



Effective OBJECTIVE-C 2.0

*52 Specific Ways to Improve Your iOS
and OS X Programs*



Matt Galloway

FREE SAMPLE CHAPTER

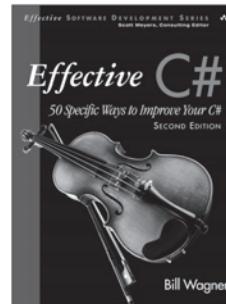
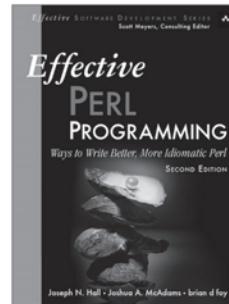
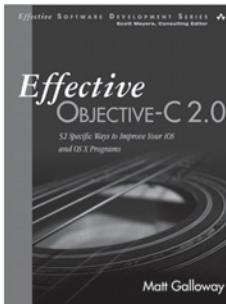
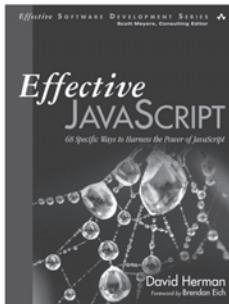
SHARE WITH OTHERS



Effective Objective-C 2.0

The Effective Software Development Series

Scott Meyers, Consulting Editor



◆ Addison-Wesley

Visit informit.com/esds for a complete list of available publications.

The Effective Software Development Series provides expert advice on all aspects of modern software development. Titles in the series are well written, technically sound, and of lasting value. Each describes the critical things experts always do — or always avoid — to produce outstanding software.

Scott Meyers, author of the best-selling books *Effective C++* (now in its third edition), *More Effective C++*, and *Effective STL* (all available in both print and electronic versions), conceived of the series and acts as its consulting editor. Authors in the series work with Meyers to create essential reading in a format that is familiar and accessible for software developers of every stripe.



Make sure to connect with us!
informit.com/socialconnect

informIT.com
the trusted technology learning source

◆ Addison-Wesley

Safari
Books Online

Effective Objective-C 2.0

**52 SPECIFIC WAYS TO IMPROVE YOUR
iOS AND OS X PROGRAMS**

Matt Galloway

 Addison-Wesley

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

The Cataloging-in-Publication Data is on file with the Library of Congress.

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-91701-0

ISBN-10: 0-321-91701-4

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing, May 2013

Editor -in-Chief
Mark Taub

Acquisitions Editor
Trina MacDonald

Development Editor
Chris Zahn

Managing Editor
John Fuller

Project Editor
Elizabeth Ryan

Packager
Vicki Rowland

Copy Editor
Evelyn W. Pyle

Indexer
Sharon Hilgenberg

Proofreader
Archie Brodsky

Technical Reviewers
Anthony Herron
Cesare Rocchi
Chris Wagner

Editorial Assistant
Olivia Bassegio

Cover Designer
Chuti Prasertsith

Compositor
Vicki Rowland

To Rosie

This page intentionally left blank

Contents

Preface	xi
Acknowledgments	xv
About the Author	xvii
Chapter 1: Accustoming Yourself to Objective-C	1
Item 1: Familiarize Yourself with Objective-C's Roots	1
Item 2: Minimize Importing Headers in Headers	4
Item 3: Prefer Literal Syntax over the Equivalent Methods	8
Item 4: Prefer Typed Constants to Preprocessor #define	12
Item 5: Use Enumerations for States, Options, and Status Codes	17
Chapter 2: Objects, Messaging, and the Runtime	25
Item 6: Understand Properties	25
Item 7: Access Instance Variables Primarily Directly When Accessing Them Internally	33
Item 8: Understand Object Equality	36
Item 9: Use the Class Cluster Pattern to Hide Implementation Detail	42
Item 10: Use Associated Objects to Attach Custom Data to Existing Classes	47
Item 11: Understand the Role of <code>objc_msgSend</code>	50
Item 12: Understand Message Forwarding	54
Item 13: Consider Method Swizzling to Debug Opaque Methods	62
Item 14: Understand What a Class Object Is	66

Chapter 3: Interface and API Design	73
Item 15: Use Prefix Names to Avoid Namespace Clashes	73
Item 16: Have a Designated Initializer	78
Item 17: Implement the <code>description</code> Method	84
Item 18: Prefer Immutable Objects	89
Item 19: Use Clear and Consistent Naming	95
Item 20: Prefix Private Method Names	102
Item 21: Understand the Objective-C Error Model	104
Item 22: Understand the <code>NSCopying</code> Protocol	109
Chapter 4: Protocols and Categories	115
Item 23: Use Delegate and Data Source Protocols for Interobject Communication	115
Item 24: Use Categories to Break Class Implementations into Manageable Segments	123
Item 25: Always Prefix Category Names on Third-Party Classes	127
Item 26: Avoid Properties in Categories	130
Item 27: Use the Class-Continuation Category to Hide Implementation Detail	133
Item 28: Use a Protocol to Provide Anonymous Objects	140
Chapter 5: Memory Management	145
Item 29: Understand Reference Counting	145
Item 30: Use ARC to Make Reference Counting Easier	153
Item 31: Release References and Clean Up Observation State Only in <code>dealloc</code>	162
Item 32: Beware of Memory Management with Exception-Safe Code	165
Item 33: Use Weak References to Avoid Retain Cycles	168
Item 34: Use Autorelease Pool Blocks to Reduce High-Memory Waterline	173
Item 35: Use Zombies to Help Debug Memory-Management Problems	177
Item 36: Avoid Using <code>retainCount</code>	183

