1. Если это не библиотечная книга, то стоит подумать о том, чтобы аккуратно вырвать эту страницу из книги и поместить так, чтобы она всегда была перед глазами.

Таблица 7.1. Комбинации клавиш Xcode

Комбинация клавиш	Описание
<⌘+⌘+E>	Раскрыть редактор
<⌘+[>	Сдвинуть блок кода влево
<⌘+]>	Сдвинуть блок кода вправо
<Табуляция>	Принять автозавершение
<Esc>	Показать меню автозавершения
<Control+.>	Прямой проход по автозавершениям
<Control+Shift+.>	Обратный проход по автозавершениям
<Control+/?>	Переход к следующему заполнителю автозавершения
<Command+Control+S>	Создание снимка
<Control+F>	Перемещение курсора вперед
<Control+B>	Перемещение курсора назад
<Control+P>	Перемещение курсора на предыдущую строку
<Control+N>	Перемещение курсора на следующую строку
<Control+A>	Перемещение курсора в начало строки
<Control+E>	Перемещение курсора в конец строки
<Control+T>	Обмен символов, соседних с курсором
<Control+D>	Удаление символа справа от курсора
<Control+K>	Удаление строки
<Control+L>	Отцентрировать курсор в текстовом редакторе
<⌘+⌘+D>	Открытие окна Open Quickly
<⌘+⌘+↑>	Открытие парного файла

Комбинация клавиш	Описание
<⌘+D>	Добавление закладки
<Option+двойной щелчок>	Поиск в документации
<⌘+Y>	Запуск программы в отладчике
<⌘+⌘+P>	Продолжение (в отладчике)
<⌘+⌘+O>	Пошаговое выполнение
<⌘+⌘+I>	Пошаговое выполнение со входом в функцию
<⌘+⌘+T>	Выполнение до выхода из функции

Резюме

Эта глава переполнена информацией, причем здесь мы практически ни слова не говорили об Objective-C. Почему? Так же как столяр должен разбираться не только в дереве, но и в инструментах, так и программист на Objective-C должен знать не только язык, но и инструментарий программиста. Чем лучше вы будете ориентироваться в том, как написать и отладить исходный текст в Xcode, тем больше времени вы сможете посвятить работе непосредственно над программой.

Материал, который вы встретите в следующей главе, представляет собой введение в некоторые классы Cocoa. Вам должно понравиться!

Краткий обзор Foundation Kit

Вы уже знаете, что Objective-C — превосходный язык, и даже знакомы с его основными возможностями. Теперь пришло время подробнее взглянуть на уже использовавшуюся нами схему Cocoa Foundation. Хотя, строго говоря, эта схема — часть Cocoa, и не встроена в Objective-C, она настолько важна, что мы рассмотрим ее в данной книге.

Как вы видели в главе 2, "Расширения C", Cocoa состоит из двух различных схем: Foundation и Application Kit. Application Kit включает все объекты пользовательского интерфейса и высокоуровневые классы. Попробовать AppKit (как его называют крутые программисты) на вкус нам удастся только в главе 14, "Введение в AppKit".

Схема Cocoa Foundation содержит набор полезных низкоуровневых классов и типов, ориентированных на работу с данными. Мы рассмотрим ряд таких классов, в частности, NSString, NSArray, NSDictionary и NSNumber. Foundation включает более сотни классов, полностью ознакомиться с которыми можно в документации, устанавливаемой с Xcode. Эту документацию можно найти в файле `/Developer/ADC Reference Library/documentation/index.html`.

Перед тем как продолжить работу, мы должны сделать замечание о проектах этой и прочих глав до конца книги. Мы будем и далее создавать проекты Foundation tool и оставлять заготовку кода, предлагаемую при создании нового файла.

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

```
int main (int argc, const char * argv[]) {  
    NSAutoreleasePool * pool  
    = [[NSAutoreleasePool alloc] init];
```

```
// Место для вставки кода...  
NSLog(@"Hello, World!");
```

```
[pool drain];
return 0;
```

```
}
```

Давайте рассмотрим этот код. Функция `main()` начинает с создания (посредством метода `alloc`) и инициализации (посредством метода `init`) `NSAutoreleasePool`. Это озеро (перевод слова `pool`) осушается (перевод слова `drain`) в конце работы программы. Все это относится к подсистеме управления памятью Сосоа, о которой мы поговорим в следующей главе. Пока же просто примите этот исходный текст как есть. Если вы выбросите его, ничего страшного с вами не случится, но вы будете получать странные сообщения при работе вашей программы.

Некоторые полезные инструменты

Перед тем как закопаться в реальные классы Сосоа, давайте взглянем на некоторые структуры, которые Сосоа предоставляет для нас.

Диапазон

Первая структура — `NSRange`.

```
typedef struct _NSRange {
    unsigned int location;
    unsigned int length;
} NSRange;
```

Эта структура используется для представления диапазона, обычно диапазона символов в строке или диапазона элементов массива. Поле `location` хранит начальную позицию диапазона, а `length` — количество элементов. Так, в строке "Objective-C отличный язык программирования" слово "язык" можно описать диапазоном, у которого `location` равно 21, а `length` — 4. Поле `location` может иметь значение `NSNotFound`, которое указывает, что диапазон указывает "в никуда", вероятно, из-за неинициализированности.

Создать новый диапазон `NSRange` можно тремя разными способами. Во-первых, можно присвоить значения полям непосредственно.

```
NSRange range;
range.location = 17;
range.length = 4;
```

Во-вторых, можно воспользоваться механизмом присваивания данных структуре в С.

```
NSRange range = { 17, 4 };
```

В-третьих, Сосоа предоставляет удобную функцию `NSMakeRange()`.

```
NSRange range = NSMakeRange(17, 4);
```

В `NSMakeRange()` особенно приятно то, что ее можно использовать везде, где можно использовать функцию, в частности, в вызове метода в качестве аргумента.

```
[anObject flarbulateWithRange: NSMakeRange(13, 15)];
```

Геометрические типы

Зачастую при работе нужны типы, работающие с геометрическими концепциями, такие как `NSPoint` и `NSSize`. Тип `NSPoint` представляет точку плоскости с декартовыми координатами (x,y).

```
typedef struct _NSPoint {
    float x;
    float y;
} NSPoint;
```

Структура `NSSize` хранит ширину и высоту.

```
typedef struct _NSSize {
    float width;
    float height;
} NSSize;
```

В семействе программ `Shapes` вместо нашей структуры для описания прямоугольника мы могли бы использовать `NSPoint` и `NSSize`, но в тот момент мы хотели, чтобы наши программы были как можно проще. Сосоа предоставляет тип прямоугольника, который представляет собой композицию точки и размера.

```
typedef struct _NSRect {
    NSPoint origin;
    NSSize size;
} NSRect;
```

Сосоа также предоставляет функции для создания объектов этих классов: `NSMakePoint()`, `NSMakeSize()` и `NSMakeRect()`.

ПРИМЕЧАНИЕ

Почему здесь использованы структуры `C`, а не полноценные объекты? Из соображений производительности. Программа, в особенности программа с графическим пользовательским интерфейсом, использует при работе массу временных точек, размеров и прямоугольников. Вспомните, что все объекты `Objective-C` создаются динамически, а динамическое выделение памяти — операция относительно дорогая, требующая определенного количества времени. Сделав эти типы структурами, разработчики смогли существенно снизить накладные расходы по их использованию.

Строки

Первый реальный класс, который мы встречаем в нашей экскурсии, — это `NSString`, класс Сосоа для работы со строками. Строка представляет собой просто последовательность символов. Поскольку компьютеры постоянно взаимодействуют с людьми, получая от них и выводя для них информацию, очень важно иметь возможность хранения и обработки "человеческого" текста. Вы уже встречались ранее с типом `NSString` и со специальными литералами `NSString` со знаком `@` перед строкой в двойных кавычках наподобие `@ "Привет!"`. Эти литералы являются такими же объектами `NSString`, как и создаваемые программно.

Если вы уже имели опыт работы со строками на C (типа описанной в книге Дэйва Марка (Dave Mark) *Learn C on the Mac*), то знаете, какая это головная боль. В языке программирования C строки реализованы как простые массивы символов, в которых конец строки маркируется нулевым байтом. Класс Cocoa NSString имеет набор встроенных методов, которые делают работу со строками существенно легче.

Создание строки

Вы знаете, что функции наподобие printf() и NSLog() получают в качестве аргументов строку формата и некоторые аргументы и создают форматированный вывод. Метод stringWithFormat: класса NSString создает новую строку NSString таким же способом — при помощи строки формата и аргументов.

```
+ (id) stringWithFormat: (NSString *) format, ...;
```

Новая строка создается примерно так.

```
NSString *height;
height = [NSString stringWithFormat:
    @"Ваш рост %d футов, %d дюймов", 5, 11];
```

В результате получится строка "Ваш рост 5 футов, 11 дюймов".

Методы класса

В объявлении метода stringWithFormat: имеется пара интересных моментов. Первый — наличие в конце троеточия (...), которое говорит вам (и компилятору), что этот метод может принимать любое количество дополнительных аргументов, разделенных запятыми, как и функции printf() и NSLog().

Второй, странный и весьма важный факт заключается в том, что объявление stringWithFormat: начинается со знака "плюс" (+). Что он тут делает? Когда система времени выполнения Objective-C создает класс, она создает *объект класса*, который представляет данный класс. Этот объект класса содержит указатель на надкласс, имя класса и список методов класса. Объект класса содержит также поле типа long, которое указывает размер в байтах вновь создаваемых экземпляров объектов класса.

Когда вы объявляете метод со знаком плюс, помечаете его как *метод класса*. Такой метод принадлежит объекту класса (а не экземпляру объекта класса) и обычно используется для создания новых экземпляров. Методы класса используются для создания новых объектов и называются *методами-фабриками* (или *фабричными методами*).

Метод stringWithFormat: является таким фабричным методом. Он создает новый объект на основе переданных ему аргументов. Использование stringWithFormat: для создания новых строк гораздо проще, чем начать с пустой строки и создавать все индивидуальные ее компоненты.

Методы класса могут также использоваться для доступа к глобальным данным. Класс NSColor из AppKit имеет несколько методов класса, названия которых соответствуют разным

цветам, такие как `redColor` и `blueColor`. Для получения синего цвета, например, можно написать нечто наподобие следующего.

```
NSColor *haveTheBlues = [NSColor blueColor];
```

Подавляющее большинство создаваемых методов — это методы экземпляра, объявленные с предшествующим знаком минус (-). Такие методы работают с конкретным экземпляром объекта, таким как цвет окружности или атмосферное давление в камере. Если же метод выполняет более обобщенные действия, такие как создание экземпляра объекта или доступ к некоторым глобальным данным класса, вы можете объявить его как метод-класса, применив предшествующий знак плюс (+).

Размер

Еще один удобный метод `NSString` (метод экземпляра) — это метод `length`, который возвращает количество символов в строке.

```
– (unsigned int) length;
```

Он используется так:

```
unsigned int length = [height length];
```

или в выражениях наподобие следующего.

```
if ([height length] > 35) {
    NSLog(@"Высок, высок!");
}
```

ПРИМЕЧАНИЕ

Указанный метод `NSString` корректно работает со строками на разных языках, содержащими, например, русские, китайские или японские символы, так как за кулисами используется кодировка `Unicode`. Работа с такими строками на обычном C особенно трудна, так как отдельный символ при этом может иметь размер более одного байта. Это значит, что функция типа `strlen()`, просто подсчитывающая байты, может возвращать неверное значение.

Сравнение строк

Частой операцией со строками является их сравнение. Иногда надо убедиться, что две строки равны (например, имеет ли строка `username` значение `'markd'?`), иногда надо выяснить относительное положение строк при их сортировке. Класс `NSString` предоставляет несколько различных функций сравнения.

Метод `isEqualToString:` сравнивает объект-получатель (которому посылается это сообщение) со строкой, переданной в качестве аргумента. Метод `isEqualToString:` возвращает значение типа `BOOL` (YES или NO), указывающее, одинаково ли содержимое этих двух строк. Метод объявлен следующим образом.

```
– (BOOL) isEqualToString: (NSString *) aString;
```


А вот как можно его использовать.

```
NSString *thing1 = @"hello 5";
NSString *thing2;
thing2 = [NSString stringWithFormat: @"hello %d", 5];
```

```
if ([thing1 isEqualToString: thing2]) {
    NSLog(@"Строки одинаковы!");
}
```

Для сравнения строк можно применить метод `compare:`, объявленный следующим образом.

```
– (NSComparisonResult) compare: (NSString *) string;
```

Метод `compare:` выполняет посимвольное сравнение получающего объекта с переданной строкой. Он возвращает значение типа `NSComparisonResult` (являющегося простым перечислением enum), указывающее результат сравнения.

```
typedef enum _NSComparisonResult {
    NSOrderedAscending = -1,
    NSOrderedSame,
    NSOrderedDescending
} NSComparisonResult;
```

ПРАВИЛЬНОЕ СРАВНЕНИЕ СТРОК

При сравнении строк на равенство следует использовать метод `isEqualToString:`, а не сравнение их указателей:

```
if ([thing1 isEqualToString: thing2]) {
    NSLog(@"Строки одинаковы!");
}
```

Это не то же самое, что и сравнение

```
if (thing1 == thing2) {
    NSLog(@"Одинаковые объекты!");
}
```

Дело в том, что оператор `==` работает только со значениями *указателей* `thing1` и `thing2`, а не с тем, на что они указывают. Поскольку `thing1` и `thing2` — разные строки в том смысле, что они расположены в разных местах памяти, второе сравнение полагает, что они различны.

Но иногда требуется проверить идентичность двух объектов: является ли объект `thing1` в точности тем же объектом, что и `thing2`? В этом случае надо применять оператор `==`. Если же вы хотите проверить равенство содержимого строк, то используйте метод `isEqualToString:`.

Если вы уже пользовались функциями `C` `qsort()` или `bsearch()`, написанное может выглядеть для вас знакомым. Если результат метода `compare:` равен `NSOrderedAscending`, то левое значение меньше правого, т.е. получатель сообщения `compare` идет раньше в алфавитном порядке, чем переданная строка. Например, `[@"aardvark" compare: @"zygote"]` вернет `NSOrderedAscending`.

Аналогично `[@"zinks" compare: @"jinkies"]` вернет `NSOrderedDescending`. Ну а `[@"fnord" compare: @"fnord"]` вернет, как и ожидалось, `NSOrderedSame`.

Чувствительность к представлению данных

Метод `compare:` выполняет сравнение, чувствительное к регистру. Другими словами, `@"Bork"` и `@"bork"` при сравнении дают `NSOrderedSame`. Однако имеется другой метод, `compare:options:`, который дает вам большие возможности управления сравнением.

– (NSComparisonResult) compare: (NSString *) string
options: (unsigned) mask;

Параметр `options` представляет собой битовую маску. Для ее образования из индивидуальных флагов используется побитовый оператор ИЛИ (`|`). Вот некоторые флаги.

- `NSCaseInsensitiveSearch`: символы в верхнем и нижнем регистрах рассматриваются как одинаковые.
- `NSLiteralSearch`: выполняется точное сравнение, с учетом регистра символов.
- `NSNumericSearch`: числа в строках сравниваются по их числовому, а не символьному значению. Без этого "100" идет перед "99", что для большинства непрограммистов неестественно или даже просто неверно.

Например, если вы хотите выполнить сравнение без учета регистра, но с корректным упорядочением чисел, можете сделать следующее.

```
if ([thing1 compare: thing2 options: NSCaseInsensitiveSearch | NSNumericSearch]
    == NSOrderedSame) {
    NSLog(@"Соответствует!");
}
```

Кто внутри?

Иногда надо проверить, не содержит ли некоторая строка другую строку внутри себя. Например, вы можете захотеть узнать, заканчивается ли имя файла на `".mov"`, чтобы узнать, можно ли открывать его в QuickTime Player, или проверить, не начинается ли оно с `"draft"` (т.е. не представляет ли собой черновую версию документа). Вот два метода, которые могут вам помочь: первый из них проверяет, начинается ли некоторая строка с другой строки, а вторая определяет, не заканчивается ли строка другой строкой.

– (BOOL) hasPrefix: (NSString *) aString;
– (BOOL) hasSuffix: (NSString *) aString;

Эти методы используются следующим образом.

```
NSString *filename = @"draft-chapter.pages";
if ([filename hasPrefix: @"draft"] {
```

```
// Это черновик
}

if ([fileName hasSuffix: @" .mov"]) {
    // Это фильм
}
```

Итак, `draft-chapters.pages` распознается как черновая версия (так как имя начинается с "draft"), но не распознается как фильм (имя завершается на ".pages", а не на ".mov").

Если вы хотите увидеть, не входит ли строка в другую, используйте метод `rangeOfString`.

```
-(NSRange) rangeOfString: (NSString *) aString;
```

Когда вы посылаете сообщение `rangeOfString:` объекту `NSString`, передаете строку, которую ищете. Метод возвращает структуру `NSRange`, которая указывает, где находится соответствующая часть строки и насколько велико соответствие. Так что после выполнения следующего фрагмента

```
NSRange range;
range = [fileName rangeOfString: @"chapter"];
```

мы получим, что `range.start` равно 6, а `range.length` равно 7. Если аргумент в получателе не найден, `range.start` будет равно `NSNotFound`.

Изменчивость

Объект `NSString` неизменяем. Это не значит, что вы не можете с ним работать; это значит, что после создания объекта вы не можете его изменять. Вы можете делать с ним множество вещей — создавать новые строки с ним, находить в нем символы, сравнивать его с другими строками, но не можете изменять его, изымая из него символы или добавляя новые.

Сосоа предоставляет подкласс `NSString` с именем `NSMutableString`. Воспользуйтесь им, если вам совершенно необходимо работать со строкой, изменяя ее.

ПРИМЕЧАНИЕ

Программисты, пришедшие в Objective-C из Java, должны чувствовать себя при таком ограничении как дома. `NSString` ведет себя, как класс `Java String`, а `NSMutableString` — как класс `Java StringBuffer`.

Новый объект `NSMutableString` можно создать при помощи метода класса `stringWithCapacity:`, объявленного следующим образом.

```
+ (id) stringWithCapacity: (unsigned) capacity;
```

Емкость (аргумент `capacity`) — не более чем предложение для `NSMutableString`, ну, как когда вы говорите своему малолетнему отпрыску, в котором часу он обязан быть дома. Строка не ограничена переданной вами емкостью — это просто оптимизация. Например, если вы знаете, что будете создавать строку размером 40 Мбайт, то `NSMutableString` может заранее

выделить соответствующий блок памяти, что сделает последующие операции существенно более быстрыми. Создается новая изменяемая строка следующим образом.

```
NSMutableString *string;
string = [NSMutableString stringWithCapacity: 42];
```

Создав изменяемую строку, вы можете делать с ней все, что захотите. Обычная операция — добавление новой строки при помощи методов `appendString:` или `appendFormat:`.

```
– (void) appendString: (NSString *) aString;
– (void) appendFormat: (NSString *) format, ...;
```

Метод `appendString` получает параметр `aString` и копирует его в конец объекта-получателя. Метод `appendFormat` работает так же, как и `stringWithFormat:`, но вместо создания нового объекта он добавляет отформатированную строку к концу строки-получателя, например.

```
NSMutableString *string;
string = [NSMutableString stringWithCapacity: 50];
[string appendString: @"Hello there "];
[string appendFormat: @"human %d!", 39];
```

По окончании выполнения этого фрагмента `string` получит значение "Hello there human 39".

Удалить символы из строки можно при помощи метода `deleteCharactersInRange:`.

```
– (void) deleteCharactersInRange: (NSRange) range;
```

Метод `deleteCharactersInRange:` часто используется вместе с `rangeOfString:`. Вспомните, что `NSMutableString` является подклассом `NSString`. Благодаря чудесам объектно-ориентированного программирования мы можем использовать все возможности `NSString` при работе с `NSMutableString`, включая метод `rangeOfString:`, методы сравнения и прочие. Например, у вас есть список ваших друзей, и вы решили, что Платон вам друг, но истина — дороже, и приняли решение убрать Платона из списка друзей.

Для начала создадим список друзей.

```
NSMutableString *friends;
friends = [NSMutableString stringWithCapacity: 50];
[friends appendString: @"Сенека Платон Диоген"];
```

Затем определим диапазон символов, занимаемый Платоном.

```
NSRange PlatonRange;
PlatonRange = [friends rangeOfString: @"Платон"];
PlatonRange.length++; // Добавляем пробел
```

В нашем случае диапазон начинается с позиции 7 и имеет длину 7. Теперь удаляем Платона из списка.

```
[friends deleteCharactersInRange: PlatonRange];
```

В результате получим строку "Сенека Диоген".

Изменяемые строки очень удобны для реализации описывающих методов. Вы можете использовать методы `appendString` и `appendFormat` для создания описания вашего объекта.

Кое-какое поведение объекта мы получаем даром, поскольку `NSMutableString` — подкласс `NSString`. Первая "халява" в том, что везде, где используется `NSString`, мы можем заменить его на `NSMutableString`. Любые методы, принимающие `NSString` в качестве аргумента, точно так же могут принимать `NSMutableString`.

Вторая дармовщина проистекает из того факта, что наследование работает как с методами экземпляра, так и с методами класса. Так что удобный метод класса `stringWithFormat:` в `NSString` может создавать новые строки `NSMutableString`. Можно легко заполнить изменяемую строку с применением формата.

```
NSMutableString *string;
string = [NSMutableString stringWithFormat: @"jo%dy", 2];
```

Теперь `string` получает значение "jo2y", но вы можете выполнять и другие операции, такие как удаление символов из заданного диапазона или вставка символов в определенную позицию. Обратитесь к документации по классам `NSString` и `NSMutableString`, чтобы получить полную информацию о всех методах, доступных в этих классах.

Коллекция коллекций

Отдельные свободные и независимые объекты — это неплохо, но коллективизм — великая сила, так что очень часто требуется эти объекты организовать во что-то более великое. Сосоа предоставляет ряд классов типа `NSArray` и `NSDictionary`, экземпляры которых предназначены исключительно для хранения других объектов.

NSArray

Вы уже использовали массивы в языке программирования С. По крайней мере, ранее в этой книге нами использовался массив для хранения колес автомобиля. Вы должны также помнить, что при этом мы столкнулись с определенными трудностями. Например, при обращении к элементу массива нам надо было выполнять проверку корректности индекса: он не мог быть меньше нуля или выходить за границы массива. Еще одна проблема заключалась в том, что размер массива — 4 элемента — был жестко зашит в исходном тексте, так что собрать автомобиль с большим количеством колес нам бы не удалось. Да, сейчас это не выглядит таким уж непреодолимым ограничением, но как знать, сколько колес будет у автомобилей в недалеком будущем?

В Сосоа имеется класс `NSArray`, предназначенный для хранения упорядоченного списка объектов. Вы можете поместить в `NSArray` объекты любого типа: `NSString`, `Car`, `Shape`, `Tire` или что-то иное.

Если у вас имеется массив `NSArray`, вы можете работать с ним по-разному: иметь переменную экземпляра, указывающую на этот массив, передавать массив в качестве аргумента в метод или функцию, получать количество элементов, хранящихся в массиве, обращаться к элементу по его индексу, находить объект в массиве, обходить все элементы массива и многое, многое другое.

Массив NSArray имеет два ограничения. Первое — он может хранить только объекты Objective-C. В нем нельзя размещать такие примитивные типы языка программирования C, как `int`, `float`, `enum`, `struct`, или произвольные указатели. Второе ограничение — вы не можете хранить в нем `nil` (нулевое значение для объектов). Однако, как вы вскоре увидите, имеются способы обойти перечисленные ограничения.

Новый NSArray можно создать при помощи метода класса `arrayWithObjects:`. Вы передаете список объектов, разделенных запятыми, со значением `nil` в качестве сигнала окончания списка (что, кстати, является одной из причин, по которым в массиве нельзя хранить значение `nil`).

```
NSArray *array;
array = [NSArray arrayWithObjects:
    @"one", @"two", @"three", nil];
```

Здесь создается трехэлементный массив, составленный из литералов NSString. После создания массива вы можете получить количество содержащихся в нем объектов.

– (unsigned) count;

Можно также выбрать элемент с определенным индексом.

– (id) objectAtIndex: (unsigned int) index;

Можно скомбинировать указанные способы работы с массивом для вывода его содержимого.

```
int i;
for (i = 0; i < [array count]; i++) {
    NSLog(@"По индексу %d находится %@.",
        i, [array objectAtIndex: i]);
}
```

Вывод программы будет следующим.

```
По индексу 0 находится one.
По индексу 1 находится two.
По индексу 2 находится three.
```

Если вы обратитесь к элементу по индексу, превышающему количество элементов в массиве, Cocoa выведет сообщение об ошибке времени выполнения. Например, код

```
[array objectAtIndex: 208000];
```

приведет к выводу сообщения

```
*** Terminating app due to uncaught exception 'NSRangeException', reason:
*** -[NSCFArray objectAtIndex:]: index (208000) beyond bounds (3)
```

Поскольку вы — как программист Cocoa, — вероятно, время от времени будете встречаться с такими сообщениями, давайте познакомимся с ними поближе. После первого ушата холодной воды, гласящего, что программа аварийно завершена, на вас выливается остальное — в част-

ности, упоминание, что это произошло из-за "непойманного исключения". Исключение в Cocoa представляет собой способ сказать "я не знаю, что с этим делать". Имеются способы перехватить исключение в коде и справиться с ним самостоятельно, но когда вы только начинаете изучение языка, не стоит забивать себе этим голову. Данное конкретное исключение — `NSRangeException`, которое означает, что что-то не так с диапазоном, переданным методу в качестве аргумента. В частности, в нашем случае неприятность произошла в методе `objectAtIndex:` класса `NSCFArray`. Этот класс очень похож на `NSArray`, что является подсказкой о том, что же пошло не так.

ПРИМЕЧАНИЕ

Когда вы видите в Cocoa буквы "CF", имеете дело с чем-то, связанным со схемой Apple Core Foundation. Схема Core Foundation подобна Cocoa, но реализована на языке программирования C, в основном в виде открытого кода, который можно загрузить для того, чтобы разобраться в нем. Многие объекты Core Foundation и Cocoa сопряжены, что означает, что они могут использоваться взаимозаменяемо. Класс `NSCFArray` представляет собой реализацию Apple класса `NSArray`, но с применением для выполнения ответственной работы `CFArray`.

Последние биты информации в сообщении об исключении наиболее интересны. Эта часть сообщения гласит, что мы запросили что-то из массива, указав индекс 208000, но наибольший элемент массива — третий. Используя эту информацию, вы можете проследить выполнение кода и найти ошибку.

Отслеживание исключений может вас только расстроить. Все, что вы получите при запуске — это данное сообщение в окне Run. В случае приложения с графическим интерфейсом пользователя выполнение программы будет продолжаться. В отладчике Xcode можно остановить программу при генерации исключения, что уже несколько легче. Для этого выберите команду меню `Run ⇒ Show ⇒ Breakpoints`. Вы получите окно, которое показывает все текущие точки останова (места, где отладчик Xcode остановит выполнение программы и даст возможность покопаться в ее внутренностях), точки останова, специфичные для текущего проекта, и глобальные точки останова, как показано на рис. 8.1.

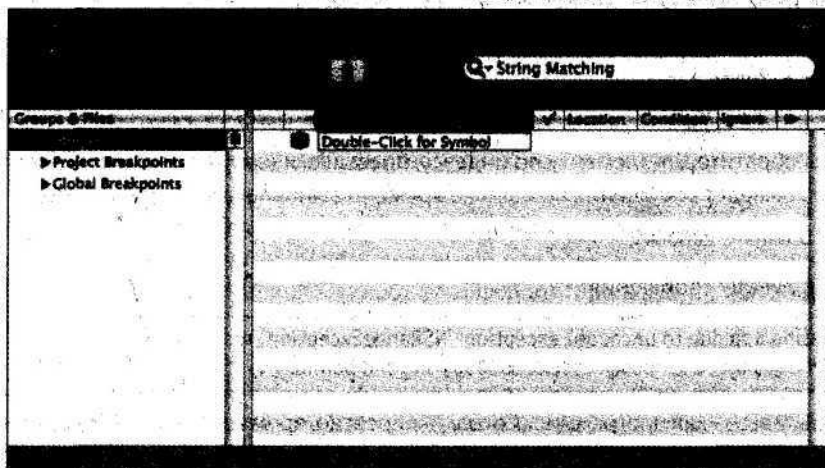


Рис. 8.1. Окно точек останова Xcode

Добавим две точки останова, которые облегчат нашу задачу. Сначала установим точку останова `-[NSException raise]`. Выберите **Global Breakpoints** и дважды щелкните на поле **Double-Click for Symbol**; введите `-[NSException raise]` и нажмите **<Enter>**. Ваше окно точек останова должно иметь вид, показанный на рис. 8.2.

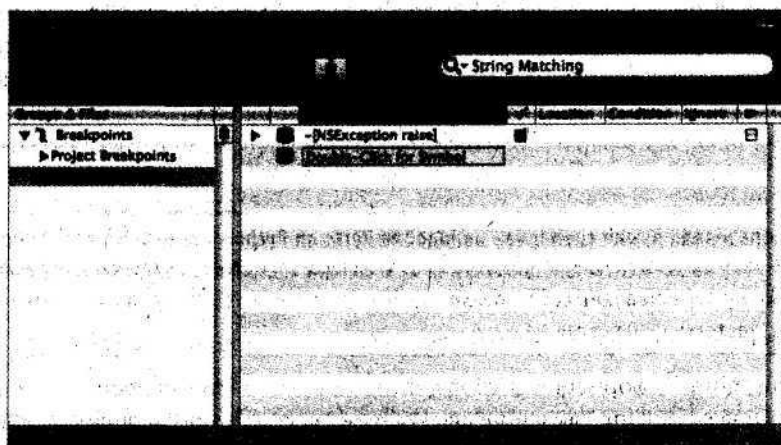


Рис. 8.2. После добавления точки останова `-[NSException raise]`

Добавим также точку останова для `objc_exception_throw`. Теперь, когда вы запустите программу и будет сгенерировано исключение, отладчик остановится и укажет на проблемную строку, как показано на рис. 8.3. Вам может понадобиться щелкнуть на стеке вызовов (верхняя левая панель на рис. 8.3) для переноса фокуса в соответствующий исходный файл. В нашем примере мы щелкаем на функции `main()` для того, чтобы увидеть наш код.

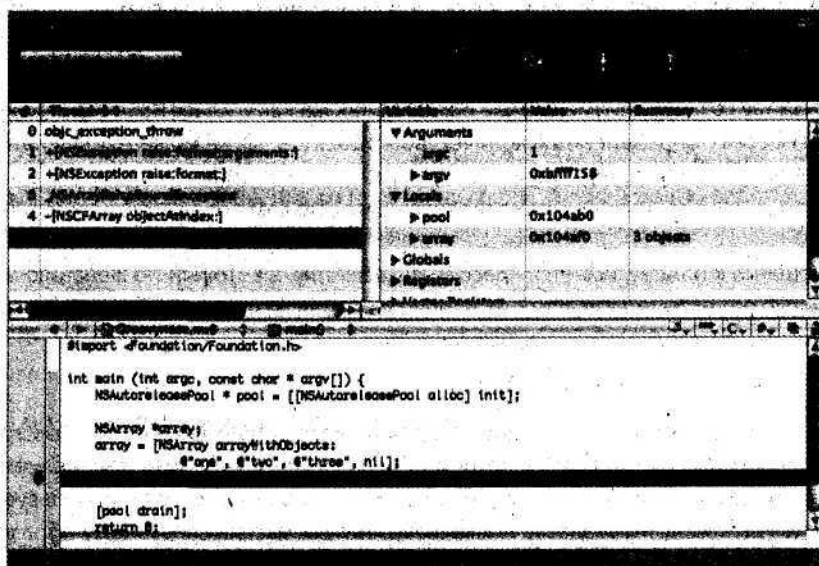


Рис. 8.3. Отладчик Xcode указывает на проблемную строку

Вам может не понравиться, что Сосоа столь ретив в своих жалобах, если ваша программа немного выйдет за рамки дозволенного. Но поверьте профессионалам: это *очень хорошо*, поскольку позволяет вам найти ошибку, которая в противном случае осталась бы незамеченной.

Так же как и NSString, NSArray обладает массой возможностей. Например, вы можете запросить массив о местоположении конкретного объекта, создать новый массив на основе диапазона элементов текущего массива, объединить все элементы в строку с заданным разделителем (удобно для составления списков с элементами, разделенными запятыми) или создать новый массив, представляющий собой отсортированную версию уже имеющегося массива.

РАЗДЕЛЕНИЕ

Если вы использовали языки сценариев наподобие Perl или Python, то, вероятно, вам приходилось разбивать строку на элементы для помещения их в массив и добавлять элементы массива в строку. Это же можно делать и при помощи NSArray.

Для разделения NSArray воспользуйтесь методом `componentsSeparatedByString:`.

```
NSString *string = @"oop:ack:bork:greeble:ponies";
NSArray *chunks = [string componentsSeparatedByString: @":"];
```

Для объединения NSArray и создания строки с его содержимым воспользуйтесь методом `componentsJoinedByString:`.

```
string = [chunks componentsJoinedByString: @": "];
```

Приведенная строка кода даст строку NSString с содержимым

```
oop :-) ack :-) bork :-) greeble :-) ponies
```

Изменяемые массивы

Класс NSArray создает неизменяемые объекты, так же как и класс NSString. После того как вы создали массив с определенным количеством объектов, вы не можете добавлять или удалять его члены. Объекты, содержащиеся в массиве, могут изменяться (так же как Car получает новый набор колес Tire, если не проходит техосмотр), но сам объект массива остается неизменным.

В дополнение к NSArray, имеется класс NSMutableArray, в который по желанию можно добавлять новые объекты (или удалять их из массива). Для создания такого изменяемого массива используется метод класса `arrayWithCapacity:`.

```
+ (id) arrayWithCapacity: (unsigned) numItems;
```

Так же как и в случае метода `stringWithCapacity:` класса NSMutableString, размер массива служит лишь подсказкой, пользуясь которой Сосоа может выполнить определенную оптимизацию кода. Сосоа не выполняет заполнение массива объектами и не использует всю емкость массива до конца. Вы создаете новый изменяемый массив следующим образом.

```
NSMutableArray *array;
array = [NSMutableArray arrayWithCapacity: 17];
```

Объекты добавляются в конец массива при помощи метода `addObject:`:

```
– (void) addObject: (id) anObject;
```

Так, добавить четыре колеса в массив можно при помощи следующего цикла.

```
for (i = 0; i < 4; i++) {
    Tire *tire = [Tire new];
    [array addObject: tire];
}
```

Можно удалить объект с конкретным индексом. Например, если вам не нужно второе колесо, то вы можете воспользоваться для его удаления методом `removeObjectAtIndex:`. Вот как определен этот метод.

```
– (void) removeObjectAtIndex: (unsigned) index;
```

А это — пример его применения.

```
[array removeObjectAtIndex: 1];
```

Обратите внимание, что второе колесо имеет индекс 1. Объекты `NSArray`, как и объекты в массивах C, нумеруются с нуля.

Теперь в массиве осталось три колеса. Пустого промежутка в массиве после удаления не остается — оставшиеся в массиве объекты, следующие за удаленным, сдвигаются и заполняют просвет.

Другие методы изменяемого массива позволяют решать множество других задач, таких как вставка объекта в определенное место, замена объекта, сортировка массива, а также выполнять все действия, доступные классу `NSArray`.

Перечисления

Весьма распространенное действие с массивом `NSArray` — выполнение некоторой операции с каждым его элементом. Например, вы можете потребовать от всех фигур изменить цвет на зеленый или спустить все колеса у автомобиля. Это можно сделать, написав цикл с индексом, пробегающим значения от 0 до `[array count]`, и поочередно работая с объектом с каждым из индексов. Но этого же можно добиться и при помощи класса `NSEnumerator`, который представляет собой метод `Cocoa` для описания итерирования коллекций. Чтобы воспользоваться `NSEnumerator`, следует запросить у массива перечислитель при помощи метода `objectEnumerator`.

```
– (NSEnumerator *) objectEnumerator;
```

Метод можно использовать следующим образом.

```
NSEnumerator *enumerator;
enumerator = [array objectEnumerator];
```

Кроме того, имеется метод `reverseObjectEnumerator`, который пригодится, если вам надо обойти коллекцию в обратном направлении, от конца к началу.

После того как получили объект перечислителя, вы создаете цикл `while`, который на каждой итерации запрашивает у перечислителя очередной объект при помощи сообщения `nextObject`.

– (id) nextObject;

Цикл завершается, когда метод `nextObject` возвращает значение `nil`. Это еще одна причина, по которой не следует хранить в массиве значения `nil`: нет никакого способа отличить, является ли `nil` значением, сохраненным в массиве, или сигналом для завершения цикла.

Полностью цикл выглядит следующим образом.

```
NSEnumerator *enumerator;
enumerator = [array objectEnumerator];
```

```
id thingie;
```

```
while (thingie = [enumerator nextObject]) {
    NSLog(@"Найден %@", thingie);
}
```

При работе перечислителей с изменяемыми массивами есть один нюанс: вы не должны изменять контейнер, добавляя или удаляя объекты. Если вы так поступите, перечислитель может привести к неопределенным результатам. "Неопределенный результат" может означать что угодно — от "Похоже, программа работает" до "Опять эта программа вылетает".

Быстрое перечисление

В Mac OS X 10.5 (Leopard) Apple добавила в Objective-C небольшие дополнительные возможности, приведшие к появлению версии языка 2.0. Первое дополнение, которое мы рассмотрим, — *быстрое перечисление*, использующее синтаксис, знакомый пользователям различных языков сценариев.

```
for (NSString *string in array) {
    NSLog(@"Найден %@", string);
}
```

Тело цикла выполняется для каждого элемента массива, который при этом находится в переменной `string`. Это существенно более краткий и быстрый синтаксис.

Подобно прочим новым возможностям Objective-C 2.0, этот синтаксис недоступен в Tiger (Mac OS X 10.4). Если вам или вашим пользователям требуется выполнять программы под управлением Tiger, использовать новый синтаксис нельзя.

Итак, у нас есть три способа обхода массива: при помощи индекса, класса `NSEnumerator` и с помощью быстрого перечисления. Каким же способом надо пользоваться?

Если вы работаете под управлением Leopard или с операционными системами более поздних версий, используйте быстрое перечисление. Оно более краткое и быстрое.

Если от вас требуется поддержка Tiger, воспользуйтесь `NSEnumerator`. Xcode включает рефакторинг для преобразования кода в Objective-C 2.0 и автоматически конвертирует циклы с `NSEnumerator` в быстрые перечисления.

Метод `-objectAtIndex` используйте только в том случае, когда вам действительно необходимо обращение посредством индекса (например, если вам надо обратиться к каждому третьему объекту массива) или если одновременно выполняется итерирование нескольких массивов.

NSDictionary

Вне всяких сомнений, вы слышали о словарях, а может быть, и использовали их в своей практике. В программировании *словарь* представляет собой коллекцию ключевых слов и их определений. Сocoa имеет класс коллекции `NSDictionary`, который обладает такой функциональностью. `NSDictionary` хранит значение (которое может быть объектом любого типа), соответствующее некоторому ключу (обычно `NSString`). Затем вы можете использовать ключ для получения соответствующего значения. Так, например, если у вас есть `NSDictionary`, который хранит контактную информацию человека, вы можете запросить такой словарь — “дай мне значение, соответствующее ключу `home-address`” или, скажем, “дай мне значение, соответствующее ключу `email-address`”.

ПРИМЕЧАНИЕ

Почему бы не использовать простой массив и поиск значений в нем? Словарь (известный также как *хеш-таблица*, или *ассоциативный массив*) использует механизм хранения, оптимизированный для поиска ключа. Вместо сканирования всего массива словарь может сразу же протянуть свою электронную руку в то место, где хранятся необходимые данные. В случае больших объемов данных и частого поиска словарь может оказаться в сотни раз быстрее массива.

Как вы, вероятно, догадываетесь, `NSDictionary`, как и `NSString` и `NSArray`, является неизменяемым объектом. Однако имеется класс `NSMutableDictionary`, который позволяет добавлять и удалять записи словаря. При создании нового `NSDictionary` вы определяете все объекты и ключи, которые будут находиться в словаре. Простейший способ сделать это — воспользоваться методом класса `dictionaryWithObjectsAndKeys:`.

```
+ (id) dictionaryWithObjectsAndKeys:
```

```
(id) firstObject, ...;
```

Этот метод получает последовательность объектов и ключей, заканчивающуюся значением `nil` (как, опять же, вы могли предположить, хранить значение `nil` в `NSDictionary` нельзя). Пусть мы хотим создать словарь для хранения колес нашего автомобиля с использованием удобочитаемых меток вместо индексов массива. Такой словарь можно создать следующим образом.

```
Tire *t1 = [Tire new];
Tire *t2 = [Tire new];
Tire *t3 = [Tire new];
Tire *t4 = [Tire new];
```

```
NSDictionary *tires;
tires = [NSDictionary dictionaryWithObjectsAndKeys:
    t1, @"front-left",
```



```
t2, @"front-right",
t3, @"back-left",
t4, @"back-right",
nil];
```

Для доступа к значениям в словаре воспользуйтесь методом `objectForKey:`, передавая ему ключ, под которым было сохранено интересное вас значение.

```
– (id) objectForKey: (id) aKey;
```

Так, для получения правого заднего колеса можно воспользоваться выражением

```
Tire *tire = [tires objectForKey: @"back-right"];
```

Если в словаре нет правого заднего колеса (у вас трехколесная машина), метод `objectForKey:` вернет значение `nil`.

Для создания нового объекта `NSMutableDictionary` пошлите сообщение `dictionary` классу `NSMutableDictionary`. Можно также создать новый изменяемый словарь с передачей подсказки о планируемом его размере с помощью метода `dictionaryWithCapacity:` (вы уже обратили внимание на последовательность `Cocoa` в именовании?).

```
+ (id) dictionaryWithCapacity: (unsigned int) numItems;
```

Как мы упоминали ранее, рассматривая `NSMutableString` и `NSMutableArray`, указываемый размер — всего лишь подсказка, а не ограничитель размера словаря.

