Мэтт Гэлловей

СИЛА OBJECTIVE-C 2.0

Эффективное программирование для iOS и OS X

Addison-Wesley

ПИТЕР

Matt Galloway

Effective Objective-C 2.0

52 Specific Ways to Improve Your iOS and OS X Programs

Addison-Wesley

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

Мэтт Гэлловей

СИЛА OBJECTIVE-C 2.0

Эффективное программирование для iOS и OS X

ПИТЕР®

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

ББК 32.973.2- 018.1
УДК 004.43
Г98

Гэлловей М.

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

ISBN 978-5-496-00963-8

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

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

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

ББК 32.973.2-018.1
УДК 004.43

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

ОГЛАВЛЕНИЕ

Предисловие

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

О КНИГЕ

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

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

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

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

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

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

ДЛЯ КОГО НАПИСАНА ЭТА КНИГА

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

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

КАКИЕ ТЕМЫ РАССМАТРИВАЮТСЯ В КНИГЕ

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

Глава 1. Осваиваем Objective-C

Основные концепции, относящиеся к языку в целом.

Глава 2. Объекты, сообщения и исполнительная среда

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

Глава 3. Проектирование интерфейса и API

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

Глава 4. Протоколы и категории

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

Глава 5. Управление памятью

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

читатель познакомится с основными проблемами, связанными с управлением памятью.

Глава 6. Блоки и Grand Central Dispatch

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

Глава 7. Системные фреймворки

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

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

Об авторе

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

Глава 1
Осваиваем Objective-C

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

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

1 Познакомьтесь с истоками Objective-C

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

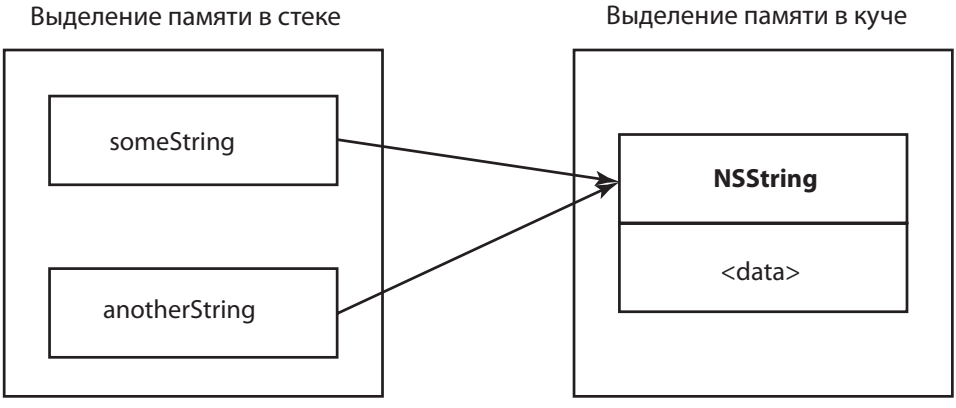

Рис. 1.1. Распределение памяти с экземпляром NSString в куче и двумя указателями на него в стеке

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

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

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

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

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

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

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

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

УЗЕЛКИ НА ПАМЯТЬ

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

2 МИНИМИЗИРУЙТЕ ИМПОРТИРОВАНИЕ В ЗАГОЛОВКАХ

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

Файлы, созданные при создании класса, могут выглядеть так:

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

```
// EOCPerson.m
```

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

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

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

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

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

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

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

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

```
@class EOCEmployer;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

УЗЕЛКИ НА ПАМЯТЬ

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

3 ИСПОЛЬЗУЙТЕ ЛИТЕРАЛЬНЫЙ СИНТАКСИС ВМЕСТО ЭКВИВАЛЕНТНЫХ МЕТОДОВ

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

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

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

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

ЧИСЛОВЫЕ ЛИТЕРАЛЫ

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

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

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

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

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

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

Литеральный синтаксис также работает и в выражениях:

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

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

ЛИТЕРАЛЬНЫЕ МАССИВЫ

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

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

С появлением литералов синтаксис заметно упростился:

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

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

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

С литералами достаточно следующей конструкции:

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

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

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

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