Chapter 6: Blocks and Grand Central Dispatch	187
Item 37: Understand Blocks	188
Item 38: Create <code>typedefs</code> for Common Block Types	194
Item 39: Use Handler Blocks to Reduce Code Separation	197
Item 40: Avoid Retain Cycles Introduced by Blocks Referencing the Object Owning Them	203
Item 41: Prefer Dispatch Queues to Locks for Synchronization	208
Item 42: Prefer GCD to <code>performSelector</code> and Friends	213
Item 43: Know When to Use GCD and When to Use Operation Queues	217
Item 44: Use Dispatch Groups to Take Advantage of Platform Scaling	220
Item 45: Use <code>dispatch_once</code> for Thread-Safe Single-Time Code Execution	225
Item 46: Avoid <code>dispatch_get_current_queue</code>	226
Chapter 7: The System Frameworks	233
Item 47: Familiarize Yourself with the System Frameworks	233
Item 48: Prefer Block Enumeration to <code>for</code> Loops	236
Item 49: Use Toll-Free Bridging for Collections with Custom Memory-Management Semantics	243
Item 50: Use <code>NSCache</code> Instead of <code>NSDictionary</code> for Caches	248
Item 51: Keep <code>initialize</code> and <code>load</code> Implementations Lean	252
Item 52: Remember that <code>NSTimer</code> Retains Its Target	258
Index	265

This page intentionally left blank

Preface

Objective-C is verbose. Objective-C is chunky. Objective-C is ugly. I have heard all these things said about Objective-C. On the contrary, I find it elegant, flexible, and beautiful. However, to get it to be these things, you must understand not only the fundamentals but also the quirks, pitfalls, and intricacies: the topic of this book.

About This Book

This book doesn't teach you the syntax of Objective-C. It is assumed that you know this already. Instead, this book teaches you how to use the language to its full potential to write good code. Objective-C is extremely dynamic, thanks to having its roots in Smalltalk. Much of the work that's usually done by a compiler in other languages ends up being done by the runtime in Objective-C. This leads to a potential for code to function fine during testing but to break in strange ways later down the line in production, perhaps when processing invalid data. Avoiding these problems by writing good code in the first place is, of course, the best solution.

Many of the topics are not, strictly speaking, related to core Objective-C. Reference is made to items found in system libraries, such as Grand Central Dispatch, which is part of libdispatch. Similarly, many classes from the Foundation framework are referred to, not least the root class, NSObject, because developing with modern Objective-C means developing for Mac OS X or iOS. When developing for either, you will undoubtedly be using the system frameworks, collectively known as Cocoa and Cocoa Touch, respectively.

Since the rise of iOS, developers have been flocking to join the ranks of Objective-C development. Some of these developers are new to programming, some have come from Java or C++ backgrounds, and some have come from web-development backgrounds. In any case, all developers should take the time to learn how to use a language effectively.

Doing so will yield code that is more efficient, easier to maintain, and less likely to contain bugs.

Even though I have been writing this book for only around six months, it has been years in the making. I bought an iPod Touch on a whim; then, when the first SDK for it was released, I decided to have a play with development. That led to me build my first “app,” which I released as Subnet Calc, which immediately got many more downloads than I could have imagined. I became certain that my future was in this beautiful language I had come to know. Since then, I have been researching Objective-C, regularly blogging about it on my web site, www.galloway.me.uk/. I am most interested in the inner workings, such as the guts of blocks and how ARC works. When I got the opportunity to write a book about this language, I jumped at the chance.

In order to get the full potential from this book, I encourage you to jump around it, hunting for the topics that are of most interest or relevant to what you’re working on right now. Each item can be read individually, and you can use the cross-references to go to related topics. Each chapter collates items that are related, so you can use the chapter headings to quickly find items relevant to a certain language feature.

Audience for This Book

This book is aimed at developers who wish to further their knowledge of Objective-C and learn to write code that will be maintainable, efficient, and less likely to contain bugs. Even if you are not already an Objective-C developer but come from another object-oriented language, such as Java or C++, you should still be able to learn. In this case, reading about the syntax of Objective-C first would be prudent.

What This Book Covers

It is not the aim of this book to teach the basics of Objective-C, which you can learn from many other books and resources. Instead, this book teaches how to use the language effectively. The book comprises Items, each of which is a bite-sized chunk of information. These Items are logically grouped into topic areas, arranged as follows:

- ◆ **Chapter 1: Accustoming Yourself to Objective-C**
 - ♦ Core concepts relating to the language in general are featured here.
- ◆ **Chapter 2: Objects, Messaging, and the Runtime**
 - ♦ Important features of any object-oriented language are how objects relate to one another and how they interact. This chapter deals with these features and delves into parts of the runtime.

◆ **Chapter 3: Interface and API Design**

- ◆ Code is rarely written once and never reused. Even if it is not released to the wider community, you will likely use your code in more than one project. This chapter explains how to write classes that feel right at home in Objective-C.

◆ **Chapter 4: Protocols and Categories**

- ◆ Protocols and categories are both important language features to master. Effective use of them can make your code much easier to read, more maintainable, and less prone to bugs. This chapter helps you achieve mastery.

◆ **Chapter 5: Memory Management**

- ◆ Objective-C's memory-management model uses reference counting, which has long been a sticky point for beginners, especially if they have come from a background of a language that uses a garbage collector. The introduction of Automatic Reference Counting (ARC) has made life easier, but you need to be aware of a lot of important things to ensure that you have a correct object model that doesn't suffer from leaks. This chapter fosters awareness of common memory-management pitfalls.