Добавить новые записи в словарь вам позволит метод `setObject:forKey:`.

```
– (void) setObject: (id) anObject forKey: (id) aKey;
```

Вот еще один способ создания уже рассматривавшегося словаря с колесами.

```
NSMutableDictionary *tires;
tires = [NSMutableDictionary dictionary];
[tires setObject: t1 forKey: @"front-left"];
[tires setObject: t2 forKey: @"front-right"];
[tires setObject: t3 forKey: @"back-left"];
[tires setObject: t4 forKey: @"back-right"];
```

Если вы воспользуетесь методом `setObject:forKey:` с ключом, который уже есть в словаре, старое значение будет заменено новым. Если вы хотите удалить запись с некоторым ключом из словаря, воспользуйтесь методом `removeObjectForKey:`.

```
– (void) removeObjectForKey: (id) aKey;
```

Так, если вы хотите убрать из автомобиля заднее левое колесо, это не составит проблем.

```
[tires removeObjectForKey: @"back-left"];
```

Никаких наследников

Возможно, вам захочется попытаться создать подклассы для `NSString`, `NSArray` или `NSDictionary`. Обуздайте свои желания. В некоторых языках программирования наследование классов массивов или строк — вполне допустимое явление, но в `Cocoa` многие классы на

самом деле реализованы как *кластеры классов*, которые представляют собой целый пакет реализующих классов, скрытых за общим интерфейсом. Когда вы создаете объект `NSString`, на самом деле можете создать `NSLiteralString`, `NSCFString`, `NSSimpleCString`, `NSBallOfString` или иное количество недокументированных реализующих объектов. В качестве примера, как это все работает, вспомним, как ранее в этой главе, когда индекс выходил за пределы границ и мы получали сообщение об ошибке, указанный в нем класс на самом деле был `NSCFArray`. Вернитесь и посмотрите еще раз.

В качестве пользователя `NSString` или `NSArray`, вам не надо знать, какой класс использован за сценой. Но если вы попытаетесь создать подкласс кластера классов, то получите головную боль и массу разочарований. Для решения своих задач вместо наследования вы можете воспользоваться композицией `NSString` или `NSArray` в одном из ваших классов или применить категории (описанные в главе 12, "Категории").

Семейства значений

Классы `NSArray` и `NSDictionary` хранят только объекты. Они не могут непосредственно хранить примитивные типы, такие как `int`, `float` или `struct`. Но вы можете воспользоваться объектами, в которые встроены примитивные значения. Например, встройте `int` в объект, и вы сможете поместить этот объект в `NSArray` или `NSDictionary`.

NSNumber

У Сосоа имеется класс `NSNumber`, который является оболочкой для примитивных числовых типов (т.е. реализует их как объекты). Вы можете создать новый объект `NSNumber` с помощью следующих методов класса.

```
+ (NSNumber *) numberWithChar: (char) value;
+ (NSNumber *) numberWithInt: (int) value;
+ (NSNumber *) numberWithFloat: (float) value;
+ (NSNumber *) numberWithBool: (BOOL) value;
```

Таких методов создания гораздо больше, включая версии для `unsigned` значений, целых чисел типа `long` и `long long`, но представленные выше — наиболее распространенные. После того как вы создали `NSNumber`, можете поместить его в словарь или массив.

```
NSNumber *number;
number = [NSNumber numberWithInt: 42];
[array addObject: number];
[dictionary setObject: number forKey: @"Bork"];
```

Имея возможность "завернуть" числа в объект `NSNumber`, вы можете и извлечь их оттуда при помощи следующих методов экземпляра.

```
- (char) charValue;
- (int) intValue;
- (float) floatValue;
```

- (BOOL) boolValue;
- (NSString *) stringValue;

Вы можете смешивать различные методы создания объектов и получения значений — например, можно создать NSNumber при помощи numberWithFloat:, а затем получить значение с использованием intValue. Класс NSNumber выполнит необходимое преобразование типов.

ПРИМЕЧАНИЕ

Такое завертывание “примитивного” типа в объект часто называется *упаковкой* (boxing), а получение примитивного типа назад — *распаковкой* (unboxing). Некоторые языки программирования обладают возможностью *автоупаковки*, которая автоматически выполняет преобразования между примитивными типами и объектами, однако Objective-C такой возможности не имеет.

NSNumber

Класс NSNumber в действительности представляет собой подкласс NSValue, который, в свою очередь, “заворачивает” любые значения. Вы можете использовать NSValue для помещения в NSArray и NSDictionary структур. Создайте новый объект NSValue с применением метода класса

```
+ (NSValue *) valueWithBytes: (const void *) value
    objCType: (const char *) type;
```

Вы передаете адрес значения, которое надо завернуть (такое, как NSSize, или вашу собственную структуру). Обычно вы получаете адрес (используя оператор C &) переменной, которую хотите сохранить. Вы также передаете строку, описывающую тип, — обычно с указанием типов и размеров полей структуры. Вам не надо самостоятельно писать код для создания такой строки — директива компилятора @encode, получающая имя типа, генерирует необходимую строку за вас. Так что для того, чтобы поместить NSRect в NSArray, вы должны написать примерно такой фрагмент исходного текста.

```
NSRect rect = NSMakeRect(1, 2, 30, 40);
NSValue *value;
value = [NSValue valueWithBytes: &rect
    objCType: @encode(NSRect)];
[array addObject: value];
```

Получить значение можно при помощи следующего метода.

- (void) getValue: (void *) value;

Когда вызываете getValue:, передаете адрес переменной, в которой хотите сохранить “завернутое” значение.

```
value = [array objectAtIndex: 0];
[value getValue: &rect];
```

ПРИМЕЧАНИЕ

В примере с getValue: вы видите применение префикса get в имени метода, который указывает, что метод получает указатель на место, где должно быть сохранено значение, сгенерированное этим методом.

Для размещения распространенных структур Cocoa в `NSValue` имеются отдельные методы.

```
+ (NSValue *) valueWithPoint: (NSPoint) point;
+ (NSValue *) valueWithSize: (NSSize) size;
+ (NSValue *) valueWithRect: (NSRect) rect;
- (NSPoint) pointValue;
- (NSSize) sizeValue;
- (NSRect) rectValue;
```

Для того чтобы сохранить значение типа `NSRect` в `NSArray` и получить его обратно, выполните следующее.

```
value = [NSValue valueWithRect: rect];
[array addObject: value];
....
NSRect anotherRect = [value rectValue];
```

NSNull

Мы уже говорили, что размещать в коллекции значение `nil` нельзя, потому что `nil` имеет специальный смысл в `NSArray` и `NSDictionary`. Но иногда действительно возникает необходимость сохранения значения, которое гласит: "Тут нет ничего". Например, у нас есть словарь, в котором хранится контактная информация человека, и под ключом `@ "home fax machine"` хранится номер домашнего факса. Если здесь хранится номер телефона, то вы знаете, что у человека дома есть факс. Но если такого значения в словаре нет — то что это означает: что у человека нет факса дома или он есть, но вы не знаете его номер? Воспользовавшись `NSNull`, можно устранить неоднозначность. Вы можете решить, что значение `NSNull` для ключа `@ "home fax machine"` означает, что человек точно не имеет дома факса, а отсутствие значения с данным ключом — что вы не знаете, есть факс или нет.

Класс `NSNull`, вероятно, самый простой класс Cocoa. Он имеет единственный метод

```
+ (NSNull *) null;
```

Вот как он добавляется в коллекцию.

```
[contact setObject: [NSNull null]
 forKey: @ "home fax machine"];
```

А так можно получить его значение.

```
id homefax;
homefax = [contact objectForKey: @ "home fax machine"];
if (homefax == [NSNull null]) {
    // ... факса дома нет
}
```

`[NSNull null]` всегда возвращает одно и то же значение, так что для сравнения можно использовать оператор `==`.

Пример: поиск файлов

Наверное, хватит трепаться, пора и серьезным делом заняться. Вот реальная работающая программа, использующая некоторые из описанных в главе классов. Программа FileWalker (которую можно найти в папке 08-01 FileWalker) просматривает ваш каталог в поисках .jpg-файлов и выводит список найденного. Да, не слишком впечатляет, зато работает.

FileWalker использует NSString, NSArray, NSArray и еще два других класса Foundation для работы с файловой системой.

В нашем примере используется также класс NSFileManager, который позволяет вам работать с файловой системой — создавать каталоги, удалять файлы, перемещать их или получать о них информацию. Мы просим NSFileManager создать для нас NSDirectoryEnumerator, которым воспользуемся для прохода по иерархии файлов.

Вся программа располагается в функции main(), так как мы не создаем собственные классы. Вот полный текст этой функции.

```
int main (int argc, const char *argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    NSFileManager *manager;
    manager = [NSFileManager defaultManager];

    NSString *home;
    home = [@"~" stringByExpandingTildeInPath];

    NSDirectoryEnumerator *direnum;
    direnum = [manager enumeratorAtPath: home];

    NSMutableArray *files;
    files = [NSMutableArray arrayWithCapacity: 42];

    NSString *filename;
    while (filename = [direnum nextObject]) {
        if ([[filename pathExtension]
            isEqualToString: @"jpg"]) {
            [files addObject: filename];
        }
    }

    NSEnumerator *fileenum;
    fileenum = [files objectEnumerator];

    while (filename = [fileenum nextObject]) {
        NSLog(@"%@ ", filename);
    }
}
```

```
[pool drain];
return (0);
} // main
```

Теперь разберем нашу программу. Вначале идет шаблонный код (который будет подробно рассмотрен в главе 9, "Управление памятью").

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
```

Затем мы создаем `NSFileManager`, с которым и будем работать. У `NSFileManager` есть метод класса `defaultManager`, который дает нам объект `NSFileManager` в полное владение.

```
NSFileManager *manager;
manager = [NSFileManager defaultManager];
```

Это распространенная идиома Cocoa. Есть ряд классов, обладающих *архитектурой синглтона*: требуется только один объект этого класса. Нужен только один менеджер файлов, один менеджер шрифтов, единственный графический контекст. Эти классы предоставляют метод класса, который дает нам доступ к единственному, совместно используемому объекту, который затем вы используете для своей работы.

В данном случае нам нужен итератор каталога. Но перед тем как запросить его у файлового менеджера, мы должны выяснить, где именно в файловой системе мы намерены искать файлы. Если начать с корня жесткого диска, это займет много времени, так что будем просматривать только собственный домашний каталог.

Как мы указываем этот каталог? Можно начать с абсолютного пути типа `/Users/markd/`, но дело в том, что это сработает при условии, что домашний каталог имеет имя `markd`. К счастью, Unix (и Mac OS X) имеет условный символ для обозначения домашнего каталога — тильду `~`. Каталог `~/Documents` представляет собой ваш подкаталог `Documents`, а `~/junk/oopsack.txt` на машине Марка — файл `/Users/markd/junk/oopsack.txt`. Класс `NSString` имеет метод, который раскрывает тильду в реальное имя. Этот метод использован нами в следующих двух строках кода.

```
NSString *home;
home = [@"~" stringByExpandingTildeInPath];
```

Метод `stringByExpandingTildeInPath` заменяет `~` на домашний каталог текущего пользователя. На машине Марка это `/Users/markd`. Затем мы передаем эту строку пути файловому менеджеру.

```
NSDirectoryEnumerator *direnum;
direnum = [manager enumeratorAtPath: home];
```

Метод `enumeratorAtPath:` возвращает объект типа `NSDirectoryEnumerator`, являющегося подклассом `NSEnumerator`. Всякий раз, когда мы вызываем метод `nextObject` этого объекта, он возвращает очередной путь к файлу в данном каталоге. Этот метод проходит и по подкаталогам. К тому времени, когда закончится выполнение цикла, мы получим пути каждого отдельного файла вашего домашнего каталога. Класс `NSDirectoryEnumerator` обеспечивает и некоторые другие возможности, например словарь атрибутов каждого файла, но мы его здесь не используем.

Так как мы ищем только .jpg-файлы (т.е. файлы, имена которых заканчиваются на ".jpg") и выводим их имена, нам надо место для их хранения. Мы могли бы просто выводить их при помощи функции NSLog() по мере перечисления, но в будущем мы, возможно, захотим выполнить над ними какие-то дополнительные действия. Класс NSMutableArray — неплохой выбор. Создадим изменяемый массив и добавим в него отвечающие нашему критерию пути.

```
NSMutableArray *files;
files = [NSMutableArray arrayWithCapacity: 42];
```

Мы не знаем, сколько именно .jpg-файлов будет найдено, так что взяли с потолка число 42. Поскольку это число не является ограничением размера массива, сойдет и оно.

Теперь самое время покончить с вегетарианскими салатиками и приступить к основному блюду. Все настроено, и пришло время для главного цикла.

```
NSString *filename;
while (filename = [direnum nextObject]) {
```

Перечислитель каталога возвращает строку NSString с путем файла. Так же как и NSEnumerator, по окончании перечисления он вернет значение nil, которое завершит работу цикла.

Класс NSString предоставляет ряд удобных утилит для работы с именами путей и файлов. Например, метод pathExtension дает вам расширение файла (без предшествующей ему точки). Так, вызов pathExtension для файла oopack.txt вернет @"txt", а для VikkiCat.jpg — @"jpg".

Мы используем вложенный вызов метода для получения расширения и отправляем этой строке сообщение isEqualTo:. Если этот вызов вернет YES, имя файла будет добавлено в массив файлов.

```
if ([[filename pathExtension] isEqualTo: @"jpg"]) {
    [files addObject: filename];
}
```

По окончании цикла мы проходим по массиву файлов и выводим его содержимое при помощи функции NSLog().

```
NSEnumerator *fileenum;
fileenum = [files objectEnumerator];
while (filename = [fileenum nextObject]) {
    NSLog(@"%@ ", filename);
}
```

Под конец мы выполняем некоторые действия по "уборке помещения", которые входят в шаблон кода, и просим функцию main() вернуть значение 0, указывающее на успешное завершение работы.

```
[pool drain];
return (0);
} // main
```

Вот как выглядит начало вывода программы на машине Марка.

cocoaheads/DSCN0798.jpg
 cocoaheads/DSCN0804.jpg
 cow.jpg
 Development/Borkware/BorkSort/cant-open-file.jpg
 Development/Borkware/BSL/BWLog/accident.jpg

Работает! Конечно, конкретный результат (и время работы программы) зависит от того, сколько тысяч фотографий валяется в вашем каталоге.

Еще быстрее

В программе FileWalker использован классический способ обхода. В проекте 08.02 FileWalkerPro показано, как выполнить те же действия с применением быстрого перечисления. Одна из приятных возможностей быстрого перечисления — то, что вы можете передать уже имеющийся объект типа `NSEnumerator` или его подкласса. А раз уж так получилось, что `NSDirectoryEnumerator` — подкласс `NSEnumerator`, то грех не воспользоваться этим и не передать результат `-enumeratorAtPath:` в быстрое перечисление.

```
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool =
        [[NSAutoreleasePool alloc] init];

    NSFileManager *manager;
    manager = [NSFileManager defaultManager];

    NSString *home;
    home = [@"~" stringByExpandingTildeInPath];

    NSMutableArray *files;
    files = [NSMutableArray arrayWithCapacity: 42];
    for (NSString *filename
         in [manager enumeratorAtPath: home]) {
        if ([filename pathExtension]
            isEqualTo: @"jpg"]) {
            [files addObject: filename];
        }
    }

    for (NSString *filename in files) {
        NSLog(@"%@", filename);
    }

    [pool drain];
    return (0);
} // main
```

Как видите, эта версия проще предыдущей: мы избавились от двух переменных и кода для их поддержки.

Резюме

В этой главе мы неплохо поработали! Мы рассмотрели три новые возможности языка: методы класса, которые работают с классом в целом, а не с каким-то конкретным экземпляром; директиву `@encode()`, применяемую при работе методов, которым для работы требуется описание типа `C`; а также быстрое перечисление.

Мы рассмотрели ряд полезных классов Cocoa, включая `NSString`, `NSArray` и `NSDictionary`. Класс `NSString` предназначен для хранения текста, а классы `NSArray` и `NSDictionary` — для хранения коллекций объектов. Объекты этих классов неизменяемы: после того как они созданы, их нельзя изменять. Но если это необходимо — Cocoa предоставляет и изменяемые версии данных классов.

Несмотря на все наши усилия, мы едва скользнули взглядом по поверхности айсберга сотен различных классов Cocoa. Растопить этот айсберг, поливая его пролитым при изучении прочих классов Cocoa потом, вы можете самостоятельно.

Наконец, мы применили рассмотренные классы в программе для поиска картинок в вашем каталоге.

В следующей главе мы погрузимся в таинственную пучину управления памятью, а еще вы узнаете, как убирать за собой — если к этому родители не приучили вас в детстве.

Управление памятью

Следующее поданное вам блюдо — это управление памятью с использованием Objective-C и Cocoa (правда, вкусно?). Управление памятью — это только часть общей проблемы, именуемой *управление ресурсами* (не бойтесь, речь идет не о нефти). Каждая компьютерная система может предоставить программе для выполнения только весьма ограниченные ресурсы, которые включают память, открытые файлы, сетевые соединения и т.п. Если вы используете ресурсы, например, путем открытия файла, после работы должны прибраться (в данном случае — закрыть файл). Если вы будете оставлять файлы открытыми и никогда не будете их закрывать, то в конечном счете исчерпаете этот ресурс и не сможете продолжать работу. Подумайте о библиотеках — если бы каждый брал в них книги и никогда не возвращал их, надолго ли хватило бы этих библиотек?

Конечно, когда программа завершает работу, операционная система освобождает выделенные ей ресурсы. Но пока ваша программа работает и использует ресурсы, ваша задача — относиться к ним бережно. Как уже говорилось, речь идет не о нефти, и практика бездумного выкачивания здесь не годится — в этом случае ваша программа просто прекратит работу из-за нехватки ресурсов.

Не каждая программа нуждается в сетевых подключениях, но нет такой программы, которая не использовала бы память. Ошибки, связанные с памятью, — сущее наказание для программистов, работающих с С-образными языками программирования. Нашим коллегам, работающим на Java и языках сценариев, гораздо проще: управление памятью у них автоматизировано; можно сказать, что родители прибирают их комнаты. Мы же сами выделяем память, когда она нужна, и освобождаем, когда выполнили с ней все действия. Если мы выделяем память, но не освобождаем ее — получаем *утечку памяти*: потребление памяти программой растет и растет, и в конечном итоге программа аварийно завершается.

С другой стороны, надо быть не менее осторожным и не использовать уже освобожденную память. Так можно воспользоваться утратившими актуальность данными и получить самые разнообразные и очень трудно обнаруживаемые ошибки.

ПРИМЕЧАНИЕ

Управление памятью — сложная задача. Решение Сосоа, скорее, элегантно, но требует умственных усилий, чтобы разобраться в нем. Даже программисты с большим стажем, впервые столкнувшись с данным материалом, испытывают определенные затруднения, так что не беспокойтесь, если после прочтения данной главы ваша голова будет некоторое время кружиться...

Если вы знаете, что ваши программы будут работать под управлением операционной системы Leopard или более поздней, можете воспользоваться возможностями Objective-C 2.0 по сборке мусора, о которой мы поговорим в конце этой главы. Но если вы планируете работать с более старыми версиями Mac OS X или разрабатывать приложения для iPhone, то вам следует прочесть всю главу полностью.

Жизненный цикл объекта

Подобно любому живому существу в нашем подлунном мире, объекты в программе также имеют свой жизненный цикл. Они рождаются (при помощи `alloc` или `new`), живут (получают сообщения и реагируют на них), заводят друзей (путем композиции и аргументов методов) и в конце концов умирают (освобождают память). Когда это происходит, их тела (память) используются новым поколением объектов.

Подсчет ссылок

Теперь, когда мы знаем о том, когда объект рождается, и немало поговорили о том, как он живет, т.е. как его использовать в программе, — пора задаться вопросом: как узнать, что жизнь объекта подошла к концу? Сосоа использует методику, известную как *подсчет ссылок* (reference counting). Каждый объект содержит связанную с ним целочисленную переменную, известную как *счетчик ссылок*. Когда некоторый фрагмент кода работает с объектом, он увеличивает его счетчик ссылок, говоря: «Я работаю с этим объектом». Когда работа завершена, код уменьшает счетчик ссылок, указывая, что ему объект более не нужен. Когда счетчик становится равен нулю, это означает, что объект больше никому не нужен (бедняжка!), так что он уничтожается, а занятая им память возвращается системе для повторного использования.

Когда объект создается при помощи `alloc` или `new` или в ответ на сообщение `copy` (которое приводит к созданию копии объекта-получателя), счетчик объекта устанавливается равным 1. Для увеличения счетчика ссылок следует послать объекту сообщение `retain`, а для уменьшения — сообщение `release`.

Когда объект должен быть уничтожен из-за обнуления его счетчика ссылок, Objective-C автоматически отправляет объекту сообщение `dealloc`. Вы можете перекрыть `dealloc` в ваших объектах для освобождения всех ресурсов, которые вы могли захватить. Никогда не вызывайте `dealloc` непосредственно — положитесь в этом на Objective-C, который вызывает метод `dealloc`, когда наступает время убить ваш объект. Чтобы узнать текущее состояние счетчика

ссылок, пошлите объекту сообщение retainCount. Вот как выглядят сигнатуры упомянутых retain, release и retainCount.

- (id) retain;
- (void) release;
- (unsigned) retainCount;

Метод retain возвращает id, так что вы можете встроить этот вызов в цепочку, сперва увеличивая счетчик ссылок, а затем запрашивая от объекта некоторые действия. Например, `[[car retain] setTire:tire atIndex: 2];` требует от car увеличить счетчик ссылок и выполнить действие setTire:.

Первый проект данной главы — RetainCount1 — расположен в папке 09.01 RetainCount-1. Эта программа создает объект (RetainTracker), который вызывает NSLog() при инициализации и уничтожении.

```
@interface RetainTracker : NSObject
@end // RetainTracker
```

```
@implementation RetainTracker
```

```
- (id) init
{
    if (self = [super init]) {
        NSLog(@"init: счетчик равен %d.",
            [self retainCount]);
    }
    return (self);
} // init
```

```
- (void) dealloc
{
    NSLog(@"Вызван dealloc. Прощайте!");
    [super dealloc];
} // dealloc
@end // RetainTracker
```

Метод init следует стандартной идиоме Сосоа для инициализации объектов, которую мы рассмотрим в следующей главе. Как упоминалось ранее, сообщение dealloc посылается (и в результате вызывается метод dealloc) автоматически, когда счетчик ссылок обнуляется. Наши версии init и dealloc используют NSLog() для вывода сообщений, информирующих об их вызовах.

В функции main() создается новый объект RetainTracker, при этом описанные выше два метода вызываются неявно. После создания нового объекта RetainTracker для увеличения и уменьшения счетчика ссылок посылаются сообщения retain и release, причем в промежутках между вызовами значение счетчика отслеживается и выводится при помощи функции NSLog().

```
int main (int argc, const char *argv[])
{
    RetainTracker *tracker = [RetainTracker new];
```

```

// count: 1

[tracker retain]; // count: 2
NSLog(@"%d", [tracker retainCount]);

[tracker retain]; // count: 3
NSLog(@"%d", [tracker retainCount]);

[tracker release]; // count: 2
NSLog(@"%d", [tracker retainCount]);

[tracker release]; // count: 1
NSLog(@"%d", [tracker retainCount]);

[tracker retain]; // count 2
NSLog(@"%d", [tracker retainCount]);

[tracker release]; // count 1
NSLog(@"%d", [tracker retainCount]);

[tracker release]; // count: 0, вызов dealloc
return (0);
} // main

```

В реальной программе, конечно, вы не будете выполнять неоднократные увеличения и уменьшения счетчика ссылок, как в этой простой функции. В процессе жизненного цикла объекта он может столкнуться с последовательностями увеличений и уменьшений счетчика ссылок, подобными приведенной, но исходящими из разных мест. Наша же программа позволяет посмотреть на изменения счетчика ссылок в процессе работы.

init: счетчик равен 1.

```

2
3
2
1
2
1

```

Вызван dealloc. Прощайте!

Так что если вы создаете объект (при помощи alloc, new or copy), то должны по окончании работы с ним освободить его — чтобы ненужный объект был уничтожен, а использованная им память была возвращена системе.

Владение объектом

"Ну и что тут сложного? — можете подумать вы. — Тоже мне бином Ньютона: создать объект, попользоваться, освободить, и все счастливы. Не вижу никаких сложностей!" Но это еще не конец, и все станет более сложным при рассмотрении концепции *владения объектом*. Когда кто-то говорит, что он "владеет объектом", это означает, что он отвечает за его удаление и освобождение памяти.

Говорят, что объект с переменными экземпляра, которые указывают на другие объекты, владеет этими другими объектами. Например, в CarParts автомобиль владеет двигателем и колесами, на которые указывает. Аналогично функция, создающая объект, владеет этим объектом. В программе CarParts функция main() создает новый объект автомобиля, так что мы говорим, что этим объектом владеет функция main().

Сложности начинаются, когда у некоторого объекта имеется несколько владельцев, т.е. его счетчик ссылок становится больше 1. В случае программы RetainCount1, функция main() владеет объектом RetainTracker, так что за его освобождение и удаление отвечает именно функция main().

Вспомните метод установки двигателя класса Car:

```
– (void) setEngine: (Engine *) newEngine;
```

и то, как он вызывается из функции main().

```
Engine *engine = [Engine new];
[car setEngine: engine];
```

Кто теперь владеет двигателем? Функция main() или объект car? Кто отвечает за отправку двигателю сообщения release о том, что он более не нужен? Это не может быть функция main(), поскольку Car использует двигатель. И это не может быть Car, поскольку функция main() может использовать двигатель позже.

Трюк заключается в том, чтобы заставить Car владеть двигателем, увеличивая его счетчик ссылок до 2. Это имеет смысл, поскольку двигатель теперь используют два владельца, Car и main(). Car должен получить двигатель во владение в методе setEngine:, а функция main() должна освободиться от владения. Тогда Car будет отвечать за освобождение двигателя по окончании работы (в своем методе dealloc), и ресурсы двигателя окажутся освобождены и возвращены системе.

Владение и освобождение в методах доступа

Первая попытка написания версии метода setEngine, корректно работающего с управлением памятью, может иметь следующий вид.

```
– (void) setEngine: (Engine *) newEngine
{
    engine = [newEngine retain];
    // ПЛОХО. См. исправленную версию ниже.
} // setEngine
```

К сожалению, этого недостаточно. Представьте следующую последовательность вызовов в функции main().

```
Engine *engine1 = [Engine new]; // count: 1
[car setEngine: engine1]; // count: 2
[engine1 release]; // count: 1
```

```
Engine *engine2 = [Engine new]; // count: 1
[car setEngine: engine2]; // count: 2
```

Ой! У нас проблема с engine1: счетчик ссылок все еще равен 1. Функция main() уже уменьшила счетчик ссылок engine1, но Car этого никогда не делает. В результате мы получаем утечку engine1, а утечка двигателей еще никогда ни к чему хорошему не приводила. Первый объект engine оказывается всеми позабыт и позаброшен, но при этом потребляет блок памяти.

Попробуем переписать setEngine:

```
– (void) setEngine: (Engine *) newEngine
{
    [engine release];
    engine = [newEngine retain];
    // ЕЩЕ ХУЖЕ. См. исправленную версию ниже.
} // setEngine
```

Этот код исправляет ранее рассмотренную утечку engine1. Но он не работает в случае, когда newEngine и старый двигатель оказываются одним и тем же объектом.

```
Engine *engine = [Engine new]; // count: 1
Car *car1 = [Car new];
Car *car2 = [Car new];
```

```
[car1 setEngine: engine]; // count: 2
[engine release]; // count: 1
```

```
[car2 setEngine: [car1 engine]]; // Ой!
```

В чем здесь проблема? Вот что произошло. [car1 engine] возвращает указатель на engine, счетчик ссылок которого равен 1. Первая строка setEngine — [engine release], которая обнуляет счетчик ссылок, и объект уничтожается. Теперь и newEngine, и переменная экземпляра engine указывают на освобожденную память, что весьма плохо. Есть лучший способ написать setEngine.

```
– (void) setEngine: (Engine *) newEngine
{
    [newEngine retain];
    [engine release];
    engine = newEngine;
} // setEngine
```

Если сначала увеличить счетчик нового двигателя, а newEngine — тот же объект, что и engine, то счетчик будет увеличен и тут же уменьшен. Но счетчик не уменьшится до нуля, и двигатель не окажется внезапно уничтоженным — что было бы весьма плохо. В методах доступа вполне безопасно увеличить счетчик ссылок нового объекта перед освобождением старого.

ПРИМЕЧАНИЕ

Имеются различные школы написания методов доступа, и между их последователями регулярно возникают бурные споры. Метод, изложенный в данном разделе, хорошо работает и достаточно прост в понимании, но не удивляйтесь, если у других программистов вы увидите другие методы (и не нервничайте, если ваш подход будут критиковать).

Автоосвобождение

Управление памятью может оказаться крепким орешком, и подтверждением тому — проблемы, с которыми мы встретились при написании устанавливающего метода доступа. А сейчас мы будем разбираться с еще одним затруднением. Вы знаете, что по окончании работы с ними объекты должны быть освобождены. В некоторых случаях получить информацию о том, что вы завершили работу с объектом, не так уж легко. Рассмотрим случай метода `description`, который возвращает строку `NSString`, описывающую объект.

```
-(NSString *) description
{
    NSString *description;
    description = [[NSString alloc]
        initWithFormat: @"Мне %d лет", 5];
    return (description);
} // description
```

Здесь при помощи вызова `alloc` создается и возвращается новый экземпляр строки, счетчик ссылок которого получает значение 1. Кто же отвечает за удаление строкового объекта?

Это не может быть метод `description`. Если вы освободите строку описания до того, как вернете ее, счетчик ссылок уменьшится до нуля и объект будет удален немедленно.

Код, который использует описание, может воспользоваться строкой, а после освободить ее, но это делает использование описаний исключительно неудобным. Одна строка должна превратиться в три.

```
NSString *desc = [someObject description];
NSLog(@"%@", desc);
[desc release];
```

Должен быть другой способ. И, к счастью, он есть!

NSAutoreleasePool

В Сосоа есть концепция *пула автоосвобождения*. Вы уже видели класс `NSAutoreleasePool` в шаблонном коде, генерируемом Xcode. Пришло время разобраться, что же это такое.

Само имя говорит о том, что это такое. Это *пул* (коллекция) чего-то, предположительно объектов, который автоматически выполняет освобождение.

У класса `NSObject` имеется метод `autorelease`.

```
-(id) autorelease;
```

Этот метод планирует отправление сообщения `release` в некоторый момент в будущем. Возвращаемое значение метода — объект, который получает это сообщение; метод `retain` использует ту же методику, которая упрощает объединение вызовов в цепочки. При отправлении сообщения `autorelease` объекту он добавляется к `NSAutoreleasePool`. При уничтожении пула всем объектам, находящимся в нем, отсылается сообщение `release`.

ПРИМЕЧАНИЕ

В концепции автоосвобождения нет никакой магии. Вы можете написать собственный пул, используя NSMutableArray для хранения объектов и отправки всем хранящимся объектам сообщения `release` в методе `dealloc` данного пула. Но это не имеет смысла — Apple уже сделала всю сложную работу за вас.

Теперь метод `description` можно переписать следующим образом.

```
-(NSString *) description
{
    NSString *description;
    description = [[NSString alloc]
        initWithFormat: @"Мне %d лет", 5];
    return ([description autorelease]);
} // description
```

Так что можно написать следующий код.

```
NSLog(@"%@@", [someObject description]);
```

Теперь управление памятью отработает корректно, поскольку метод `description` создаст новую строку, передаст ей сообщение `autorelease` и вернет ее функции `NSLog()` для вывода. Поскольку эта строка описания получила сообщение `autorelease`, она помещена в текущий пул автоосвобождения, и в некоторый момент позже, после того как код функции `NSLog()` завершит свою работу, пул будет уничтожен.

О деструкции

Когда же уничтожается пул автоосвобождения (и тем самым шлет сообщения `release` всем содержащимся в нем объектам)? А когда он создается? При использовании инструментария Foundation создание и уничтожение пула выполняются явно.

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
...
[pool release];
```

Когда вы создаете пул автоосвобождения, он автоматически становится активным. Когда вы освобождаете этот пул, его счетчик ссылок обнуляется, так что он удаляется, а в процессе этого он освобождает все находящиеся в нем объекты.

ПРИМЕЧАНИЕ

В генерируемом Xcode исходном тексте вы могли встретить альтернативный способ уничтожения объектов пула автоосвобождения: метод `-drain`. Этот метод опустошает пул, не уничтожая его. Метод `-drain` доступен только в операционной системе Mac OS X 10.4 (Tiger) и более поздних. В вашем собственном коде (не генерируемом Xcode) лучше использовать `-release`, поскольку этот метод работает в любой операционной системе от начала времен.

При использовании AppKit Cocoa автоматически регулярно создает и удаляет пул автоосвобождения. Он делает это после обработки программой текущего события (такого, как

щелчок мыши или нажатие клавиши). Вы можете использовать столько автоосвобождаемых объектов, сколько хотите, и пул автоматически будет их освобождать, как только пользователь выполнит какое-то действие.

Пулы в действии

Программа RetainTracker2 демонстрирует пул автоосвобождения в действии. Ее можно найти в папке 09.02 RetainTracker-2. Эта программа использует тот же класс RetainTracker, что и программа RetainTracker1 (этот класс сообщает посредством функции NSLog() о создании и освобождении объектов).

Вот как выглядит функция main() этой программы.

```
int main (int argc, const char *argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    RetainTracker *tracker;
    tracker = [RetainTracker new]; // count: 1

    [tracker retain];           // count: 2
    [tracker autorelease];     // count: все еще 2
    [tracker release];         // count: 1

    NSLog (@\"Освобождение пула\");
    [pool release];

    return (0);
} // main
```

Начинаем с создания пула автоосвобождения.

```
NSAutoreleasePool *pool;

pool = [[NSAutoreleasePool alloc] init];
```

Когда мы посылаем сообщение autorelease объекту, тот попадает в данный пул.

```
RetainTracker *tracker;
tracker = [RetainTracker new]; // count: 1
```

Здесь создается новый объект tracker. Так как он создается при помощи сообщения new, его счетчик ссылок равен 1.

```
[tracker retain]; // count: 2
```

Теперь объект удерживается нами — просто в демонстрационных целях. Его счетчик ссылок становится равным 2.

```
[tracker autorelease]; // count: все еще 2
```


После этого объект с неизменным счетчиком ссылок попадает в пул автоосвобождения. Когда пул, который теперь содержит ссылку на объект, будет уничтожен, объекту tracker будет послано сообщение release.

```
[tracker release]; // count: 1
```

Теперь захваченный ранее объект освобожден. Его счетчик ссылок остается больше нуля, так что жизненный цикл объекта продолжается.

```
NSLog(@"Освобождение пула");
```

```
[pool release];
```

Теперь мы освобождаем пул. Объект `NSAutoreleasePool` — такой же обычный объект, как и другие, — подчиняется тем же правилам управления памятью, что все объекты. Поскольку пул был создан при помощи сообщения `alloc`, его счетчик ссылок равен 1. Сообщение `release` обнуляет счетчик ссылок, так что пул уничтожается (с вызовом его метода `dealloc`).

Наконец, функция `main` возвращает значение 0, указывающее, что программа выполнена успешно.

```
return (0);
```

```
} // main
```

Можете ли вы предположить, каким будет вывод рассмотренной программы? Не будем вас томить — вот он.

```
init: счетчик равен 1.
```

```
Освобождение пула
```

```
Вызван dealloc. Прощайте!
```

Как вы, вероятно, и предполагали, сначала вызывается `NSLog()` из функции `main()`, а потом — `NSLog()` из `RetainTracker`.

Правила управления памятью Cocoa

Теперь вы знакомы со всеми сообщениями — `retain`, `release` и `autorelease`. У Cocoa имеется ряд соглашений, связанных с управлением памятью. Это достаточно простые правила, последовательно применяемые во всем инструментарии.

ПРИМЕЧАНИЕ

Распространенная ошибка при программировании с использованием Cocoa заключается в "забывании" этих правил, а также в попытке считать их сложнее, чем они есть на самом деле. Если вы рассыпаете по всему исходному тексту `retain` и `release` в надежде исправить некоторую ошибку, значит, вы не поняли эти правила. Это означает, что пора остановиться, глубоко вдохнуть, может быть, даже сделать перерыв и перекусить, а потом прочесть их снова со свежей головой.

Вот эти правила.

- Когда создаете объект с использованием `new`, `alloc` или `copy`, счетчик ссылок объекта становится равен 1. Вы отвечаете за отправку объекту сообщения `release` или `autorelease` по окончании работы с ним. Таким образом, этот объект уничтожается, когда становится более ненужным.

- Если вы захватываете объекты при помощи некоторого другого механизма, считайте, что его счетчик ссылок равен 1 и что ему было послано сообщение autorelease. Вы не должны предпринимать никакие дальнейшие действия для гарантии его уничтожения. Если объект вам нужен в течение некоторого времени, перед началом работы захватите его при помощи сообщения retain, а по окончании — освободите сообщением release.
- Если вы захватили объект, в конечном счете обязаны освободить его. Сообщения retain и release должны быть сбалансированы.

И это все — только три правила.

Можете запомнить и повторять мантру: полученное через new, alloc и copy надо либо освободить, либо автоосвободить.

Временные объекты

Давайте рассмотрим обычный сценарий жизненного цикла. Мы используем объект в некотором фрагменте кода временно и не намерены хранить его очень долго. Если мы получаем объект при помощи new, alloc или copy, то должны обеспечить его удаление, обычно при помощи сообщения release.

```
NSMutableArray *array;
array = [[NSMutableArray alloc] init]; // count: 1
```

// Использование массива

```
[array release]; // count: 0
```

Если вы получаете объект как-то иначе, например при помощи arrayWithCapacity:, то не должны беспокоиться об его уничтожении.

```
NSMutableArray *array;
array = [NSMutableArray arrayWithCapacity: 17];
// count: 1, autorelease
// Использование массива
```

Метод arrayWithCapacity: — не alloc, не new и не copy, так что можете считать, что возвращенный методом объект имеет счетчик ссылок, равный 1, и уже помещен в пул автоосвобождения. При уничтожении этого пула объекту array посылается сообщение release, его счетчик ссылок обнуляется, а память возвращается системе.

Вот фрагмент кода, использующего NSColor.

```
NSColor *color;
color = [NSColor blueColor];
// Использование color
```

Метод blueColor — не alloc, не new и не copy, так что можете считать, что возвращенный методом объект имеет счетчик ссылок, равный 1, и уже помещен в пул автоосвобождения. blueColor возвращает глобальный синглтон, т.е. объект, который используется всеми нуж-

дающимися в нем программами, и этот объект никогда не уничтожается, но вам незачем беспокоиться о деталях его реализации. Все, что вам надо знать — это то, что вы не должны явно освобождать объект `color`.

Захват объектов

Зачастую вам надо, чтобы объект существовал дольше, чем просто пару строк кода. Обычно такие объекты помещаются в переменные экземпляра других объектов, добавляясь в коллекции наподобие `NSArray` или `NSDictionary` или, что встречается существенно реже, хранятся как глобальные переменные.

Если вы получаете объект при помощи `alloc`, `new` или `copy`, не должны выполнять никакие особые действия. Счетчик ссылок такого объекта оказывается равен 1, так что он никуда не исчезает. Просто следует освободить этот объект в методе `dealloc` объекта-владельца, захватившего его.

```
- (void) doStuff
{
    // flonkArray — переменная экземпляра
    flonkArray = [NSMutableArray new]; // count: 1
} // doStuff

- (void) dealloc
{
    [flonkArray release]; // count: 0
    [super dealloc];
} // dealloc
```

Если вы получаете объект каким-то другим способом, вам надо не забыть захватить его. При написании приложений с графическим интерфейсом пользователя следует думать в терминах цикла событий. Следует обязательно захватывать автоосвобождаемые объекты, продолжительность жизни которых превышает текущий цикл событий.

Что же такое цикл событий? Типичное графическое приложение затрачивает массу времени на ожидание ввода пользователя. Программа дремлет в ожидании, пока этот тормоз-пользователь наконец-то щелкнет мышью или нажмет клавишу. Когда это случится, программа немедленно проснется и выполнит необходимые действия в ответ на происшедшее событие. После обработки события приложение снова засыпает в ожидании очередного события. Чтобы ваше приложение потребляло поменьше памяти, `Socoa` создает пул автоосвобождения перед началом обработки события и уничтожает его после того, как событие обработано. Тем самым количество памяти, выделяемой для временных объектов, поддерживается на минимальном уровне.

При использовании автоосвобождаемых объектов предыдущие методы должны быть записаны следующим образом.

```
- (void) doStuff
{
    // flonkArray — переменная экземпляра
    flonkArray
        = [NSMutableArray arrayWithCapacity: 17];
```

```

        [flonkArray retain]; // count: 1, autorelease
    } // doStuff // count: 2, 1 autorelease

- (void) dealloc
{
    [flonkArray release]; // count: 0
    [super dealloc];
} // dealloc

```

В конце текущего цикла событий (в случае программы с графическим интерфейсом пользователя) или при уничтожении пула автоосвобождения flonkArray получает сообщение release, которое снижает значение счетчика ссылок с 2 до 1. Так как значение счетчика больше нуля, объект остается в живых. Мы должны освободить объект в своем методе dealloc, чтобы освободить занимаемую им память. Если в doStuff сообщение retain не будет послано, объект flonkArray окажется неожиданно уничтоженным.

Вспомните, что пул автоосвобождения очищается в точно определенные моменты: когда он явно уничтожается в вашем коде и, при использовании AppKit, в конце цикла события. Вам не надо беспокоиться ни об освобождении пула в случайные моменты времени, ни о захвате каждого используемого объекта при помощи сообщения retain, так как пул не будет очищен прямо среди выполнения вашей функции.

ПРИНУДИТЕЛЬНАЯ ОЧИСТКА

Иногда пулы автоосвобождения очищаются не так часто, как хотелось бы. Очень часто в различных рассылках, посвященных Cocoa, поднимается один и тот же вопрос: "Я послал всем используемым мною объектам сообщение autorelease, но потребляемая моей программой память вырастает до невероятных размеров". Такая проблема обычно вызывается кодом наподобие следующего.

```

int i;
for (i = 0; i < 1000000; i++) {
    id object = [someArray objectAtIndex: i];
    NSString *desc = [object description];
    // Какие-то действия с описанием
}

```

Здесь работает цикл, который генерирует автоосвобождаемый объект (а может, два или десять) всякий раз при выполнении очередной итерации. Вспомните, что пул автоосвобождения очищается только в точно определенные моменты времени, и середина цикла в эти моменты не входит. В данном цикле создается миллион строк описания, и все они помещаются в текущий пул автоосвобождения, так что у нас получается миллион строк, ждущих своего часа. Когда пул будет уничтожен, всему этому миллиону строк придет конец, но не ранее.