◆ **Chapter 6: Blocks and Grand Central Dispatch**

- ◆ Blocks are lexical closures for C, introduced by Apple. Blocks are commonly used in Objective-C to achieve what used to involve much boilerplate code and introduced code separation. Grand Central Dispatch (GCD) provides a simple interface to threading. Blocks are seen as GCD tasks that can be executed, perhaps in parallel, depending on system resources. This chapter enables you to make the most from these two core technologies.

◆ **Chapter 7: The System Frameworks**

- ◆ You will usually be writing Objective-C code for Mac OS X or iOS. In those cases, you will have the full system frameworks stack at your disposal: Cocoa and Cocoa Touch, respectively. This chapter gives a brief overview of the frameworks and delves into some of their classes.

If you have any questions, comments, or remarks about this book, I encourage you to contact me. You can find my full contact details on the web site for this book at www.effectiveobjectivec.com.

This page intentionally left blank

Acknowledgments

When asked whether I would like to write a book about Objective-C, I instantly became excited. I had already read other books in this series and knew that the task of creating one for Objective-C would be a challenge. But with the help of many people, this book became a reality.

Much inspiration for this book has come from the many excellent blogs that are dedicated to Objective-C. Mike Ash, Matt Gallagher, and “bbum” are a few of the individuals whose blogs I read. These blogs have helped me over the years to gain a deeper understanding of the language. *NSHipster* by Mattt Thompson has also provided excellent articles that gave me food for thought while compiling this book. Finally, the excellent documentation provided by Apple has also been extremely useful.

I would not have been in a position to write this book had it not been for the excellent mentoring and knowledge transfer that happened while I was working at MX Telecom. Matthew Hodgson in particular gave me the opportunity to develop the company’s first iOS application, building on top of a mature C++ code base. The knowledge I picked up from this project formed the basis of much of my subsequent work.

Over the years, I have had many excellent colleagues with whom I have always stayed in touch either for academic reasons or purely just being there for a beer and a chat. All have helped me while writing this book.

I’ve had a fantastic experience with the team from Pearson. Trina MacDonald, Olivia Basegio, Scott Meyers, and Chris Zahn have all provided help and encouragement when required. They have provided the tools for me to get the book written without distraction and answered my queries when necessary.

The technical editors I have had the pleasure of working with have been incredibly helpful. Their eagle eyes have pushed the content of the book to be the very best. They should all be proud of the level of detail they used when analyzing the manuscript.

Finally, I could not have written this book without the understanding and support from Helen. Our first child was born the day I was supposed to start writing, so I naturally postponed for a short time. Both Helen and Rosie have been fantastic at keeping me going throughout.

About the Author

Matt Galloway is an iOS developer from London, UK. He graduated from the University of Cambridge, Pembroke College, in 2007, having completed an M.Eng. degree, specializing in electrical and information sciences. Since then, he has been programming, mostly in Objective-C. He has been developing for iOS ever since the first SDK was released. You'll find him on Twitter as @mattjgalloway, and he is a regular contributor to *Stack Overflow* (<http://stackoverflow.com>).

This page intentionally left blank

1

Accustoming Yourself to Objective-C

Objective-C brings object-oriented features to C through an entirely new syntax. Often described as verbose, Objective-C syntax makes use of a lot of square brackets and isn't shy about using extremely long method names. The resulting source code is very readable but is often difficult for C++ or Java developers to master.

Writing Objective-C can be learned quickly but has many intricacies to be aware of and features that are often overlooked. Similarly, some features are abused or not fully understood, yielding code that is difficult to maintain or to debug. This chapter covers fundamental topics; subsequent chapters cover specific areas of the language and associated frameworks.

Item 1: Familiarize Yourself with Objective-C's Roots

Objective-C is similar to other object-oriented languages, such as C++ and Java, but also differs in many ways. If you have experience in another object-oriented language, you'll understand many of the paradigms and patterns used. However, the syntax may appear alien because it uses a messaging structure rather than function calling. Objective-C evolved from Smalltalk, the origin of messaging. The difference between messaging and function calling looks like this:

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

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

The key difference is that in the messaging structure, the runtime decides which code gets executed. With function calling, the compiler

decides which code will be executed. When polymorphism is introduced to the function-calling example, a form of runtime lookup is involved through what is known as a virtual table. But with messaging, the lookup is always at runtime. In fact, the compiler doesn't even care about the type of the object being messaged. That is looked up at runtime as well, through a process known as dynamic binding, covered in more detail in Item 11.

The Objective-C runtime component, rather than the compiler, does most of the heavy lifting. The runtime contains all the data structures and functions that are required for the object-oriented features of Objective-C to work. For example, the runtime includes all the memory-management methods. Essentially, the runtime is the set of code that glues together all your code and comes in the form of a dynamic library to which your code is linked. Thus, whenever the runtime is updated, your application benefits from the performance improvements. A language that does more work at compile time needs to be recompiled to benefit from such performance improvements.

Objective-C is a superset of C, so all the features in the C language are available when writing Objective-C. Therefore, to write effective Objective-C, you need to understand the core concepts of both C and Objective-C. In particular, understanding the memory model of C will help you to understand the memory model of Objective-C and why reference counting works the way it does. This involves understanding that a pointer is used to denote an object in Objective-C. When you declare a variable that is to hold a reference to an object, the syntax looks like this:

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

This syntax, mostly lifted straight from C, declares a variable called `someString` whose type is `NSString*`. This means that it is a pointer to an `NSString`. All Objective-C objects must be declared in this way because the memory for objects is always allocated in heap space and never on the stack. It is illegal to declare a stack-allocated Objective-C object:

```
NSString stackString;  
// error: interface type cannot be statically allocated
```

The `someString` variable points to some memory, allocated in the heap, containing an `NSString` object. This means that creating another variable pointing to the same location does not create a copy but rather yields two variables pointing to the same object:

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

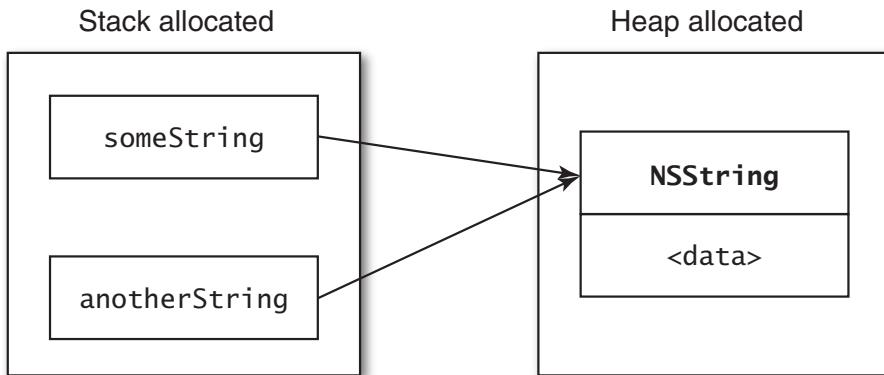


Figure 1.1 Memory layout showing a heap-allocated `NSString` instance and two stack-allocated pointers to it

There is only one `NSString` instance here, but two variables are pointing to the same instance. These two variables are of type `NSString*`, meaning that the current stack frame has allocated 2 bits of memory the size of a pointer (4 bytes for a 32-bit architecture, 8 bytes for a 64-bit architecture). These bits of memory will contain the same value: the memory address of the `NSString` instance.

Figure 1.1 illustrates this layout. The data stored for the `NSString` instance includes the bytes needed to represent the actual string.

The memory allocated in the heap has to be managed directly, whereas the stack-allocated memory to hold the variables is automatically cleaned up when the stack frame on which they are allocated is popped.

Memory management of the heap memory is abstracted away by Objective-C. You do not need to use `malloc` and `free` to allocate and deallocate the memory for objects. The Objective-C runtime abstracts this out of the way through a memory-management architecture known as reference counting (see Item 29).

Sometimes in Objective-C, you will encounter variables that don't have a * in the definition and might use stack space. These variables are not holding Objective-C objects. An example is `CGRect`, from the CoreGraphics framework:

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

```
frame.size.height = 150.0f;
```

A CGRect is a C structure, defined like so:

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

typedef struct CGRect CGRect;
```

These types of structures are used throughout the system frameworks, where the overhead of using Objective-C objects could affect performance. Creating objects incurs overhead that using structures does not, such as allocating and deallocating heap memory. When nonobject types (int, float, double, char, etc.) are the only data to be held, a structure, such as CGRect, is usually used.

Before embarking on writing anything in Objective-C, I encourage you to read texts about the C language and become familiar with the syntax. If you dive straight into Objective-C, you may find certain parts of the syntax confusing.

Things to Remember

- ◆ Objective-C is a superset of C, adding object-oriented features. Objective-C uses a messaging structure with dynamic binding, meaning that the type of an object is discovered at runtime. The runtime, rather than the compiler, works out what code to run for a given message.
- ◆ Understanding the core concepts of C will help you write effective Objective-C. In particular, you need to understand the memory model and pointers.

Item 2: Minimize Importing Headers in Headers

Objective-C, just like C and C++, makes use of header files and implementation files. When a class is written in Objective-C, the standard approach is to create one of each of these files named after the class, suffixed with .h for the header file and .m for the implementation file. When you create a class, it might end up looking like this:

```
// 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
// Implementation of methods
@end
```

The importing of Foundation.h is required pretty much for all classes you will ever make in Objective-C. Either that, or you will import the base header file for the framework in which the class's superclass lives. For example, if you were creating an iOS application, you would subclass UIViewController often. These classes' header files will import UIKit.h.

As it stands, this class is fine. It imports the entirety of Foundation, but that doesn't matter. Given that this class inherits from a class that's part of Foundation, it's likely that a large proportion of it will be used by consumers of EOCPerson. The same goes for a class that inherits from UIViewController. Its consumers will make use of most of UIKit.

As time goes on, you may create a new class called EOCEmployer. Then you decide that an EOCPerson instance should have one of those. So you go ahead and add a property for it:

```
// 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
```

A problem with this, though, is that the EOCEmployer class is not visible when compiling anything that imports EOCPerson.h. It would be wrong to mandate that anyone importing EOCPerson.h must also import EOCEmployer.h. So the common thing to do is to add the following at the top of EOCPerson.h:

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

This would work, but it's bad practice. To compile anything that uses EOCPerson, you don't need to know the full details about what an EOCEmployer is. All you need to know is that a class called EOCEmployer exists. Fortunately, there is a way to tell the compiler this:

```
@class EOCEmployer;
```

This is called forward declaring the class. The resulting header file for EOCPerson would look like this:

```
// 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
```

The implementation file for EOCPerson would then need to import the header file of EOCEmployer, as it would need to know the full interface details of the class in order to use it. So the implementation file would end up looking like this:

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

@implementation EOCPerson
// Implementation of methods
@end
```

Deferring the import to where it is required enables you to limit the scope of what a consumer of your class needs to import. In the example, if EOCEmployer.h were imported in EOCPerson.h, anything importing EOCPerson.h would also import all of EOCEmployer.h. If the chain of importing continues, you could end up importing a lot more than you bargained for, which will certainly increase compile time.