Обходной путь заключается в создании собственного пула автоосвобождения внутри цикла. Таким образом, например, каждую тысячу итераций вы можете уничтожать пул и создавать новый (как показано далее; новый код выделен полужирным шрифтом).

```

NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];

```

```

int i;
for (i = 0; i < 1000000; i++) {
    id object = [someArray objectAtIndex:i];
    NSString *desc = [object description];
    // Какие-то действия с описанием
    if (i % 1000 == 0) {
        [pool release];
        pool = [[NSAutoreleasePool alloc] init];
    }
}
[pool release]

```

Каждую тысячу проходов по циклу новый пул уничтожается и создается вновь. Теперь одновременно сосуществует не более тысячи строк описания, так что потребляемая память падает на порядки. Создание и уничтожение пула автоосвобождения — операция достаточно дешевая, так что новый пул можно создавать даже на каждой итерации цикла.

Пулы автоосвобождения поддерживаются в виде стека: при создании нового пула он добавляется на вершину стека. Сообщение autorelease помещает его получателя в верхний пул в стеке. Если вы помещаете объект в пул, а затем создаете новый пул и уничтожаете его, указанный автоосвобождаемый объект продолжает существование, так как пул, хранящий этот объект, продолжает существовать.

Чисто там, где убирают

В Objective-C 2.0 введена возможность так называемой сборки мусора. Программисты на Java или Python хорошо знакомы с этой концепцией. Вы просто создаете и используете объекты, а затем — вы не поверите! — просто забываете о них. Система автоматически разбирается, какие объекты все еще используются, а какие можно убирать. Выполнить включение сборки мусора очень просто — перейдите на вкладку Build окна информации о проекте и выберите на ней Required [-fobjc-gc-only], как показано на рис. 9.1.

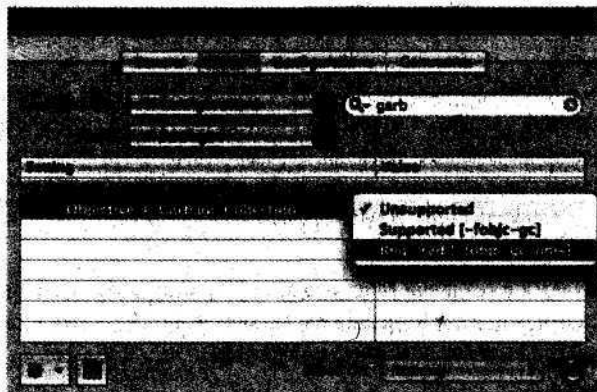


Рис. 9.1. Включение сборки мусора

ПРИМЕЧАНИЕ

Опция `-fobjc-gc` предназначена для создания кода, поддерживающего как сборку мусора, так и счетчики ссылок, как, например, код библиотеки, которая может быть использована в обеих средах.

При включении сборки мусора вызовы обычной системы управления памятью превращаются в холостые команды, не выполняющие никаких действий.

Вот как работает сборщик мусора в Objective-C. Время от времени сборщик мусора начинает просмотр ваших переменных и объектов, а также следует по всем исходящим из них указателям. Любой объект, на который в программе нет ни одного указателя, считается мусором, и его память освобождается. Худшее, что вы можете сделать — это сохранить указатель на более не требующийся вам объект. Поэтому, если вы указали на объект в переменной экземпляра (вспомните композицию), то убедитесь, что этой переменной присвоено значение `nil`, которое удаляет ссылку на этот объект и позволяет сборщику мусора удалить его.

Подобно пулу автоосвобождения, сборка мусора запускается в конце цикла событий. Вы можете также самостоятельно запустить ее, если ваша программа не использует графический интерфейс пользователя, но эта тема выходит за рамки нашей главы.

При использовании сборки мусора вам не приходится беспокоиться об управлении памятью. Есть некоторые нюансы при работе с памятью, выделенной при помощи функции `malloc`, или с объектами Core Foundation, но здесь мы не будем их обсуждать. Пока что вы можете просто создавать объекты и не беспокоиться об их освобождении.

Заметим, что вы не можете использовать сборку мусора при разработке программного обеспечения для iPhone. В действительности Apple рекомендует при программировании для iPhone избегать использования `autorelease` в вашем собственном коде и функций, которые возвращают автоосвобожденные объекты.

Резюме

В этой главе вы познакомились с методами управления памятью Cocoa: `retain`, `release` и `autorelease`.

Каждый объект поддерживает счетчик ссылок. Объекты создаются со счетчиком ссылок, равным 1. При захвате объекта его счетчик увеличивается на 1, а при освобождении объекта — уменьшается на 1. Когда счетчик ссылок обнуляется, объект уничтожается. Сначала вызывается метод `dealloc` объекта, а затем его память возвращается системе и может быть использована при создании другого объекта.

Когда объект получает сообщение `autorelease`, его счетчик ссылок немедленно не изменяется; вместо этого объект помещается в пул `NSAutoreleasePool`. При уничтожении этого пула всем находящимся в нем объектам посылается сообщение `release`. При этом у всех объектов, которые были автоосвобождены, счетчик ссылок уменьшается на 1. Если счетчик обнуляется, объект уничтожается. При работе с AppKit пул автоосвобождения создается и уничтожается в точно определенные моменты времени, в частности, по окончании обработки события пользователя. В прочих случаях вы отвечаете за создание вашего собственного пула автоосвобождения (соответствующий код включается в шаблон исходного текста Foundation).

У Сосоа есть три правила по работе с объектами и их счетчиками ссылок.

- Когда вы создаете объект с использованием `new`, `alloc` или `copy`, счетчик ссылок объекта становится равен 1. Вы отвечаете за отправку объекту сообщения `release` или `autorelease` по окончании работы с ним. Таким образом, этот объект уничтожается, когда становится более ненужным.
- Если вы захватываете объекты при помощи некоторого другого механизма, считайте, что его счетчик ссылок равен 1 и что ему было послано сообщение `autorelease`. Вы не должны предпринимать никакие дальнейшие действия для гарантии его уничтожения. Если объект вам нужен в течение некоторого времени, перед началом работы захватите его при помощи сообщения `retain`, а по окончании — освободите сообщением `release`.
- Если вы захватили объект, в конечном счете обязаны освободить его. Сообщения `retain` и `release` должны быть сбалансированы.

Далее мы с вами познакомимся с методом `init`.

Инициализация объекта

До сих пор мы создавали объекты двумя разными путями. Первый из них — `[SomeClass new]`, а второй — `[[SomeClass alloc] init]`. Эти методики эквивалентны, но обычное соглашение Сосоа заключается в использовании `alloc` и `init` вместо `new`. Обычно программисты Сосоа используют `new` в качестве первых шагов до тех пор, пока не станут достаточно хорошо знакомы с `alloc` и `init`. Пришло время и вам пересечь с трехколесного велосипеда на двухколесный.

Выделение объектов

Выделение (allocation) представляет собой процесс рождения нового объекта. Это счастливое время, когда операционная система выделяет для новорожденного немного памяти, в частности, для всех его переменных экземпляра. Это делается путем отправки сообщения `alloc` классу. Сообщение `alloc` также инициализирует всю выделенную память нулями, так что у вас не должно быть проблем с ошибками из-за неинициализированной памяти, заполненной всяким мусором. Все значения типа `BOOL` получают значение `NO`; все числа типа `int` становятся равны 0, а типа `float` — 0.0; все указатели получают значение `nil`.

Вновь выделенный объект не готов к немедленному использованию: перед тем как работать с ним, его следует инициализировать. В некоторых языках, включая C++ и Java, выделение и инициализация объекта выполняются единой операцией с применением конструктора. Objective-C разделяет эту единую операцию на две стадии — выделения и инициализации. Обычно начинающие забывают об этом и используют только одну операцию `alloc`.

```
Car *car = [Car alloc];
```

Это может и сработать, но без инициализации позже вы получите странное поведение (известное под сленговым именем "bag"). Данная глава посвящена этой жизненно важной концепции инициализации.

Инициализация объектов

Напарником выделения является инициализация. Инициализация получает блок памяти и делает из него полезного члена общества. Методы `init` — т.е. методы, выполняющие инициализацию, — почти всегда возвращают инициализированный объект. Вы можете (и должны) объединять выделение и инициализацию в цепочку так

```
Car *car = [[Car alloc] init];
```

но не так:

```
Car *car = [Car alloc];  
[car init];
```

Такая методика важна в связи с тем, что метод инициализации может вернуть объект, отличающийся от выделенного. Если считаете это странным — вы правы, но такое может случиться.

Почему программист может захотеть, чтобы метод `init` вернул другой объект? Если вы вспомните обсуждение кластеров классов в конце главы 8, "Краткий обзор Foundation Kit", то увидите, что классы типа `NSString` и `NSArray` — просто декоративный фасад, скрывающий массу специализированных классов. Метод `init` может получать аргументы, так что код метода может, рассмотрев полученные аргументы, решить, что более подходящим будет объект иного класса. Пусть, например, новая строка создается из очень длинной строки или, скажем, из строки арабских символов. На основании этой информации инициализатор строки может решить создать объект иного класса, лучше отвечающего требованиям исходной строки, и вернуть его вместо исходного объекта.

Написание методов инициализации

Ранее при рассмотрении методов инициализации встречались моменты, когда мы просили вас потерпеть и принять все как есть, несмотря на определенную загадочность того или иного фрагмента кода. Вот метод `init` из ранней версии `CarParts`.

```
- (id) init  
{  
    if (self = [super init]) {  
        engine = [Engine new];  
        tires[0] = [Tire new];  
        tires[1] = [Tire new];  
        tires[2] = [Tire new];  
        tires[3] = [Tire new];  
    }  
    return (self);  
} // init
```

Основная странность в этом методе заключена в первой строке.

```
if (self = [super init]) {
```

Этот код подразумевает, что `self` может изменяться. Изменение `self` посреди метода? Что за глупость? Может, и глупость, но не в этот раз. Первый фрагмент кода, выполняющийся в этой инструкции, — `[super init]`. Этот код предлагает надклассу выполнить его работу по инициализации. У классов, порожденных из `NSObject`, вызов инициализации надкласса позволяет `NSObject` выполнить все действия, необходимые для того, чтобы данный объект мог отвечать на сообщения и работать со счетчиком ссылок. В случае классов, унаследованных от других классов, это позволяет последним выполнить их собственный код инициализации.

Мы уже говорили, что методы наподобие `init` могут возвращать совершенно иные объекты. Вспомните, что переменные экземпляра располагаются в памяти на фиксированном расстоянии от скрытого параметра `self`. Если метод `init` возвращает новый объект, требуется соответствующее обновление `self`, с тем чтобы все последующие ссылки на переменные экземпляра указывали на корректные места в памяти. Вот почему нам требуется выполнение присваивания `self = [super init]`. Учтите, что это присваивание действует на значение `self` только для данного метода. Оно не изменяет ничего за пределами области видимости метода.

Метод `init` может вернуть `nil` при наличии проблем при инициализации объекта. Например, вы можете использовать метод `init`, который получает URL и инициализирует объект изображения файлом, полученным с веб-сайта. Если сеть не работает или в связи с перепроектированием веб-сайта этот URL больше не ссылается на нужный файл, вы не в состоянии создать корректный объект. В этом случае метод `init` должен вернуть `nil`, указывая, что объект не может быть инициализирован. Проверка `if (self = [super init])` не допустит выполнения кода, если `[super init]` вернет значение `nil`. Объединение присваивания с проверкой на ненулевое значение — классическая идиома C, которая благополучно применяется и в Objective-C.

Наш код получает объект и выполняет исходный текст в фигурных скобках инструкции `if`. В исходном методе `init` класса `Car` тело `if` создает двигатель и четыре колеса. С точки зрения управления памятью код выполняет корректные действия, так как объекты, возвращаемые сообщением `new`, получают счетчик ссылок, равный 1.

Последняя строка метода —

```
return (self);
```

Метод `init` возвращает только что инициализированный объект. Поскольку мы присвоили значение, возвращаемое вызовом `[super init]`, переменной `self`, то ее мы и должны вернуть.

СПОРЫ ОБ INIT

Некоторые программисты не любят объединять присваивание и проверку и предпочитают начинать свои методы `init` следующим образом.

```
self = [super init];
if (self) {
    ....
}
return (self);
```

Это нормально. Главное — присваивание возвращаемого значения переменной `self`, в особенности при обращении к любым переменным экземпляра. Как именно вы это сделаете — значения не имеет. Просто объединение присваивания и проверки — распространенная практика, которую можно встретить в коде у многих программистов.

Стиль `self = [super init]` вызывает определенные дебаты. Одни говорят, что вы должны всегда так поступать, другие утверждают, что смена объекта — настолько редкое явление, что можно использовать просто вызов `[super init]`. Сторонники этого мнения указывают, что даже если `init` надклассом изменяет объект, вероятно, в новом объекте не будет никаких добавленных вами переменных экземпляра.

Это действительно сложная проблема, если подходить к ней с абстрактной точки зрения, но в реальной жизни такое случается нечасто. Мы рекомендуем использование метода `if (self = [super init])` по той причине, что оно безопасно и перехватывает поведение "init вернул nil" некоторых методов `init`. Но если вы будете использовать простой вызов `[super init]`, это тоже нормально. Просто будьте готовы немного повозиться с отладчиком, если вдруг вы в встретитесь с одним из упомянутых редких случаев.

Что делать при инициализации

Что должно находиться в ваших методах `init`? Это место для размещения кода инициализации. Вы присваиваете значения переменным экземпляра и создаете другие объекты, в которых ваши объекты нуждаются для корректной работы. При написании своих методов `init` вы должны решить, какое количество работы следует здесь выполнить. Программы семейства CarParts демонстрируют два разных подхода к этой проблеме.

Первый использует метод `init` для создания двигателя и четырех колес. Это делает объект Car немедленно готовым к действию: вызовите `alloc` и `init` и садитесь за руль. В следующей версии в методе `init` мы вообще ничего не делали. Мы просто оставили пустое место для двигателя и колес. Код, создавший объект, должен был затем создать двигатель и колеса и установить их при помощи методов доступа.

Какой способ правильнее? Решение зависит от выбранного вами соотношения гибкости и производительности, как и многие компромиссы в программировании. Исходный метод `init` очень удобен. Если предполагаемое использование класса Car заключается в создании базового автомобиля и последующем его применении, то это правильный дизайн.

С другой стороны, если автомобиль будет часто перенастраиваемым с применением различных двигателей и колес, как в играх-гонках, то установленные в методе `init` двигатель и колеса будут тут же выброшены и заменены другими. Какое расточительство! Объекты создаются и уничтожаются, никак не используясь.

ПРИМЕЧАНИЕ

Даже если вы не предоставляете методы для настройки атрибутов ваших объектов, можете подождать с созданием этих атрибутов до тех пор, пока они реально не понадобятся. Такая методика называется *отложенным вычислением*; она может обеспечить определенное повышение производительности в ситуации, когда в методе `-init` создаются сложные объекты, которые в действительности могут оказаться невостребованными.

Инициализаторы для удобства

У некоторых объектов имеется по несколько методов, начинающихся со слова `init`. В действительности важно понять и запомнить, что в методах `init` нет ничего выдающегося. Это обычные методы, следующие соглашению об именовании.

Многие классы имеют *инициализаторы для удобства*. Эти методы `init` выполняют дополнительные действия, экономя ваши силы и время. Чтобы помочь вам понять, о чем мы говорим, рассмотрим некоторые из методов `init` класса `NSString`.

– (id) `init`;

Этот базовый метод инициализирует новую пустую строку. В случае неизменяемой строки `NSString` этот метод не слишком полезен. Но вы можете выделить и инициализировать новую строку `NSMutableString` и добавлять символы в нее.

```
NSString *emptyString = [[NSString alloc] init];
```

Этот код дает вам пустую строку.

– (id) `initWithFormat: (NSString *) format, ...;`

Эта версия инициализирует новую строку результатом операции форматирования, как при применении функции `NSLog()` и метода класса `stringWithFormat:`, с которым мы встречались в главе 7, “Еще об Xcode”. Вот пример, в котором используется этот новый метод инициализации.

```
string = [[NSString alloc]
initWithFormat: @"%d or %d", 25, 624];
```

Здесь в результате получаем строку со значением “25 or 624”.

– (id) `initWithContentsOfFile: (NSString *) path;`

Метод `initWithContentsOfFile:` открывает текстовый файл с указанным именем, считывает его содержимое и инициализирует им строку. Приведенные далее строки читают файл `/tmp/words.txt`.

```
string = [[NSString alloc]
initWithContentsOfFile: @"/tmp/words.txt"];
```

Это очень удобно. Для выполнения этих действий потребовалось бы множество строк кода на C (открыть файл, считать блок данных, добавить к строке, убедиться в наличии в нужном месте завершающего нулевого байта, закрыть файл). В Objective-C все это превращается в одну строку кода. Прелестно!

Практический пример

Вернемся к программе `CarParts`, с которой мы в последний раз встречались в главе 6, “Организация исходных файлов”, когда выделяли каждый класс в свой собственный исходный файл. В этот раз мы добавим инициализацию к классу `Tire`, а попутно разберемся с управ-

лением памятью в классе Car. Окончательную версию программы можно найти в каталоге 10.01 CarPartsInit (или в каталоге 10.01 CarPartsInit-GC, где находится версия программы с использованием сборки мусора).

Инициализация колес

Реальные колеса на самом деле существенно более интересные вещи, чем моделируемые нами до сих пор в программе CarParts. В реальных колесах приходится постоянно следить за давлением в камере и глубиной рисунка протектора. Расширим наш класс Tire, чтобы он мог следить за давлением в камере и глубиной рисунка протектора. Вот как выглядит класс с двумя добавленными переменными экземпляра и соответствующими методами доступа.

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

```
@interface Tire : NSObject
{
    float pressure;
    float treadDepth;
}
```

```
- (void) setPressure: (float) pressure;
```

```
- (float) pressure;
```

```
- (void) setTreadDepth: (float) treadDepth;
```

```
- (float) treadDepth;
```

```
@end // Tire
```

А вот как выглядит реализация класса Tire — в ней нет никаких хитростей.

```
#import "Tire.h"
```

```
@implementation Tire
```

```
- (id) init
```

```
{
    if (self = [super init]) {
        pressure = 34.0;
        treadDepth = 20.0;
    }
    return (self);
} // init
```

```
- (void) setPressure: (float) p
```

```
{
    pressure = p;
} // setPressure
```

```
- (float) pressure
```



```

{
    return (pressure);
} // pressure

- (void) setTreadDepth: (float) td
{
    treadDepth = td;
} // setTreadDepth

- (float) treadDepth
{
    return (treadDepth);
} // treadDepth

- (NSString *) description
{
    NSString *desc;
    desc = [NSString stringWithFormat:
        @"Колесо: давление: %.1f протектор: %.1f",
        pressure, treadDepth];
    return (desc);
} // description
@end // Tire

```

Методы доступа предоставляют пользователю способ изменения давления и глубины рисунка. Давайте рассмотрим метод `init`.

```

- (id) init
{
    if (self = [super init]) {
        pressure = 1.5;
        treadDepth = 20.0;
    }
    return (self);
} // init

```

Для вас здесь не должно быть никаких сюрпризов. Инициализируется надкласс (в данном случае `NSObject`), и возвращаемое при этом значение присваивается переменной `self`. Затем присваиваются значения по умолчанию переменным экземпляра. Вот как создается новое колесо.

```
Tire *tire = [[Tire alloc] init];
```

Давление в камере при этом равно 1.5 атм., а глубина рисунка протектора — 20 мм.

Мы также должны изменить метод `description`.

```

- (NSString *) description
{
    NSString *desc;
    desc = [NSString stringWithFormat:
        @"Колесо: давление: %.1f протектор: %.1f",

```

```

        pressure, treadDepth];
    return (desc);
} // description

```

Теперь метод `description` использует метод `stringWithFormat`: класса `NSString` для создания строки, включающей давление и глубину рисунка. Следует ли данный метод нашим правилам управления памятью? Да. Так как объект не создан при помощи `alloc`, `copy` или `new`, его счетчик ссылок равен 1, и мы можем рассматривать его как автоосвобожденный. Так что данная строка будет удалена при уничтожении пула автоосвобождения.

Обновление функции `main()`

Вот как теперь выглядит файл `main.m` — он стал несколько сложнее, чем раньше.

```

#import "Engine.h"
#import "Car.h"
#import "Slant6.h"
#import "AllWeatherRadial.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    Car *car = [[Car alloc] init];

    int i;
    for (i = 0; i < 4; i++) {
        Tire *tire;
        tire = [[Tire alloc] init];

        [tire setPressure: 2.0 + i/10.0];
        [tire setTreadDepth: 33 - i];

        [car setTire: tire
         atIndex: i];

        [tire release];
    }
    Engine *engine = [[Slant6 alloc] init];
    [car setEngine: engine];

    [car print];
    [car release];

    [pool release];
    return (0);
} // main

```

Разберем функцию `main()` строка за строкой. Начинается функция с создания пула автоосвобождения для автоосвобождаемых объектов.

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
```

Затем создается новый автомобиль с использованием сообщений `alloc` и `init`.

```
Car *car = [[Car alloc] init];
```

После этого идет цикл, выполняющий четыре итерации, в котором создаются новые колеса.

```
int i;
for (i = 0; i < 4; i++) {
```

На каждой итерации цикла создается и инициализируется новое колесо.

```
Tire *tire;
tire = [[Tire alloc] init];
```

Каждое колесо начинает свое существование с давлением и глубиной рисунка по умолчанию, заданными в методе `init`. Но мы изменяем эти значения — просто так, шутки ради. Делаем мы это при помощи методов доступа.

```
[tire setPressure: 2.0 + i/10.0];
[tire setTreadDepth: 33 - i];
```

Затем мы прикручиваем колесо к автомобилю.

```
[car setTire: tire
 atIndex: i];
```

Когда колесо установлено на место, мы освобождаем его.

```
[tire release];
```

Этот код предполагает, что `Car` корректно управляет памятью, выполняя захват объекта. Заметим, что класс `Car`, показанный в главе 6, "Организация исходных файлов", не следовал правилам управления памятью — тогда нам о них еще не было известно — но, как мы увидим, в новой версии программы все сделано так, как надо.

После того как все колеса оказываются на своих местах, создается новый двигатель, который монтируется в автомобиль.

```
Engine *engine = [[Slant6 alloc] init];
[car setEngine: engine];
[engine release];
```

Двигатель освобождается, как и колеса, поскольку мы больше не используем его. Ответственность за удаление двигателя переходит к автомобилю.

Наконец, `Car` получает сообщение `print`, которое заставляет его вывести информацию о самом себе, после чего объект освобождается (так как мы завершили на этом работу с ним).

```
[car print];
[car release];
```

После этого освобождается пул автоосвобождения. Это приводит к обнулению его счетчика ссылок, освобождению пула и отправке сообщений `release` всем объектам в пуле. Когда это происходит, уничтожаются строки `NSString`, сгенерированные методом `description` класса `Tire`.

```
[pool release];
```

На этом функция `main()` завершается. Но перед тем как запустить программу, нам надо исправить класс `Car`, с тем чтобы он корректно работал с памятью. Но перед этим давайте взглянем на вариант функции `main()`, предназначенной для работы в системе со сборкой мусора.

```
int main (int argc, const char * argv[])
{
    Car *car = [[Car alloc] init];

    int i;
    for (i = 0; i < 4; i++) {
        Tire *tire;
        tire = [[Tire alloc] init];

        [tire setPressure: 2.0 + i/10.0];
        [tire setTreadDepth: 33 - i];

        [car setTire: tire
         atIndex: i];
    }
    Engine *engine = [[Slant6 alloc] init];
    [car setEngine: engine];

    [car print];

    return (0);
} // main
```

Как видите, этот вариант несколько короче и проще за счет отсутствия вызовов, связанных с системой управления памятью.

Автомойка

Пора заняться кодом класса `Car` и довести его до блеска. Начнем с того, что вместо применения обычного массива `C` в классе `Car` воспользуемся `NSMutableArray`. Почему? Потому что это обеспечит нас автоматическим контролем границ. Для этого нам надо изменить раздел `@interface` класса `Car` (изменения выделены полужирным шрифтом).

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

```
@class Tire;
@class Engine;
```

```
@interface Car : NSObject
{
```

```

NSMutableDictionary *tires;
Engine *engine;
}

- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;
- (void) setTire: (Tire *) tire
    atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
- (void) print;

@end // Car

```

Обновлен практически каждый метод класса Car — теперь эти методы следуют правилам управления памятью. Начнем с метода `init`.

```

- (id) init
{
    if (self = [super init]) {
        tires = [[NSMutableArray alloc] init];
        int i;
        for (i = 0; i < 4; i++) {
            [tires addObject: [NSNull null]];
        }
    }
    return (self);
} // init

```

Здесь вы видите уже неоднократно встречавшуюся конструкцию `self = [super init]`. Как вы теперь знаете, она просто обеспечивает инициализацию объекта надкласса.

Затем создается массив `NSMutableArray`. У этого класса имеется очень удобный метод `replaceObjectAtIndex:withObject:`, который идеально подходит для реализации метода `setTire:atIndex:`. Чтобы использовать этот метод, должны иметься элементы массива с заданными именами, чтобы было что заменять на новые элементы. Но только что созданный массив `NSMutableArray` пуст, так что нам надо чем-то его заполнить — любыми пустышками. Такими пустышками служат объекты `NSNull`. Вообще говоря, без этого можно обойтись, но в нашем случае такой трюк существенно облегчает жизнь.

В конце метода `init` мы возвращаем `self`, так как это тот объект, который мы только что инициализировали.

Теперь перейдем к методам доступа двигателя. Это `setEngine:` и `engine`. Метод `setEngine:` использует методику "захватывать переданный объект, а затем освобождать текущий", с которой мы встречались ранее.

```

- (void) setEngine: (Engine *) newEngine
{
    [newEngine retain];
    [engine release];
    engine = newEngine;
} // setEngine

```

Метод доступа engine просто возвращает текущий двигатель.

```
- (Engine *) engine
{
    return (engine);
} // engine
```

Теперь перейдем к функциям доступа к колесам. Вот как выглядит устанавливающий метод доступа.

```
- (void) setTire: (Tire *) tire
    atIndex: (int) index
{
    [tires replaceObjectAtIndex: index
        withObject: tire];
} // setTire:atIndex:
```

Рассматриваемый метод доступа использует метод `replaceObjectAtIndex:withObject:` для удаления текущего объекта из коллекции и замены его новым объектом. Здесь мы не выполняем никакие явные действия по управлению памятью, поскольку класс `NSMutableArray` автоматически захватывает новое колесо и освобождает то, которое до этого находилось в позиции с данным индексом, будь то заглушка `NSNull` или ранее сохраненный объект колеса. `NSMutableArray` освобождает все свои объекты при уничтожении, так что колеса будут своевременно уничтожены.

Получающий метод доступа `tireAtIndex:` для получения колеса из массива использует метод `objectAtIndex:` класса `NSArray`.

```
- (Tire *) tireAtIndex: (int) index
{
    Tire *tire;
    tire = [tires objectAtIndex: index];
    return (tire);
} // tireAtIndex:
```

БЫСТРЫЙ ВОЗВРАТ

Можно совершенно корректно записать метод `tireAtIndex:` в одну строку, возвращая результат вызова `objectAtIndex:`.

```
- (Tire *) tireAtIndex: (int) index
{
    return ([tires objectAtIndex: index]);
} // tireAtIndex:
```

Лишняя переменная в оригинальном коде делает его более простым для чтения и понимания (как минимум, для нас) и упрощает установку точек останова при отладке, так что мы можем просмотреть возвращаемое значение. Она упрощает отладку еще и тем, что позволяет вставить отладочный вызов `NSLog()` между вызовом `objectAtIndex:` и окончанием метода.

Мы должны обеспечить еще и корректное освобождение автомобилем захваченных объектов — в частности, engine и массива tires. Это делается в методе dealloc.

```
- (void) dealloc
{
    [tires release];
    [engine release];
    [super dealloc];
} // dealloc
```

Этого достаточно, чтобы гарантировать, что когда автомобиль отправится на свалку, вся память будет возвращена системе. Не забудьте вызвать метод dealloc надклассом! Такая забывчивость — ошибка весьма распространенная.

Наконец, вот метод print автомобиля, который выводит информацию о двигателе и колесах.

```
- (void) print
{
    int i;
    for (i = 0; i < 4; i++) {
        NSLog(@"%@", [self tireAtIndex: i]);
    }
    NSLog(@"%@", engine);
} // print
```

Метод print обходит все колеса при помощи цикла и выводит информацию о каждом из них. Обратите внимание, что в цикле используется метод tireAtIndex:, а не обращение к массиву. Если вы хотите обращаться непосредственно к массиву, можете это сделать, но применение методов доступа (даже в реализации класса) изолирует ваш код от любых изменений. Например, если в будущем будет изменен механизм крепления колес (например, вы вернетесь к массиву C), то метод print при этом переписывать не придется.

Теперь, наконец-то, мы можем выполнить нашу программу и получить ее вывод, который должен иметь следующий вид.

```
Колесо: давление: 2.0 протектор: 33.0
Колесо: давление: 2.1 протектор: 32.0
Колесо: давление: 2.2 протектор: 31.0
Колесо: давление: 2.3 протектор: 30.0
Я — двигатель slant-6. P-p-p-p!
```

Использование сборки мусора

Ну а как все это выглядит при использовании сборки мусора? Как класс выглядит в этом случае? Метод setEngine становится проще.

```
- (void) setEngine: (Engine *) newEngine
{
    engine = newEngine;
} // setEngine
```

Мы изменяем переменную экземпляра engine. Когда сборщик мусора Сосоа приступит к работе, он обнаружит, что на старый двигатель никто не указывает, так что его можно уничтожить, а занимаемую им память вернуть системе. С другой стороны, переменная экземпляра указывает на newEngine, так что его уничтожать нельзя — это не мусор, раз на него есть ссылка.

Метод dealloc полностью исчезает: в мире, где есть сборщик мусора, dealloc не нужен. Если вам надо выполнить некоторую работу при уничтожении объекта, можно перекрыть метод -finalize, который вызывается при окончательном уничтожении объекта (однако с этим связаны кое-какие тонкости). Но при программировании с Сосоа вам вряд ли понадобится беспокоиться об этом методе.

Инициализаторы для удобства

Никакой код не совершенен — в любой фрагмент можно внести усовершенствования. Давайте вернемся к функции main() и посмотрим, как мы создаем колеса.

```
tire = [[Tire alloc] init];
[tire setPressure: 2.0 + i/10.0];
[tire setTreadDepth: 33 - i];
```

Здесь в трех строках кода посылаются четыре сообщения. Было бы неплохо выполнить все эти действия в одной операции. Этого можно добиться при использовании инициализатора, который одновременно получает давление и глубину рисунка. Вот как выглядит класс Tire с таким инициализатором (выделен полужирным шрифтом).

```
@interface Tire : NSObject
{
    float pressure;
    float treadDepth;
}

- (id) initWithPressure: (float) pressure
    treadDepth: (float) treadDepth;

- (void) setPressure: (float) pressure;
- (float) pressure;

- (void) setTreadDepth: (float) treadDepth;
- (float) treadDepth;
@end // Tire
```

Реализация этого метода достаточно проста, без каких-либо сюрпризов.

```
- (id) initWithPressure: (float) p
    treadDepth: (float) td
{
    if (self = [super init]) {
        pressure = p;
        treadDepth = td;
    }
}
```

```
return (self);
} // initWithPressure:treadDepth:
```

Теперь выделение и инициализация колеса выполняются одной операцией.

```
Tire *tire;
tire = [[Tire alloc]
initWithPressure: 2.0 + i/10.0
treadDepth: 33 - i];
```

Стандартный инициализатор

К сожалению, не все еще хорошо в инициализаторском королевстве. При добавлении инициализаторов обнаруживаются некоторые тонкости. Давайте добавим к классу Tire еще два инициализатора.

```
@interface Tire : NSObject
{
    float pressure;
    float treadDepth;
}

- (id) initWithPressure: (float) pressure;
- (id) initWithTreadDepth: (float) treadDepth;

- (id) initWithPressure: (float) pressure
treadDepth: (float) treadDepth;

- (void) setPressure: (float) pressure;
- (float) pressure;

- (void) setTreadDepth: (float) treadDepth;
- (float) treadDepth;
@end // Tire
```

Эти новые инициализаторы, `initWithPressure:` и `initWithTreadDepth:`, предназначены для тех пользователей, кто знает конкретное давление или глубину рисунка протектора, но не знает значения второго атрибута, и готов принять значения по умолчанию. Вот первая попытка такой инициализации (будет исправлена позже).

```
- (id) initWithPressure: (float) p
{
    if (self = [super init]) {
        pressure = p;
        treadDepth = 20.0;
    }
    return (self);
} // initWithPressure
```

```

- (id) initWithTreadDepth: (float) td
{
    if (self = [super init]) {
        pressure = 1.5;
        treadDepth = td;
    }
    return (self);
} // initWithTreadDepth

```

Теперь у нас есть четыре метода инициализации: `init`, `initWithPressure:`, `initWithTreadDepth:` и `initWithPressure:treadDepth:`. Каждый из них знает давление по умолчанию (1.5), глубину рисунка протектора (20) или оба эти значения. Все это нормально работает, и код вполне корректен.

Проблемы начинаются при наследовании класса `Tire`.

Проблема наследования

У нас уже есть подкласс `AllWeatherRadial` класса `Tire`. Предположим теперь, что мы хотим добавить в `AllWeatherRadial` две новые переменные экземпляра, `rainHandling` и `snowHandling`, которые представляют собой значения с плавающей точкой, указывающие, как колеса ведут себя на мокрой и заснеженной дороге. Нам надо обеспечить получение подходящих значений этими переменными при создании нового `AllWeatherRadial`.

Вот новый интерфейс `AllWeatherRadial`, с новыми переменными экземпляра и функциями доступа.

```

@interface AllWeatherRadial : Tire
{
    float rainHandling;
    float snowHandling;
}

- (void) setRainHandling: (float) rainHandling;
- (float) rainHandling;

- (void) setSnowHandling: (float) snowHandling;
- (float) snowHandling;
@end // AllWeatherRadial

```

Вот как реализованы методы доступа.

```

- (void) setRainHandling: (float) rh
{
    rainHandling = rh;
} // setRainHandling

- (float) rainHandling
{
    return (rainHandling);
} // rainHandling

```

```

- (void) setSnowHandling: (float) sh
{
    snowHandling = sh;
} // setSnowHandling

- (float) snowHandling
{
    return (snowHandling);
} // snowHandling

```

Кроме того, мы обновим метод `description`, который возвращает информацию о колесе.

```

- (NSString *) description
{
    NSString *desc;
    desc = [[NSString alloc] initWithFormat:
        @"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",
        [self pressure], [self treadDepth],
        [self rainHandling],
        [self snowHandling]];
    return (desc);
} // description

```

А вот исходный текст цикла `for` в функции `main()`, который создает новые колеса с их значениями по умолчанию.

```

int i;
for (i = 0; i < 4; i++) {
    AllWeatherRadial *tire;
    tire = [[AllWeatherRadial alloc] init];
    [car setTire: tire
        atIndex: i];
    [tire release];
}

```

При запуске программы мы получим следующее.

```

AllWeatherRadial: 1.5 / 20.0 / 0.0 / 0.0
AllWeatherRadial: 1.5 / 20.0 / 0.0 / 0.0
AllWeatherRadial: 1.5 / 20.0 / 0.0 / 0.0
AllWeatherRadial: 1.5 / 20.0 / 0.0 / 0.0
Я — двигатель slant-6. P-p-p-p-p!

```

Непохоже, чтобы все параметры `AllWeatherRadial` получили подходящие значения по умолчанию. Что же произошло? Мы должны установить значения в методе `init`, а значит, должны перекрыть этот метод. Но помимо метода `init` класс `Tire` имеет и другие методы инициализации — `initWithPressure:`, `initWithTreadDepth:` и `initWithPressure:treadDepth:`. Должны ли мы перекрывать их все? И даже если мы так и поступим, кто может гарантировать, что у класса `Tire` не появится новый инициализатор? Будет невесело, если изменения в классе `Tire` сделают неработоспособным класс `AllWeatherRadial`.

К счастью, создатели Сосоа предвидели эту проблему и разработали концепцию *стандартного инициализатора* (designated initializer). Метод `init` рассматривается как стандартный инициализатор класса. Все методы инициализации класса для выполнения своей инициализирующей деятельности используют этот стандартный инициализатор (вы не запутались?). Подклассы для инициализации надкласса используют его стандартный инициализатор. Обычно таким инициализатором является метод `init`, который принимает наибольшее количество аргументов. Если вы используете чужой код — обратитесь к его документации, чтобы узнать, какой из методов является стандартным инициализатором.

Исправление инициализаторов Tire

Сначала мы должны решить, какой из инициализаторов Tire должен быть посвящен в стандартные. Хорошим выбором может стать `initWithPressure:treadDepth:`. У него наибольшее количество аргументов, и он самый гибкий среди всех инициализаторов.

Все прочие инициализаторы при этом должны быть реализованы посредством стандартного инициализатора, т.е. в терминах `initWithPressure:treadDepth:`. Это выглядит примерно следующим образом.

```
- (id) init
{
    if (self = [self initWithPressure: 1.5
                                treadDepth: 20]) {
    }
    return (self);
} // init

- (id) initWithPressure: (float) p
{
    if (self = [self initWithPressure: p
                                treadDepth: 20.0]) {
    }
    return (self);
} // initWithPressure

- (id) initWithTreadDepth: (float) td
{
    if (self = [self initWithPressure: 34.0
                                treadDepth: td]) {
    }
    return (self);
} // initWithTreadDepth
```

ПРИМЕЧАНИЕ

На самом деле пустые тела конструкций `if` не нужны — мы использовали их только для того, чтобы все методы `init` имели одинаковый вид.

Добавление инициализатора AllWeatherRadial

Теперь пришло время добавить инициализатор к классу AllWeatherRadial. Единственный метод, который следует добавить, — это перекрытие стандартного инициализатора.

```

- (id) initWithPressure: (float) p
    treadDepth: (float) td
{
    if (self = [super initWithPressure: p
        treadDepth: td]) {
        rainHandling = 23.7;
        snowHandling = 42.5;
    }
    return (self);
} // initWithPressure:treadDepth

```

Теперь при запуске программы получим корректные значения по умолчанию.

```

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5
AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5
Я- двигатель slant-6. P-p-p-p-p!

```

И правда, p-p-p-p-p!

Правила инициализаторов

Вы не обязаны создавать метод инициализации для своего класса. Если у класса нет настраиваемых состояний или стандартное поведение alloc, обнуляющее все переменные, вас устраивает, можете не беспокоиться о методе init.

Если вы пишете инициализатор, не забудьте вызвать в своем стандартном инициализаторе стандартный инициализатор надкласса.

Если у вас имеется несколько инициализаторов, выберите один в качестве стандартного. Именно в нем вызывается стандартный инициализатор надкласса. Реализуйте все остальные инициализаторы через стандартный, как мы делали это ранее.

Резюме

В этой главе вы познакомились с выделением и инициализацией объектов. В Cocoa это две разные операции. Метод alloc (метод класса NSObject) выделяет блок памяти и обнуляет его. Методы init, являющиеся методами экземпляра, инициализируют объект.

Класс может иметь несколько методов init. Обычно это методы, настраивающие объекты разными способами, и вы можете выбрать наиболее простой и подходящий вам. Один из

этих методов выбирается в качестве стандартного инициализатора. Все прочие методы инициализации работают посредством данного стандартного инициализатора.

В ваших собственных методах инициализации вы должны вызвать либо собственный стандартный инициализатор, либо стандартный инициализатор надкласса. Не забудьте присвоить значение, возвращаемое инициализатором надкласса, переменной `self` и вернуть это значение из своего метода инициализации. Возможна ситуация, когда надкласс решает вернуть совершенно другой объект.

Следующей темой нашего рассмотрения будут свойства — быстрый и простой способ создания методов доступа.

Свойства

Помните, как мы писали — кажется, уже давным-давно — методы доступа к нашим переменным экземпляра? Мы писали массу стандартного шаблонного кода, создавая метод `-setBlah` для установки значения атрибута `blah` и метод `-blah` — для его получения. Если атрибут являлся объектом, то нам требовалось захватить новый объект, а затем отпустить старый. Имеются утилиты, которые превращают определение вашего класса в объявления методов и определения, которые можно вставить в ваши файлы. Тем не менее написание методов доступа остается ужасно скучной и монотонной работой.

В Objective-C 2.0 Apple ввела *свойства*, комбинацию новых директив компилятора и нового синтаксиса методов доступа. Эта новая возможность существенно снижает количество бессмысленного кода, который вам приходится писать. В этой главе мы будем модифицировать проект 10.01 CarParts-Init, обучая его использовать свойства. Окончательный вид проекта, который он приобретает в этой главе, можно найти в папке 11.01 CarProperties.

Не забывайте, что возможности Objective-C 2.0 можно применять только в операционной системе Mac OS X 10.5 (Leopard) или более поздней. Свойства широко используются в новых частях Cocoa (в особенности в Core Animation), а также при разработке приложений для iPhone, так что лучше быть с ними хорошо знакомым.

Упрощение кода при помощи свойств

Начнем с преобразования для использования свойств одного из простейших классов, AllWeatherRadial. Чтобы сделать работу более интересной, добавим в функцию `main()` пару вызовов, изменяющих значения создаваемого объекта AllWeatherRadial. Мы смоделируем покупку колес с другими характеристиками на распродаже.

Вот как теперь выглядит функция main() (новые строки выделены полужирным шрифтом).

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    Car *car = [[Car alloc] init];

    int i;
    for (i = 0; i < 4; i++) {
        AllWeatherRadial *tire;

        tire = [[AllWeatherRadial alloc] init];
        [tire setRainHandling: 20 + i];
        [tire setSnowHandling: 28 + i];
        NSLog(@"Свойства колеса: %.f %.f",
                [tire rainHandling],
                [tire snowHandling]);

        [car setTire: tire
                atIndex: i];

        [tire release];
    }

    Engine *engine = [[Slant6 alloc] init];
    [car setEngine: engine];

    [car print];
    [car release];

    [pool release];

    return (0);
} // main
```

Если вы запустите эту программу, то увидите следующий ее вывод.

```
Свойства колеса: 20 28
Свойства колеса: 21 29
Свойства колеса: 22 30
Свойства колеса: 23 31
AllWeatherRadial: 1.5 / 20.0 / 20.0 / 28.0
AllWeatherRadial: 1.5 / 20.0 / 21.0 / 29.0
AllWeatherRadial: 1.5 / 20.0 / 22.0 / 30.0
AllWeatherRadial: 1.5 / 20.0 / 23.0 / 31.0
Я - двигатель slant-6. P-p-p-p-p!
```

Упрощение интерфейса

Давайте еще раз посмотрим на интерфейс класса AllWeatherRadial.

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

```
@interface AllWeatherRadial : Tire {
    float rainHandling;
    float snowHandling;
}
```

```
- (void) setRainHandling: (float) rainHandling;
- (float) rainHandling;
```

```
- (void) setSnowHandling: (float) snowHandling;
- (float) snowHandling;
```

```
@end // AllWeatherRadial
```

Все это вам должно быть хорошо знакомо. А вот как выглядит интерфейс с применением свойств.

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

```
@interface AllWeatherRadial : Tire {
    float rainHandling;
    float snowHandling;
}
```

```
@property float rainHandling;
```

```
@property float snowHandling;
```

```
@end // AllWeatherRadial
```

Попроще, не правда ли? Нет четырех определений методов и добавлены два ключевых слова с предшествующим знаком @. Вспомните, что символ @ сигнализирует о следующем: "Вы встретились с чем-то, специфичным для Objective-C". Директива @property представляет собой новую возможность компилятора, гласящую об объявлении нового атрибута объекта.

Строка @property float rainHandling; говорит о том, что объекты класса AllWeatherRadial имеют атрибут типа float с именем rainHandling. Она также говорит о том, что вы можете установить свойство при помощи вызова -setRainHandling; и получить его значение при помощи вызова -rainHandling. Вы можете заново собрать и запустить программу, и она будет вести себя так же, как и ранее. Все строки @property автоматически объявляют устанавливающий и получающий методы доступа к атрибуту. Атрибут не обязан соответствовать имени переменной экземпляра, но в большинстве случаев это именно так. Мы поговорим об этом (и о некоторых других особенностях свойств) немного позже.

Упрощение реализации

Обратимся к реализации класса AllWeatherRadial.

```
#import "AllWeatherRadial.h"
@implementation AllWeatherRadial

- (id) initWithPressure: (float) p
    treadDepth: (float) td
{
    if (self = [super initWithPressure: p
        treadDepth: td]) {
        rainHandling = 23.7;
        snowHandling = 42.5;
    }
    return (self);
} // initWithPressure:treadDepth

- (void) setRainHandling: (float) rh
{
    rainHandling = rh;
} // setRainHandling

- (float) rainHandling
{
    return (rainHandling);
} // rainHandling

- (void) setSnowHandling: (float) sh
{
    snowHandling = sh;
} // setSnowHandling

- (float) snowHandling
{
    return (snowHandling);
} // snowHandling

- (NSString *) description
{
    NSString *desc;
    desc = [[NSString alloc] initWithFormat:
        @"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",
        [self pressure], [self treadDepth],
        [self rainHandling],
        [self snowHandling]];
    return (desc);
} // description
@end // AllWeatherRadial
```


В предыдущей главе мы рассматривали метод `init`, стандартный инициализатор, все устанавливающие и получающие методы доступа и описание. Теперь мы безжалостно уберем все методы доступа, заменив их двумя строками исходного текста.

```
#import "AllWeatherRadial.h"

@implementation AllWeatherRadial

@synthesize rainHandling;
@synthesize snowHandling;

- (id) initWithPressure: (float) p
    treadDepth: (float) td
{
    if (self = [super initWithPressure: p
        treadDepth: td]) {
        rainHandling = 23.7;
        snowHandling = 42.5;
    }
    return (self);
} // initWithPressure:treadDepth

- (NSString *) description
{
    NSString *desc;
    desc = [[NSString alloc] initWithFormat:
        @"AllWeatherRadial: %.1f / %.1f / %.1f / %.1f",
        [self pressure], [self treadDepth],
        [self rainHandling],
        [self snowHandling]];
    return (desc);
} // description
@end // AllWeatherRadial
```

Директива `@synthesize` представляет собой новую возможность компилятора, которая гласит: "Создай методы доступа для этого атрибута". Так, вместо строки `@synthesize rainHandling;` компилятор вставляет код методов `-setRainHandling:` и `-rainHandling`.

ПРИМЕЧАНИЕ

Возможно, вы уже знакомы с *генерацией кода*: утилиты создания методов доступа *Cocoa* и построители пользовательского интерфейса для других платформ создают исходный текст, который затем компилируется. Но `@synthesize` не является генерацией кода в том смысле, что вы не увидите явно написанный код реализации методов `-setRainHandling:` и `-rainHandling`, но эти методы будут существовать и могут быть вызваны. Это дает Apple возможность изменять генерируемые Objective-C методы доступа, возможно, обеспечивая большую безопасность или лучшую производительность.

Если вы запустите программу, то увидите те же результаты, что и до внесения изменений.

...и точка!

Свойства Objective-C 2.0 позволяют упростить синтаксис обращения к атрибутам объекта, более привычный для тех, кто работал с языками программирования наподобие C++ или Java.

Вспомним две строки, добавленные к функции `main()` для изменения атрибутов колеса.

```
[tire setRainHandling: 20 + i];
[tire setSnowHandling: 28 + i];
```

Этот код мы можем заменить следующим.

```
tire.rainHandling = 20 + i;
tire.snowHandling = 28 + i;
```

Если вы запустите программу, то увидите те же результаты, что и до внесения изменений. Точно так же можно внести изменения и в код вызова функции `NSLog()`, выводящего информацию о колесе.

```
NSLog(@"Свойства колеса: %.f %.f",
      [tire rainHandling],
      [tire snowHandling]);
```

заменяется на

```
NSLog(@"Свойства колеса: %.f %.f",
      tire.rainHandling,
      tire.snowHandling);
```

Такая "запись с точкой" выглядит очень похоже на обращение к структуре в C или объекту в Java. Когда вы видите точку слева от знака равенства, вызывается устанавливающий метод доступа для данного атрибута (`-setRainHandling:` и `-setSnowHandling:`). В противном случае вызывается получающий метод доступа для данного атрибута (`-rainHandling` и `-snowHandling`).

ПРИМЕЧАНИЕ

Запись с точкой представляет собой всего лишь сокращенную запись вызова метода доступа. Никаких дополнительных закулисных хитростей не происходит. В главе 15, "Работа с файлами", поговорим о кодировании "ключ/значение", когда действительно используется определенное шаманство. Но в данном случае ничего подобного нет.

Если вы используете свойства и при этом получаете странные сообщения об ошибках доступа к чему-то, отличающемуся от `struct`, убедитесь, что вы включили все необходимые заголовочные файлы используемых вами классов.

Возражения против свойств

До сих пор мы имели дело со свойствами скалярных типов — в частности, `float`, но та же методика применима и для `int`, `char`, `BOOL` и `struct`. Например, при желании у нас может быть свойство `NSRect`.

Объекты привносят некоторые сложности. Вспомните, что мы должны захватывать и освобождать объекты, проходящие через наши методы доступа. В случае некоторых объектов, в частности, строк, мы всегда копируем их. В случае других объектов, наподобие делегатов (о которых мы поговорим в следующей главе), захват вообще не должен осуществляться.

ПРИМЕЧАНИЕ

Стоп, стоп, стоп! Что там насчет копирования и незахвата?

Вы хотите сделать копию строкового аргумента. Распространенная ошибка заключается в получении строки от пользовательского интерфейса (типа текстового поля ввода) и использовании ее в качестве имени чего-то. Строка, которую вы получаете от текстового поля, — обычная изменяемая строка, она будет изменяться при новом вводе пользователя. Создание копии строки предохраняет значение от неожиданного изменения.

Теперь — о захвате объектов. Имеется специальный случай — *цикл захвата* — когда счетчики ссылок не работают. Если у вас есть отношение “владелец/собственность”, как между Car и Engine, вы хотите, чтобы автомобиль владел двигателем, но не наоборот. Двигатель не должен владеть автомобилем, в который он вмонтирован. Если автомобиль захватывает двигатель, а двигатель — автомобиль, то ни один счетчик ссылок не обнуляется и ни один объект не будет уничтожен. Метод dealloc класса Car не будет вызван до тех пор, пока двигатель не освободит автомобиль в своем методе dealloc, а метод dealloc двигателя не будет вызван до тех пор, пока метод dealloc автомобиля не освободит двигатель. Они так и будут сидеть друг напротив друга, широко раскрыв глаза и ожидая, кто же моргнет первым. Общее правило — владелец захватывает собственность, и никогда наоборот.

Счастливым пользователям сборщика мусора не надо беспокоиться о такой ситуации.¹

Продолжим наши игры и добавим новую возможность в класс Car. Давайте дадим автомобилю имя! Начнем как отличники старой школы и используем традиционные методы доступа. Сначала внесем изменения в файл Car.h (они выделены полужирным шрифтом).

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

```
@class Tire;
```

```
@class Engine;
```

```
@interface Car : NSObject {
```

```
    NSString *name;
```

```
    NSMutableArray *tires;
```

```
    Engine *engine;
```

```
}
```

```
- (void)setName: (NSString *) newName;
```

```
- (NSString *) name;
```

¹ У них есть свои заботы — например, два объекта, причем указатель в переменной экземпляра одного объекта указывает на второй, а указатель в переменной экземпляра второго — на первый (цепочка может оказаться и более длинной). Других указателей на эти объекты нет, и они могли бы быть уничтожены сборщиком мусора, но наличие цепочки указателей не позволяет это сделать. — *Примеч. ред.*

```

- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;

- (void) setTire: (Tire *) tire
  atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;

- (void) print;
@end // Car

```

Теперь добавим реализацию методов доступа (обратите внимание, что мы копируем имя) вместе с выбором имени по умолчанию и выводом его в описании.

```

#import "Car.h"

@implementation Car
- (id) init
{
    if (self = [super init]) {
        name = @"Car";
        tires = [[NSMutableArray alloc] init];
        int i;
        for (i = 0; i < 4; i++) {
            [tires addObject: [NSNull null]];
        }
    }
    return (self);
} // init

- (void) dealloc
{
    [name release];
    [tires release];
    [engine release];

    [super dealloc];
} // dealloc

- (void) setName: (NSString *) newName {
    [name release];
    name = [newName copy];
} // setName

- (NSString *) name {
    return (name);
} // name

- (Engine *) engine
{

```

```

    return (engine);
} // engine

- (void) setEngine: (Engine *) newEngine
{
    [newEngine retain];
    [engine release];
    engine = newEngine;
} // setEngine

- (void) setTire: (Tire *) tire
    atIndex: (int) index
{
    [tires replaceObjectAtIndex: index
        withObject: tire];
} // setTire:atIndex:

- (Tire *) tireAtIndex: (int) index
{
    Tire *tire;
    tire = [tires objectAtIndex: index];
    return (tire);
} // tireAtIndex:

- (void) print
{
    NSLog(@"%@ имеет:", name);
    int i;
    for (i = 0; i < 4; i++) {
        NSLog(@"%@", [self tireAtIndex: i]);
    }
    NSLog(@"%@ ", engine);
} // print
@end // Car

```

Установим имя автомобиля в функции main().

```

Car *car = [[Car alloc] init];
[car setName: @"Herbie"];

```

Запустите программу, и в первой строке вывода вы увидите имя автомобиля. Ну, а теперь добавим в класс Car свойства. Вот как теперь выглядит Car.h во всей своей красе.

```

#import <Cocoa/Cocoa.h>

@class Tire;
@class Engine;

@interface Car : NSObject {
    NSString *name;

```

```

NSMutableArray *tires;
Engine *engine;
}

@property (copy) NSString *name;
@property (retain) Engine *engine;

- (void) setTire: (Tire *) tire
  atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;

- (void) print;
@end // Car

```

Обратите внимание, что объявления простых методов доступа заменены на объявления `@property`. Вы можете приукрасить `@property` дополнительными атрибутами, которые выражают ваши точные намерения по поводу поведения свойства. Добавив `copy` к `name`, вы говорите компилятору и пользователям класса, что значение `name` должно копироваться. Это может упростить жизнь программиста, использующего данный класс, поскольку он будет знать, что ему не надо делать копию строки, полученной из текстового поля ввода. С другой стороны, `engine` управляется при помощи захвата и освобождения. Если вы не укажете ни то, ни другое, компилятор по умолчанию использует `assign`, что, как правило, не то, что вы хотите добиться от объектов.

ПРИМЕЧАНИЕ

Можно использовать и другие атрибуты, такие как `nonatomic`, которые делают методы доступа более быстрыми, если они не будут использоваться в многопоточной среде. Настольные компьютеры слишком быстрые, чтобы маркировка свойства как `nonatomic` давала заметные преимущества, но разработчики для iPhone часто используют это ключевое слово для выжимания всего, что можно, из этого устройства с очень ограниченными ресурсами. Можно также использовать ключевое слово `assign`, если вы не хотите, чтобы объект был захвачен (во избежание возникновения циклов захвата).

В файле `Car.m` два основных изменения: удалены методы доступа для `name` и `engine` и добавлены две директивы `@synthesize`.

```
@implementation Car
```

```

@synthesize name;
@synthesize engine;

```

И последнее — в функции `main()` используется запись с точкой.

```

Car *car = [[Car alloc] init];
car.name = @"Herbie";
...
car.engine = [[Slant6 alloc] init];

```


Разноименность

Во всех исходных текстах данной главы имя свойства совпадало с именем переменной экземпляра, которая хранит это свойство. Это распространенное явление, но иногда вам может понадобиться, чтобы имя переменной экземпляра и общедоступное имя атрибута были разными.

Допустим, мы хотим назвать переменную экземпляра с именем автомобиля иначе — например, `appellation`. Мы соответствующим образом изменим имя переменной экземпляра в файле `Car.h`.

```
@interface Car : NSObject {
    NSString *appellation;
    NSMutableArray *tires;
    Engine *engine;
}
```

```
@property (copy) NSString *name;
@property (retain) Engine *engine;
```

Затем надо изменить директиву `synthesize`.

```
@synthesize name = appellation;
```

Компилятор, как и ранее, создаст методы `-setName:` и `-name`, но при их реализации использует переменную экземпляра `appellation`.

Однако при компиляции вы увидите пару сообщений об ошибках. Вы можете вспомнить, что мы непосредственно обращались к переменной экземпляра `name`, которая была изменена. Можно решить проблему при помощи поиска и замены, а можно заменить непосредственное обращение к переменной на применение метода доступа. Замените в `init`

```
name = @"Car";
```

на

```
self.name = @"Car";
```

В чем идея использования записи с точкой? Когда используется просто имя `name`, компилятор считает, что мы непосредственно обращаемся к переменной экземпляра. Чтобы воспользоваться методом доступа, можно записать `[self setName:@"Car"]`. Вспомните, что точка представляет собой просто сокращенную запись для такого вызова, так что `self.name = @"Car"` — это всего лишь другой способ сказать то же самое.

В `dealloc` мы используем такой же трюк.

```
self.name = nil;
```

В этой строке вызывается `setName:` с аргументом `nil`. Сгенерированный метод автоматически освобождает предыдущее имя и заменяет его на `nil`. Этот метод завершает работу по освобождению памяти. Конечно, можно просто освободить `name`, чтобы вернуть память системе. При освобождении свойства вне `dealloc` применение присваивания `nil` предотвращает появление "висячей" ссылки на память, которая может быть освобождена.

Наконец, метод `-description` требует исправления первого вызова `NSLog`.

```
NSLog(@"%@ имеет:", self.name);
```

Теперь мы можем переименовать `appellation` на что-то еще — например, `nickname` или `moniker`. Все, что нам надо будет сделать — это изменить имя переменной экземпляра и имя, используемое в `@synthesize`.

Защита от записи

Можно создать объект с атрибутом, доступным только для чтения. Этот атрибут может быть вычисляемым на лету значением, типа площади поверхности банана, или может быть значением, которое не должно изменяться другими объектами, — например номера водительских прав. Такие ситуации обрабатываются дополнительным атрибутом в директиве `@property`.

По умолчанию свойства изменяемы: вы можете и читать их, и записывать. Свойства имеют атрибут `readwrite`, который можно и не использовать, так как это значение по умолчанию; но если вы хотите ясно выразить свои намерения — вполне можете им воспользоваться. Вот как можно использовать этот атрибут в файле `Car.h`.

```
@property (readwrite, copy) NSString *name;
@property (readwrite, retain) Engine *engine;
```

Но мы так не делаем, поскольку стараемся избегать избыточности везде, где только это возможно.

Возвращаясь к нашему обсуждению свойства “только для чтения”, предположим, что у нас есть такое свойство, как номер прав или размер обуви, которое мы не хотим отдавать на всеобщее изменение. Для этого можно воспользоваться атрибутом `readonly` в директиве `@property`. Вот пример такого применения в классе.

```
@interface Me : NSObject {
    float shoeSize;
    NSString *licenseNumber;
}
```

```
@property (readonly) float shoeSize;
@property (readonly) NSString *licenseNumber;
@end
```

Когда компилятор видит, что `@property` имеет атрибут `readonly`, он генерирует только получающий метод доступа, но не устанавливающий. Пользователи `Me` могут вызывать `-shoeSize` и `-licenseNumber`, но если вы попытаетесь вызвать `-setShoeSize`, то компилятор начнет жаловаться. То же самое поведение вы получите и при использовании записи с точкой.

Увы, свойства — не панацея

Обратите внимание, что мы не конвертировали методы `Car` для колес в свойства.

```
- (void) setTire: (Tire *) tire
    atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
```

Это связано с тем, что эти методы не укладываются в достаточно широкий диапазон, охватываемый свойствами. Свойства позволяют замещать методы вида `-setBlah` и `-blah`, но не методы, получающие дополнительные аргументы, наподобие положения колеса в автомобиле.

Резюме

В этой главе мы рассматривали свойства, позволяющие уменьшить количество кода, который надо написать (а потом и прочитать) для реализации обычных операций с атрибутами объектов. Директива `@property` используется для того, чтобы сообщить всему миру: "У данного объекта имеется атрибут данного типа с таким-то именем". Эта директива, кроме того, передает информацию о свойстве, такую как можно ли его изменять (`readonly` или `readwrite`) и как выполняется управление памятью (`retain`, `assign` или `copy`). Компилятор автоматически генерирует объявления методов доступа для указанного атрибута объекта.

Директива `@synthesize` указывает компилятору на необходимость генерации реализации методов доступа. Вы можете при этом решать, с какой переменной экземпляра должна работать генерируемая реализация. Если вы не хотите использовать поведение по умолчанию, то можете сами написать собственный код для методов доступа.

Запись с точкой, хотя и представлена в контексте свойств, представляет собой всего лишь сокращенную запись для вызова устанавливающего и получающего методов доступа объектов. Например, инструкция `dealie.blah = greeble` в точности эквивалентна вызову `[dealie setBlah: greeble]`, а `shronk = dealie.greeble` — вызову `shronk = [dealie greeble]`. Запись с точкой снижает количество вводимых символов и несколько удобнее для программистов, которые перешли из некоторых других языков программирования.

Следующим номером нашей программы будут категории — способ расширения существующих классов даже при отсутствии доступа к их коду! Не пропустите эту захватывающую тему.

Категории

П

ри написании объектно-ориентированных программ часто требуется добавить некоторое поведение в существующий класс. Например, вы можете разработать новый вид колеса, для чего вам придется создавать подкласс класса `Tire` и добавлять в него новую функциональность. В основном, при желании добавить поведение в существующий класс вы создаете подкласс.

Но иногда создание подкласса неудобно. Например, вы можете захотеть добавить некоторое новое поведение в класс `NSString`, но, как вы помните, `NSString` на самом деле представляет собой фасад кластера классов, что затрудняет создание подкласса. В других случаях вы можете оказаться способны создать подкласс, но при этом используете некий инструментарий или библиотеку, которые не способны работать с объектами нового класса. Например, ваш новый подкласс класса `NSString` не может возвращаться при создании новой строки с использованием метода класса `stringWithFormat:`.

Динамический механизм диспетчеризации времени выполнения Objective-C позволяет добавлять методы в существующие классы. Это, как минимум, звучит очень круто! Objective-C использует для этих новых методов термин "категории".

Создание категории

Категория представляет собой способ добавления новых методов к существующим классам. Хотите добавить новый метод в класс? Нет проблем! Это можно сделать с любым классом, даже с тем, для которого нет исходного текста.

Пусть, например, вы пишете головоломку-крестворд, в которой программа получает ряд строк, определяет длину каждой строки и помещает эти длины в массив `NSArray` или `NSDictionary`. Перед добавлением каждой длины в `NSArray` или `NSDictionary` ее следует "завернуть" в объект `NSNumber`.

Вы можете написать такой код.

```
NSNumber *number;  
number = [NSNumber numberWithInt: [string length]];  
// ... Работаем с number
```

Но это неинтересно и утомительно. Вместо этого можно добавить к NSString категорию, которая сделает эту работу за вас. Давайте сделаем это. Код проекта LengthAsNSNumber можно найти в папке 12.01 LengthAsNSNumber.

@interface

Объявление категории внешне напоминает объявление класса.

```
@interface NSString (NumberConvenience)
```

```
- (NSNumber *) lengthAsNumber;
```

```
@end // NumberConvenience
```

Вы должны заметить в этом объявлении пару интересных моментов. Здесь упоминается существующий класс, за именем которого следует новое имя в круглых скобках. Это означает, что категория называется NumberConvenience, и она добавляет методы к классу NSString. Это же можно сказать следующими словами: "мы добавляем категорию NumberConvenience к классу NSString". Вы можете добавить к классу столько категорий, сколько хотите, лишь бы их имена были уникальными.

Вы указываете класс, в который добавляются категории (NSString), и имя категории (NumberConvenience) и перечисляете добавляемые методы, после чего идет директива @end. Вы не можете добавлять новые переменные экземпляра, поэтому раздела с переменными экземпляра, имеющегося в объявлении класса, в объявлении категории нет.

@implementation

В разделе @implementation нет таких сюрпризов, как в разделе @interface. Вы помещаете добавляемые методы в раздел @implementation.

```
@implementation NSString (NumberConvenience)
```

```
- (NSNumber *) lengthAsNumber
```

```
{
    unsigned int length = [self length];
    return ([NSNumber numberWithInt: length]);
} // lengthAsNumber
```

```
@end // NumberConvenience
```

Так же, как и в случае раздела @interface, в разделе @implementation указываются имена класса и категории, а также помещаются тела новых методов.

Метод lengthAsNumber получает длину строки при помощи вызова [self length], а затем создается новый объект NSNumber с этой длиной.

Давайте сделаем небольшой перерыв для одной из наших излюбленных тем — управления памятью. Корректен ли этот код? Да! Метод numberWithInt не является методом alloc, сору или new. Поскольку он не является ни одним из трех перечисленных методов, объект, который он возвращает, считается имеющим счетчик ссылок, равный 1, и при этом

автоосвобожденным. Созданный нами объект `NSNumber` будет освобожден при уничтожении текущего активного пула автоосвобождения.

А вот новая категория в действии. Функция `main()` создает новый объект `NSMutableDictionary` и добавляет в него три строки в качестве ключей и длины этих строк — в качестве значений.

```
int main (int argc, const char *argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    NSMutableDictionary *dict;
    dict = [NSMutableDictionary dictionary];

    [dict setObject: [@"hello" lengthAsNumber]
     forKey: @"hello"];

    [dict setObject: [@"iLikeFish" lengthAsNumber]
     forKey: @"iLikeFish"];

    [dict setObject: [@"Once upon a time" lengthAsNumber]
     forKey: @"Once upon a time"];

    NSLog ("%@", dict);

    [pool release];

    return (0);
} // main
```

Давайте, как обычно, строка за строкой разберем наш код. Сначала мы создаем пул автоосвобождения, о котором, надо понимать, слушать вам уже надоело.

```
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];
```

Просто в качестве напоминания: этот пул — то место, куда попадают все автоосвобожденные объекты. В частности, там окончат свои дни и изменяемый словарь, и объекты `NSNumber`, создаваемые нашей категорией.

После пула создаем новый изменяемый словарь. Напомним, что этот удобный класс `Cocoa` позволяет нам хранить пары ключей и объектов.

```
NSMutableDictionary *dict;
dict = [NSMutableDictionary dictionary];
```

Мы не можем помещать в словарь примитивные типы наподобие `int`, так что приходится использовать классы-оболочки наподобие `NSNumber`. К счастью, наша новая категория позволяет легко встраивать длину строки в `NSNumber`. Вот код, который добавляет значение 5 в словарь с ключом `@"hello"`.

```
[dict setObject: [@"hello" lengthAsNumber]
 forKey: @"hello"];
```


Этот код выглядит странно, но на самом деле он делает правильные вещи. Вспомните, что строки типа @"string" на самом деле представляют собой объекты NSString. Они реагируют на сообщения точно так же, как и любой объект NSString. Благодаря новой категории у NSString, любая строка теперь реагирует на сообщение lengthAsNumber, в том числе и литеральные строки наподобие приведенной.

Еще раз и медленно: *любая* строка NSString отвечает на сообщение lengthAsNumber — включая литералы, строки, возвращаемые методами description, изменяемые строки, строки из файлов, из Интернета и т.д. Именно такая всеядность и делает категории столь ценной идеей. Нам не надо порождать подкласс из NSString для того, чтобы добиться указанного поведения — все работает и без этого!

При запуске программы вы получите на экране что-то наподобие следующего.

```
{
    "Once upon a time" = 16;
    hello = 5;
    iLikeFish = 9;
}
```

Плохие категории

Теперь, когда вы воспарили с категориями к небесам, опустим вас на грешную землю. У категорий имеется два ограничения. Первое заключается в невозможности добавления к классу переменных экземпляра.

Второе ограничение связано с коллизиями имен, когда один из методов ваших категорий имеет то же имя, что и существующий метод. При таком столкновении побеждает категория. Ваш метод категории полностью заменяет исходный метод, и нет никакого способа достучаться до оригинала. Некоторые программисты добавляют к именам методов категорий специальные префиксы, чтобы избежать конфликтов имен.

ПРИМЕЧАНИЕ

Существуют методы обойти ограничение, заключающееся в невозможности добавлять переменные экземпляра. Например, можно использовать глобальный словарь для хранения отображения между объектами и дополнительными переменными, которые вы хотите с ними связать. Но в этом случае следует задуматься, а действительно ли категории — наилучшее средство для решения стоящей перед вами задачи?

Хорошие категории

В Cocoa категории используются для трех основных целей: разделение реализации класса между несколькими файлами или схемами, создание опережающих ссылок для закрытых методов и добавление неформальных протоколов к объекту. Не беспокойтесь, если слова "неформальные протоколы" не вызывают у вас ничего, кроме недоумения. Мы поговорим о них немного позже.

Разделение реализации по категориям

Как вы видели в главе 6, "Организация исходных файлов", интерфейс класса можно разместить в заголовочном файле, а реализацию — в .m-файле. Но вы не можете разбить @implementation на несколько .m-файлов. Если у вас имеется один большой класс, который вы хотите разбить на несколько .m-файлов, можете воспользоваться категориями.

Возьмем, например, класс `NSWindow` из `AppKit`. Если вы посмотрите на документацию `NSWindow`, то найдете в ней сотни методов (если распечатать документацию класса `NSWindow`, получится более 60 страниц).

Размещение всего кода `NSWindow` в одном файле делает его слишком большим и неудобным даже для команды разработчиков `Cocoa`, не говоря уж об обычных пользователях. Если вы рассмотрите заголовочный файл `/System/Library/Frameworks/AppKit.framework/Headers/NSWindow.h` и поищите в нем подстроку `"@interface"`, то найдете официальный интерфейс класса.

```
@interface NSWindow : NSResponder
```

За ним следует множество категорий, включая следующие:

```
@interface NSWindow(NSKeyboardUI)
@interface NSWindow(NSToolbarSupport)
@interface NSWindow(NSDrag)
@interface NSWindow(NSCarbonExtensions)
@interface NSObject(NSWindowDelegate)
```

Такое применение категорий позволяет разместить в одном файле все, связанное с клавиатурой, во втором — код панели инструментов, в третьем — код, обеспечивающий возможность перетаскивания объектов, и т.д. Эти категории разделяют методы на логические группы, что делает заголовочный файл более удобочитаемым. Это то, что мы попытаемся сделать самостоятельно, но в очень скромных масштабах.

Использование категорий в нашем проекте

Проект `CategoryThing`, расположенный в каталоге 12.02 `CategoryThing`, содержит простой класс, распределенный по нескольким файлам реализации.

Первый файл — `CategoryThing.h`, в котором находится объявление класса и некоторые категории. Этот файл начинается с директивы импорта схемы `Foundation` и объявления класса с тремя целыми переменными экземпляра.

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

```
@interface CategoryThing : NSObject {
    int thing1;
    int thing2;
    int thing3;
}
@end // CategoryThing
```

После объявления класса идут три категории, каждая из которых содержит методы доступа к одной из переменных экземпляра. Их реализации будут размещены в отдельных файлах.

```
@interface CategoryThing (Thing1)
```

```
- (void) setThing1: (int) thing1;
- (int) thing1;
```

```
@end // CategoryThing (Thing1)
```

```
@interface CategoryThing (Thing2)
```

```
- (void) setThing2: (int) thing2;
- (int) thing2;
```

```
@end // CategoryThing (Thing2)
```

```
@interface CategoryThing (Thing3)
```

```
- (void) setThing3: (int) thing3;
- (int) thing3;
```

```
@end // CategoryThing (Thing3)
```

На этом файл CategoryThing.h заканчивается.

Файл CategoryThing.m весьма прост и содержит метод description, который мы можем использовать со спецификатором формата %@ в функции NSLog().

```
#import "CategoryThing.h"
```

```
@implementation CategoryThing
```

```
- (NSString *) description
```

```
{
    NSString *desc;
    desc = [NSString stringWithFormat: @"%d %d %d",
        thing1, thing2, thing3];
    return (desc);
} // description
@end // CategoryThing
```

Как всегда, наступает время проверки, все ли корректно с точки зрения управления памятью. Правильно ли работает метод description? Да, правильно. Поскольку stringWithFormat не является ни alloc, ни copy, и ни new, можно считать, что возвращаемый этим методом объект имеет счетчик ссылок, равный 1, и что он автоосвобожден, так что он будет уничтожен при удалении текущего пула автоосвобождения.

Теперь можно переходить к категориям. В файле Thing1.m содержится реализация категории Thing1.

```
#import "CategoryThing.h"

@implementation CategoryThing (Thing1)

- (void) setThing1: (int) t1
{
    thing1 = t1;
} // setThing1

- (int) thing1
{
    return (thing1);
} // thing1
@end // CategoryThing
```

Здесь следует отметить, что категория свободно обращается к переменным экземпляра класса. Методы категорий обладают всеми правами "граждан" класса, как и методы самого класса.

Содержимое файла Thing2.m очень похоже на содержимое файла Thing1.m.

```
#import "CategoryThing.h"

@implementation CategoryThing (Thing2)

- (void) setThing2: (int) t2
{
    thing2 = t2;
} // setThing2

- (int) thing2
{
    return (thing2);
} // thing2
@end // CategoryThing
```

Пожалуй, после прочтения приведенного исходного текста вы без труда сможете воссоздать содержимое файла Thing3.m при помощи операций копирования, вставки и поиска с заменой.

```
#import "CategoryThing.h"

@implementation CategoryThing (Thing3)

- (void) setThing3: (int) t3
{
    thing3 = t3;
} // setThing3

- (int) thing3
{

```

```

    return (thing3);
} // thing3
@end // CategoryThing

```

Файл `main.m` содержит функцию `main()`, которая использует только что созданные категории. Начнем с конструкций `#import`.

```

#import <Foundation/Foundation.h>
#import "CategoryThing.h"

```

Нам надо импортировать заголовочный файл класса `CategoryThing`, чтобы компилятор мог увидеть определение класса и категории. После этого идет тело самой функции `main()`.

```

int main (int argc, const char *argv[])
{
    NSAutoreleasePool * pool;
    pool = [[NSAutoreleasePool alloc] init];

    CategoryThing *thing;
    thing = [[CategoryThing alloc] init];

    [thing setThing1: 5];
    [thing setThing2: 23];
    [thing setThing3: 42];

    NSLog (@"Things are %@", thing);

    [thing release];

    [pool release];

    return (0);
} // main

```

Первые две строки функции `main()` представляют собой стандартный, набивший оскомину код создания пула автоосвобождения. При уничтожении этого пула он потянет за собой и автоосвобожденную строку описания, использованную функцией `NSLog()`.

Далее выделяется и инициализируется объект `CategoryThing`.

Наш обязательный отчет в полицию управления памятью: объект выделяется при помощи `alloc`, так что его счетчик ссылок равен 1, и он не находится в пуле автоосвобождения, так что мы отвечаем за его освобождение по окончании работы.

Далее объекту посылаются сообщения, устанавливающие значения переменных экземпляра `thing1`, `thing2` и `thing3`.

```

[thing setThing1: 5];
[thing setThing2: 23];
[thing setThing3: 42];

```

При использовании объекта не имеет значения, объявлен ли метод в интерфейсе, надклассе или категории.

После установки значений переменных экземпляра объекта `thing` функция `NSLog()` выводит информацию об объекте. Как вы видели в методе `description` класса `CategoryThing`, эта информация содержит значения трех переменных экземпляра.

```
NSLog(@"Things are %@", thing);
```

Поскольку при создании объекта `thing` использовался метод `alloc`, мы отвечаем за освобождение этого объекта по окончании работы, что и происходит прямо сейчас.

```
[thing release];
```

И наконец, освобождается пул автоосвобождения, после чего функция `main()` возвращает нулевое значение.

```
[pool release];  
return (0);  
} // main
```

На этом наша маленькая программа завершается. При ее запуске мы получаем следующий результат.

```
Things are 5 23 42
```

Можно не только разбить реализацию класса на несколько исходных файлов, но и разделить ее между несколькими схемами. Класс `NSString` располагается в схеме `Foundation`, где нашли пристанище множество ориентированных на данные классов, таких как строки, числа и коллекции. Все графические элементы (окна, цвета, рисунки и т.п.) располагаются в `AppKit`. Несмотря на то что класс `NSString` объявлен в `Foundation`, в `AppKit` имеется категория `NSString` по имени `NSStringDrawing`, которая позволяет отправлять сообщения о выводе строковым объектам. Этот метод обеспечивает вывод текстовых строк на экран. Поскольку это сугубо графический вывод, он относится к `AppKit`. Но `NSString` — объекты `Foundation`. Разработчики `Cocoa` использовали категории для размещения функциональности по работе с данными в схеме `Foundation`, а функциональности графического вывода — в `AppKit`. Мы, как программисты, имеем дело просто с `NSString`, и нас, вообще говоря, не волнует, где именно находится тот или иной метод.

Опережающие ссылки и категории

Как мы упоминали ранее, в `Cocoa` нет никаких истинно закрытых методов. Если вы знаете имя метода, поддерживаемого объектом, можете его вызвать, даже если в разделе `@interface` класса нет его объявления.

Компилятор при этом пытается быть честным. Увидев вызов метода объекта, для которого он еще не видел объявления или определения, он жалуется примерно так: `Warning: 'CategoryThing' may not respond to '-setThing4:'`. Вообще говоря, такие предупреждения — это очень хорошо, поскольку они могут помочь вам выловить массу опечаток.

Однако бдительность компилятора способна породить проблемы, если у вас есть методы, используемые в вашей реализации, но не перечисленные в разделе `@interface` вашего класса. Имеется масса уважительных причин, по которым вы можете не захотеть перечис-

лять здесь ваши методы. Методы могут быть просто деталями реализации или, например, вы просто окончательно не определились с используемыми именами. Но если вы не объявляете ваши методы до их использования, то получите предупреждения от компилятора. Заставить компилятор замолчать — идея хорошая, но как это сделать?

Если можете определить метод до его использования, компилятор увидит это определение и не будет выводить предупреждение. Но если это неудобно сделать или если вы используете непубликуемые методы в другом классе, то вам надо какое-то другое решение.

Спасительные категории

Объявления метода в категории достаточно для того, чтобы компилятор удовлетворенно сказал: "Отлично, такой метод существует. Я не буду жаловаться, если программист использует его". Вы не обязаны реализовывать его, если не хотите этого делать.

Наша методика зачастую размещает категории в начале файла реализации. Скажем, класс Car имеет метод rotateTires. Мы можем реализовать rotateTires посредством другого метода с именем moveTireFromPosition:toPosition:, который обменивает колеса в двух позициях. Этот второй метод представляет собой детали реализации, и мы не намерены помещать его в открытый интерфейс класса Car. При объявлении этого метода в категории метод rotateTires может применять его, не рискуя получить какие-либо предупреждения от компилятора. Такая категория может иметь вид наподобие следующего.

```
@interface Car (PrivateMethods)
```

```
- (void) moveTireFromPosition: (int) pos1
```

```
    toPosition: (int) pos2;
```

```
@end // Private Methods
```

При реализации этого метода он не обязан находиться в блоке @implementation Car (PrivateMethods). Вы можете оставить его в разделе @implementation Car. Это позволит разделить ваши методы на категории для удобства организации и документирования, в то время как все методы будут продолжать находиться в одной большой куче в файле реализации. При обращении к закрытым методам других классов вам не требуется предоставлять их реализацию. Просто объявите их в категории — и этого будет вполне достаточно для счастья компилятора (кстати, в действительности вы не должны обращаться к закрытым методам других классов, но иногда это приходится делать при написании тестового кода или для обхода ошибок в Cocoa или в коде других программистов).

Неформальные протоколы и категории делегирования

Пришло время познакомиться с более сложными концепциями, чем обычно встречающиеся в объектно-ориентированном программировании, — впрочем, они выглядят сложнее, чем есть на самом деле.

Классы Сосоа часто используют технологии, включающие *делегирование*, когда один объект запрашивает у другого выполнения некоторой его работы. Например, класс `AppKit NSApplication` запрашивает у своего делегата, следует ли создавать окно `Untitled` при запуске приложения; объекты `NSWindow` запрашивают у своих делегатов, следует ли позволять закрывать окно.

Чаще всего приходится писать один объект-делегат и передавать его другому объекту, обычно предоставляемому Сосоа. Путем реализации определенных методов вы можете управлять поведением объектов Сосоа.

Прокручиваемые списки в Сосоа обрабатываются классом `AppKit NSTableView`. Когда `tableView` готов к выполнению некоторой работы, такой как выбор строки, на которой щелкнул пользователь, объект запрашивает у своего делегата, не может ли он выбрать строку. Объект `tableView` посылает своему делегату сообщение:

```
-(BOOL) tableView: (NSTableView *) tableView
shouldSelectRow: (int) row;
```

Метод делегата может рассмотреть `tableView` и строку и решить, следует ли ее выбирать. Если таблица включает строки, которые не должны быть выбраны, делегат может реализовать концепцию отключенных строк, которые нельзя выбирать.

Проект iTunesFinder

Класс Сосоа, который позволяет находить сетевые сервисы, опубликованные Bonjour (ранее эта технология именовалась `Rendezvous`), называется `NSNetServiceBrowser`. Вы говорите браузеру сетевых сервисов, какой именно сервис вы ищете, и передаете ему объект-делегат. Затем объект браузера посылает сообщение объекту-делегату, сообщая о новых сервисах.

Проект `iTunesFinder` из папки 12.03 `iTunesFinder` использует класс `NSNetServiceBrowser` для перечисления всех совместно используемых музыкальных библиотек `iTunes`, которые он может найти.

В этом проекте мы начнем с функции `main()` из файла `main.m`. Объект-делегат представляет собой экземпляр класса `iTunesFinder`, так что нам надо импортировать соответствующий заголовочный файл.

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

Затем начинается сама функция `main()`. Мы создаем пул автоосвобождения.

```
int main (int argc, const char *argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];
```

После этого создается новый объект `NSNetServiceBrowser`:

```
NSNetServiceBrowser *browser;
browser = [[NSNetServiceBrowser alloc] init];
```

и новый объект `iTunesFinder`.

```
ITunesFinder *finder;
finder = [[ITunesFinder alloc] init];
```

Так как для создания мы используем метод alloc, должны принять ответственность за освобождение этих объектов по окончании работы с ними.

Теперь предложим браузеру сетевых сервисов использовать в качестве делегата объект ITunesFinder.

```
[browser setDelegate: finder];
```

Затем попросим браузер найти совместно используемые библиотеки iTunes.

```
[browser searchForServicesOfType: @"_daap._tcp"
inDomain: @"local."];
```

Строка "_daap._tcp" говорит браузеру о том, что следует искать сервисы типа daap ("daap" — сокращенное "Digital Audio Access Protocol") с применением сетевого протокола TCP. Так находятся все библиотеки, опубликованные iTunes. Домен local. означает, что поиск выполняется в локальной сети. Комитет по цифровым адресам в Интернете (Internet Assigned Numbers Authority, IANA) поддерживает список семейств протоколов Интернета, которые обычно отображаются на сервис Bonjour.

Затем функция main() сообщает о факте начала просмотра и запускает цикл run.

```
NSLog(@"Начинаем поиск");
[[NSRunLoop currentRunLoop] run];
```

Цикл run представляет собой конструкцию Cocoa, которая блокируется (т.е. не выполняет никаких действий), пока не произойдет что-то интересное. В данном случае "интересное" означает, что браузер сетевых сервисов обнаружил новую библиотеку iTunes.

В дополнение к прослушиванию сетевого трафика, run выполняет и другие задачи, например ожидает пользовательские события, такие как нажатие клавиши или щелчок мыши. Метод run в действительности не возвращает управление; он продолжает работать вечно, так что код, следующий за ним, никогда не будет выполнен. Но мы все же оставляем его, чтобы те, кто будут его читать, знали, что мы побеспокоились о корректном управлении памятью (можно создать цикл run, который будет работать только ограниченное количество времени, но такой код сложнее и ничего не дает с точки зрения нашего рассмотрения делегатов). Так что в данной ситуации код очистки памяти в действительности выполняться не будет.

```
[browser release];
[finder release];
[pool release];
return (0);
} // main
```

Теперь у нас есть браузер сетевых сервисов и цикл run. Браузер посылает сетевые пакеты в поисках определенных сервисов, и возвращающиеся назад пакеты докладывают: "Я здесь". Когда пакеты возвращаются, цикл run сообщает браузеру: "Для тебя есть пакеты". Браузер просматривает пакеты, и если это пакеты от неизвестного ранее сервиса, он посылает сообщение объекту-делегату, рассказывая ему о происшедшем.