Using forward declaration also alleviates the problem of both classes referring to each other. Consider what would happen if EOCEmployer had methods to add and remove employees, defined like this in its header file:

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

This time, the EOCPerson class needs to be visible to the compiler, for the same reasons as in the opposite case. However, achieving this by importing the other header in each header would create a chicken-and-egg situation. When one header is parsed, it imports the other,

which imports the first. The use of `#import` rather than `#include` doesn't end in an infinite loop but does mean that one of the classes won't compile correctly. Try it for yourself if you don't believe me!

Sometimes, though, you need to import a header in a header. You must import the header that defines the superclass from which you are inheriting. Similarly, if you declare any protocols that your class conforms to, they have to be fully defined and not forward declared. The compiler needs to be able to see the methods the protocol defines rather than simply that a protocol does exist from a forward declaration.

For example, suppose that a rectangle class inherits from a shape class and conforms to a protocol allowing it to be drawn:

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

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

The extra import is unavoidable. For such protocols, it is prudent to put them in their own header file for this reason. If the `EOCDrawable` protocol were part of a larger header file, you'd have to import all of that, thereby creating the same dependency and extra compilation-time problems as described before.

That said, not all protocols, such as delegate protocols (see Item 23), need to go in their own files. In such cases, the protocol makes sense only when defined alongside the class for which it is a delegate. In these cases, it is often best to declare that your class implements the delegate in the class-continuation category (see Item 27). This means that the import of the header containing the delegate protocol can go in the implementation file rather than in the public header file.

When writing an import into a header file, always ask yourself whether it's really necessary. If the import can be forward declared, prefer that. If the import is for something used in a property, instance variable, or protocol conformance and can be moved to the class-continuation category (see Item 27), prefer that. Doing so will keep compile time as low as possible and reduce interdependency, which can cause problems with maintenance or with exposing only parts of your code in a public API should ever you want to do that.

Things to Remember

- ◆ Always import headers at the very deepest point possible. This usually means forward declaring classes in a header and importing their corresponding headers in an implementation. Doing so avoids coupling classes together as much as possible.
- ◆ Sometimes, forward declaration is not possible, as when declaring protocol conformance. In such cases, consider moving the protocol-conformance declaration to the class-continuation category, if possible. Otherwise, import a header that defines only the protocol.

Item 3: Prefer Literal Syntax over the Equivalent Methods

While using Objective-C, you will come across a few classes all the time. They are all part of the Foundation framework. Although technically, you do not have to use Foundation to write Objective-C code, you usually do in practice. The classes are `NSString`, `NSNumber`, `NSArray`, and `NSDictionary`. The data structures that each represent are self-explanatory.

Objective-C is well known for having a verbose syntax. That's true. However, ever since Objective-C 1.0, there has been a very simple way to create an `NSString` object. It is known as a string literal and looks like this:

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

Without this type of syntax, creating an `NSString` object would require allocating and initializing an `NSString` object in the usual `alloc` and then `init` method call. Fortunately, this syntax, known as literals, has been extended in recent versions of the compiler to cover `NSNumber`, `NSArray`, and `NSDictionary` instances as well. Using the literal syntax reduces source code size and makes it much easier to read.

Literal Numbers

Sometimes, you need to wrap an integer, floating-point, or Boolean value in an Objective-C object. You do so by using the `NSNumber` class, which can handle a range of number types. Without literals, you create an instance like this:

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

This creates a number that is set to the integer 1. However, using literals makes this cleaner:

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

As you can see, the literal syntax is much more concise. But there's more to it than that. The syntax also covers all the other types of data that NSNumber instances can represent. For example:

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

The literal syntax also works for expressions:

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

Making use of literals for numbers is extremely useful. Doing so makes using NSNumber objects much clearer, as the bulk of the declaration is the value rather than superfluous syntax.

Literal Arrays

Arrays are a commonly used data structure. Before literals, you would create an array as follows:

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

Using literals, however, requires only the following syntax:

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

But even though this is a much simpler syntax, there's more to it than that with arrays. A common operation on an array is to get the object at a certain index. This also is made easier using literals. Usually, you would use the `objectAtIndex:` method:

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

With literals, it's a matter of doing the following:

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

This is known as subscripting, and just like the rest of the literal syntax, it is more concise and much easier to see what's being done. Moreover, it looks very similar to the way arrays are indexed in other languages.

However, you need to be aware of one thing when creating arrays using the literal syntax. If any of the objects is nil, an exception is

thrown, since literal syntax is really just syntactic sugar around creating an array and then adding all the objects within the square brackets. The exception you get looks like this:

```
*** Terminating app due to uncaught exception
'NSInvalidArgumentException', reason: '***  
-[__NSPlaceholderArray initWithObjects:count:]: attempt to  
insert nil object from objects[0]'
```

This brings to light a common problem when switching to using literals. The following code creates two arrays, one in each syntax:

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

Now consider the scenario in which object1 and object3 point to valid Objective-C objects, but object2 is nil. The literal array, arrayB, will cause the exception to be thrown. However, arrayA will still be created but will contain only object1. The reason is that the arrayWithObjects: method looks through the variadic arguments until it hits nil, which is sooner than expected.