Теперь самое время познакомиться с кодом нашего делегата, iTunesFinder. Интерфейс класса iTunesFinder минимален.

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

```
@interface iTunesFinder : NSObject
@end // iTunesFinder
```

Вспомните, что мы не обязаны объявлять методы в разделе @interface. Чтобы быть делегатом, достаточно реализовать методы, в вызове которых мы заинтересованы.

Реализация содержит два метода. Сначала, как обычно, идет вступление.

```
#import "iTunesFinder.h"
@implementation iTunesFinder
```

Затем начинается настоящая работа — первый метод делегата.

```
-(void) netServiceBrowser: (NSNetServiceBrowser *) b
    didFindService: (NSNetService *) service
    moreComing: (BOOL) moreComing
{
    [service resolveWithTimeout: 10];
    NSLog(@"Найден сервис %@",
        [service name]);
} // didFindService
```

Когда NSNetServiceBrowser находит новый сервис, он посылает сообщение netServiceBrowser:didFindService:moreComing: делегату. В качестве первого аргумента передается сам браузер (то же значение, которое хранится в переменной browser в функции main()). Если поиск одновременно выполняют несколько браузеров, этот аргумент позволяет определить, какой именно что-то нашел.

Объект NSNetService, который передается в качестве второго аргумента, является объектом, описывающим найденный сервис (такой, как iTunes). Последний аргумент, moreComing, используется для сигнала о завершении пакета уведомлений. Зачем разработчики Cocoa включили этот параметр moreComing? Если вы запустите эту программу в большой сети с сотней библиотек iTunes, этот метод будет вызван 99 раз с аргументом moreComing, имеющим значение YES, а затем один раз со значением NO. Эта информация особенно удобна при создании пользовательского интерфейса, так как дает вам знать, когда можно обновить окно. Пока находятся новые библиотеки iTunes, этот метод будет вызываться вновь и вновь.

Сообщение [service resolveWithTimeout: 10] говорит системе Bonjour о необходимости получить всю информацию о сервисе, в частности, его имя (типа Scott's Groovy Tunes), чтобы мы могли вывести его. Сообщение [service name] дает нам искомое имя.

Библиотеки iTunes могут появляться и исчезать — по мере того как пользователи включают или отключают свои компьютеры. Класс iTunesFinder реализует второй метод делегата, который вызывается, когда сетевой сервис исчезает.

```
-(void) netServiceBrowser: (NSNetServiceBrowser *) b
    didRemoveService: (NSNetService *) service
    moreComing: (BOOL) moreComing
```

```
{
    [service resolveWithTimeout: 10];
    NSLog(@"Сервис %@ недоступен",
          [service name]);
} // didRemoveService
```

Этот метод в точности такой же, как и `didFindService`, за исключением того, что он информирует о моментах, когда сервис становится недоступен.

Теперь запустим программу и посмотрим, что у нас получится. В сети Марка имеется старый G4 iMac с именем `iLamp`, на котором имеется музыка iTunes, доступная в его доме. Это приводит к следующему выводу программы.

Начинаем поиск

Найден сервис `iLamp`

Запустим iTunes на ноутбуке и откроем музыку под именем `mark's music` для общего пользования.

Найден сервис `mark's music`

После отключения iTunes на ноутбуке `ITunesFinder` сообщает:

Сервис `mark's music` недоступен

Делегаты и категории

Хорошо, но какое отношение это делегирование имеет к категориям? Делегаты выдвигают на первый план еще одно применение категорий: методы, которые могут быть вызваны соответствующими сообщениями, посланными делегату, объявлены как категория `NSObject`. Вот часть объявления методов делегата `NSNetService`.

```
@interface NSObject
(NSNetServiceBrowserDelegateMethods)

- (void) netServiceBrowserWillSearch:
(NSNetServiceBrowser *) browser;

- (void) netServiceBrowser:
(NSNetServiceBrowser *) aNetServiceBrowser
didFindService: (NSNetService *) service
moreComing: (BOOL) moreComing;

- (void) netServiceBrowserDidStopSearch:
(NSNetServiceBrowser *) browser;

- (void) netServiceBrowser:
(NSNetServiceBrowser *) browser
didRemoveService: (NSNetService *) service
moreComing: (BOOL) moreComing;
@end
```


Благодаря объявлению этих методов как категории NSObject, реализация NSNetServiceBrowser может посылать одно из этих сообщений *любому* объекту, безотносительно его настоящего класса. Это также означает, что делегатом может быть любой тип объекта, лишь бы он реализовал нужный метод.

ПРИМЕЧАНИЕ

Добавление категории к классу NSObject позволяет использовать в качестве делегата объект любого типа. Нам не надо наследовать специализированный класс serviceBrowserDelegate (как пришлось бы делать в C++) или обеспечивать соответствие определенному интерфейсу (как в Java).

Добавление категории к классу NSObject называется 'созданием неформального протокола'. Как вы знаете, "протокол" в компьютерном языке означает набор правил, управляющих коммуникацией. Неформальный протокол — просто способ сказать "вот несколько методов, которые вы можете захотеть реализовать, что обеспечит возможность получить неплохие результаты". В классе ITunesFinder мы реализовали некоторые из методов, объявленных в неформальном протоколе NSNetServiceBrowserDelegateMethods. Это нормально. При работе с неформальными протоколами вы реализуете только то, что необходимо.

Как можно предположить, существует концепция формального протокола. Ее мы рассмотрим в следующей главе.

Ответы на селекторы

Вы можете задаться вопросом "откуда NSNetServiceBrowser знает, что делегат может обрабатывать посылаемые ему сообщения?" Возможно, вы сталкивались с сообщениями об ошибке времени выполнения Objective-C, возникающей при попытке отправить сообщение, которое объект не понимает.

```
-[ITunesFinder addSnack:]: selector not recognized
```

Так почему NSNetServiceBrowser уверен, что делегат обработает посланное сообщение? Да он и не уверен. NSNetServiceBrowser сначала спрашивает у объекта: "Сможешь ли ты ответить на этот селектор?" Если да, то NSNetServiceBrowser посылает сообщение.

Что же такое селектор? Это просто имя метода, но закодированное специальным образом, используемым Objective-C для быстрого поиска. Вы указываете селектор при помощи директивы компилятора @selector() с именем метода в скобках. Так, селектор метода setEngine: класса Car имеет вид

```
@selector(setEngine:)
```

А вот как выглядит селектор метода setTire:atIndex: класса Car.

```
@selector(setTire:atIndex:)
```

Класс NSObject предоставляет метод respondsToSelector:, который запрашивает объект, может ли он отвечать на указанное сообщение. В приведенном далее фрагменте кода используется этот метод.

```
Car *car = [[Car alloc] init];
if ([car respondsToSelector: @selector(setEngine:)] {
```



```

NSLog(@"Попал!");
}

```

Приведенный код выведет строку "Попал!", так как объект класса Cat в состоянии ответить на сообщение setEngine:.

Рассмотрим еще один фрагмент.

```

ITunesFinder *finder = [[ITunesFinder alloc] init];
if ([finder respondsToSelector:@selector(setEngine:)]) {
    NSLog(@"Попал!");
}

```

В этот раз строки "Попал!" вы не увидите, так как у ITunesFinder нет метода setEngine:.

Возвращаясь назад, NSNetServiceBrowser вызывает respondsToSelector:@selector(netServiceBrowser:didFindService:moreComing:). Если делегат в состоянии ответить на это сообщение, браузер пошлет его делегату. В противном случае браузер игнорирует наличие делегата.

Другие применения селекторов

Селекторы могут использоваться в качестве аргументов методов и даже сохраняться в переменных экземпляра. Это приводит к некоторым очень мощным и гибким конструкциям.

Один из классов AppKit носит имя NSTimer; он может посылать неоднократные сообщения объекту, что очень удобно в играх для перемещения всяческих монстров. При создании нового объекта NSTimer, вы передаете ему объект, которому хотите отправлять сообщения, и селектор, указывающий, какой метод вы хотите вызывать. Например, у вас может быть таймер, вызывающий метод moveMonsterTowardPlayer: вашего игрового движка. Иной таймер может вызывать, например, метод animateOneFrame:.

Резюме

В этой главе вы познакомились с категориями, которые предоставляют способ добавлять новые методы к существующим классам, даже если у вас нет доступа к их исходным текстам.

Кроме добавления функциональности к существующим классам, категории обеспечивают возможность разделения реализации объекта на несколько исходных файлов или даже схем. Так, в качестве примера возвращаясь к методам класса NSString, мы видим, что методы, предназначенные для работы с данными, находятся в схеме Foundation, а методы для вывода на экран — в AppKit.

Категории позволяют объявлять неформальные протоколы. Неформальный протокол представляет собой категорию класса NSObject, перечисляющую методы, при помощи которых объект в состоянии ответить на соответствующие сообщения. Неформальные протоколы используются для реализации делегирования, методики, обеспечивающей легкую настройку поведения объекта. Попутно вы узнали о селекторах, которые являются способом указать определенное сообщение Objective-C в вашем коде.

Далее мы поговорим о формальных протоколах Objective-C — родственниках неформальных протоколов, с которыми вы познакомились в этой главе.

Протоколы

В предыдущей главе мы говорили о магии категорий и неформальных протоколов. При использовании неформального протокола, как видно из главы 12, "Категории", мы реализуем только те методы, на которые хотим отвечать. В случае делегата `NSNetServiceBrowser` из предыдущей главы мы реализовали только два метода, которые вызывались при добавлении или удалении нового сетевого сервиса, и не реализовывали шесть прочих методов из неформального протокола `NSNetServiceBrowserDelegate`. Мы также не обязаны ничего объявлять в нашем объекте, говоря, что он может быть применен в качестве делегата `NSNetServiceBrowser`. Все заработало при минимальных усилиях.

Как можно предположить, Objective-C и Cocoa включают также концепцию формального протокола, и в этой главе мы рассмотрим, как она работает.

Формальные протоколы

Формальный протокол (подобно неформальному) представляет собой именованный список методов. Но, в отличие от неформального протокола, он требует явного принятия протокола. Вы *принимаете* протокол путем указания его имени в объявлении `@interface` класса. Когда делаете это, ваш класс *следует* протоколу. Принятие протокола означает, что вы обещаете реализовать все методы данного протокола. Если вы этого не делаете, компилятор постарается испортить вам настроение своими предупреждениями.

ПРИМЕЧАНИЕ

Формальные протоколы похожи на интерфейсы Java. Фактически протоколы Objective-C и были вызваны к жизни интерфейсами Java.

Почему вы можете захотеть создать или принять формальный протокол? Похоже, реализация всех методов требует большого количества работы? Это зависит от протокола. Бывает, что требуется выполнить большое количество работы, но чаще, пожалуй, протокол содержит только небольшое количество методов, которые надо реализовать, так что обычно формальный протокол представляет собой не такое уж непосильное бремя. В Objective-C 2.0 в язык программирования добавлены некоторые возможности, которые делают использование протоколов еще менее затруднительным (о них поговорим в конце этой главы).

Объявление протоколов

Давайте взглянем на протокол `NSCopying`, объявленный Cocoa. Если вы принимаете `NSCopying`, ваш объект знает, как сделать копию самого себя.

```
@protocol NSCopying
```

```
– (id) copyWithZone: (NSZone *) zone;
@end
```

Синтаксис выглядит примерно так же, как и синтаксис объявления класса или категории. Вместо директивы `@interface`, здесь используется директива `@protocol`, говорящая компилятору: "Хочу показать тебе, как выглядит новый формальный протокол". За директивой следует имя протокола (имена протоколов должны быть уникальны).

Затем идет список объявлений методов, которые должен реализовать каждый объект, принимающий данный протокол. Объявление протокола заканчивается директивой `@end`. В протоколе не могут быть введены никакие переменные экземпляра.

Давайте рассмотрим еще один пример. Вот протокол `NSCoding` из Cocoa.

```
@protocol NSCoding
```

```
– (void) encodeWithCoder: (NSCoder *) aCoder;
– (id) initWithCoder: (NSCoder *) aDecoder;
@end
```

Когда класс принимает протокол `NSCoding`, он обещает реализовать оба эти сообщения. Сообщение `encodeWithCoder:` используется для получения переменных экземпляра объекта и преобразования их в объект `NSCoder`. Сообщение `initWithCoder:` извлекает переменную экземпляра из `NSCoder` и использует ее для инициализации нового объекта. Они всегда реализуются вместе: нет никакого смысла кодировать объект, если вы никогда не будете восстанавливать его в виде нового объекта, и если вы не кодируете объект, то вы не будете и создавать новый.

Принятие протокола

Для принятия протокола следует указать его в объявлении класса в угловых скобках. Например, если класс `Car` принимает `NSCopying`, объявление выглядит следующим образом.

```
@interface Car : NSObject <NSCopying>
{
    // Переменные экземпляра
}
// Методы
@end // Car
```

Если класс Car принимает как NSCopying, так и NSCoding, объявление имеет вид

```
@interface Car : NSObject <NSCopying, NSCoding>
{
    // Переменные экземпляра
}
// Методы экземпляра
@end // Car
```

Вы можете перечислять протоколы в любом порядке.

При принятии протокола вы сообщаете читающим исходный текст программистам, что объекты этого класса могут делать следующее: выполнять кодирование и декодирование и копировать сами себя.

Реализация протокола

Это по сути все, что надо знать о протоколах (не считая мелких деталей синтаксиса, которые мы рассмотрим позже). Теперь мы рассмотрим практический пример по использованию протокола NSCopying в проекте CarParts.

Копирование

Давайте в очередной раз хором повторим наше заклинание: “если объект получен при помощи alloc, сору или new, его счетчик ссылок равен 1 и вы отвечаете за его освобождение”. Мы уже рассмотрели alloc и new, но до сору у нас пока что руки не дошли. Метод сору, конечно же, делает копию объекта. Сообщение сору говорит объекту о том, что надо создать новый объект и сделать его таким же, как и объект-получатель.

Давайте расширим проект CarParts и займемся в нем копированием автомобилей (только тихонько, чтобы нас не услышали в Детройте или в Тольятти). Код проекта находится в папке 13.01 CarParts-Сору. По ходу дела мы коснемся некоторых интересных тонкостей реализации кода создания копии.

КОПИРОВАНИЕ

Создавать копии можно по-разному. Большинство объектов ссылается — т.е. указывает — на другие объекты. При *мелком* копировании вы не копируете объекты, на которые ссылается основной объект; ваша новая копия просто указывает на те же существующие объекты, что и оригинал. Метод сору класса NSArray выполняет мелкое копирование. Когда вы делаете копию NSArray, копируете указатели на объекты, но не сами объекты. Например, если вы копируете NSArray, содержащий пять

строк NSString, у вас не становится десять строк — их так и остается пять. В этом случае каждый объект содержит указатель на каждую строку. Глубокое же копирование создает копии всех указываемых объектов. При глубоком копировании NSArray у вас получается десять строк. В проекте CarParts мы используем глубокое копирование. При этом оказывается возможным изменить значение, на которое ссылается копия (например, давление в колесах), не меняя его в обоих автомобилях одновременно. Глубокое и мелкое копирование могут использоваться одновременно, в зависимости от требований конкретного класса.

Чтобы копировать автомобиль, нам надо уметь копировать двигатель и колеса.

Копирование двигателей

Первый класс, который начнем портить, — Engine. Чтобы иметь возможность создавать копии двигателей, класс должен принять протокол NSCopying. Вот новый интерфейс класса Engine.

```
@interface Engine : NSObject <NSCopying>
@end // Engine
```

В силу принятия протокола NSCopying необходимо реализовать метод copyWithZone:. Зона — это объект NSZone, который представляет собой область памяти, из которой можно выделять память. Когда вы посылаете сообщение copy объекту, оно превращается в copyWithZone: перед достижением вашего кода. В свое время класс NSZone играл более важную роль, чем сейчас, но и теперь мы не можем бросить этот чемодан без ручки...

Вот реализация метода copyWithZone: класса Engine.

```
- (id) copyWithZone: (NSZone *) zone
{
    Engine *engineCopy;
    engineCopy = [[[self class]
                  allocWithZone: zone]
                 init];
    return (engineCopy);
} // copyWithZone
```

У класса Engine нет переменных экземпляра, так что все, что мы должны сделать, — это новый объект двигателя. Однако это не так просто, как кажется. Взгляните на сложную конструкцию справа от engineCopy. Посылаемое сообщение имеет три уровня вложенности!

Первое, что делает данный метод, — получает класс self. Затем он посылает классу сообщение allocWithZone:, чтобы выделить некоторое количество памяти и создать новый объект этого класса. И наконец, новому объекту посылается сообщение init, что должно привести к его инициализации.

Вспомните, что метод alloc является методом класса. Метод allocWithZone: — также метод класса, о чем говорит ведущий плюс в объявлении этого метода.

```
+ (id) allocWithZone: (NSZone *) zone;
```

Нам надо послать это сообщение классу, а не его экземпляру. Какому классу мы должны его послать? Первое инстинктивное побуждение — послать сообщение `allocWithZone:` классу `Engine`.

```
[Engine allocWithZone: zone];
```

Этот способ сработает для класса `Engine`, но не для его подклассов. Почему? Рассмотрим класс `Slant6`, являющийся подклассом `Engine`. Если вы пошлете сообщение `copy` объекту `Slant6`, в конечном итоге код достигнет метода `copyWithZone:` класса `Engine`, так как мы используем логику копирования класса `Engine`. А если вы посылаете сообщение `allocWithZone:` непосредственно классу `Engine`, будет создан новый объект класса `Engine`, а не `Slant6`. Все окажется совсем плохо, если у класса `Slant6` имеются добавленные переменные экземпляра. В этом случае объект `Engine` окажется слишком мал для размещения этих переменных, и копирование приведет к ошибке записи в некорректный участок памяти.

Теперь вы понимаете, почему мы используем `[self class]`? В этом случае сообщение `allocWithZone:` посылается классу объекта-получателя сообщения `copy`. Если `self` является объектом `Slant6`, будет создан новый объект `Slant6`. Если в будущем мы добавим в программу новый вид двигателя (ну, например, `MatterAntiMatterReactor`), то он также будет корректно копироваться.

Последняя строка метода возвращает вновь созданный объект.

Давайте еще раз проверим управление памятью. Операция `copy` должна возвращать объект, счетчик ссылок которого равен единице (а сам объект не автоосвобожден). Мы размещаем новый объект при помощи метода `alloc`, который всегда возвращает объект со счетчиком ссылок, равным 1, и не освобождает его, так что со стороны департамента управления памятью никаких заявлений последовать не должно.

Все это делает класс `Engine` копируемым. Нам не надо трогать класс `Slant6`. Поскольку этот класс не привносит никаких новых переменных экземпляра, то никаких дополнительных действий при копировании не требуется. Класс `Slant6` в состоянии копировать самого себя благодаря наследованию и применению конструкции `[self class]`.

Копирование колес

Копирование колес немного хитрее, чем копирование двигателя. У класса `Tire` есть две переменные экземпляра (`pressure` и `treadDepth`), которые следует скопировать в новое колесо, а подкласс `AllWeatherRadial` вводит две новые дополнительные переменные экземпляра (`rainHandling` и `snowHandling`), которые также следует скопировать в новый объект.

Начнем с `Tire`. Интерфейс использует синтаксис принятия протокола.

```
@interface Tire : NSObject <NSCopying>
{
    float pressure;
    float treadDepth;
}
// ... Методы
@end // Tire
```


Класс теперь реализует метод `copyWithZone:`.

```

- (id) copyWithZone: (NSZone *) zone
{
    Tire *tireCopy;
    tireCopy = [[[self class]
                  allocWithZone: zone]
                 initWithPressure: pressure
                 treadDepth: treadDepth];
    return (tireCopy);
} // copyWithZone

```

Здесь вы видите знакомый уже по классу `Engine` шаблон `[[self class] allocWithZone: zone]`. Поскольку при создании объекта мы должны вызвать `init`, можем воспользоваться методом `initWithPressure:treadDepth:` класса `Tire` для установки значений `pressure` и `treadDepth` нового колеса равными соответствующим значениям копируемого колеса. Это стандартный метод инициализации `Tire`, но вы не обязаны использовать его при копировании. Если хотите, можно использовать обычный метод `init` и методы доступа для изменения атрибутов.

ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ

Вы можете обращаться к переменным экземпляра непосредственно при помощи указателя `C`.

```

tireCopy->pressure = pressure;
tireCopy->treadDepth = treadDepth;

```

Обычно мы используем методы `init` и методы доступа в том редком случае, когда установка атрибута вызывает выполнение дополнительных действий.

Теперь пора взяться за `AllWeatherRadial`. Раздел `@interface` класса `AllWeatherRadial` остается неизменным.

```

@interface AllWeatherRadial : Tire
{
    float rainHandling;
    float snowHandling;
}
// ... Методы
@end // AllWeatherRadial

```

Минутку! А где же `<NSCopying>`? Нам это не надо, и, вероятно, вы понимаете, почему. Когда `AllWeatherRadial` наследует `Tire`, он тащит с собой весь багаж класса `Tire`, включая соответствие протоколу `NSCopying`.

Однако мы должны реализовать метод `copyWithZone:`, так как должны гарантировать копирование переменных экземпляра, вносимых классом `AllWeatherRadial`.

```

- (id) copyWithZone: (NSZone *) zone
{
    AllWeatherRadial *tireCopy;
    tireCopy = [super copyWithZone: zone];
    [tireCopy setRainHandling: rainHandling];
}

```

```
[tireCopy setSnowHandling: snowHandling];
return (tireCopy);
} // copyWithZone
```

Поскольку AllWeatherRadial является подклассом класса, который может быть копирован, нет необходимости возиться со всеми этими [self class], как раньше. Этот класс просто просит надкласс сделать копию и надеется, что надкласс все сделает правильно и использует [self class] при выделении памяти для объекта. Поскольку метод copyWithZone: класса Tire использует [self class] для определения вида создаваемого объекта, он создаст новый объект AllWeatherRadial, который представляет собой именно то, что нам надо. Этот код, кроме того, копирует значения pressure и treadDepth. Разве это не удобно?

Остальная часть работы состоит в копировании значения переменных экземпляра, добавляемых классом AllWeatherRadial. Методы доступа для этого вполне подходят.

Копирование автомобиля

Теперь, когда мы умеем делать копии двигателей и колес и их подклассов, пришло время сделать копируемым и сам автомобиль.

Как вы, вероятно, ожидаете, класс Car должен принять протокол NSCopying.

```
@interface Car : NSObject <NSCopying>
{
    NSMutableArray *tires;
    Engine *engine;
}
// ... Методы
@end // Car
```

Чтобы выполнить свое обещание протоколу NSCopying, класс Car должен реализовать нашего старого знакомого — метод copyWithZone:. Вот какой вид он имеет.

```
-(id) copyWithZone: (NSZone *) zone
{
    Car *carCopy;
    carCopy = [[[self class]
                allocWithZone: zone]
               init];

    carCopy.name = self.name;

    Engine *engineCopy;
    engineCopy = [[engine copy] autorelease];
    carCopy.engine = engineCopy;

    int i;
    for (i = 0; i < 4; i++) {
        Tire *tireCopy;

        tireCopy = [[self tireAtIndex: i] copy];
```

```

[tireCopy autorelease];

[carCopy setTire: tireCopy
 atIndex: i];
}
return (carCopy);
} // copyWithZone

```

Здесь кода немного больше, чем мы писали ранее, но он примерно такой же, как и ранее.

Сначала при помощи сообщения `allocWithZone:` выделяется память для нового объекта `Car`:

```

Car *carCopy;
carCopy = [[[self class]
 allocWithZone: zone]
 init];

```

В нашем проекте у класса `Car` нет подклассов, но в один прекрасный день они могут появиться. Наш автомобиль — не DeLorean, и мы не знаем, что ждет наш класс в будущем.¹ Защита от неизвестного будущего — в выделении нового объекта с использованием класса `self`, как мы делали до сих пор.

Нам надо скопировать имя автомобиля.

```
carCopy.name = self.name;
```

Вспомните, что свойство `name` копирует строку, так что новый автомобиль получит корректное со всех точек зрения имя.

Следом выполняется копирование двигателя, и копия автомобиля получает во владение копию двигателя.

```

Engine *engineCopy;
engineCopy = [[engine copy] autorelease];
carCopy.engine = engineCopy;

```

Вы увидели сообщение `autorelease` и раздумываете, насколько оно необходимо? Давайте подумаем вместе. Конструкция `[engine copy]` возвращает объект со счетчиком ссылок, равным 1. Метод `setEngine:` захватывает двигатель, увеличивая его счетчик ссылок до 2. Когда копия автомобиля (в конечном счете) будет уничтожена, двигатель будет освобожден методом `dealloc` класса `Car`, так что его счетчик ссылок уменьшится до 1. Больше никто освобождать двигатель не будет, так что у нас окажется утечка памяти. Автоосвобождение через некоторое время, когда пул автоосвобождения будет сброшен, уменьшит счетчик ссылок двигателя, что уберет нас от утечки памяти.

А нельзя ли просто освободить объект при помощи `[engineCopy release]` вместо его автоосвобождения? Да, можно — после вызова `setEngine:`; в противном случае копия двигателя будет уничтожена до использования. Какой путь вы выберете — дело вашего вкуса. Некото-

¹ Намек на кинотрилогию Р. Земекиса "Назад в будущее", герои которой перемещались во времени на автомобиле указанной марки. — *Примеч. пер.*

рые программисты предпочитают сами убираться за собой, некоторые — чтобы не забыть освободить объект, предпочитают сразу же по его созданию сообщить о нем уборщику-автоосвобождению и больше о нем не вспоминать. Оба подхода корректны.

После оснащения carCopy новым двигателем цикл for выполняет четыре итерации, копируя и устанавливая в новый автомобиль четыре колеса.

```
int i;
for (i = 0; i < 4; i++) {
    Tire *tireCopy;
    tireCopy = [[self tireAtIndex: i] copy];
    [tireCopy autorelease];
    [carCopy setTire: tireCopy
      atIndex: i];
}
```

Код в цикле использует метод доступа для получения колеса в позиции 0, затем — в позиции 1 и так до окончания цикла. Затем колесо копируется и автоосвобождается, так что проблем с памятью не возникает. После этого копия автомобиля получает сообщение о том, что копию колеса надо установить в ту же позицию, что и оригинал. Благодаря аккуратному написанию методов copyWithZone: классов Tire и AllWeatherRadial, этот код корректно работает с колесами любого типа.

Наконец, вот исходный текст функции main(). В основном код ее такой же, как и в предыдущих главах; вновь добавленный код выделен полужирным шрифтом.

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    Car *car = [[Car alloc] init];
    car.name = @"Herbie";

    int i;
    for (i = 0; i < 4; i++) {
        AllWeatherRadial *tire;
        tire = [[AllWeatherRadial alloc] init];
        [car setTire: tire
          atIndex: i];
        [tire release];
    }

    Slant6 *engine = [[Slant6 alloc] init];
    car.engine = engine;
    [engine release];

    [car print];

    Car *carCopy = [car copy];
    [carCopy print];
}
```

```

[car release];
[carCopy release];

[pool release];
return (0);
} // main

```

После печати информации об автомобиле создается его копия и выводится информация о копии. Таким образом, мы должны получить два одинаковых набора выводимых строк.

Herbie имеет:

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5

Я — двигатель slant-6. P-p-p-p-p!

Herbie имеет:

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5

AllWeatherRadial: 1.5 / 20.0 / 23.7 / 42.5

Я — двигатель slant-6. P-p-p-p-p!

Протоколы и типы данных

В типах данных, используемых для переменных экземпляра и аргументов методов, можно указывать имена протоколов. Этим вы даете компилятору Objective-C дополнительную информацию, которая может помочь ему в проверке вашего кода на наличие ошибок.

Вспомните, что тип `id` представляет собой указатель на объект произвольного типа — это обобщенный тип объекта. Вы можете присвоить переменной `id` указатель на любой объект. Если после `id` указано имя протокола в угловых скобках, тем самым вы говорите компилятору (а также программистам, которые будут читать ваш код), что здесь ожидается объект любого типа, следующий протоколу.

Например, `NSControl` имеет метод `setObjectValue:`, который должен быть у объекта, следующего протоколу `NSCopying`.

```
-(void) setObjectValue: (id<NSCopying>) obj;
```

При компиляции этого кода компилятор выполнит проверку типа аргумента и выдаст предупреждение наподобие "class 'Triangle' does not implement the 'NSCopying' protocol" ("класс 'Triangle' не реализует протокол 'NSCopying'"). Правда, удобно?

Дополнения Objective-C 2.0

Apple никогда не оставляет в покое хорошее, делая из него лучшее. Objective-C 2.0 добавляет к протоколам два новых модификатора: `@optional` и `@required`. Подождите минутку

с вашими комментариями. Мы говорили, что если вы следуете протоколу, то должны реализовать все его методы? Да, это так — в случае старых версий Objective-C. Если же у вас Objective-C 2.0, то вы можете поступить примерно так.

```
@protocol BaseballPlayer
```

```
- (void)drawHugeSalary;
```

```
@optional
```

```
- (void)slideHome;
```

```
- (void)catchBall;
```

```
- (void)throwBall;
```

```
@required
```

```
- (void)swingBat;
```

```
@end // BaseballPlayer
```

Так, класс, который принимает протокол `BaseballPlayer`, должен реализовать `-drawHugeSalary` и `-swingBat`, но остальные методы может и не реализовывать.

Зачем Apple поступает таким образом, притом что неформальные протоколы, кажется, работают вполне корректно? Просто это еще один инструмент в нашем арсенале, предназначенный для явного выражения наших намерений в объявлениях классов и методов. Скажем, в заголовочном файле вы увидели следующее.

```
@interface CalRipken : Person <BaseballPlayer>
```

Вам немедленно становится ясно, какие именно методы будут обязательно реализованы, что невозможно при применении неформальных протоколов. Точно так же могут быть украшены и аргументы методов.

```
-(void)draft:(Person<BaseballPlayer>);
```

Этот код делает очевидным, на что способен передаваемый объект. При разработке приложений iPhone вы можете заметить, как неформальные протоколы Cocoa становятся формальными протоколами с массой методов, помеченных как `@optional`.

Резюме

В этой главе мы ввели концепцию формального протокола. Формальный протокол определяется путем перечисления множества методов в блоке `@protocol`. Объекты принимают формальный протокол путем указания его имени в угловых скобках после имени класса в конструкции `@interface`. Когда объект принимает формальный протокол, он обещает реализовать все необходимые методы, перечисленные в протоколе. Компилятор помогает выполнить обещанное, выдавая предупреждения, если реализованы не все методы протокола.

Попутно мы разобрались с некоторыми нюансами объектно-ориентированного программирования, в частности, с вопросами, возникающими при копировании объектов, находящихся в иерархии классов.

А теперь — поздравляем! Вы познакомились с большей частью языка программирования Objective-C и глубоко изучили ряд вопросов, которые наиболее часто встают перед программистами в области объектно-ориентированного программирования. Теперь у вас заложен неплохой фундамент, позволяющий перейти к программированию Cocoa или к собственным проектам. В следующей главе вы попробуете, каково на вкус написание графических приложений Cocoa с применением Interface Builder и AppKit. Interface Builder и AppKit — душа программирования Cocoa и центральная тема большинства книг и проектов, посвященных Cocoa. Позже мы еще глубже зароемся в низкоуровневые возможности Cocoa.

Введение в AppKit

До сих пор в этой книге все наши программы использовали Foundation Kit и общались с нами при помощи старого доброго вывода на консоль. Для начала этого достаточно, но настоящая работа начнется, когда вы увидите Мас-образный интерфейс, включающий управляющие элементы, на которых можно вдоволь пощелкать мышью. В этой главе мы совершим обзорную экскурсию, чтобы показать вам избранные места Application Kit (или просто AppKit), сущий клад пользовательского интерфейса Cocoa.

Программа, которой мы займемся в данной главе, называется CaseTool, и ее можно найти в папке 14:01 CaseTool. CaseTool выводит окно, показанное на рис. 14.1. Это окно содержит фиксированный текст, метку и пару кнопок. После того как введете некоторый текст в текстовое поле и щелкнете на кнопке, введенный текст будет приведен к верхнему или нижнему регистру.

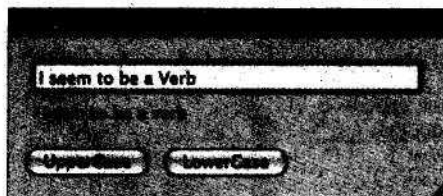


Рис. 14.1. Окончательная программа

Создание проекта

Мы будем использовать Xcode и Interface Builder и проведем вас шаг за шагом по всему процессу построения проекта. Первое, что надо сделать, — это создать файлы проекта. Затем надо составить схему пользовательского интерфейса и, наконец, соединить пользовательский интерфейс и код.

Начнем с запуска Xcode и создадим новый проект Cocoa Application, выбрав пункт меню **File**⇒**New Project**, а в диалоговом окне **New Project** — **Cocoa Application** (рис. 14.2). Затем присвоим имя новому проекту (рис. 14.3).

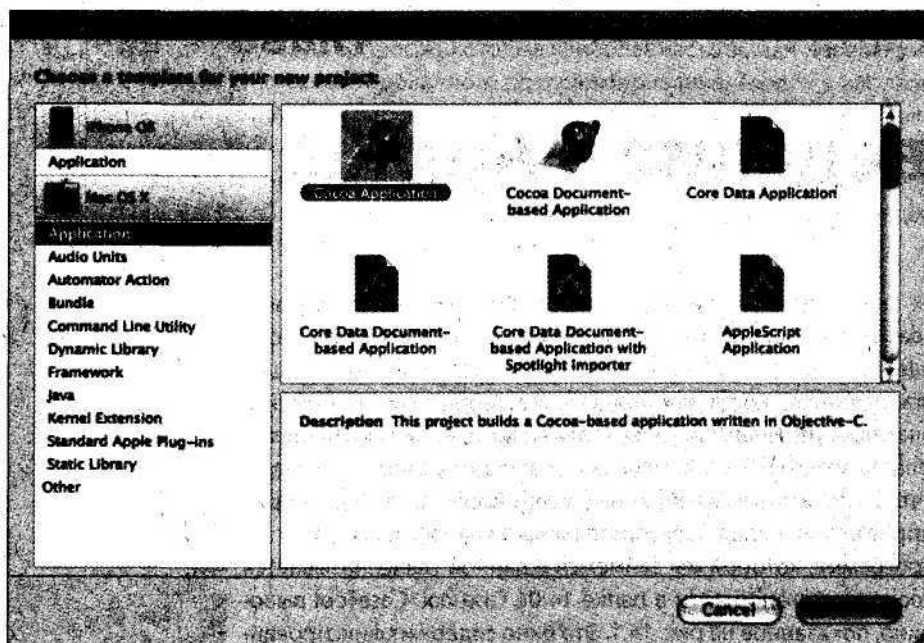


Рис. 14.2. Создание нового приложения Cocoa

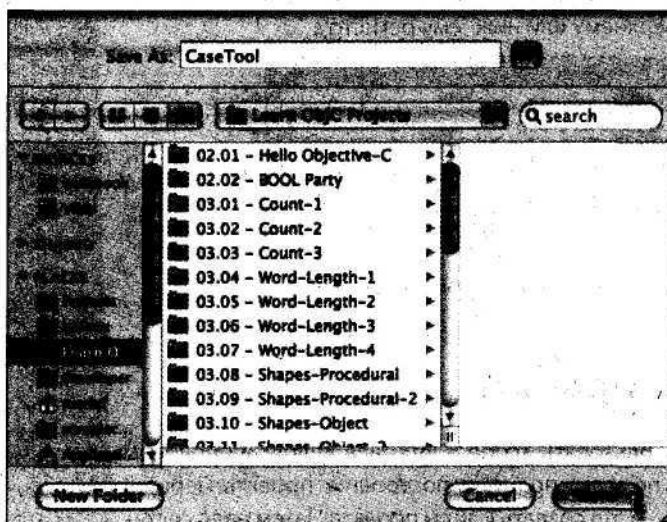


Рис. 14.3. Именованное нового проекта

Теперь добавим новый файл класса Objective-C, который назовем AppController (так как он управляет объектом нашего приложения). Выберите папку Sources в панели Groups & Files окна проекта. Затем выберите пункт меню File⇒New File. На рис. 14.4 показано, как Xcode запрашивает, какого вида файл должен быть создан (в данном случае это Objective-C class),

а на рис. 14.5 показано именование создаваемого файла. Убедитесь в том, что переключатель Also create AppDelegate.h включен.

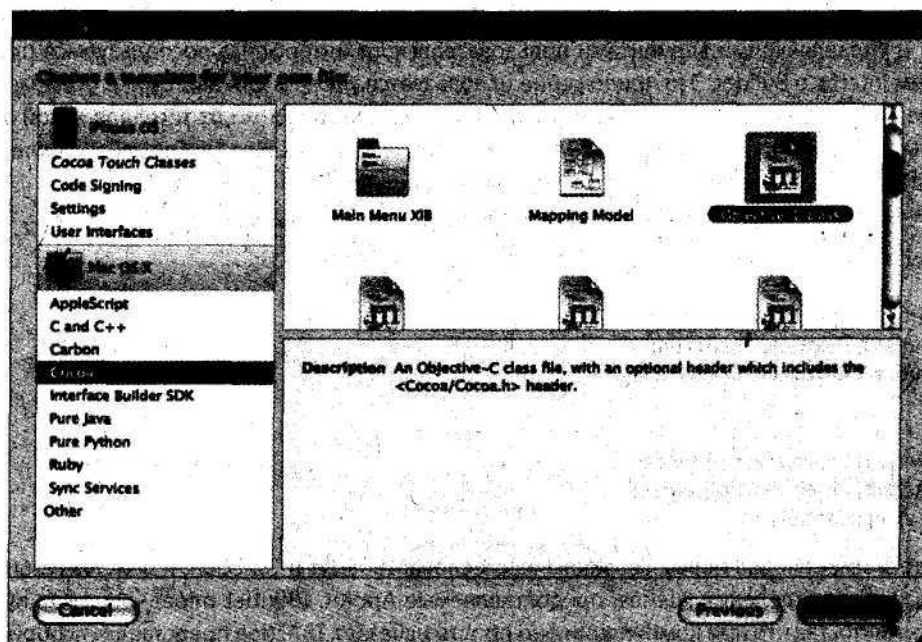


Рис. 14.4. Создание нового класса Objective-C

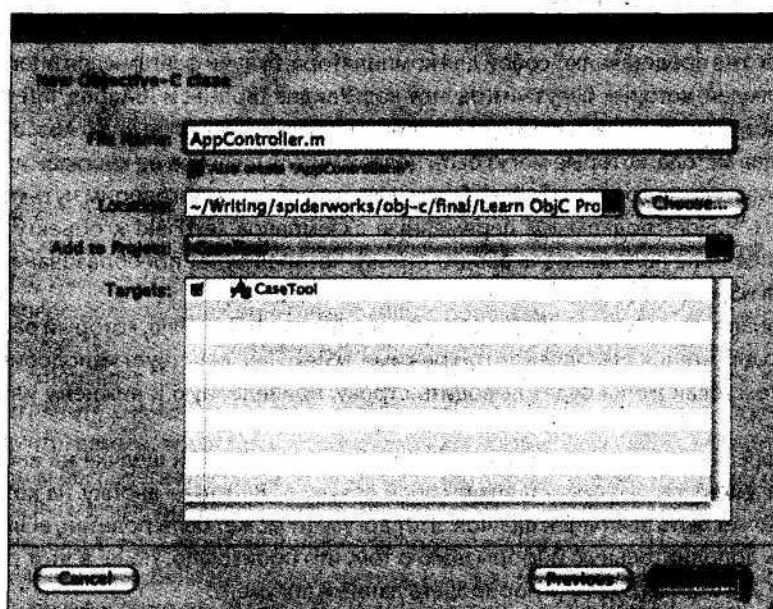


Рис. 14.5. Именование нового класса

Создание интерфейса класса

Для размещения содержимого окна и подключения соединений между классом `AppController` и управляющими элементами пользовательского интерфейса воспользуемся приложением `Interface Builder`. Это приложение используется также и для настройки приложений iPhone, так что затратить время на его изучение стоит, независимо от того, для какой платформы вы будете программировать. Мы внесем добавления в класс `AppController`, после чего `Interface Builder` позволит нам построить пользовательский интерфейс.

Начнем с заголовочного файла `AppController`.

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

@interface AppController : NSObject {
    IBOutlet NSTextField *textField;
    IBOutlet NSTextField *resultsField;
}

- (IBAction) uppercase: (id) sender;
- (IBAction) lowercase: (id) sender;
@end // AppController
```

Здесь имеется два новых квазиключевых слова: `IBOutlet` и `IBAction`. В действительности это просто определения `#define`, предоставляемые `AppKit`. `IBOutlet` определено как пустое место, ничто, так что при компиляции оно просто исчезает. `IBAction` представляет собой `void`, что означает, что тип возвращаемого значения объявленных методов в `AppController` — `void` (т.е. методы ничего не возвращают).

Если `IBOutlet` и `IBAction` ничего не делают, то зачем они здесь? Ответ заключается в том, что они ничего не представляют собой для компилятора, будучи флагами для `Interface Builder`, а также для людей, которые будут читать этот код. Увидев `IBOutlet` и `IBAction`, `Interface Builder` узнает, что объекты `AppController` имеют две переменные экземпляра и что `AppController` предоставляет два метода, которые могут быть целевыми для щелчков на кнопках (и других действий пользовательского интерфейса). О том, как это работает, поговорим чуть позже.

В `Interface Builder` подключим переменную экземпляра `textField` к объекту `NSTextField`. В это текстовое поле пользователь вводит строку, которая будет преобразована. Это типичная роль для `NSTextField`.

Переменная экземпляра `resultsField` будет соединена с `NSTextField`, который работает в режиме только для чтения. Находясь в этом режиме, `NSTextField` действует наподобие текстовой метки. Эта текстовая метка будет выводить строку, приведенную к нижнему или верхнему регистру.

Метод `uppercase`: — это тот метод, который будет вызван при щелчке на кнопке `UpperCase`. Его аргумент, `sender`, представляет собой объект `NSButton` — кнопку, на которой пользователем может быть выполнен щелчок. Этот аргумент может быть полезен, если необходимо получить дополнительную информацию о том, что произошло. Но не в этот раз: в нашей программе `CaseTool` мы будем игнорировать данный аргумент.

Метод `lowercase`: — это тот метод, который будет вызван при щелчке на кнопке `LowerCase`.

Interface Builder

Теперь перейдем к Interface Builder (для друзей — просто IB). Мы хотим отредактировать файл MainMenu.xib, который имеется в составе проекта. Этот файл содержит меню вместе с окном, в котором будут размещены пользовательские управляющие элементы.

Найдите в окне проекта Xcode файл MainMenu.xib и дважды щелкните на нем (рис. 14.6).

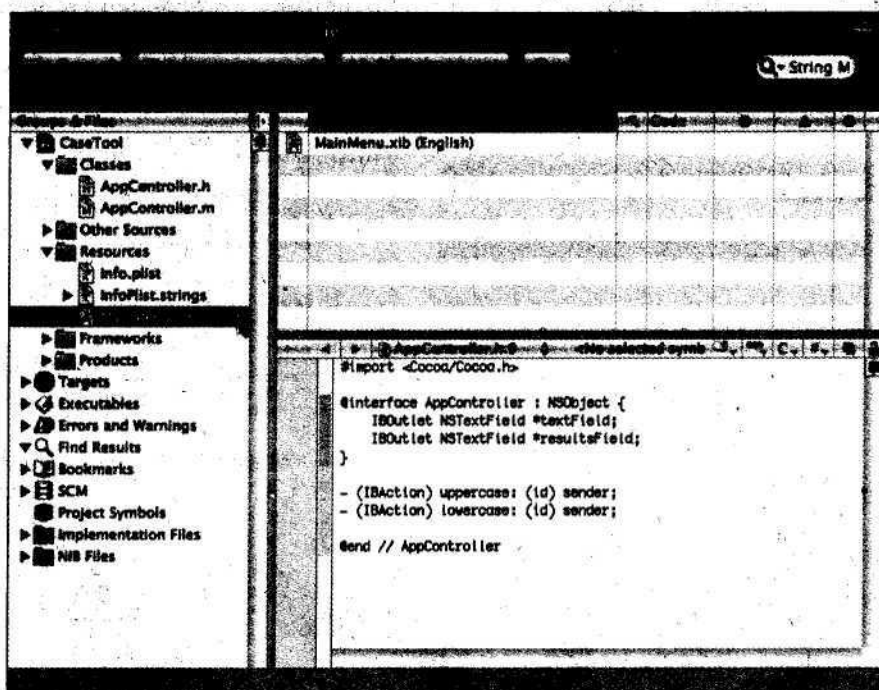


Рис. 14.6. Открытие MainMenu.xib

Это действие запустит Interface Builder, который откроет указанный файл. Несмотря на расширение .xib, мы называем такие файлы nib-файлами. "Nib" — аббревиатура для NeXT Interface Builder, указание на то, что Cocoa является наследием компании, носившей имя NeXT. Nib-файлы представляют собой бинарные файлы с объектами, а .xib-файлы — это просто nib-файлы в формате XML. Они преобразуются в nib-формат в процессе компиляции проекта.

Когда IB открывает файл, вы видите нечто похожее на четыре окна, показанные на рис. 14.7. Взгляните сначала в верхний левый угол — там вы видите окно Dock, в котором находятся пиктограммы, представляющие содержимое nib-файла. Это — основное окно nib-файла. Под ним располагается очень коротенькая (не пропустите ее случайно!) полоса меню приложения. Вы можете добавлять новые меню и пункты и редактировать существующие.

Теперь воспользуемся Interface Builder для продолжения работы над нашей программой. Мы попросим Interface Builder создать один из наших объектов AppDelegate. При запуске программы Cocoa загрузит nib-файл, и мы будем работать с объектом AppDelegate. Но сначала нам надо создать AppDelegate.

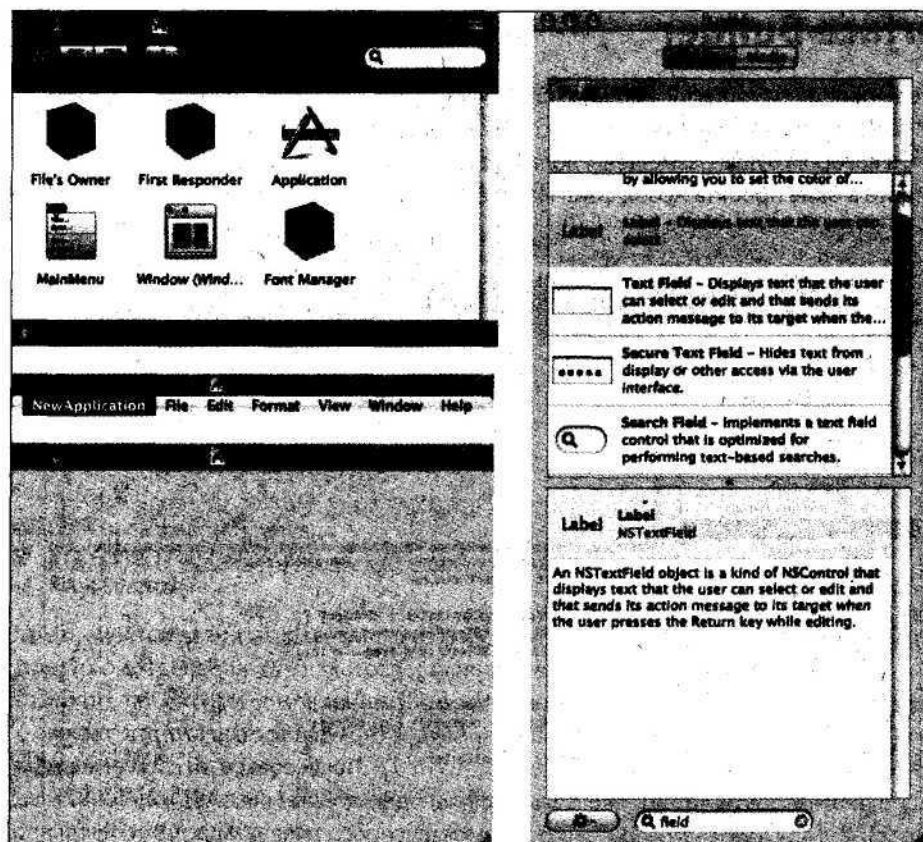


Рис. 14.7. Работа в Interface Builder

Перетащите NSObject из библиотеки в окно Dock файла CaseTool.xib. Соответствующая пиктограмма имеет имя Object, как показано на рис. 14.8.

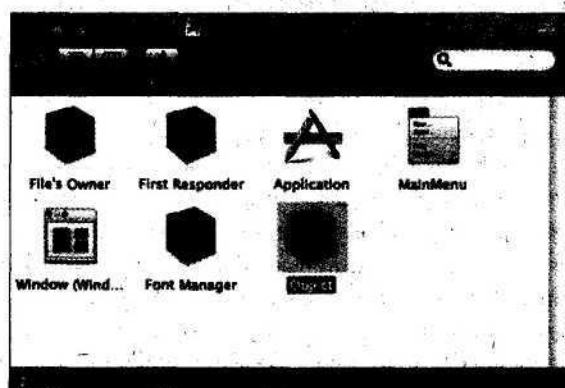


Рис. 14.8. После перетаскивания объекта из библиотеки

ки, которые помогут вам разместить объект в соответствии со спецификациями пользовательского интерфейса Apple.

Теперь перетащим управляющий элемент Label. Захватите объект Label в библиотеке и перетащите его в окно, как показано на рис. 14.12. Здесь будут выводиться результаты преобразования строк.

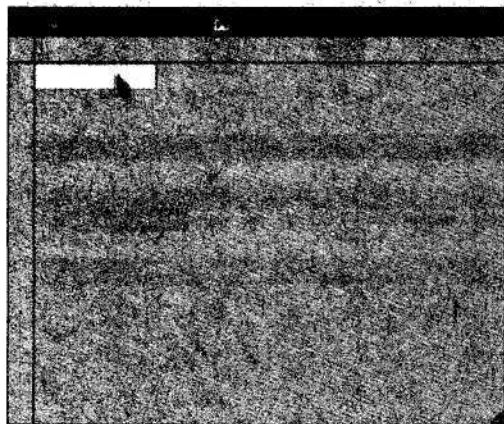


Рис. 14.11. Перетаскивание поля текстового ввода

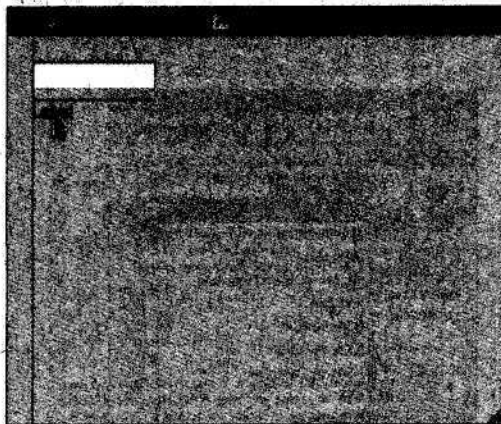


Рис. 14.12. Перетаскивание метки

Затем найдите в библиотеке кнопку и перетащите ее в окно, разместив ниже метки, как показано на рис. 14.13. Просто и красиво, правда?

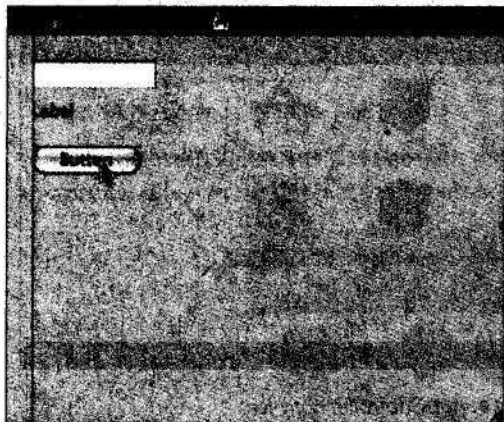


Рис. 14.13. Перетаскивание кнопки

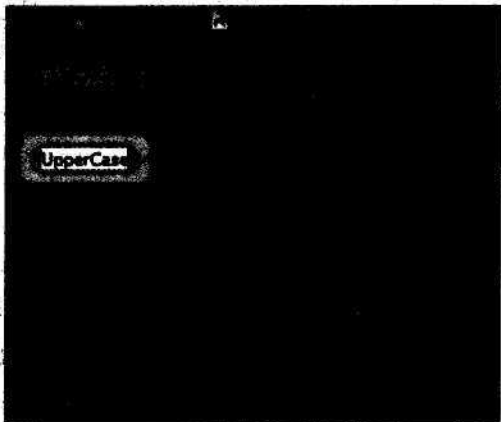


Рис. 14.14. Изменение подписи кнопки

Теперь выполните на размещенной кнопке двойной щелчок. Подпись на кнопке при этом станет редактируемой. Введите в поле UpperCase и нажмите <Enter> для принятия введенного текста в качестве новой надписи на кнопке (рис. 14.14).

Теперь точно так же перетащите еще одну кнопку и назовите ее LowerCase. На рис. 14.15 показан общий вид окна после добавления второй кнопки.

Теперь надо немного "причесать" пользовательский интерфейс, изменив размер поля текстового ввода и самого окна, как показано на рис. 14.16. Мы также изменили размер управляющего элемента Label так, чтобы он был растянут на все окно. Размера метки должно быть достаточно для вывода любого текста, вводимого в поле. Теперь окно приобрело завершённый вид.

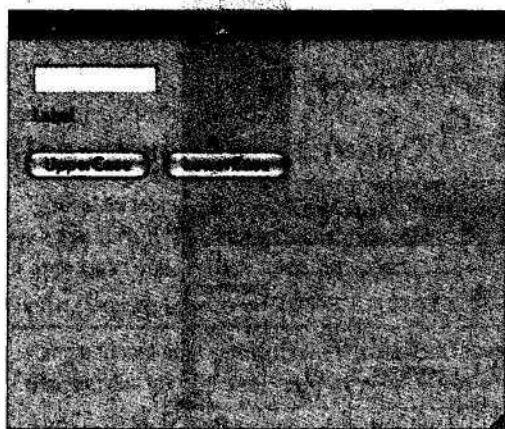


Рис. 14.15. Все элементы добавлены

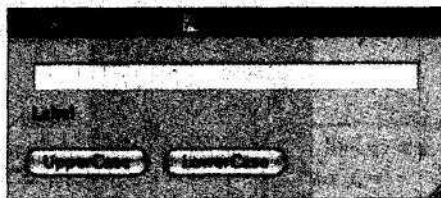


Рис. 14.16. Окончательный вид пользовательского интерфейса

Установка соединений

В этом разделе мы узнаем, как связать код с только что созданными элементами пользовательского интерфейса.

Подключение выходов

Теперь пришло время выполнить несколько подключений. Сначала нам надо пояснить объекту AppController, на какие объекты типа NSTextField должны указывать его переменные экземпляра textField и resultsField.

Начнем с того, что разместим окно MainMenu.xib и окно пользовательского интерфейса с текстовыми полями так, чтобы они были одновременно видимы на экране. Затем, при нажатой клавише <Control>, перетащите AppController в текстовое поле (рис. 14.17) — при этом за указателем мыши потянется синяя линия. При движении указателя над текстовым полем вы должны увидеть маленькую метку Text Field.

Когда вы отпустите кнопку мыши, появится меню с возможными вариантами подключения (меню Outlets), как показано на рис. 14.18. Выберите в нем textField.

Теперь выполните то же самое для поля метки, выбрав в этот раз в меню поле resultsField.

Проверьте установленные соединения, выбрав панель Connections (или воспользовавшись комбинацией клавиш <⌘+5>). Вы должны увидеть оба соединения в верхней части окна, как показано на рис. 14.19.

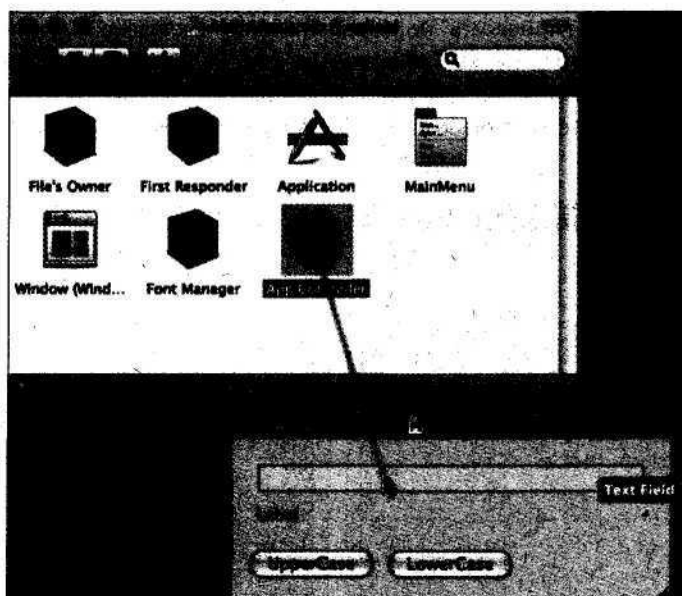


Рис. 14.17. Начало подключения

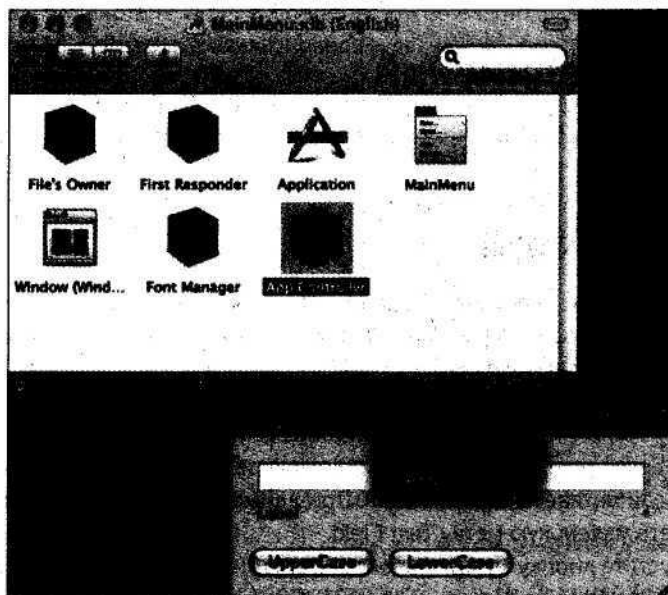


Рис. 14.18. Создание соединения

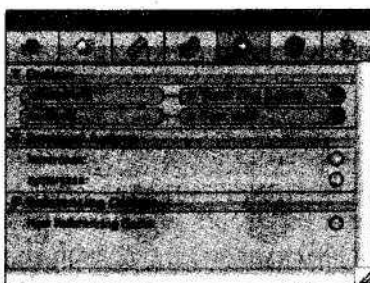


Рис. 14.19. Проверка соединений

Подключение действий

Теперь мы готовы назначить кнопкам действия, чтобы щелчок на них запускал выполнение нашего кода. Мы вновь выполняем перетягивание при нажатой клавише <Control>, но в этот раз от кнопок к AppController.

ПРИМЕЧАНИЕ

Направление перетягивания — вечный источник недоразумений в Interface Builder. Перетягивание направлено *от* объекта, который должен что-то знать, к объекту, о котором он должен знать.

AppController должен знать, какой объект NSTextField использовать для пользовательского ввода, так что перетягивание осуществляется от AppController к текстовому полю.

Кнопка должна знать, кому сказать “Эгей! На мне щелкнули!”, так что перетягивание выполняется от кнопки к AppController.

Щелкните при нажатой клавише <Control> на кнопке UpperCase и выполните перетаскивание к AppController, как показано на рис. 14.20.

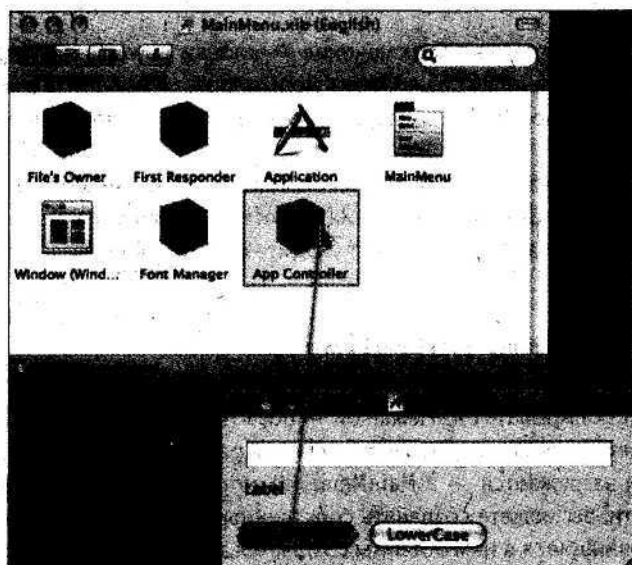


Рис. 14.20. Подключение кнопки UpperCase

После завершения перетягивания появится небольшое меню, в котором следует выбрать `uppercase`, как показано на рис. 14.21.

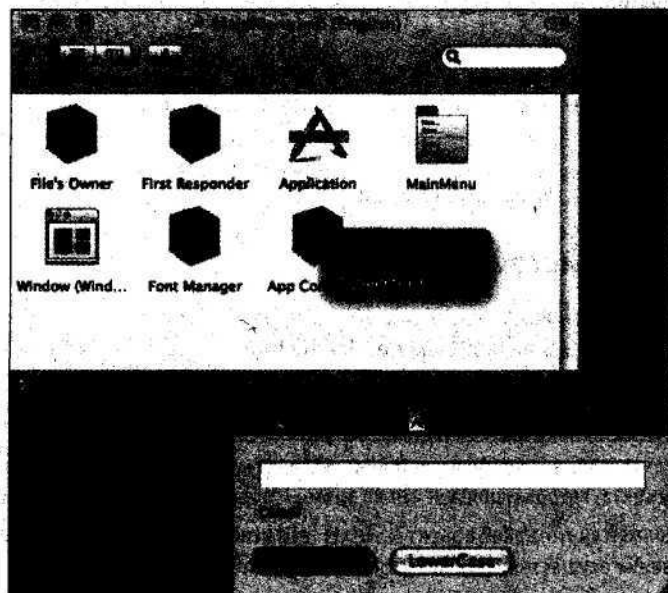


Рис. 14.21. Продолжение подключения кнопки `UpperCase`

Теперь при щелчке на кнопке экземпляру `AppController` будет послано сообщение `uppercase`. Все, что мы хотим сделать в ответ на щелчок кнопки, можно разместить в нашем методе `uppercase`.

Теперь выполняем последнее подключение — кнопки `LowerCase`. Перетащите при нажатой клавише `<Control>` кнопку `LowerCase` к `AppController` и выберите в меню `lowercase`. На этом наша работа в Interface Builder завершена. Сохраните свой `nib`-файл и можете выходить из Interface Builder.

Скорее всего, пока что перетягивание выполняется вами очень медленно, объекты с трудом находят свои места в окне, а соединения никак не соединяются. Все это дело практики, и со временем на все описанное у вас будет уходить меньше минуты.

Реализация `AppController`

А теперь хватит полировать стол мышкой — пора немного поработать. Займемся реализацией `AppController`. Но сначала пару слов о том, как работает `IBOutlet`.

Когда `nib`-файл загружается — а `MainMenu.nib` автоматически загружается при запуске приложения (кстати, вы можете создавать собственные `nib`-файлы и самостоятельно их загружать) — все хранящиеся в нем объекты создаются заново. Это означает, что за кулисами выполняются `alloc` и `init`. Так что при запуске приложения выделяется и инициализируется

экземпляр `AppController`. В процессе работы метода `init` все переменные экземпляра `IBOutlet` равны `nil`. И только после того, как созданы все объекты из `nib`-файла (включая окно, текстовые поля и кнопки), будут установлены все соединения.

После установления всех соединений (что просто представляет собой размещение адресов объектов `NSTextField` в переменных экземпляра `AppController`), каждому созданному объекту посылается сообщение `awakeFromNib`. Очень, очень распространенная ошибка — пытаться выполнить какую-то работу с `IBOutlet` в методе `init`. Поскольку все переменные экземпляра равны `nil`, все сообщения для них не приводят ни к каким действиям, так что любые действия, которые вы пытаетесь выполнить в `init`, молча не выполняются (это одно из тех мест, где Cocoa может заставить вас потерять немало времени на отладку). Если вы столкнетесь с тем, что у вас неполадки в методе `init`, выведите при помощи `NSLog` значения переменных экземпляра и убедитесь, что они равны `nil`. Кроме того, не имеется как предопределенного порядка создания объектов, так и порядка, в котором посылаются сообщения `awakeFromNib`.

Итак, за дело. Реализация `AppController` начинается с обычной "увертюры".

```
#import "AppController.h"
```

```
@implementation AppController
```

Затем идет метод `init`, в котором выводятся значения переменных экземпляра `IBOutlet` (которые равны `nil` — поверьте нам на слово).

```
-(id) init
{
    if (self = [super init]) {
        NSLog(@"init: text %@ / results %@",
              textField, resultsField);
    }
    return (self);
} // init
```

Чтобы пользовательский интерфейс имел более привлекательный вид, следует дать текстовым полям некоторые разумные значения по умолчанию вместо стандартного `Label`. Да, это верное значение по умолчанию, но совсем уж не интересное. Давайте поместим в поле ввода текст `Enter text here`, а в поле результата преобразования — `Results`. Идеальное место для этого — метод `awakeFromNib`.

```
-(void) awakeFromNib
{
    NSLog(@"awake: text %@ / results %@",
          textField, resultsField);

    [textField setStringValue: @"Enter text here"];
    [resultsField setStringValue: @"Results"];
} // awakeFromNib
```

Класс `NSString` имеет метод `setStringValue:`, который получает в качестве параметра `NSString` и изменяет содержимое текстового поля на переданное значение. Мы используем этот метод для изменения текстовых полей на что-то более интересное, чем значения по умолчанию.

Теперь переходим к методам действий, первым из которых является `uppercase:`.

```
- (IBAction) uppercase: (id) sender
{
    NSString *original;
    original = [textField stringValue];

    NSString *uppercase;
    uppercase = [original uppercaseString];

    [resultsField setStringValue: uppercase];
} // uppercase
```

Мы получаем исходную строку от `textField` при помощи сообщения `stringValue` и приводим ее к верхнему регистру. Класс `NSString` предоставляет нам удобный метод `uppercaseString`, который создает новую строку из исходной, переводя каждый ее символ в верхний регистр. Затем эта строка становится содержимым `resultsField`.

Пришло время задаться вечным вопросом: все ли в порядке в смысле управления памятью? Можете не сомневаться. Оба вновь создаваемых новых объекта (исходная строка и строка в верхнем регистре) получают при помощи методов, которые не являются ни `alloc`, ни `copy` и ни `new`, так что эти объекты находятся в пуле автоосвобождения. За копирование или захват входящей строки отвечает метод `setStringValue:`, и что и как он делает — это его дело. Мы знаем, что наши действия корректны с точки зрения управления памятью.

Метод `lowercase:` такой же, как и `uppercase:`, но приводит строку к нижнему регистру.

```
- (IBAction) lowercase: (id) sender
{
    NSString *original;
    original = [textField stringValue];

    NSString *lowercase;
    lowercase = [original lowercaseString];

    [resultsField setStringValue: lowercase];
} // lowercase
```

И это все! Запустив программу, вы увидите знакомое окно. Введите в поле ввода строку и измените ее регистр, как показано на рис. 14.22.

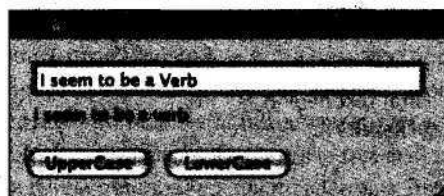


Рис. 14.22. Программа *CaseTool* работает так, как и предполагалось

Резюме

В этой главе мы заглянули в область графических приложений, где познакомились с Interface Builder и Application Kit. Непосредственно мы использовали только один класс AppKit (NSTextField) и пару классов косвенно (NSButton, отвечающий за кнопки, и NSWindow, объект которого управляет окном). В AppKit имеется более 100 различных классов, с которыми вы можете работать и многие из которых имеются в Interface Builder.

Теперь вы достаточно подготовлены, чтобы взяться за книгу, посвященную Cocoa, или за соответствующий проект. В следующей главе мы продолжим наше исследование и познакомимся с некоторыми низкоуровневыми возможностями Foundation Kit.

Работа с файлами

Большинство компьютерных программ в результате работы создают некий “полупостоянный” результат работы пользователя. Это может быть отредактированная фотография, глава книги или результат подсчетов трат супруги в косметическом магазине. В любом случае результат работы представляет собой некоторый дисковый файл.

Стандартная библиотека C предоставляет набор функций для создания, чтения и записи файлов, такие как `open()`, `read()`, `write()`, `fopen()` и `fread()`. Эти функции хорошо документированы, так что не будем о них говорить. В Cocoa имеется API Core Data, где вся работа с файлами выполняется для вас за сценой. Но мы не хотим говорить и на эту тему.

Тогда о чем будем говорить? О двух классах Cocoa для работы с файлами: списках свойств и кодировании объектов.

Списки свойств

В Cocoa имеется класс объектов, известный как *список свойств* (property list, часто сокращаемый до *plist*). Эти списки содержат небольшие множества объектов, с которыми Cocoa умеет работать, в частности, Cocoa знает, как сохранить их в файл и загрузить назад. Классы списков свойств включают `NSArray`, `NSDictionary`, `NSString`, `NSNumber`, `NSDate` и `NSData`, а также их изменяемые аналоги, если таковые существуют.

Первые четыре класса вы уже встречали, последние два — пока нет. Вкратце познакомимся с ними, прежде чем идти дальше. Весь код этой части главы можно найти в папке 15.01 PropertyListing.

NSDate

Работа с датой и временем — весьма распространенная задача в программировании. iPhoto знает, когда вы сфотографировали своего пса, а ваша программа для домашней бухгалтерии зна-

ет, когда вы позволили себе потратить немного лишнего. Класс NSDate — фундаментальный класс Сосоа для работы с датой и временем.

Для получения текущих даты и времени воспользуйтесь конструкцией [NSDate date], которая дает вам автоосвобожденный объект. Фрагмент кода

```
NSDate *date = [NSDate date];
NSLog(@"today is %@", date);
```

выведет на экран нечто вроде следующего.

```
today is 2008-08-23 11:32:02 -0400
```

У класса есть методы для сравнения двух дат, так что вы можете сортировать списки. Имеются методы для получения даты через некоторое время после (или за некоторое время до) текущего момента. Например, нас может интересовать дата, которая была ровно 24 часа назад.

```
NSDate *yesterday =
[NSDate dateWithTimeIntervalSinceNow: -(24 * 60 * 60)];
NSLog(@"yesterday is %@", yesterday);
```

Этот фрагмент кода приведет к такому выводу.

```
yesterday is 2008-08-22 11:32:02 -0400
```

Метод +dateWithTimeIntervalSinceNow: получает в качестве аргумента объект NSTimeInterval, который представляет собой typedef для double и является некоторым интервалом в секундах. Это позволяет вам указать время в будущем при помощи положительного значения и в прошлом — при помощи отрицательного.

NSData

Распространенной идиомой в языке программирования С является передача буфера данных функциям. Для этого функции обычно передается указатель на буфер и его длина. Кроме того, при этом в С возникает масса вопросов, связанных с управлением памятью. Например, если буфер был динамически выделен, то кто отвечает за его освобождение, когда он становится более не нужен?

Сосоа предоставляет класс NSData, который представляет собой "обертку" для блока байтов. Вы можете получить указатель на начало блока и его длину от объекта этого класса. Поскольку вы работаете с объектом, вопросы управления памятью разрешаются сами собой при помощи обычных правил. Так что, вместо передачи функции указателя на буфер памяти, вы можете передать автоосвобожденный объект NSData и не беспокоиться об его освобождении. Вот объект NSData, хранящий обычную строку С, которая представляет собой не что иное, как последовательность байтов, и выводит данные на экран.

```
const char *string = "Hi there, this is a C string!";
NSData *data = [NSData dataWithBytes: string
length: strlen(string) + 1];
NSLog(@"data is %@", data);
```


А вот как выглядит вывод на экран.

```
data is <48692074 68657265 2c207468 69732069 73206120 43207374 72696e67 2100>
```

Выглядит непривычно? Но если взять в руки таблицу ASCII (ее можно найти, запустив терминал и введя команду `man ascii`), то вы увидите, что все эти шестнадцатеричные числа на самом деле и есть наша строка. 0x48 — это "H", 0x69 — "i" и т.д. Метод `-length` дает количество байтов, а метод `-bytes` — указатель на начало строки. Вы обратили внимание на +1 в вызове `+dataWithBytes:?` Это необходимо для того, чтобы был включен завершающий C-строку нулевой символ (взгляните еще раз на вывод функции `NSLog` и убедитесь, что он завершается символами 00). При включении нулевого символа можно использовать спецификатор формата `%s` и вывести строку.

```
NSLog(@"%d byte string is '%s'", [data length], [data bytes]);
```

В результате на экран выводится следующее.

```
30 byte string is 'Hi there, this is a C string!'
```

Объекты `NSData` неизменяемы. После того как объект создан, вы можете использовать его, но не можете изменить его содержимое. Как обычно, имеется класс `NSMutableData`, который позволяет вам добавлять и удалять байты данных.

Запись и чтение списков свойств

Теперь, когда вы увидели все классы, относящиеся к спискам свойств, разберемся, что же мы можем с ними делать? Классы коллекций (`NSArray`, `NSDictionary`) имеют метод `-writeToFile:atomically:`, который записывает списки свойств в файл. `NSString` и `NSData` также имеют метод `-writeToFile:atomically:`, но он записывает в файл только строки или блоки данных.

Итак, мы можем загрузить массив строками, а затем сохранить его.

```
NSArray *phrase;
phrase = [NSArray arrayWithObjects: @"I", @"seem", @"to",
    @"be", @"a", @"verb", nil];
[phrase writeToFile: @"/tmp/verbiage.txt" atomically: YES];
```

Теперь взгляните на файл `/tmp/verbiage.txt`, и вы должны увидеть нечто наподобие следующего.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <array>
    <string>I</string>
    <string>seem</string>
    <string>to</string>
    <string>be</string>
    <string>a</string>
```

```

    <string>verb</string>
  </array>
</plist>

```

Этот код, несмотря на свою многословность, представляет собой именно то, что мы пытались сохранить: массив строк. Эти файлы списков свойств могут быть любой степени сложности, например с массивами словарей, которые содержат массивы строк, чисел и дат. В состав Xcode входит редактор списков свойств, так что вы можете открывать такие файлы и изменять их. Если оглянитесь в операционной системе, то найдете массу файлов списков свойств, наподобие файлов настроек в Library/Preferences в вашем домашнем каталоге или системных конфигурационных файлов в /System/Library/LaunchDaemons.

ПРИМЕЧАНИЕ

Некоторые файлы списков свойств, в особенности файлы настроек, хранятся в сжатом бинарном формате. Их можно конвертировать в удобочитаемый вид утилитой `plutil`, введя команду `plutil -convert xml1 filename.plist`.

Теперь, имея дисковый файл `verbiage.txt`, можно прочесть его при помощи метода `arrayWithContentsOfFile:`.

```

NSArray *phrase2 = [NSArray arrayWithContentsOfFile:
    @"tmp/verbiage.txt"];
NSLog(@"%@ ", phrase2);

```

Полученный вывод соответствует ранее сохраненной информации.

```

(
    I,
    seem,
    to,
    be,
    a,
    verb
)

```

ПРИМЕЧАНИЕ

Вы обратили внимание на слово "atomically" в методе `writeToFile:`? Это что, какой-то радиоактивный метод? Нет, все проще. Аргумент `atomically`, принимающий значение типа `BOOL`, говорит Сосоа, следует ли сначала сохранить содержимое файла во временном файле, а затем, после успешного сохранения, обменять этот временный файл на постоянный. Этот аргумент представляет собой механизм безопасности — если при сохранении что-то пойдет не так, вы не получите огрызок файла с непонятно чем внутри. Но такая безопасность имеет свою цену — двойной расход дискового пространства в процессе сохранения. Если вы не сохраняете огромного количества информации, которое в состоянии переполнить диск, лучше использовать атомарное сохранение файлов.

Если вы в состоянии привести ваши данные к типам списков свойств, то можете использовать эти весьма удобные вызовы для сохранения данных на диск и чтения их оттуда. Когда работаете с новыми идеями или с новым проектом, для ускорения дела можете воспользоваться описанными удобствами. Даже если вам надо всего лишь сохранить на диске блок данных и вы вообще не используете объекты, то для упрощения работы можете применить `NSData`. Просто оберните ваши данные в объект `NSData` и вызовите его метод `writeToFile:atomically:`.

Недостатком этих функций является то, что они не возвращают информацию об ошибках. Если вы не можете загрузить файл, то получите от метода указатель `nil`, без каких-либо комментариев, что же именно произошло.

Кодирование объектов

К сожалению, вы не всегда в состоянии выразить информацию ваших объектов в виде классов списков свойств. Если бы мы могли выразить все в виде словарей массивов, зачем бы нам нужны были собственные классы? К счастью, Сосоа имеет механизм для преобразования объектов в формат, который может быть сохранен на диск. Объекты могут кодировать свои переменные экземпляра и другую информацию в блоки данных, которые можно сохранять на диске. Такой блок данных позже может быть считан обратно в память, и на основании этих данных могут быть созданы новые объекты. Такой процесс называется *кодированием и декодированием*, или *сериализацией и десериализацией*.

Если вы помните нашу вылазку в Interface Builder в прошлой главе, то не забыли, как мы перетаскивали объекты из библиотеки в окно и все это сохранялось в `nib`-файле. Другими словами, объекты `NSWindow` и `NSTextField` были сериализованы и сохранены на диске. Когда при запуске программы `nib`-файл загружался в память, объекты десериализовались и создавались и связывались вместе новые объекты `NSWindow` и `NSTextField`.

Как вы, вероятно, можете предположить, то же самое можете делать со своими собственными объектами путем принятия протокола `NSCoding`. Протокол имеет следующий вид.

```
@protocol NSCoding
- (void) encodeWithCoder: (NSCoder *) aCoder;
- (id) initWithCoder: (NSCoder *) aDecoder;
@end
```

Принимая этот протокол, вы обещаете реализовать оба указанных метода. Когда вашим объектам поступает запрос о сохранении, вызывается метод `-encodeWithCoder:`; в ответ на запрос о загрузке вызывается метод `-initWithCoder:`.

Так что же такое этот кодировщик? `NSCoder` — это абстрактный класс, определяющий набор методов для преобразования ваших объектов в `NSData` и обратно. Вы никогда не создаете новый объект `NSCoder`, поскольку сам он ничего не делает. Однако имеется несколько конкретных подклассов `NSCoder`, которые используются при кодировании и декодировании ваших объектов. Мы будем работать с двумя из них, `NSKeyedArchiver` и `NSKeyedUnarchiver`.

Вероятно, простейший способ разобраться с ними — это конкретный пример. Взгляните на проект в папке 15.02 `SimpleEncoding`.

Начнем с простого класса с несколькими переменными экземпляра.

```

@interface Thingie : NSObject <NSCoding> {
    NSString *name;
    int magicNumber;
    float shoeSize;
    NSMutableArray *subThingies;
}

@property (copy) NSString *name;
@property int magicNumber;
@property float shoeSize;
@property (retain) NSMutableArray *subThingies;

- (id)initWithName: (NSString *) n
    magicNumber: (int) mn
    shoeSize: (float) ss;
@end // Thingie

```

Все это должно быть для вас знакомо. У нас имеется четыре переменные экземпляра — скаляры и объекты, в том числе одна коллекция. Для каждой из переменных объявлено соответствующее свойство (свойства по умолчанию являются `readwrite`, так что мы не упоминаем об этом в их объявлениях). Кроме того, имеется инициализатор, создающий новый объект Thingie с нуля.

Обратите внимание, что класс Thingie принимает протокол `NSCoding`. Это означает, что мы должны обеспечить реализацию методов `encodeWithCoder` и `initWithCoder`. Пока что оставим их пустыми.

```

@implementation Thingie

@synthesize name;
@synthesize magicNumber;
@synthesize shoeSize;
@synthesize subThingies;

- (id)initWithName: (NSString *) n
    magicNumber: (int) mn
    shoeSize: (float) ss {
    if (self = [super init]) {
        self.name = n;
        self.magicNumber = mn;
        self.shoeSize = ss;
        self.subThingies = [NSMutableArray array];
    }
    return (self);
}

- (void)dealloc {
    [name release];
    [subThingies release];
    [super dealloc];
}

```

```

} // dealloc

- (void) encodeWithCoder: (NSCoder *) coder {
    // Никого нет дома...
} // encodeWithCoder

- (id) initWithCoder: (NSCoder *) decoder {
    return (nil);
} // initWithCoder

- (NSString *) description {
    NSString *description =
        [NSString stringWithFormat:@"%@@: %d/%.1f %@",
            name, magicNumber, shoeSize, subThingies];
    return (description);
} // description
@end // Thingie

```

Приведенный код инициализирует объект, освобождает память по окончании работы, создает заготовки методов, необходимых для полного счастья компилятора, — чтобы он ни на что не жаловался, зная, что класс принимает протокол `NSCoding`, и возвращает описание.

Обратите внимание, что в методе инициализации мы используем в левой части присваиваний конструкцию `self.attribute`. Вспомните, что это значит — в действительности для этих атрибутов мы вызываем методы доступа и эти методы создаются при помощи директив `@synthesize`. Мы не выполняем непосредственное присваивание переменным экземпляра. Такая методика создания объектов обеспечивает корректное управление памятью при передаче `NSString` и при работе с создаваемым объектом `NSMutableArray`, так что нам не требуется явно заниматься этим вопросом.

В функции `main()` мы создаем объект `Thingie` и выводим его описание.

```

Thingie *thing1;
thing1 = [[Thingie alloc]
    initWithName: @"thing1"
    magicNumber: 42
    shoeSize: 10.5];
NSLog(@"some thing: %@", thing1);

```

Этот код выводит на экран следующее.

```

some thing: thing1: 42/10.5 (
)

```

Это было просто. Теперь давайте архивируем наш объект. Реализуем метод `encodeWithCoder:` класса `Thingie` следующим образом.

```

- (void) encodeWithCoder: (NSCoder *) coder {
    [coder encodeObject: name
        forKey: @"name"];
    [coder encodeInt: magicNumber
        forKey: @"magicNumber"];
}

```



```
[coder encodeFloat: shoeSize
    forKey: @"shoeSize"];
[coder encodeObject: subThingies
    forKey: @"subThingies"];
} // encodeWithCoder
```

Для выполнения работы по архивированию наших объектов в NSData мы используем NSKeyedArchiver. Этот архиватор использует пары "ключ/значение" для хранения информации об объекте. Метод `-encodeWithCoder` класса Thingie кодирует каждую переменную экземпляра с ключом, соответствующим имени переменной экземпляра. Вы не обязаны поступать именно так — ничто не мешает вам закодировать имя с ключом `flarblewhazzit`. Просто соответствие ключей именам переменных экземпляра упрощает запоминание, что чему соответствует.

Для ключей кодирования можно использовать как обычные литеральные строки, так и, во избежание опечаток, константы. Вы можете определить константу при помощи директивы `#define`, например, как `#define kSubthingiesKey @"subThingies"`, или воспользоваться переменной, локальной для данного файла, наподобие `static NSString* kSubthingiesKey = @"subThingies";`

Обратите внимание на наличие различных `encodeSomethingForKey:` для каждого типа. Вы должны убедиться в применении корректного метода для кодирования ваших типов. Для любого типа Objective-C используйте `encodeObjectForKey:`.

При восстановлении объекта используем методы `decodeSomethingForKey:`.