This subtle difference means that literals are much safer. It's much better that an exception is thrown, causing a probable application crash, rather than creating an array having fewer than the expected number of objects in it. A programmer error most likely caused nil to be inserted into the array, and the exception means that the bug can be found more easily.

Literal Dictionaries

Dictionaries provide a map data structure in which you add key-value pairs. Like arrays, dictionaries are commonly used in Objective-C code. Creating one used to look like this:

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

This is rather confusing, because the order is <object>, <key>, <object>, <key>, and so on. However, you usually think about dictionaries the other way round, as in key to object. Therefore, it doesn't read particularly well. However, literals once again make the syntax much clearer:

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

This is much more concise, and the keys are before the objects, just as you'd expect. Also note that the literal number in the example shows where literal numbers are useful. The objects and keys have to all be Objective-C objects, so you couldn't store the integer 28; instead, it must be wrapped in an NSNumber instance. But the literal syntax means that it's simply one extra character.

Just like arrays, the literal syntax for dictionaries suffers from an exception being thrown if any values are nil. However, for the same reason, this is a good thing. It means that instead of creating a dictionary with missing values, owing to the `dictionaryWithObjectsAndKeys:` method stopping at the first nil, an exception is thrown.

Also similar to arrays, dictionaries can be accessed using literal syntax. The old way of accessing a value for a certain key is as follows:

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

The equivalent literal syntax is:

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

Once again, the amount of superfluous syntax is reduced, leaving an easy-to-read line of code.

Mutable Arrays and Dictionaries

In the same way that you can access indexes in an array or keys in a dictionary through subscripting, you can also set them if the object is mutable. Setting through the normal methods on mutable arrays and dictionaries looks like this:

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

Setting through subscripting looks like this:

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

Limitations

A minor limitation with the literal syntax is that with the exception of strings, the class of the created object must be the one from the Foundation framework. There's no way to specify your own custom subclass that should be created instead. If you wanted to create an instance of your own custom subclass, you'd need to use the nonliteral syntax. However, since NSArray, NSDictionary, and NSNumber are class clusters (see Item 9), they are rarely subclassed, as it's non-trivial to do so. Also, the standard implementations are usually good enough. Strings can use a custom class, but it must be changed through a compiler option. Use of this option is discouraged because unless you know what you are doing, you will always want to use NSString anyway.

Also, in the case of strings, arrays, and dictionaries, only immutable variants can be created with the literal syntax. If a mutable variant is required, a mutable copy must be taken, like so:

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

This adds an extra method call, and an extra object is created, but the benefits of using the literal syntax outweigh these disadvantages.

Things to Remember

- ◆ Use the literal syntax to create strings, numbers, arrays, and dictionaries. It is clearer and more succinct than creating them using the normal object-creation methods.
- ◆ Access indexes of an array or keys in a dictionary through the subscripting methods.
- ◆ Attempting to insert nil into an array or dictionary with literal syntax will cause an exception to be thrown. Therefore, always ensure that such values cannot be nil.

Item 4: Prefer Typed Constants to Preprocessor

#define

When writing code, you will often want to define a constant. For example, consider a UI view class that presents and dismisses itself using animations. A typical constant that you'd likely want to factor out is the animation duration. You've learned all about Objective-C and its C foundations, and so you take the approach of defining the constant like this:

```
#define ANIMATION_DURATION 0.3
```

This is a preprocessor directive; whenever the string ANIMATION_DURATION is found in your source code, it is replaced with 0.3. This might seem exactly what you want, but this definition has no type information. It is likely that something declared as a “duration” means that the value is related to time, but it’s not made explicit. Also, the preprocessor will blindly replace all occurrences of ANIMATION_DURATION, so if that were declared in a header file, anything else that imported that header would see the replacement done.

To solve these problems, you should make use of the compiler. There is always a better way to define a constant than using a preprocessor define. For example, the following defines a constant of type NSTimeInterval:

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

Note that with this style, there is type information, which is beneficial because it clearly defines what the constant is. The type is NSTimeInterval, and so it helps to document the use of that variable. If you have a lot of constants to define, this will certainly help you and other people who read the code later.

Also note how the constant is named. The usual convention for constants is to prefix with the letter k for constants that are local to a translation unit (implementation file). For constants that are exposed outside of a class, it is usual to prefix with the class name. Item 19 explains more about naming conventions.

It is important where you define your constants. Sometimes, it is tempting to declare preprocessor defines in header files, but that is extremely bad practice, especially if the defines are not named in such a way that they won’t clash. For example, the ANIMATION_DURATION constant would be a bad name to appear in a header file. It would be present in all other files that imported the header. Even the static const as it stands should not appear in a header file. Since Objective-C has no namespaces, it would declare a global variable called kAnimationDuration. Its name should be prefixed with something that scopes it to the class it is to be used with, such as EOCViewClassAnimationDuration. Item 19 explains more about using a clear naming scheme.