```
- (id) initWithCoder: (NSCoder *) decoder {
    if (self = [super init]) {
        self.name =
            [decoder decodeObjectForKey: @"name"];
        self.magicNumber =
            [decoder decodeIntForKey: @"magicNumber"];
        self.shoeSize =
            [decoder decodeFloatForKey: @"shoeSize"];
        self.subThingies =
            [decoder decodeObjectForKey: @"subThingies"];
    }
    return (self);
} // initWithCoder
```

Метод `initWithCoder:` подобен прочим инициализирующим методам. Перед тем как приступать к каким-либо действиям, вы должны инициализировать ваш надкласс. У вас есть два способа сделать это — в зависимости от того, что собой представляет родительский класс. Если родительский класс принимает протокол `NSCoding`, вы должны вызвать `[super initWithCoder: decoder]`. В противном случае вы должны вызвать просто `[super init]`. Класс `NSObject` не следует протоколу `NSCoding`, поэтому мы вызываем простейший метод `init`.

При использовании `decodeIntForKey:` вы извлекаете из декодера значение типа `int`. Применяя `decodeObjectForKey:`, вы извлекаете из декодера объект с рекурсивным использованием `initWithCoder:` для всех вложенных объектов. Управление памятью работает так, как вы ожидаете, — вы получаете объекты от метода, который не является ни `alloc`, ни `copy` и ни `new`,

так что можете считать, что объекты автоосвобождены. Наши объявления свойств обеспечивают корректное управление памятью.

Обратите внимание, что кодирование и декодирование используют один и тот же порядок, совпадающий с порядком переменных экземпляра. Вообще говоря, вы не обязаны поступать именно так, но такая привычка позволит вам кодировать и декодировать все переменные, ничего не забывая. Одна из причин применения ключей заключается именно в том, что в таком случае вы можете выполнять кодирование и декодирование в произвольном порядке.

А теперь давайте воспользуемся всем этим. У нас есть ранее созданный объект `thing1`. Давайте заархивируем его.

```
NSData *freezeDried;
freezeDried = [NSKeyedArchiver
    archivedDataWithRootObject: thing1];
```

Метод класса `+archivedDataWithRootObject:` выполняет кодирование объекта. Сначала он создает за сценой экземпляр `NSKeyedArchiver`, а затем передает его методу `-encodeWithCoder` объекта `thing1`. Когда `thing1` кодирует свои атрибуты, это может привести к кодированию других объектов, наподобие строк или массивов, а также содержимого, которое может быть помещено в массив. Когда завершается кодирование всей этой горы объектов, архиватор `NSKeyedArchiver` упаковывает полученные данные в объект `NSData` и возвращает его.

При желании мы можем сохранить этот объект `NSData` на диск при помощи метода `-writeToFile:atomically:`. Здесь же мы просто удалим `thing1`, а затем воссоздадим его из архива и выведем информацию на консоль.

```
[thing1 release];
thing1 = [NSKeyedUnarchiver
    unarchiveObjectWithData: freezeDried];
NSLog (@"reconstituted thing: %@", thing1);
```

Вывод в точности тот же, что и ранее.

```
reconstituted thing: thing1: 42/10.5 (
)
```

В начале работы с классом на стене висело ружье `NSMutableArray`. Согласно Чехову, пришла пора выстрела из него. Мы можем добавлять объекты в массив `subThingies`, и они будут автоматически закодированы при кодировании массива. Реализация метода `encodeWithCoder:` класса `NSArray` вызывает `encodeWithCoder` для всех его объектов, так что все объекты будут кодированы. Добавим несколько объектов в `subThingies`.

```
Thingie *anotherThing;
anotherThing = [[[Thingie alloc]
    initWithName: @"thing2"
    magicNumber: 23
    shoeSize: 13.0] autorelease];
[thing1.subThingies addObject: anotherThing];
anotherThing = [[[Thingie alloc]
    initWithName: @"thing3"
```

```

        magicNumber: 17
        shoeSize: 9.0] autorelease];
[thing1.subThings addObject: anotherThing];
NSLog(@"thing with things: %@", thing1);

```

Вот что мы получим при работе программы.

```

thing with things: thing1: 42/10.5 (
  thing2: 23/13.0 (
  ),
  thing3: 17/9.0 (
  )
)

```

Кодирование и декодирование работают точно так же.

```

freezeDried =
[NSKeyedArchiver archivedDataWithRootObject: thing1];
thing1 =
[NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];
NSLog(@"reconstituted multithing: %@", thing1);

```

И вывод на экран получаем в точности таким же, что и ранее.

А что произойдет при наличии циклов в кодируемых данных? Например, что если thing1 находится в собственном массиве subThings? Не будет ли thing1 кодировать массив, который вызовет кодирование thing1, который будет кодировать массив, который...? К счастью, реализация архиваторов и разархиваторов Cocoa достаточно интеллектуальна, чтобы справиться с циклами объектов.

Чтобы убедиться в этом, поместим thing1 в его собственный массив subThings.

```

[thing1.subThings addObject: thing1];

```

Но не пытайтесь использовать NSLog с thing1. Функция NSLog недостаточно разумна для обнаружения циклов объектов, так что она может войти в бесконечную рекурсию, пытаясь построить строку для вывода, что в конечном счете завершится для вас тысячами вызовов -description в стеке вызовов отладчика...

Но если вы попытаетесь выполнить кодирование и декодирование thing1, все будет корректно работать без каких-либо неприятностей.

```

freezeDried =
[NSKeyedArchiver archivedDataWithRootObject: thing1];
thing1 =
[NSKeyedUnarchiver unarchiveObjectWithData: freezeDried];

```

Резюме

Как вы узнали из этой главы, Cocoa предоставляет два способа загрузки и сохранения файлов. Типы данных списков свойств представляют собой коллекции классов, которые знают, каким образом сохранять и загружать самих себя. Если у вас имеется коллекция объек-

тов, каждый из которых имеет тип списка свойств, вы можете легко сохранять их на диск и загружать обратно в память.

Если, как и большинство программистов Сосоа, вы разрабатываете собственные объекты, которые не являются списками свойств, можете принять протокол `NSCoding` и реализовать методы для кодирования и декодирования объектов. Благодаря этим методам вы можете превратить свои объекты в объекты `NSData`, которые затем можно сохранять на диск и позже загружать с диска. Затем из такого загруженного объекта `NSData` вы можете восстанавливать свои объекты.

Далее мы рассмотрим кодирование "ключ/значение", которое позволит вам взаимодействовать со своими объектами на более высоком уровне абстракции.

Кодирование "ключ/значение"

Косвенность — вот та концепция, к которой мы возвращаемся опять и опять. Многие методы программирования основаны на косвенности, включая объектно-ориентированное программирование в целом. В этой главе мы обратимся еще к одному механизму косвенности. Это — возможность не языка программирования Objective-C, а Cocoa. До сих пор мы изменяли состояние объекта непосредственно путем вызова методов — как прямо, так и через использование записи с точкой для свойств — или путем установки значений переменных экземпляра. Кодирование "ключ/значение" (key/value coding, или KVC) представляет собой способ косвенного изменения состояния объекта, при помощи строк описания изменения состояния объекта. Вся эта глава посвящена данному вопросу. Некоторые из более передовых возможностей Cocoa, такие как Core Data или Cocoa Bindings (которые выходят за рамки данной книги), используют KVC в качестве шестеренок своего механизма.

Исходный проект

Мы будем продолжать работать с нашим старым знакомым — проектом CarParts (см. папку 16.01 Car-Value-Coding). Добавим к классу Car несколько атрибутов, таких как изготовитель и модель, и вернем переменной appellation имя name.

```
@interface Car : NSObject <NSFirstSCopying> {  
    NSString *name;  
    NSMutableArray *tires;  
    Engine *engine;  
    NSString *make;  
    NSString *model;  
    int modelYear;  
    int numberOfDoors;  
    float mileage;  
}
```

```

@property (readwrite, copy) NSString *name;
@property (readwrite, retain) Engine *engine;
@property (readwrite, copy) NSString *make;
@property (readwrite, copy) NSString *model;
@property (readwrite) int modelYear;
@property (readwrite) int numberOfDoors;
@property (readwrite) float mileage;
...
@end // Car

```

Кроме того, добавим директивы @synthesize, чтобы компилятор мог автоматически генерировать методы доступа.

```
@implementation Car
```

```

@synthesize name;
@synthesize engine;
@synthesize make;
@synthesize model;
@synthesize modelYear;
@synthesize numberOfDoors;
@synthesize mileage;
...

```

Обновим метод -copyWithZone с тем, чтобы он был в состоянии работать с новыми атрибутами.

```

- (id) copyWithZone: (NSZone *) zone
{
    Car *carCopy;
    carCopy = [[[self class]
                allocWithZone: zone]
               init];
    carCopy.name = name;
    carCopy.make = make;
    carCopy.model = model;
    carCopy.numberOfDoors = numberOfDoors;
    carCopy.mileage = mileage;
    // А также копирование двигателя и колес,
    // как в главе 13.
}

```

Чтобы обеспечить вывод информации о новых атрибутах, внесем добавления в метод -description.

```

- (NSString *) description {
    NSString *desc;
    desc = [NSString stringWithFormat:
            @"%@, a %d %@ %%, has %d doors, "
            "%.1f miles, and %d tires.",
            name, modelYear, make, model,

```

```

    numberOfDoors, mileage, [tires count]);
    return desc;
} // description

```

Наконец, в функции `main()` устанавливаем все указанные свойства автомобиля и выводим информацию о нем на экран. Мы используем сообщение `autorelease` вместе с `alloc` и `init`, так что все требования управления памятью соблюдены.

```

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    Car *car = [[[Car alloc] init] autorelease];
    car.name = @"Herbie";
    car.make = @"Honda";
    car.model = @"CRX";
    car.numberOfDoors = 2;
    car.modelYear = 1984;
    car.mileage = 110000;

    int i;
    for (i = 0; i < 4; i++) {
        AllWeatherRadial *tire;
        tire = [[AllWeatherRadial alloc] init];
        [car setTire:tire
         atIndex:i];
        [tire release];
    }

    Slant6 *engine = [[[Slant6 alloc] init] autorelease];
    car.engine = engine;

    NSLog(@"Car is %@", car);

    [pool release];
    return (0);
} // main

```

При выполнении программы получите строку описания автомобиля следующего вида.

```
Car is Herbie, a 1984 Honda CRX, has 2 doors, 110000.0 miles, and 4 tires.
```

Введение в KVC

Фундаментальными вызовами в кодировании "ключ/значение" являются `-valueForKey:` и `-setValue:forKey:`. Вы посылаете сообщение объекту и передаете строку, которая является ключом интересующего вас атрибута.

Так, мы можем запросить имя автомобиля.

```
NSString *name = [car valueForKey:@"name"];
NSLog(@"%@", name);
```

Этот код даст нам имя Herbie. Аналогично можно запросить производителя.

```
NSLog(@"make is %@", [car valueForKey:@"make"]);
```

Метод `valueForKey:`, проявляя чудеса хитрости, находит название производителя и возвращает его.

Метод `valueForKey:` сначала просматривает имена получающих методов доступа в поисках имени `-key` или `-isKey` для значения `key`. Так что в приведенных выше вызовах выполняется поиск `-name` и `-make`. Если получающий метод доступа не обнаруживается, выполняется поиск переменной экземпляра с именем `_key` или `key`. Таким образом, если мы не предоставляем методы доступа с применением директивы `@synthesize`, метод `valueForKey` будет искать переменные экземпляра `_name` и `name`, а также `_make` и `make`.

Последнее не так просто: метод `-valueForKey` использует метаданные системы времени выполнения Objective-C для "взлома" объекта и поиска интересующей информации. Выполнить такие действия в C или C++ невозможно. Используя кодирование "ключ/значение", вы можете получить значения при отсутствии методов доступа и без непосредственного доступа к переменной экземпляра посредством указателя на объект.

Та же методика работает и для года выпуска модели:

```
NSLog(@"model year is %@", [car valueForKey:@"modelYear"]);
```

что даст нам строку `model year is 1984`.

Стоп! %@ в `NSLog` выводит объект, но ведь `modelYear` представляет собой `int`, а не объект! Что же происходит? В кодировании "ключ/значение" Cocoa автоматически упаковывает и распаковывает скалярные значения — т.е. автоматически помещает скалярные значения (`int`, `float`, некоторые `struct`) в `NSNumber` или `NSValue` при использовании `valueForKey` и так же автоматически извлекает их оттуда при применении метода `-setValueForKey`. Такую автоупаковку выполняет только кодирование "ключ/значение". Обычные методы и синтаксис свойств этого не делают.

В дополнение к получению значений, вы можете устанавливать их при помощи `-setValueForKey:`.

```
[car setValue:@"Harold" forKey:@"name"];
```

Этот метод работает аналогично `-valueForKey:`. Сначала он ищет устанавливающий метод доступа с соответствующим именем, типа `-setName`, и вызывает его с аргументом `@ "Harold"`. Если устанавливающего метода доступа нет, просматривается класс в поисках переменной экземпляра с именем `name` или `_name` и выполняется присваивание этой переменной.

ЭТО ПРАВИЛО СТОИТ ПОДЧЕРКНУТЬ

Компилятор и Apple резервируют имена переменных экземпляра, начинающиеся с символа подчеркивания, обещая гром и молнию на вашу голову, если вы попытаетесь сами использовать такое имя. В настоящее время принуждения следовать данному правилу со стороны компилятора не наблюдается, но никто не застрахован от этого в будущем — так что если хотите нарушить это правило, делайте это на свой страх и риск.

При установке скалярного значения перед вызовом `-setValue:forKey:` следует его упаковать.

```
[car setValue: [NSNumber numberWithFloat: 25062.4]
 forKey: @"mileage"];
```

Метод `-setValue:forKey:` распакует переданное значение перед тем, как вызвать `-setMileage:` или изменить переменную экземпляра непосредственно.

В путь

В дополнение к установке значений с применением ключа, кодирование "ключ/значение" позволяет указать путь ключей, который, как путь файловой системы, обеспечивает продвижение по цепочке связей.

В качестве примера рассмотрим, как добавить значение мощности к двигателю. Мы добавляем новую переменную экземпляра в класс `Engine`.

```
@interface Engine : NSObject <NSCopying> {
    int horsepower;
}
@end // Engine
```

Обратите внимание, что мы не добавляем никакие методы доступа или свойства. Обычно мы обеспечиваем класс свойствами или методами доступа для интересующих нас атрибутов объекта, но здесь мы их избегаем для того, чтобы продемонстрировать, как кодирование "ключ/значение" в состоянии работать с объектами непосредственно.

Добавим метод инициализации для того, чтобы двигатель создавался с ненулевой мощностью.

```
-(id) init {
    if (self = [super init]) {
        horsepower = 145;
    }
    return (self);
} // init
```

Еще мы добавим в метод `-copyWithZone` копирование нашей переменной экземпляра, а также подправим метод `-description`, который несколько устарел, так чтобы он выводил, среди прочего, информацию о мощности двигателя.

Протестируем наши возможности получать и устанавливать значения при помощи следующего кода.

```
NSLog(@"horsepower is %@",
      [engine valueForKey: @"horsepower"]);
[engine setValue: [NSNumber numberWithInt: 150]
 forKey: @"horsepower"];
NSLog(@"horsepower is %@",
      [engine valueForKey: @"horsepower"]);
```

Он выводит следующее.

```
horsepower is 145
horsepower is 150
```

Но вернемся к путям ключей. Что же это такое? Вы можете указывать различные имена атрибутов, разделенные точками. Запрашивая автомобиль о его "engine.horsepower", мы получим значение мощности двигателя. Давайте испытаем путь ключей с методами -valueForKeyPath и -setValueForKeyPath. Отправим соответствующие сообщения не двигателю, а автомобилю.

```
[car setValue: [NSNumber numberWithInt: 155]
 forKeyPath: @"engine.horsepower"];
NSLog(@"horsepower is %@",
 [car valueForKeyPath: @"engine.horsepower"]);
```

Эти пути ключей могут быть произвольной глубины, в зависимости от сложности графа нашего объекта; например, могут быть пути наподобие "car.interior.airconditioner.fan.velocity". Зачастую такое погружение вглубь объекта оказывается проще при помощи пути ключей, чем с применением ряда вложенных вызовов методов.

Агрегаты

Одно из приятных свойств кодирования "ключ/значение" в том, что если вы запрашиваете в качестве значения ключа NSArray, то в действительности запрашивается каждый объект массива, и все они упаковываются в другой массив, который и возвращается вам. Так же выполняется работа и с массивами внутри объекта (вспомните композицию), к которым вы обращаетесь посредством пути ключей.

Вставленные в другие объекты массивы NSArray известны на языке кодирования "ключ/значение" как имеющие связь "ко многим". Например, автомобиль связан со многими (ну ладно, четырьмя) колесами. Поэтому можно говорить о связи Car ко многим Tire. Если путь ключей включает атрибут массива, оставшаяся часть пути отправляется каждому из объектов в массиве.

МНОГИЕ И ОДИН

Теперь, зная о связи "ко многим", вы можете задаться вопросом, а что же такое связь "к одному"? Обычная композиция объектов и есть связь "к одному". Автомобиль, например, имеет связь "к одному" с двигателем.

Вспомните, что Car имеет массив колес, а каждое колесо имеет свое давление pressure. Давление во всех колесах можно получить единственным вызовом.

```
NSArray *pressures = [car valueForKeyPath: @"tires.pressure"];
```

После этого вызов

```
NSLog(@"pressures %@", pressures);
```

выведет на экран следующие результаты.

```
pressures (
  1.5,
  1.5,
  1.5,
  1.5
)
```

Что же в точности здесь произошло, помимо того, что мы убедились, что с колесами все в порядке? Метод `valueForKeyPath:` разбивает ваш путь и обрабатывает его слева направо. Сначала он запрашивает автомобиль о его колесах. После того как колеса оказываются в его руках, он запрашивает объект `tires` при помощи его метода `valueForKeyPath:` с аргументом, представляющим собой остаток пути, в данном случае — `"pressure"`. Метод `valueForKeyPath:` класса `NSArray` проходит по всему содержимому массива и посылает сообщение каждому его элементу, так что `NSArray` отправляет каждому своему колесу сообщение `valueForKeyPath:` с `"pressure"` в качестве пути ключей. В результате возвращаются давления в колесах, упакованные в объекты `NSNumber`. Весьма удобно!

К сожалению, в путях ключей нельзя использовать индексы, так что вы не можете, например, использовать путь `"tires[0].pressure"` для получения давления в первом колесе.

Пит-стоп

Перед тем как направиться дальше, сделаем пит-стоп¹ и добавим новый класс, `Garage`, предназначенный для хранения набора автомобилей. Весь код можно найти в папке 16.02 Car-Value-Garaging. Вот интерфейс класса `Garage`.

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

```
@class Car;
```

```
@interface Garage : NSObject {
    NSString *name;
    NSMutableArray *cars;
}
```

```
@property (readwrite, copy) NSString *name;
- (void) addCar: (Car *) car;
- (void) print;
@end // Garage
```

Здесь нет ничего нового. Имеется опережающее объявление класса `Car`, поскольку все, что нам надо знать, — это то, что данный тип объекта используется в качестве аргумента метода `-addCar:`. Переменная `name` является свойством, а конструкция `@property` говорит о том, что пользователи класса `Garage` могут получить доступ к этой переменной и изменить

¹ Заезд гонщика в гараж для смены колес, дозаправки или ремонта во время гонки. — *Примеч. пер.*

ее. Имеется также метод для вывода на экран содержимого гаража. Чтобы реализовать коллекцию автомобилей, за сценой используется изменяемый массив.

Реализация класса также проста.

```
#import "Garage.h"

@implementation Garage

@synthesize name;

- (void) addCar: (Car *) car {
    if (cars == nil) {
        cars = [[NSMutableArray alloc] init];
    }
    [cars addObject: car];
} // addCar

- (void) dealloc {
    [name release];
    [cars release];
    [super dealloc];
} // dealloc

- (void) print {
    NSLog(@"%@:", name);
    for (Car *car in cars) {
        NSLog(@"  %@", car);
    }
} // print
@end // Car
```

Как обычно, мы включаем заголовочный файл `Garage.h` и используем конструкцию `@synthesize` для методов доступа к переменной экземпляра `name`.

Метод `-addCar:` представляет собой пример отложенной инициализации массива `cars`; мы создаем его только тогда, когда это необходимо. Метод `-dealloc` освобождает память, выделенную для имени и массива, а `-print` проходит по массиву и выводит информацию об автомобилях.

Мы полностью переделываем исходный файл `Car-Value-Garage.m` по сравнению с предыдущими версиями программы. В этот раз программа создает коллекцию автомобилей и размещает их в гараже.

Сначала, как обычно, идут все необходимые директивы `#import` для используемых объектов.

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

```
#import "Car.h"
#import "Garage.h"
#import "Slant6.h"
#import "Tire.h"
```

Далее, у нас есть функция, которая создает автомобиль из набора атрибутов. Мы могли бы разработать метод класса `Car` или создать класс-фабрику, но Objective-C остается языком

программирования C, так что мы можем использовать функции. Здесь воспользуемся именно функцией — так код создания автомобиля окажется ближе к коду его применения.

```
Car *makeCar(NSString *name, NSString *make,
            NSString *model, int modelYear,
            int numberOfDoors, float mileage,
            int horsepower) {
    Car *car = [[[Car alloc] init] autorelease];
    car.name = name;
    car.make = make;
    car.model = model;
    car.modelYear = modelYear;
    car.numberOfDoors = numberOfDoors;
    car.mileage = mileage;

    Slant6 *engine = [[[Slant6 alloc] init] autorelease];
    [engine setValue: [NSNumber numberWithInt: horsepower]
     forKey: @"horsepower"];
    car.engine = engine;

    // Создание колес
    int i;
    for (i = 0; i < 4; i++) {
        Tire *tire = [[[Tire alloc] init] autorelease];
        [car setTire: tire atIndex: i];
    }

    return (car);
} // makeCar
```

Кое-что из написанного может оказаться вам незнакомо. Новый автомобиль создается и автоосвобождается в соответствии с соглашениями Сосоа, так как те, кто будут использовать данную функцию, получают автомобиль не при помощи вызовов new, сору или alloc. Затем мы устанавливаем некоторые свойства, и методика их установки отлична от кодирования "ключ/значение", поскольку мы не используем метод setValue:forKey:. После этого создаем двигатель и в этот раз применяем кодирование "ключ/значение" для установки значения его мощности, поскольку так и не сделали метод доступа для этой переменной экземпляра. Наконец, мы создаем колеса и монтируем их в автомобиль. И, конечно, возвращаем готовую продукцию.

А вот новая версия функции main().

```
int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool;
    pool = [[NSAutoreleasePool alloc] init];

    Garage *garage = [[Garage alloc] init];
    garage.name = @"Joe's Garage";

    Car *car;
```



```

car = makeCar(@ "Herbie", @ "Honda", @ "CRX",
              1984, 2, 110000, 58);
[garage addCar: car];

car = makeCar(@ "Badger", @ "Acura", @ "Integra",
              1987, 5, 217036.7, 130);
[garage addCar: car];

car = makeCar(@ "Elvis", @ "Acura", @ "Legend",
              1989, 4, 28123.4, 151);
[garage addCar: car];

car = makeCar(@ "Phoenix", @ "Pontiac", @ "Firebird",
              1969, 2, 85128.3, 345);
[garage addCar: car];

car = makeCar(@ "Streaker", @ "Pontiac",
              @ "Silver Streak", 1950, 2, 39100.0, 36);
[garage addCar: car];

car = makeCar(@ "Judge", @ "Pontiac", @ "GTO", 1969, 2,
              45132.2, 370);
[garage addCar: car];

car = makeCar(@ "Paper Car", @ "Plymouth", @ "Valiant",
              1965, 2, 76800, 105);
[garage addCar: car];

[garage print];
[garage release];

[pool release];
return (0);
} // main

```

Функция `main()` создает гараж и несколько автомобилей, которые будут в нем находиться. Наконец, она выводит информацию о гараже и освобождает его.

Запуск программы приводит к следующему выводу.

Joe's Garage:

Herbie, a 1984 Honda CRX, has 2 doors, 110000.0 miles, 58 hp and 4 tires
 Badger, a 1987 Acura Integra, has 5 doors, 217036.7 miles, 130 hp and 4 tires
 Elvis, a 1989 Acura Legend, has 4 doors, 28123.4 miles, 151 hp and 4 tires
 Phoenix, a 1969 Pontiac Firebird, has 2 doors, 85128.3 miles, 345 hp and 4 tires
 Streaker, a 1950 Pontiac Silver Streak, has 2 doors, 39100.0 miles, 36 hp and 4 tires
 Judge, a 1969 Pontiac GTO, has 2 doors, 45132.2 miles, 370 hp and 4 tires
 Paper Car, a 1965 Plymouth Valiant, has 2 doors, 76800.0 miles, 105 hp and 4 tires

Теперь у нас есть основа для обещанного дальнейшего рассмотрения возможностей кодирования "ключ/значение".

Операторы

Пути ключей могут указывать не только на значения объектов. В путях может применяться ряд операторов, выполняющих определенные действия наподобие вычисления среднего значения элементов массива или возврата минимального или максимального значения.

Например, вот как можно подсчитать количество автомобилей.

```
NSNumber *count;
count = [garage valueForKeyPath: @"cars.@count"];
NSLog(@"У нас имеется %@ автомобилей", count);
```

При выполнении этого фрагмента мы получим такую строку.

У нас имеется 7 автомобилей

Давайте разберем путь "cars.@count". Часть cars заставляет обратиться к свойству cars объекта garage, которое, как мы знаем, представляет собой NSArray. Да, мы, конечно, знаем, что на самом деле это NSMutableArray, но до тех пор, пока не собираемся что-то в нем изменять, его можно рассматривать как NSArray. Далее в строке идет @count. Символ @, как вы знаете, сигнализирует о какой-то очередной хитрости Objective-C. Так, для компилятора @"blah" является строкой, @interface — введением в класс. Ну, а @count просит механизм кодирования "ключ/значение" подсчитать количество результатов в левой части пути ключей.

Мы можем получить сумму определенных значений — например, общий пробег всей нашей автоколонны. Так, фрагмент

```
NSNumber *sum;
sum = [garage valueForKeyPath: @"cars.@sum.mileage"];
NSLog(@"Общий пробег равен %@ миль", sum);
```

при выполнении выводит такую строку.

Общий пробег равен 601320.6 миль

Как это работает? Оператор @sum разбивает путь ключей на две части. Первая часть рассматривается как путь ключей к чему-то со связью "ко многим", в нашем случае — к массиву cars. Другая часть трактуется как путь ключей, в середине которого имеется связь "ко многим" и который применяется к каждому объекту этой связи. Таким образом, сообщение mileage отправляется каждому объекту cars, а полученные результаты суммируются. Конечно, каждый из этих путей может иметь произвольную длину.

Если вы хотите найти средний пробег одного автомобиля, то для этого можно разделить суммарный пробег на количество автомобилей. Но есть способ и попроще — фрагмент

```
NSNumber *avgMileage;
avgMileage = [garage valueForKeyPath: @"cars.@avg.mileage"];
NSLog(@"Средний пробег - %.2f", [avgMileage floatValue]);
```

дает в результате такую строку.

Средний пробег — 85902.95

Просто, правда? Если не пользоваться кодированием "ключ/значение", для получения данного результата следует написать цикл, проходящий по всем автомобилям и запрашивающий пробег каждого из них. Полученные значения следует сложить вместе и разделить на количество автомобилей — в общем-то, все не так уж сложно, но достаточно многословно.

Рассмотрим использованный в этот раз путь "cars.@avg.mileage". Как и при применении оператора @sum, оператор @avg разбивает путь ключей на две части: часть до оператора (cars) представляет собой путь до связи "ко многим", часть после @avg является другим путем ключей (в нашем случае — mileage). За кулисами кодирование "ключ/значение" выполняет описанный цикл, суммирует значения и делит сумму на количество автомобилей.

Имеются и другие операторы. Так, операторы @min и @max выполняют совершенно очевидные вычисления. Так, фрагмент

```
NSNumber *min, *max;
min = [garage valueForKeyPath: @"cars.@min.mileage"];
max = [garage valueForKeyPath: @"cars.@max.mileage"];
NSLog(@"minimax: %@ / %@", min, max);
```

выводит строку minmax: 28123.4 / 217036.7.

ЗА ВСЕ НАДО ПЛАТИТЬ

Кодирование "ключ/значение" существенно упрощает работу с коллекциями. Так почему бы не использовать его везде и забыть о методах доступа и написании кода вручную? Ну, бесплатных обедов не бывает — по крайней мере в странах, считающих себя цивилизованными. Кодирование "ключ/значение" приводит к снижению производительности кода, поскольку, как минимум, требуется выполнить анализ строки и понять, чего же хочет программист. Кроме того, компилятор не в состоянии выполнить всестороннюю проверку на наличие ошибок, так что при запросе kargz.@avg.millage компилятор не знает, что вы передали некорректный путь, и вы получите сообщение об ошибке уже во время работы приложения.

Иногда приходится иметь дело с атрибутом, который может принимать значения только из небольшого множества доступных, например, как в случае атрибута производителя автомобиля. Даже если у нас миллион автомобилей, список производителей будет очень небольшим. Его можно получить при помощи пути ключей "cars.@distinctUnionOfObjects.make".

```
NSArray *manufacturers;
manufacturers = [garage valueForKeyPath:
    @"cars.@distinctUnionOfObjects.make"];
NSLog(@"Производители: %@", manufacturers);
```

Приведенный фрагмент кода даст следующий вывод.

```
Производители: (
Honda,
Plymouth,
Pontiac,
Acura
)
```

Оператор в середине данного пути ключей (@distinctUnionOfObjects) выполняет именно то, что сказано в его названии, — получает коллекцию, определяемую левой частью пути, и использует правую часть пути для каждого объекта коллекции, а затем превращает полученные значения в новую коллекцию, которую и возвращает. Часть "union" имени оператора как раз и говорит об объединении в коллекцию. Часть имени "distinct" говорит об устранении дубликатов в этой коллекции.

Имеется еще несколько операторов, которые можно использовать в путях ключей, но мы оставим их для вашего самостоятельного поиска. Последнее замечание, которое мы вынуждены сделать, — это то, что вы не можете создавать свои собственные операторы.

Пакетные изменения

Кодирование "ключ/значение" имеет пару вызовов, которые позволяют вам выполнять пакетные изменения объектов. Первым из них является dictionaryWithValuesForKeys:. Вы передаете ему массив строк. Вызов принимает ключи, использует для каждого из них метод valueForKey: и строит словарь со строками ключей и только что полученными значениями.

Выберем из гаража автомобиль и сделаем словарь некоторых из его атрибутов.

```
car = [[garage valueForKeyPath: @"cars"] lastObject];
NSArray *keys = [NSArray arrayWithObjects: @"make",
    @"model", @"modelYear", nil];
NSDictionary *carValues = [car
    dictionaryWithValuesForKeys: keys];
NSLog(@"Параметры автомобиля: %@", carValues);
```

Выполнение этого фрагмента кода дает нам некоторую информацию о "Paper Car".

```
Параметры автомобиля: {
    make = Plymouth;
    model = Valiant;
    modelYear = 1965;
}
```

Мы можем изменять эти значения, превратив наш автомобиль в совсем другой.

```
NSDictionary *newValues =
    [NSDictionary dictionaryWithObjectsAndKeys:
        @"Chevy", @"make",
        @"Nova", @"model",
        [NSNumber numberWithInt:1964], @"modelYear", nil];
[car setValuesForKeysWithDictionary: newValues];
NSLog(@"Новый автомобиль: %@", car);
```

После выполнения этого фрагмента кода получаем новый автомобиль.

```
Новый автомобиль: Paper Car, a 1964 Chevy Nova, has 2 doors, 76800.0 miles, and 4 tires.
```

Обратите внимание на то, что некоторые значения были изменены (производитель, модель, год выпуска), в то время как другие остались неизменными.

Для данной программы такой инструмент не очень полезен, но он позволяет выполнять некоторые отличные трюки в пользовательском интерфейсе.

Вы можете спросить, а что будет со значениями `nil`, например, в случае с машиной без имени — ведь словари не могут содержать значения `nil`? Вспомните главу 7, "Еще об Xcode", где для представления значения `nil` использовалась конструкция `[NSNull null]`. То же самое происходит и здесь. Для безымянного автомобиля для ключа `@ "name"` при вызове `dictionaryWithValuesForKeys` вернется значение `[NSNull null]`; в свою очередь, вы можете передать это значение при вызове `setValuesForKeysWithDictionary` для решения обратной задачи.

nil

Данное обсуждение значений `nil` приводит к интересному вопросу. Что означает `nil` в случае скалярного значения наподобие `mileage`. Это ноль? -1? π ? Попробуем посмотреть, как Cocoa отреагирует на следующую строку кода.

```
[car setValue: nil forKey: @"mileage"];
```

Но Cocoa тут же ставит нас на место.

```
'[<Car 0x105740> setNilValueForKey]: could not set nil as the value for the key mileage.'
```

Для решения проблемы перекроем метод `-setNilValueForKey` и обеспечим некоторую имеющую смысл логику. Будем считать, что пробег `nil` — это на самом деле нулевой пробег.

```
-(void) setNilValueForKey: (NSString *) key {
    if ([key isEqualToString: @"mileage"]) {
        mileage = 0;
    } else {
        [super setNilValueForKey: key];
    }
}
// setNilValueForKey
```

Обратите внимание, что при получении неожиданного значения ключа мы вызываем метод надкласса. Таким образом, если кто-то попытается использовать кодирование "ключ/значение" с неизвестным нам ключом, он получит корректные результаты. Вообще говоря, при перекрытии следует всегда вызывать метод надкласса, за исключением крайне редких ситуаций с уважительными причинами поступать иначе.

Обработка необработанного

Последняя остановка в нашем путешествии по кодированию "ключ/значение" — обработка неопределенных ключей. Если вы допустите опечатку при вводе ключа, то получите сообщение примерно такого вида.

```
'[<Car 0x105740> valueForKey:]: this class is
not key value coding-compliant for the key garbanzo.'
```

Это означает, что Сосоа не в состоянии понять, что за ключ передан, и вынужденно прекращает работу.

Если вы внимательно посмотрите на сообщение об ошибке, то увидите, что в нем упоминается метод `valueForKey:`. Как вы, вероятно, можете предположить, чтобы обработать неопределенные ключи, следует перекрыть этот метод. Можно также догадываться о существовании соответствующего метода `setValue:forUndefinedKey:` для случая изменения значения с неопределенным ключом.

Если механизм кодирования "ключ/значение" не в состоянии найти способ выполнить свою работу, он отступает и запрашивает класс о том, что ему делать. Реализация по умолчанию просто поднимает руки, как вы только что видели. Мы можем изменить такое поведение. Давайте превратим `Garage` в очень гибкий объект, способный работать с любым ключом. Начнем с добавления изменяемого словаря.

```
@interface Garage : NSObject {
    NSString *name;
    NSMutableArray *cars;
    NSMutableDictionary *stuff;
}
...
@end // Garage
```

Теперь добавим методы для работы с неопределенными ключами.

```
-(void) setValue: (id) value
forUndefinedKey: (NSString *) key {
    if (stuff == nil) {
        stuff = [[NSMutableDictionary alloc] init];
    }
    [stuff setValue: value forKey: key];
} // setValueForUndefinedKey

-(id) valueForKey: (NSString *) key {
    id value = [stuff valueForKey: key];
    return (value);
} // valueForKey
```

Мы освобождаем добавленный словарь в методе `-dealloc`.

Теперь мы можем как добавлять произвольные значения в гараж

```
[garage setValue: @"bunny" forKey: @"fluffy"];
[garage setValue: @"greeble" forKey: @"bork"];
[garage setValue: [NSNull null] forKey: @"snorgle"];
[garage setValue: nil forKey: @"gronk"];
```

так и получать их обратно.

```
NSLog(@"values are %@ %@ and %@",
    [garage valueForKey: @"fluffy"],
    [garage valueForKey: @"bork"],
```



```
[garage valueForKey: @"snorgle"],
[garage valueForKey: @"gronk"]]);
```

Этот вызов NSLog выводит на экран такую строку.

```
values are bunny greeble <null> and (null)
```

Обратите внимание на различия между `<null>` и `(null)`. `<null>` представляет собой объект `[NSNull null]`, а `(null)` — это реальное значение `nil`, полученное потому, что ключ `"gronk"` в словаре отсутствует. Обратите также внимание, что при работе со словарем `stuff` мы использовали метод кодирования "ключ/значение" `setValue:forKey:`. Применение этого метода позволяющей функции передавать значение `nil` без дополнительных проверок в коде. Метод `NSDictionary setObject:forKey:` будет жаловаться при передаче значения `nil`, в то время как метод `setValue:forKey:` при получении значения `nil` удалит соответствующий ключ из словаря.

Резюме

Кодирование "ключ/значение" на самом деле гораздо обширнее, чем мы могли рассказать в этой главе, но здесь вы должны были получить достаточную базу для самостоятельного изучения иных аспектов кодирования "ключ/значение". Вы видели примеры установки и получения значений с одним ключом, когда кодирование "ключ/значение" находило соответствующие методы доступа для выполнения вашего запроса. Если методы оставались не найдены, кодирование "ключ/значение" обращается непосредственно к объекту и его переменным экземпляра.

Вы также познакомились с путями ключей, которые представляют собой ключи, разделенные точками, указывающими путь по графу объектов. Эти пути внешне могут быть похожи на обращение к свойствам, но на самом деле их механизм совершенно иной. Вы можете использовать в путях различные операторы для выполнения дополнительных действий по вашему желанию. Наконец, мы рассмотрели некоторые методы, которые можно перекрыть для настройки поведения механизма кодирования "ключ/значение" в сложных ситуациях.

Следующей темой нашего рассмотрения будут предикаты Cocoa.

NSPredicate

Достаточно распространенная при написании программного обеспечения операция — отбор из коллекции тех объектов, которые соответствуют некоторому критерию. Вы оставляете объекты, для которых некоторое условие истинно, и отбрасываете те, которые этому условию не соответствуют.

Вы постоянно встречаетесь с этой проблемой в используемых приложениях наподобие iPhoto. Например, если вы просите iPhoto показать только рисунки с рейтингом три звезды и выше, то определяемый вами критерий звучит как “фото должно иметь рейтинг не ниже трех звезд”. Все ваши фото пропускаются через указанный фильтр. Те из них, у которых рейтинг три и более звезды, проходят через фильтр, а остальные — нет. После этого iPhoto показывает все рисунки, отвечающие вашему критерию отбора.

Аналогично в iTunes имеется поле поиска. Критерий, по которому вы выполняете поиск, может быть, например, таким — “исполнителем должен быть Иосиф Кобзон или Муслим Магомаев”. При этом все песни в исполнении Пугачевой окажутся скрытыми с глаз долой.

Cocoa предоставляет класс NSPredicate, который позволяет вам определять такие фильтры отбора. Вы можете создать объекты NSPredicate, которые в точности описывают критерий отбора, и пропускать все объекты через такой предикат в поисках соответствующих ему. Слово “предикат” имеет здесь строгий математический смысл — функция, возвращающая булево значение.

Класс NSPredicate представляет собой средство Cocoa для описания запросов, подобных используемым при работе с базами данных. Вы можете использовать объекты NSPredicate с API баз данных наподобие Core Data и Spotlight (впрочем, здесь мы не будем рассматривать данный вопрос). Вы можете рассматривать NSPredicate как еще одно средство косвенности. Можно использовать объект предиката вместо явного задания в коде вопроса “Достаточно ли глуп данный Иванушка, чтоб быть царевичем?” Благодаря возможности смены предикатов можно использовать для отсеивания данных один и тот же код, в котором не будет

жестко закодирован критерий отбора. Это еще одно применение принципа открытости-закрытости из главы 3, "Введение в объектно-ориентированное программирование".

Создание предиката

Перед тем как использовать один из объектов NSPredicate с одним из ваших объектов, вы должны создать его, что можно сделать двумя основными способами. Один из них включает создание множества объектов и их сборку. Это требует большого количества кода и удобно при построении пользовательского интерфейса для конкретных поисков. Другой способ заключается в помещении строк запроса в ваш код. Это проще для начинающих, так что в этой книге мы будем иметь дело именно с ним.

Нам не удастся избежать нашего старого проекта CarParts — примеры в этой главе основаны на построенном в предыдущей главе гараже с автомобилями. Соответствующие исходные тексты находятся в папке 17.01 Car-Part-Predicate.

Для начала рассмотрим один автомобиль.

```
Car *car;
car = makeCar(@"Herbie", @"Honda", @"CRX",
             1984, 2, 110000, 58);
[garage addCar: car];
```

Вспомните, что мы написали функцию makeCar для создания автомобиля и настройки его двигателя и колес. В данном случае мы создаем автомобиль Herbie, двухдверную модель 1984 Honda CRX с двигателем мощностью 58 л.с. и пробегом 110 000 миль.

Теперь создадим предикат.

```
NSPredicate *predicate;
predicate = [NSPredicate
             predicateWithFormat:@"name == 'Herbie'"];
```

Давайте разберем этот код. Переменная predicate представляет собой один из наших обычных указателей на объекты Objective-C, указывающий на объект NSPredicate. Для создания предиката применяем метод +predicateWithFormat: класса NSPredicate. Этот метод в качестве аргумента получает строку и строит за сценой дерево объектов, которое будет использоваться для вычисления предиката.

Имя predicateWithFormat звучит почти так же, как и stringWithFormat, метод, предоставляемый классом NSString и обеспечивающий вставку информации с помощью спецификаторов формата в стиле функции printf. Как увидите позже, то же самое позволяет и метод predicateWithFormat. Все же хорошо, что Cocoa придерживается последовательных схем именования!

Эта строка предиката похожа на выражение C. В левой ее части находится путь ключей, name. Затем идет оператор равенства == и строка в кавычках в правой части. Если блок текста в строке предиката не взят в кавычки, он рассматривается как путь ключей; в противном случае он рассматривается как строковый литерал. Вы можете использовать как двойные, так и одинарные кавычки, — главное, чтобы они были сбалансированы. Обычно используются одинарные кавычки, так как в противном случае каждый символ двойных кавычек в строке должен быть заменен соответствующей управляющей последовательностью.

Вычисление предиката

Итак, у нас есть предикат. Что теперь? Мы применяем его к объекту.

```
BOOL match = [predicate evaluateWithObject: car];
NSLog(@"%s", (match) ? "YES" : "NO");
```

Метод `-evaluateWithObject:` требует от объекта-получателя (предиката) вычислить самого себя для данного объекта. В нашем случае это автомобиль, полученный путем применения `valueForKeyPath:` с `name` в качестве пути ключей. Затем выполняется сравнение имени на равенство со строкой "Herbie". Если они равны, `-evaluateWithObject:` возвращает YES, в противном случае — NO. Функция `NSLog` использует для преобразования числового значения `BOOL` в удобочитаемую строку тернарный оператор.

Вот еще один предикат.

```
predicate = [NSPredicate predicateWithFormat:
    @"engine.horsepower > 150"];
match = [predicate evaluateWithObject: car];
```

Строка предиката содержит путь ключей в левой части. Этот путь ключей входит в автомобиль, находит двигатель, а затем его мощность. Затем это значение сравнивается со 150.

После вычисления этого предиката для автомобиля Herbie, переменная `match` получает значение NO, поскольку мощность Herbie (58 л.с.) не превосходит 150.

Проверка объекта на соответствие определенному предикату — это хорошо и интересно, но все становится еще интереснее при наличии коллекции объектов. Скажем, мы хотим выяснить, какие автомобили в нашем гараже самые мощные. Мы можем циклически обойти автомобили и выполнить их проверку на соответствие предикату.