A constant that does not need to be exposed to the outside world should be defined in the implementation file where it is used. For example, if the animation duration constant were used in a UIView subclass, for use in an iOS application that uses UIKit, it would look like this:

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

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

// EOCAutowiredView.m
#import "EOCAutowiredView.h"

static const NSTimeInterval kAnimationDuration = 0.3;

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

It is important that the variable is declared as both `static` and `const`. The `const` qualifier means that the compiler will throw an error if you try to alter the value. In this scenario, that's exactly what is required. The value shouldn't be allowed to change. The `static` qualifier means that the variable is local to the translation unit in which it is defined. A translation unit is the input the compiler receives to generate one object file. In the case of Objective-C, this usually means that there is one translation unit per class: every implementation (.m) file. So in the preceding example, `kAnimationDuration` will be declared locally to the object file generated from `EOCAutowiredView.m`. If the variable were not declared `static`, the compiler would create an external symbol for it. If another translation unit also declared a variable with the same name, the linker would throw an error with a message similar to this:

```
duplicate symbol _kAnimationDuration in:
    EOCAutowiredView.o
    EOCAotherView.o
```

In fact, when declaring the variable as both `static` and `const`, the compiler doesn't end up creating a symbol at all but instead replaces occurrences just like a preprocessor define does. Remember, however, the benefit is that the type information is present.

Sometimes, you will want to expose a constant externally. For example, you might want to do this if your class will notify others using

NSNotificationCenter. This works by one object posting notifications and others registering to receive them. Notifications have a string name, and this is what you might want to declare as an externally visible constant variable. Doing so means that anyone wanting to register to receive such notifications does not need to know the actual string name but can simply use the constant variable.

Such constants need to appear in the global symbol table to be used from outside the translation unit in which they are defined. Therefore, these constants need to be declared in a different way from the static const example. These constants should be defined like so:

```
// In the header file
extern NSString *const EOCStringConstant;

// In the implementation file
NSString *const EOCStringConstant = @"VALUE";
```

The constant is “declared” in the header file and “defined” in the implementation file. In the constant’s type, the placement of the `const` qualifier is important. These definitions are read backward, meaning that in this case, `EOCStringConstant` is a “constant pointer to an `NSString`.“ This is what we want; the constant should not be allowed to change to point to a different `NSString` object.

The `extern` keyword in the header tells the compiler what to do when it encounters the constant being used in a file that imports it. The keyword tells the compiler that there will be a symbol for `EOCStringConstant` in the global symbol table. This means that the constant can be used without the compiler’s being able to see the definition for it. The compiler simply knows that the constant will exist when the binary is linked.

The constant has to be defined once and only once. It is usually defined in the implementation file that relates to the header file in which it is declared. The compiler will allocate storage for the string in the data section of the object file that is generated from this implementation file. When this object file is linked with other object files to produce the final binary, the linker will be able to resolve the global symbol for `EOCStringConstant` wherever else it has been used.

The fact that the symbol appears in the global symbol table means that you should name such constants carefully. For example, a class that handles login for an application may have a notification that is fired after login has finished. The notification may look like this:

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

```
extern NSString *const EOCLoginManagerDidLoginNotification;

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

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

NSString *const EOCLoginManagerDidLoginNotification =
    @"EOCLoginManagerDidLoginNotification";

@implementation EOCLoginManager

- (void)login {
    // Perform login asynchronously, then call 'p_didLogin'.
}

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

@end
```

Note the name given to the constant. Prefixing with the class name that the constant relates to is prudent and will help you avoid potential clashes. This is common throughout the system frameworks as well. UIKit, for example, declares notification names as global constants in the same way. The names include UIApplicationDidEnterBackgroundNotification and UIApplicationWillEnterForegroundNotification.

The same can be done with constants of other types. If the animation duration needed to be exposed outside of the EOCAnimatedView class in the preceding examples, you could declare it like so:

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

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

Defining a constant in this way is much better than a preprocessor define because the compiler is used to ensure that the value cannot change. Once defined in EOCAnimatedView.m, that value is used

everywhere. A preprocessor define could be redefined by mistake, meaning that different parts of an application end up using different values.

In conclusion, avoid using preprocessor defines for constants. Instead, use constants that are seen by the compiler, such as `static const` globals declared in implementation files.

Things to Remember

- ◆ Avoid preprocessor defines. They don't contain any type information and are simply a find and replace executed before compilation. They could be redefined without warning, yielding inconsistent values throughout an application.
- ◆ Define translation-unit-specific constants within an implementation file as `static const`. These constants will not be exposed in the global symbol table, so their names do not need to be namespaced.
- ◆ Define global constants as external in a header file, and define them in the associated implementation file. These constants will appear in the global symbol table, so their names should be namespaced, usually by prefixing them with the class name to which they correspond.

Item 5: Use Enumerations for States, Options, and Status Codes

Since Objective-C is based on C, all the features of C are available. One of these is the enumeration type, `enum`. It is used extensively throughout the system frameworks but is often overlooked by developers. It is an extremely useful way to define named constants that can be used, for example, as error status codes and to define options that can be combined. Thanks to the additions of the C++11 standard, recent versions of the system frameworks include a way to strongly type such enumeration types. Yes, Objective-C has benefitted from the C++11 standard as well!

An enumeration is nothing more than a way of naming constant values. A simple enumeration set might be used to define the states through which an object goes. For example, a socket connection might use the following enumeration:

```
enum EOConnectionState {  
    EOConnectionStateDisconnected,  
    EOConnectionStateConnecting,  
    EOConnectionStateConnected,  
};
```

Using an enumeration means that code is readable, since each state can be referred to by an easy-to-read value. The compiler gives a unique value to each member of the enumeration, starting at 0 and increasing by 1 for each member. The type that backs such an enumeration is compiler dependent but must have at least enough bits to represent the enumeration fully. In the case of the preceding enumeration, this would simply need to be a `char` (1 byte), since the maximum value is 2.