```
NSArray *cars = [garage cars];
for (Car *car in [garage cars]) {
    if ([predicate evaluateWithObject: car]) {
        NSLog(@"%@", car.name);
    }
}
```

Мы получаем автомобили, находящиеся в гараже, циклически обходим их и вычисляем предикат для каждого из них. Приведенный фрагмент кода выводит автомобили с наибольшей мощностью.

```
Elvis
Phoenix
Judge
```

Похоже на правду? Перед тем как идти дальше, убедимся, что нам понятен использованный синтаксис. Взгляните на имя автомобиля в вызове `NSLog`. Здесь использован синтаксис Objective-C 2.0 с точкой, который эквивалентен вызову `[car name];`. Тут все понятно. Строка предиката — `"engine.horsepower > 150"`. В ней `engine.horsepower` представляет собой путь ключей, который, по сути, и выполняет всю работу за сценой.

Фильтры

Одно из главных качеств настоящего программиста — лень. Так что с точки зрения программиста было бы хорошо, если бы можно было обойтись без написания цикла `for` и конструкции `if`. Это только пара строк, но нуль строк — это еще меньше. К счастью, в Cocoa имеется пара категорий, добавляющих в классы коллекций методы фильтрации при помощи предикатов.

Метод `-filteredArrayUsingPredicate`: представляет собой метод категории для `NSArray`, выполняющий проход по содержимому массива, вычисляющий предикат для каждого из объектов массива и накапливающий объекты, вычисление для которых дает значение YES, в новый возвращаемый массив.

```
NSArray *results;
results = [cars filteredArrayUsingPredicate: predicate];
NSLog(@"%@", results);
```

Этот фрагмент дает следующий результат.

```
(
Elvis, a 1989 Acura Legend, has 4 doors, 28123.4 miles, 151 hp and 4 tires,
Phoenix, a 1969 Pontiac Firebird, has 2 doors, 85128.3 miles, 345 hp and 4 tires,
Judge, a 1969 Pontiac GT0, has 2 doors, 45132.2 miles, 370 hp and 4 tires
)
```

Да, этот результат не идентичен предыдущему. Это — массив автомобилей; в предыдущем же примере у нас были имена. Давайте воспользуемся кодированием "ключ/значение" для выделения имен. Вспомним, что когда массиву посылается сообщение `valueForKey:`, ключ применяется ко всем элементам массива.

```
NSArray *names;
names = [results valueForKey:@"name"];
```

Если мы выведем имена, то получим следующее.

```
(
Elvis,
Phoenix,
Judge
)
```

Скажем, у вас есть изменяемый массив и вы хотите убрать все лишние элементы. У класса `NSMutableArray` есть метод `-filterUsingPredicate`, который, что достаточно удобно, способен выполнить такой отбор.

```
NSMutableArray *carsCopy = [cars mutableCopy];
[carsCopy filterUsingPredicate: predicate];
```


Если теперь вывести carsCopy, то вы увидите, что в этом массиве три автомобиля, которые мы видели ранее.

Вы, конечно же, можете использовать -filteredArrayUsingPredicate: с тем же классом NSMutableArray для создания нового (неизменяемого) массива, поскольку NSMutableArray является подклассом NSArray. Аналогичные вызовы имеются и у NSSet.

Как мы уже упоминали при рассмотрении кодирования "ключ/значение", применение предикатов действительно удобно, но это не намного быстрее, чем написание всего кода самостоятельно. Это связано с тем, что не имеется способа избежать циклического прохода по всем автомобилям и выполнения определенных действий с ними. Большей частью этот цикл — не такое уж важное дело, так как современные компьютеры очень быстрые. Если у вас будут проблемы со скоростью работы — напишите код и исследуйте его производительность с помощью инструментария Apple. Программисты iPhone должны уделять вопросам эффективности существенно большее внимание.

Спецификаторы формата

Опытные программисты знают, что "прошивание" данных в программе — не слишком хорошая идея. Что если надо узнать, у каких автомобилей мощность выше 200 л.с., а позже потребуется найти автомобили с мощностью более 50 л.с.? У нас могут иметься строки предикатов наподобие "engine.horsepower > 200" и "engine.horsepower > 50", но необходимость recompilation программы с разными строками отбрасывает нас назад на позиции, где мы были до главы 3, "Введение в объектно-ориентированное программирование".

Можно поместить различную информацию в строку предиката двумя способами — с помощью спецификаторов формата и имен переменных. Начнем со спецификаторов формата. Так, в строке можно разместить числовые значения при помощи уже знакомых вам спецификаторов %d и %f.

predicate =

```
{NSPredicate predicateWithFormat:
 @"engine.horsepower > %d", 50};
```

Конечно, вместо применения жестко закодированного числа 50 можно воспользоваться значением, введенным пользователем или полученным при помощи иного внешнего механизма.

В дополнение к обычным спецификаторам printf можно также воспользоваться %@ для вставки строкового значения. %@ рассматривается так же, как и обычная строка в кавычках.

predicate = [NSPredicate predicateWithFormat:

```
@"name == %@", @"Herbie"];
```

Обратите внимание на то, что %@ в форматной строке в кавычки не берется. Если вы возьмете %@ в кавычки, например "name == '%@'", то символы % и @ будут рассматриваться как часть строки предиката.

Строки `NSPredicate` позволяют также использовать спецификатор `%K` для указания пути ключей. Приведенный далее предикат таков же, как и предыдущие, использовавшие строку `name == 'Herbie'`.

```
predicate =
[NSPredicate predicateWithFormat:
@"%K == %@", @"name", @"Herbie"];
```

Применение спецификаторов формата — первый способ обеспечения гибкости предикатов. Второй способ заключается в помещении в строку имен переменных, подобно переменным среды.

```
NSPredicate *predicateTemplate =
[NSPredicate predicateWithFormat:@"name == $NAME"];
```

Теперь, когда у нас есть предикат с переменной в нем, можем создавать новые специализированные предикаты с помощью вызова `predicateWithSubstitutionVariables`. Вы создаете словарь пар «ключ/значение», в котором ключом служит имя переменной (без знака доллара), а значением — то, что должно быть подставлено в предикат.

```
NSDictionary *varDict;
varDict = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Herbie", @"NAME", nil];
```

Здесь строка `"Herbie"` использована в качестве значения для ключа `"NAME"`. Теперь мы можем создать новый предикат.

```
predicate = [predicateTemplate
    predicateWithSubstitutionVariables: varDict];
```

Этот предикат работает в точности так же, как и остальные, с которыми мы уже имели дело.

В качестве значений переменных можно использовать различные виды объектов, в частности, `NSNumber`.

```
predicateTemplate =
[NSPredicate predicateWithFormat:
@"engine.horsepower > $POWER"];
varDict = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt: 150],
    @"POWER", nil];
predicate =
[predicateTemplate
    predicateWithSubstitutionVariables: varDict];
```

Здесь создается предикат, возвращающий значение `YES` для автомобилей, мощность двигателя которых превышает 150 л.с.

В дополнение к `NSNumber` и `NSString` можно использовать `[NSNull null]` для получения значений `nil`; можно даже использовать массивы, как вы увидите немного позже в этой главе. Заметим, что для путей ключей применять имена переменных `$VARIABLE` нельзя — только

значения. Если вы хотите программно изменять пути ключей при использовании форматных строк предиката, вам нужен спецификатор формата %K.

Механизм предикатов не выполняет проверку типов, так что можно случайно вставить строку там, где требуется числовое значение. Такая ошибка может привести к сообщению об ошибке времени выполнения программы или, что еще хуже, к неверному ее поведению.

Операторы

Форматная строка `NSPredicate` позволяет использовать множество различных операторов. Здесь мы бегло рассмотрим большую их часть; остальные вы можете найти в оперативной документации Apple.

Операторы сравнения и логические операторы

Синтаксис строк предикатов поддерживает некоторые из хорошо знакомых операторов C, такие как операторы равенства `==` и `=`.

Операторы неравенства включают:

`>` Больше, чем

`>=` и `=>` Больше или равно

`<` Меньше, чем

`<=` и `=<` Меньше или равно

`!=` и `<>` Не равно

Синтаксис поддерживает выражения со скобками, логические операторы `AND`, `OR` и `NOT`, а также их C-эквиваленты `&&`, `||` и `!`.

Рассмотрим конкретный пример. Вот как можно отфильтровать самые мощные и самые слабые автомобили, оставив только автомобили средней мощности.

```
predicate = [NSPredicate predicateWithFormat:
```

```
    @"(engine.horsepower > 50) AND
```

```
    (engine.horsepower < 200)"];
```

```
results = [cars filteredArrayUsingPredicate: predicate];
```

```
NSLog(@"%@@", results);
```

Применяя этот код к нашему гаражу, получим следующие результаты.

```
Herbie, a 1984 Honda CRX, has 2 doors, 34000.0 miles, 58 hp and 4 tires,
Badger, a 1987 Acura Integra, has 5 doors, 217036.7 miles, 130 hp and 4 tires,
Elvis, a 1989 Acura Legend, has 4 doors, 28123.4 miles, 151 hp and 4 tires,
Paper Car, a 1965 Plymouth Valiant, has 2 doors, 76800.0 miles, 105 hp and 4 tires
```

Операторы в строках предикатов нечувствительны к регистру. Вы можете использовать `AND`, `AND`, `AND` или `and` — как вам больше нравится. Мы будем использовать прописные буквы, но вы не обязаны поступать именно так.

Операторы неравенства работают как с числовыми значениями, так и со строками. Чтобы увидеть все автомобили из начала алфавита, воспользуйтесь следующим предикатом.

```
predicate = [NSPredicate predicateWithFormat:
    @"name < 'Newton'"];
results = [cars filteredArrayUsingPredicate: predicate];
NSLog(@"%@ ", [results valueForKey: @"name"]);
```

Вот результат работы этого фрагмента.

```
(
Herbie,
Badger,
Elvis,
Judge
)
```

Массивы

Строка предиката типа "(engine.horsepower > 50) OR (engine.horsepower < 200)" — весьма распространенный шаблон. Мы ищем значения мощности между двумя значениями. Правда, было бы неплохо, если можно было бы использовать некий оператор, который обеспечивал бы поиск между двумя значениями? Оказывается, такой оператор есть.

```
predicate = [NSPredicate predicateWithFormat:
    @"engine.horsepower BETWEEN { 50, 200 }"];
```

Фигурные скобки обозначают массив, а оператор BETWEEN рассматривает первый элемент массива как нижнюю границу, а второй — как верхнюю.

Вы можете выполнить вставку в собственный массив NSArray с помощью спецификатора формата %@.

```
NSArray *betweens =
    [NSArray arrayWithObjects:
        [NSNumber numberWithInt: 50],
        [NSNumber numberWithInt: 200], nil];
predicate =
    [NSPredicate predicateWithFormat:
        @"engine.horsepower BETWEEN %@, betweens];
```

Вы можете использовать и переменные.

```
predicateTemplate =
    [NSPredicate predicateWithFormat:
        @"engine.horsepower BETWEEN $POWERS"];
varDict =
    [NSDictionary dictionaryWithObjectsAndKeys:
        betweens, @"POWERS", nil];
predicate =
    [predicateTemplate
    predicateWithSubstitutionVariables: varDict];
```

Массивы нужны не только для того, чтобы указывать конечные точки интервала. Можно использовать оператор IN для проверки наличия конкретного значения в массиве. Это должно быть знакомо для программистов с опытом работы с SQL.

```
predicate = [NSPredicate predicateWithFormat:
    @"name IN { 'Herbie', 'Snugs',
        'Badger', 'Flap' }"];
```

У нас есть автомобили Herbie и Badger, так что они должны остаться после фильтрации.

```
results = [cars filteredArrayUsingPredicate: predicate];
NSLog(@"%@", [results valueForKey: @"name"]);
```

В этом легко убедиться, взглянув на результат работы программы.

```
(
Herbie,
Badger
)
```

SELF

Иногда требуется применять предикаты к простым значениям наподобие добрых старых строк, а не к объектам, с которыми можно работать при помощи путей ключей. Скажем, у нас есть массив имен автомобилей и мы хотим применить к ним тот же фильтр, что и раньше. Что мы должны использовать вместо name (поскольку NSString не в состоянии корректно отреагировать на запрос name)?

Нас спасет ключевое слово SELF! SELF означает объект, для которого вычисляется предикат. Фактически мы можем выразить все наши пути ключей в предикатах относительно SELF. Рассматриваемый ранее предикат в точности соответствует следующему.

```
predicate = [NSPredicate predicateWithFormat:
    @"SELF.name IN { 'Herbie', 'Snugs',
        'Badger', 'Flap' }"];
```

Теперь вернемся к массиву строк. Как нам выяснить, находится ли строка в данном массиве строк?

Для начала нам нужен массив имен автомобилей. Поскольку мы уже закопались в проект CarParts по уши, можем получить его при помощи кодирования "ключ/значение".

```
NSArray *names = [cars valueForKey: @"name"];
```

Полученный массив строк содержит все имена наших машин. Теперь займемся предикатом.

```
predicate = [NSPredicate predicateWithFormat:
    @"SELF IN { 'Herbie', 'Snugs',
        'Badger', 'Flap' }"];
```

и вычислим его.

```
results = [names filteredArrayUsingPredicate: predicate];
```

Если мы рассмотрим полученный массив `results`, то увидим в нем те же имена, что и ранее: `Herbie` и `Badger`.

А теперь загадка. Каким будет результат выполнения следующего фрагмента?

```
NSArray *names1 =
    [NSArray arrayWithObjects:
        @"Herbie", @"Badger", @"Judge", @"Elvis", nil];
NSArray *names2 =
    [NSArray arrayWithObjects:
        @"Judge", @"Paper Car", @"Badger", @"Finto", nil];
predicate =
    [NSPredicate predicateWithFormat:
        @"SELF IN %@", names1];
results =
    [names2 filteredArrayUsingPredicate: predicate];
NSLog(@"%@", results);
```

Вот ответ.

```
(
    Judge,
    Badger
)
```

Это очевидный способ вычисления пересечения двух множеств. Как это работает? Предикат включает содержимое первого массива, так что он имеет следующий вид.

```
SELF IN {"Herbie", "Badger", "Judge", "Elvis"}
```

После этого при помощи данного предиката выполняется фильтрация второго массива. Строки, находящиеся в обоих массивах, будут находиться в `names2` и при вычислении конструкции `SELF IN` дадут результат `YES`, так что они попадут в результирующий массив. Объекты, которые имеются только во втором массиве, будут отброшены, так как они не соответствуют ни одной строке в предикате. Строки, которые имеются только в первом массиве, также будут отброшены, поскольку будут дожидаться сравнения, которое так никогда и не произойдет.

Операции со строками

Операторы сравнения, с которыми мы познакомились выше, успешно работают со строками. Но есть еще несколько операций, предназначенных для работы только со строками.

- **BEGINSWITH:** проверяет, начинается ли строка с некоторой другой
- **ENDSWITH:** проверяет, заканчивается ли строка некоторой другой
- **CONTAINS:** проверяет, содержит ли строка некоторую другую

Применение этих операторов позволяет запросить "name BEGINSWITH 'Bad'" для получения "Badger", "name ENDSWITH 'vis'" — для получения "Elvis" и "name CONTAINS 'udg'" — для получения "Judge".

Что произойдет при применении предиката "name BEGINSWITH 'HERB'"? Он не найдет "Herbie", поскольку описанные проверки чувствительны к регистру. Точно так же не сработает и предикат "name BEGINSWITH 'Hérb'", так как здесь использован символ "е" с ударением. Для ослабления правил сравнения можно добавить к указанным операторам [c], [d] или [cd]: c означает нечувствительность к регистру, d — нечувствительность к диакритическим знакам (в частности, к ударению), [cd] обеспечивает оба режима одновременно. Обычно стоит использовать [cd], если только у вас нет веских причин поступать иначе. Мало ли, вдруг у пользователя залипла клавиша и он вводит все ПРОПИСНЫМИ БУКВАМИ?

Приведенный далее предикат обнаружит строку "Herbie".

"name BEGINSWITH[cd] 'HERB'"

LIKE

Иногда проверки соответствия начала или конца (или середины) строки другой строке недостаточно. Для этих ситуаций форматная строка предиката включает оператор LIKE. В этом операторе вопросительный знак соответствует одному символу, а звездочка — любому количеству символов. Для программистов, работавших с SQL или языком оболочки Unix, это не в новинку.

Строка предиката "name LIKE '*er*'" находит те имена, в середине которых имеется сочетание "er". Это эквивалент оператора CONTAINS.

Строка предиката "name LIKE '???er*'" найдет имя "Paper Car", так как оно состоит из трех символов, за которыми следует "er" и произвольное количество любых символов. Однако шаблон не соответствует имени "Badger", поскольку перед "er" находится четыре символа.

К LIKE также можно добавить [cd] для нечувствительности к регистру и диакритическим символам.

Если вы знакомы с регулярными выражениями, то можете использовать оператор MATCHES. Передайте ему регулярное выражение, и предикат будет проверять строки на соответствие ему.

ВЫРАЗИТЕ СЕБЯ

Регулярные выражения представляют собой очень мощное и компактное средство описания логики проверки соответствия строк шаблону. Регулярные выражения могут становиться очень сложными и запутанными, и этой теме посвящены целые книги. Регулярные выражения в NSPredicate используют синтаксис ICU (International Components for Unicode, международные компоненты Unicode), подробное описание которого вы легко найдете в Интернете.

У мощи регулярных выражений имеется оборотная сторона — высокие вычислительные затраты. Если в вашем предикате имеются более простые операции — начните с них, прежде чем переходить к MATCHES. Это должно обеспечить более высокую эффективность вашей программы.

Это все

На этом наша книга, по сути, завершена. Она охватывает достаточно большой и разнообразный материал — от концепции косвенности до объектно-ориентированного программирования, кодирования "ключ/значение" и предикатов.

Поздравляем вас с окончанием книги! Теперь вы официально готовы к следующим этапам вашей карьеры Мас-программиста. Спасибо, что были с нами.

Переход к Objective-C с других языков

Множество программистов переходят к работе с Objective-C и Cocoa с опытом работы на других языках и должны затратить много времени и усилий на изучение Objective-C — поскольку он существенно отличается от большинства других языков программирования. Новички в Objective-C часто полагают, что Objective-C — плохой язык программирования, поскольку в нем нет возможностей X, Y, Z, которые есть в их Любимом Языке. Впрочем, в некоторых аспектах Objective-C весьма странный язык.

Совет, который можно дать новичкам в Objective-C и Cocoa, — даже тем, у кого за плечами годы опыта работы с другими языками и платформами, — отбросить предвзятое мнение о том, как что должно работать, и принять Objective-C, Cocoa и Xcode таковыми, какими они есть. Работайте с небольшим количеством книг и учебников. После того как получите определенный опыт работы с Objective-C, сможете разобраться, какие методы и подходы из других языков программирования применимы к Cocoa и Objective-C, а какие — нет. Идеального языка программирования не существует, как не существует и идеального инструмента для любых работ. Лучшее, что вы можете сделать, — это изучить язык программирования в достаточной степени, чтобы со знанием дела решать, какой язык и инструментарий в наибольшей степени отвечает стоящим перед вами задачам.

В этом приложении мы представим некоторую информацию, которая должна помочь вашему переходу к Objective-C с других языков программирования.

C

Objective-C — просто старый добрый C с некоторыми дополнительными возможностями для объектной ориентированности. В основном, материал данной книги посвящен именно этим дополнительным возможностям, так что не будем повторяться. Тем не менее имеется кое-что, о чем следует упомянуть.

Помните, что программисты Objective-C имеют доступ ко всем возможностям C, в частности, к стандартной библиотеке C. Например, вы можете использовать функции `malloc()` и `free()` для работы с динамической памятью или `open()` и `fgets()` — для работы с файлами.

Иногда в почтовых рассылках Сосоа встречается вопрос: "Я вызываю библиотеку ANSI C, где для работы используется обратный вызов. Как я могу в этой ситуации использовать вызов метода?" Этот вопрос возникает и при использовании низкоуровневых схем Apple типа Core Foundation и Core Graphics.

Краткий ответ — никак. Обратный вызов работает только с функциями C с сигнатурами, которые требуются библиотекой. Функции, реализующие методы Objective-C, требуют передачи дополнительных аргументов, в частности, объекта-получателя, так что они никогда не соответствуют требуемым сигнатурам.

Большинство библиотек, ориентированных на применение обратных вызовов, позволяют вам передавать некоторые пользовательские данные или указатель. Вы передаете библиотеке некоторый значащий указатель при регистрации обратного вызова, и библиотека передает его вам назад при вызове вашей функции.

Все объекты Objective-C выделяются динамически, так что в качестве этого контекстного указателя можно вполне безопасно использовать адрес объекта. Вам не надо беспокоиться об объектах, находящихся в стеке ниже вас. В функции обратного вызова вы преобразуете контекстный указатель в тип вашего объекта и посылаете ему сообщения.

Пусть, например, вы работаете с воображаемой C-библиотекой для анализа XML и передаете данные классу `TreeWalker` в виде анализируемого XML-файла. Создаем новый объект `TreeWalker`.

```
TreeWalker *walker = [[TreeWalker alloc] init];
```

Затем создаем XML-анализатор.

```
XMLParser parser;  
parser = XMLParserLoadFile ("/tmp/badgers.xml");
```

Теперь назначаем для анализатора функцию обратного вызова (функцию C) и используем в качестве контекста объект `TreeWalker`.

```
XMLSetParserCallback (parser, elementCallback, walker);
```

После анализа XML-файла вызывается функция обратного вызова.

```
void elementCallback(XMLParser *parser,  
                    XMLElement *element,  
                    userData *context)  
{  
    TreeWalker *walker = (TreeWalker *) context;
```

```
[walker handleNewElement: element
    inParser: parser];
} // someElementCallback
```

Как видите, контекстный указатель преобразуется в указатель на объект, которому и посылается сообщение.

C++

Язык C++ обладает массой возможностей, которых нет в Objective-C: множественное наследование; пространства имен, перегрузка операторов, шаблоны, переменные классов, абстрактные классы, STL (Standard Template Library, стандартная библиотека шаблонов) и т.д. Если вам их не хватает — Objective-C в состоянии их заменить или, как минимум, имитировать.

Например, вы можете использовать категории и протоколы как разновидность множественного наследования или для реализации абстрактных базовых классов. Одно из распространенных применений множественного наследования заключается в обеспечении интерфейса, с тем чтобы другой код мог вызывать методы вашего объекта. Категории и протоколы идеальны для этой ситуации. Вы можете использовать протоколы для обеспечения чисто абстрактных базовых классов.

Категории и протоколы не могут вам помочь в случае, когда множественное наследование применяется для привнесения дополнительных переменных экземпляра (в C++ они называются *переменными-членами*). Чтобы добиться этого, можно применить композицию для включения объекта в другой объект, а затем использовать соответствующие методы для перенаправления сообщений второму объекту (распространенная методика в Java). Имитировать множественное наследование можно также путем перекрытия метода `forwardInvocation:`. Этот метод вызывается, если объект получает сообщение, которое не умеет обработать. При помощи изучения объекта `NSInvocation` вы можете выяснить, следует ли передать сообщение вашему "множественно наследованному" объекту, а затем при необходимости переслать его. Эта методика предохраняет вас от написания большого количества специализированных пересылающих сообщение методов, но она существенно менее производительна, чем реальное множественное наследование, а ее реализация — сплошная головная боль.

Прочие соглашения в состоянии заменить еще больше возможностей C++ — как, например, специальные методы-пустышки, вызывающие `abort` для абстрактных классов, которые не должны инстанцироваться. Некоторые замены существенно более убоги по сравнению с настоящими возможностями C++ — как, например, использование префиксов имен вместо пространств имен.

Таблица виртуальных функций C++ и динамическая диспетчеризация Objective-C

Одно из наибольших отличий C++ и Objective-C заключается в механизме диспетчеризации методов (или *функций-членов*, как их называют в C++). C++ использует механизм таблиц виртуальных функций (vtable) для определения того, какой код должен работать при вызове виртуальной функции.

Каждый объект C++ можно рассматривать как имеющий указатель на массив указателей на функции. Когда компилятор видит, что код должен вызвать виртуальную функцию, он вычисляет смещение от начала таблицы и генерирует машинный код для получения указателя на функцию по данному смещению и вызова функции посредством полученного указателя. Этот процесс требует, чтобы в процессе компиляции компилятор знал тип объекта для вычисления корректного смещения в таблице. Это очень быстрый вид диспетчеризации, требующий всего лишь пары операций с указателями и одного чтения для получения окончательного указателя на функцию.

Objective-C, как описано в главе 3, "Введение в объектно-ориентированное программирование", использует функцию времени выполнения для поиска выполняемого кода. Эта методика может быть в несколько раз медленнее, чем способ C++.

Скорость и безопасность — вот цена, которую Objective-C платит за гибкость и удобство. В модели C++ диспетчеризация функций-членов быстра и, к тому же, очень безопасна, так как компилятор и компоновщик гарантируют, что используемый объект в состоянии выполнить указанный метод. Но технология C++ может быть негибкой, так как вы не в состоянии изменить тип используемого объекта. Для того чтобы позволить объектам различных классов реагировать на одно и то же сообщение, следует использовать наследование.

Компилятор C++ не хранит огромное количество информации об объекте, такой как цепочка наследования, составляющие члены и т.д. Во время выполнения программы возможность обобщенной работы с объектами ограничена. Наибольшее, что можно сделать во время работы программы, — это выполнить динамическое приведение типа, которое выясняет, является ли тип объекта определенным подклассом другого объекта.

Иерархия наследования C++ не может быть изменена во время выполнения программы. После того как программа скомпилирована и скомпонована, она напоминает монолит. Динамическая загрузка библиотек C++ зачастую проблематична, в частности, из-за сложностей системы имен C++.

В Objective-C достаточно наличия реализации метода у объекта, чтобы он был вызываем. Это позволяет произвольным объектам становиться источниками данных и/или делегатами для других объектов. Отсутствие множественного наследования может оказаться неудобным, но это компенсируется возможностью посылать любые сообщения любым объектам, не беспокоясь о вопросах наследования.

Конечно, эта возможность отправлять любое сообщение любому объекту делает Objective-C менее безопасным, по сравнению с C++, с точки зрения типов. Вы можете получить ошибку времени выполнения, если объект, которому послано сообщение, не в состоянии его обработать. В C++ нет безопасных с точки зрения типов контейнеров. В контейнер может быть помещен любой объект.

Objective-C поддерживает большое количество метаданных класса, так что вы можете выяснить, в состоянии ли объект отвечать на определенное сообщение. Это весьма распространенная практика для объектов, имеющих источники данных или делегатов. Можно избежать ряда возможных ошибок времени выполнения, если предварительно выполнять проверку, в состоянии ли делегат ответить на сообщение. Можно также воспользоваться категориями для добавления методов к другим классам.

Благодаря упомянутым метаданным, упрощается анализ классов программы. Вы можете определить наличие переменных экземпляра, схему их размещения в структуре объекта, а также методы, определенные классом. Метаданные остаются на месте даже при удалении из программы отладочной информации. Если у вас имеется конфиденциальный алгоритм, его лучше реализовать на C++ или, как минимум, использовать для его реализации запутывающие имена (например, не стоит именовать метод или класс как `SerialNumberVerifier`).

В Objective-C можно посылать сообщения нулевым объектам. Нет никакой необходимости сравнивать получателя с нулем: сообщения для `nil` просто не выполняют никаких действий. Возвращаемые значения сообщений, посланных нулевым объектам, зависят от возвращаемого типа метода. Если метод возвращает указатель (например, указатель на объект), то возвращаемое значение будет равно `nil` (так что можно безопасно организовывать цепочки сообщений для нулевых объектов). Если метод возвращает `int`, это будет нуль. В случае возврата числа с плавающей точкой или структуры результат не определен. Благодаря такой особенности языка программирования, вы можете использовать нулевые объекты и сэкономить на проверках указателей. С другой стороны, эта особенность может маскировать ошибки и усложнять отладку программ.

Все объекты Objective-C выделяются динамически. В языке программирования нет стековых объектов, как нет и автоматического создания и уничтожения временных объектов или автоматического приведения типов классов. В этом, кстати, одна из причин того, что малые легковесные типы (наподобие `NSPoint` и `NSRange`) являются структурами, а не классами.

Наконец, Objective-C — язык очень свободный. В то время как в C++ имеются спецификаторы `public`, `protected` и `private` для функций-членов и членов-данных, в Objective-C имеется только определенная базовая поддержка защищенных переменных экземпляра (которую очень легко обойти) и нет никакой защиты для функций-членов. Любой, кто знает имя метода, может послать сообщение объекту. Более того, Objective-C позволяет узнать все методы, поддерживаемые данным объектом. Методы могут быть вызваны, даже если о них нет упоминания в заголовочном файле и нет надежного способа определить, какой объект вызывает метод, поскольку сообщения могут посылаться и из функций C.

Как вы видели, не обязательно повторно объявлять методы, перекрываемые в подклассах. Имеются две школы по этому поводу. Одни говорят, что повторное объявление предоставляет информацию для читающего программу о том, какие изменения сделаны в классе по сравнению с его надклассом, в то время как другие говорят, что это всего лишь детали реализации, которые не должны интересовать пользователей класса и не стоят полной перекомпиляции всех зависимых классов при новом перекрытии метода.

В Objective-C нет переменных класса. Их можно имитировать при помощи глобальных переменных в области видимости файла, предоставляя методы доступа к ним. Пример объявления класса может выглядеть следующим образом.

```
@interface Blarg : NSObject
{
}
+ (int) classVar;
+ (void) setClassVar: (int) cv;
@end // Blarg
```


Реализация класса при этом имеет следующий вид.

```
#import "Blarg.h"
static int g_cvar;
@implementation Blarg
+ (int) classVar
{
    return (g_cvar);
} // classVar

+ (void) setClassVar: (int) cv
{
    g_cvar = cv;
} // setClassVar
@end // Blarg
```

Иерархия объектов Cocoa имеет общий класс-предок NSObject. При создании нового класса вы практически всегда создаете подкласс NSObject или существующего класса Cocoa. Иерархии объектов C++ обычно представляют собой несколько деревьев с различными корнями.

Objective-C++

Имеется способ получить лучшее из двух миров. Компилятор GCC, поставляемый с Xcode, поддерживает гибридный язык Objective-C++. Этот компилятор позволяет свободно комбинировать код C++ и Objective-C, при наличии небольших ограничений. При необходимости можно обеспечить безопасность типов и низкоуровневую производительность C++ и при этом использовать динамическую природу Objective-C и инструментарий Cocoa там, где это имеет смысл.

Обычный сценарий разработки состоит в размещении всей центральной логики приложения в переносимой библиотеке C++ (если вы создаете кроссплатформенное приложение), а пользовательский интерфейс создается с применением инструментария для конкретной платформы. Objective-C++ идеально подходит для этого стиля разработки. Вы получаете производительность и безопасность типов C++, а пользователи — приложения, созданные при помощи "родного" инструментария, полностью использующие возможности конкретной платформы.

Чтобы компилятор рассматривал ваш код как код Objective-C++, используйте расширение файла .mm. Применяется также расширение .M, однако HFS+ — файловая система Mac — нечувствительна к регистру (хотя и сохраняет регистр символов в именах), так что лучше всего избегать любой зависимости от регистра символов.

Иерархии объектов Objective-C и C++ не могут смешиваться, как материя и антиматерия. Так что у вас не может быть класса C++, унаследованного от NSObject, или класса Objective-C, унаследованного от std::string.

Вы можете размещать указатели на объекты Objective-C в объектах C++. Поскольку все объекты Objective-C создаются динамически, у вас не может быть завершенных объектов, внедренных в класс или объявленных в стеке. Вы должны вызывать alloc и init для объектов

Objective-C в ваших конструкторах C++ (или в ином удобном месте) и освобождать эти объекты в ваших деструкторах (или где-то еще). Вот корректное объявление класса C++ с объектами Objective-C.

```
class ChessPiece {
    ChessPiece::PieceType type;
    int row, column;
    NSImage *pieceImage;
};
```

Вы можете помещать объекты C++ в объекты Objective-C.

```
@interface SWChessBoard : NSView
{
    ChessPiece *piece[32];
}
@end // SWChessBoard
```

У объектов C++, размещенных в объектах Objective-C (а не связанных с ними указателями), конструкторы вызываются при выделении объекта Objective-C, а деструкторы — при освобождении объекта Objective-C.

Java

Как и C++, Java имеет ряд возможностей, которых либо нет в Objective-C, либо они реализованы иными способами. Например, в классическом Objective-C отсутствует сборка мусора, но есть счетчики ссылок и пул автоосвобождения. При желании вы можете включить сборку мусора в ваших программах Objective-C.

Интерфейсы Java подобны формальным протоколам Objective-C, в том смысле что они оба требуют реализации набора методов. У Java есть абстрактные классы, а у Objective-C их нет. В Java имеются переменные класса, в то время как в Objective-C приходится использовать статические глобальные переменные в области видимости файла и предоставлять методы доступа к ним (метод, описанный в разделе, посвященном C++). Как упоминалось ранее, в Objective-C может быть вызван любой метод, поддерживаемый объектом, даже если он нигде не виден извне, например в заголовочном файле. Java позволяет вам объявлять классы как `final`, предотвращая создание подклассов. Objective-C оказывается на другом полюсе — он позволяет добавлять методы к любому классу во время работы программы.

Реализация классов в Objective-C обычно разделяется на два файла: заголовочный файл и файл реализации, но такое разделение не является обязательным. Заголовочный файл (с расширением `.h`) содержит открытую информацию о классе, такую как перечисления, типы, структуры и объекты, которые будут использоваться кодом, применяющим данный класс. Прочие модули импортируют этот файл при помощи директивы препроцессора `#import`. В Java нет препроцессора C, представляющего собой инструмент текстуальной подстановки, автоматически обрабатывающий исходные тексты C, Objective-C и C++ перед передачей их компилятору. Когда вы встречаете директивы, начинающиеся с `#`, знаете, что эта строка — команда препроцессора. Препроцессор C ничего не знает о семействе языков C; все, что

он умеет делать, — это подстановка текста. Препроцессор может быть очень мощным — и очень опасным! — инструментом. Многие программисты рассматривают отсутствие препроцессора в Java как достоинство этого языка программирования.

В Java почти каждая ошибка обрабатывается при помощи исключений. В Objective-C обработка ошибок зависит от используемого вами API. API в Unix обычно возвращает значение -1 и устанавливает переменную errno равной коду ошибки. API Cocoa обычно генерирует исключения только для ошибок программиста или в ситуациях, когда восстановление невозможно. Язык программирования Objective-C предоставляет возможности по обработке исключений, подобные возможностям Java и C++: @try, @catch и @finally.

В Objective-C нулевой объект называется nil. Вы можете посылать сообщения для nil и не беспокоиться об исключении NullPointerException. Сообщения к nil не выполняют никаких действий, так что нет необходимости проверять, не является ли нулевым объект, которому посылается сообщение. Сообщения к nil рассматривались ранее в разделе, посвященном C++.

В Objective-C вы можете изменить поведение класса в процессе работы программы путем добавления методов к существующим классам с помощью категорий. В Objective-C нет таких сущностей, как классы, объявленные как final; вы можете создавать подклассы любого класса — лишь бы имелся соответствующий заголовочный файл (поскольку компилятор должен знать размер объекта надкласса).

На практике создание подклассов в Objective-C используется существенно реже, чем в Java. Благодаря механизмам наподобие категорий и динамической системе времени выполнения, позволяющей отправлять любые сообщения любым объектам, вы можете добавить функциональность в те классы, где она имеет наибольший смысл. Например, вы можете добавить в NSString категорию, обеспечивающую новые возможности класса, такие как обращение строки или удаление всех пробельных символов. После этого вы можете вызывать этот метод у любого объекта NSString, независимо от его происхождения. Вы не ограничены в применении новых возможностей только собственными строковыми подклассами.

Вообще говоря, создание подклассов в Cocoa требуется только при создании новой разновидности объектов (объектов на вершине иерархии), фундаментальном изменении поведения объекта или при работе с классом, которому нужен подкласс из-за того, что сам класс не выполняет никакой полезной работы. Например, класс NSView, используемый Cocoa при создании компонентов пользовательского интерфейса, не имеет реализации метода drawRect:. Вам надо создать подкласс NSView и перекрыть указанный метод. Но для многих других объектов применяется делегирование и источники данных. Так как Objective-C может посылать любое сообщение любому объекту, объект не обязан принадлежать определенному подклассу или соответствовать определенному интерфейсу, так что один класс может быть делегатом и источником данных для любого количества различных объектов.

Поскольку методы источника данных и делегата объявляются в категориях, вы не обязаны реализовать их все. Программирование Cocoa в Objective-C основано на небольшом количестве методов, которые в свою очередь вызывают одноименные методы встроенных объектов просто для того, чтобы компилятор не возмущался при принятии формального протокола.

Сила предполагает ответственность. Система ручного выделения и освобождения памяти Objective-C, а также система автоосвобождения позволяют программисту очень легко допустить ошибку, найти и исправить которую будет очень трудно. Помещение категорий

в другие классы — механизм очень мощный, но при злоупотреблении им код становится очень запутанным и его практически невозможно передать для поддержки другому программисту. Кроме того, Objective-C основан на C, так что он отягощен всеми неприятностями C, в частности, опасностями применения препроцессора и ошибками, связанными с применением указателей.

BASIC

Для тех, кто начинал учиться программировать на языках программирования Visual Basic или REALbasic, переход к Cocoa и Objective-C может быть весьма трудным.

Интегрированная среда разработчика BASIC (Visual и REAL) представляет собой единое завершённое рабочее пространство, в то время как Cocoa разделяет среду разработки на две части: Interface Builder и Xcode. Interface Builder используется для создания пользовательского интерфейса и передачи ему имен методов для работы с конкретными объектами, после чего вся управляющая логика помещается в исходный текст, редактируемый в Xcode (или TextMate, BBEdit, emacs либо ином текстовом редакторе).

В BASIC элементы пользовательского интерфейса и код, с которым они работают, тесно связаны. Вы помещаете блоки кода в кнопки и текстовые поля, чтобы заставить вести их так, как это вам требуется. Этот код можно вынести в общий класс, а в кнопке останется только код, передающий сообщения этому классу. По большей части программирование в BASIC использует размещение кода в элементах пользовательского интерфейса. Если вы не будете аккуратны, такой стиль может привести к плохо построенным программам, в которых логика разбросана по множеству разных элементов. Обычно программирование на BASIC включает для изменения поведения объектов изменение их свойств.

В Cocoa имеется четкое разделение между интерфейсом и логикой, скрывающейся за ним. У вас имеется коллекция объектов, общающихся друг с другом. Вместо непосредственной установки свойств объекта вы просите его самостоятельно изменить свое свойство. Это тонкое, но важное отличие. Основная работа программиста в Cocoa заключается в выяснении, какое сообщение следует послать, в отличие от работы в BASIC, когда следует думать о том, какое свойство нужно изменить.

У BASIC имеется большой рынок управляющих элементов сторонних производителей и поддерживающего их кода. Часто можно просто купить готовое решение и интегрировать его в свой код, а не создавать решение самостоятельно.

Языки сценариев

Пожалуй, наиболее сложен переход к миру Objective-C и Cocoa для программистов, работавших с языками сценариев, такими как Perl, PHP, Python и Tcl.

Языки сценариев выделяются из прочих языков программирования различными удобствами для программистов, такими как очень мощная обработка строк, автоматическое управление памятью (с применением как счетчиков ссылок, так и сборки мусора), быстрая

разработка, гибкая система типов (легкий переход от чисел к строкам и спискам и обратно), а также изобилие готовых пакетов, которые можно загрузить и использовать. Среда времени выполнения у языков сценариев зачастую очень гибкая, позволяющая вам создавать собственные типы объектов и управляющих структур.

Если вы перешли к Objective-C из языков сценариев, то он может показаться вам большим шагом назад. Это язык 1980-х годов, в то время как языки сценариев развивались в 1990-х годах. Обработка строк, в связи с отсутствием встроенных возможностей работы с регулярными выражениями, может стать большой головной болью. Несмотря на наличие в современном Objective-C сборки мусора, подавляющая часть существующего кода (который вы можете найти в Интернете) использует ручное управление памятью с применением сообщений `retain` и `release`. Построение приложения включает этапы компиляции и компоновки, что приводит к задержке между изменением кода и получением результата. Вы должны вручную работать с различными типами, такими как целые числа, символьные массивы и строки. Кроме того, вы получаете весь багаж C, в том числе указатели, побитовые операции и легко совершаемые при работе с памятью ошибки.

А нужна ли вся эта головная боль, связанная с использованием Objective-C? Одной из причин является производительность: в зависимости от вида приложения, Objective-C может обеспечить существенно более высокую производительность, чем язык сценариев. Еще одним преимуществом является доступ к "родному" инструментарию для создания пользовательского интерфейса (Cocoa). Большинство языков сценариев поддерживает инструментарий Tk, изначально разработанный для языка Tcl. Это вполне работоспособный пакет, но он не в состоянии обеспечить те возможности пользовательского интерфейса, которые вам дает Cocoa. И, что достаточно важно, приложения, построенные с применением Tk, обычно выглядят не так, как должны выглядеть программы Mac.

Вы можете взять лучшее из обоих миров с помощью так называемых "сценарных мостов". Имеются такие мосты между Objective-C и Python (PyObjC) и Ruby (RubyObjC). При их использовании вы можете создавать подклассы Cocoa в Python или Ruby и получать доступ ко всем возможностям Cocoa.

Резюме

Objective-C и Cocoa не похожи на другие языки программирования и инструменты. Objective-C имеет ряд возможностей, связанных с динамической диспетчеризацией времени выполнения. В Objective-C вы можете делать то, что не позволяют другие языки программирования.

При этом в Objective-C отсутствуют некоторые возможности других языков программирования, в частности, расширенные возможности работы со строками, пространства имен, метапрограммирование.

Все в программировании имеет свою цену. Вы должны решить, стоят ли получаемые при работе с Objective-C возможности тех потерь, которые неизбежны при переходе от языка программирования, который вы используете в настоящее время. Что касается нас, то мы считаем, что затраты времени и усилий на изучение Objective-C полностью компенсируются возможностью использовать Cocoa для создания своих приложений.