This style of defining an enumeration is not particularly useful, though, and requires the following syntax:

```
enum EOConnectionState state = EOConnectionStateDisconnected;
```

It would be much easier if you didn't have to type `enum` each time but rather use `EOConnectionState` on its own. To do this, you add a `typedef` to the enumeration definition:

```
enum EOConnectionState {  
    EOConnectionStateDisconnected,  
    EOConnectionStateConnecting,  
    EOConnectionStateConnected,  
};  
typedef enum EOConnectionState EOConnectionState;
```

This means that `EOConnectionState` can be used instead of the full `enum EOConnectionState`:

```
EOConnectionState state = EOConnectionStateDisconnected;
```

The advent of the C++11 standard brought some changes to enumerations. One such change is the capability to dictate the underlying type used to store variables of the enumerated type. The benefit of doing this is that you can forward declare enumeration types. Without specifying the underlying type, an enumeration type cannot be forward declared, since the compiler cannot know what size the underlying type will end up being. Therefore, when the type is used, the compiler doesn't know how much space to allocate for the variable.

To specify the type, you use the following syntax:

```
enum EOConnectionStateConnectionState : NSInteger { /* ... */ };
```

This means that the value backing the enumeration will be guaranteed to be an `NSInteger`. If you so wished, the type could be forward declared like so:

```
enum EOConnectionStateConnectionState : NSInteger;
```

It's also possible to define the value a certain enumeration member relates to rather than letting the compiler choose for you. The syntax looks like this:

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

This means that EOCConnectionStateDisconnected will use the value 1 rather than 0. The other values follow, incrementing by 1 each time, just as before. Thus, EOCConnectionStateConnected will use the value 3, for example.

Another reason to use enumeration types is to define options, especially when the options can be combined. If the enumeration is defined correctly, the options can be combined using the bitwise OR operator. For example, consider the following enumeration type, found in the iOS UI framework, used to define which dimensions of a view can be resized:

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

Each option can be either on or off, and using the preceding syntax enables this because each option has just a single bit set in the value represented by it. Multiple options can be bitwise OR'ed together: for example, UIViewAutoresizingFlexibleWidth | UIViewAutoresizingFlexibleHeight. Figure 1.2 shows the bit layout of each enumeration member and the combination of two of the members.

It's then possible to determine whether one of the options is set by using the bitwise AND operator:

```
enum UIViewAutoresizing resizing =  
    UIViewAutoresizingFlexibleWidth |  
    UIViewAutoresizingFlexibleHeight;  
if (resizing & UIViewAutoresizingFlexibleWidth) {  
    // UIViewAutoresizingFlexibleWidth is set  
}
```

UIViewAutoresizingFlexibleLeftMargin	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	0	0	1
0	0	0	0	0	1		
UIViewAutoresizingFlexibleWidth	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	0	0	1	0
0	0	0	0	1	0		
UIViewAutoresizingFlexibleRightMargin	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	0	1	0	0
0	0	0	1	0	0		
UIViewAutoresizingFlexibleTopMargin	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	1	0	0	0
0	0	1	0	0	0		
UIViewAutoresizingFlexibleHeight	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	0	0	0	0
0	1	0	0	0	0		
UIViewAutoresizingFlexibleBottomMargin	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	0	0	0	0	0
1	0	0	0	0	0		
UIViewAutoresizingFlexibleWidth UIViewAutoresizingFlexibleHeight	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	0	0	1	0
0	1	0	0	1	0		

Figure 1.2 Binary representation of three options values and two of those values bitwise OR'd together

This is used extensively throughout the system libraries. Another example from UIKit, the iOS UI framework, uses it as a way of telling the system what device orientations your view supports. It does this with an enumerated type called `UIInterfaceOrientationMask`, and you implement a method called `supportedInterfaceOrientations` to indicate the supported orientations:

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

A couple of helpers defined within the Foundation framework help define enumeration types that also allow you to specify the integral type that will be used to store values that use the enumeration type. These helpers provide backward compatibility such that if you're targeting a compiler that supports the new standard, that syntax is used, but it falls back to the old syntax if not. The helpers are provided in the form of preprocessor `#define` macros. One is provided for normal enumeration types, such as the `ECCConnectionState` example. The other is provided for the case in which the enumeration defines a list of options like the `UIViewAutoresizing` example. You use them as follows:

```

typedef NS_ENUM(NSUInteger, EOConnectionState) {
    EOConnectionStateDisconnected,
    EOConnectionStateConnecting,
    EOConnectionStateConnected,
};

typedef NS_OPTIONS(NSUInteger, EOCPermittedDirection) {
    EOCPermittedDirectionUp      = 1 << 0,
    EOCPermittedDirectionDown   = 1 << 1,
    EOCPermittedDirectionLeft   = 1 << 2,
    EOCPermittedDirectionRight  = 1 << 3,
};

}

```

This is what the macro definitions look like:

```

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

```

The reason for the various ways of defining the macros is that there are different scenarios. The first case that is checked is whether the compiler supports the new style enumerations at all. This is checked with what looks like some rather complex Boolean logic, but all that it's checking is that the feature is there. If the feature is not there, it defines the enumeration by using the old style.

If the feature is available, the NS_ENUM type is defined such that it expands out like this:

```

typedef enum EOConnectionState : NSUInteger EOConnectionState;
enum EOConnectionState : NSUInteger {
    EOConnectionStateDisconnected,
    EOConnectionStateConnecting,

```

```
EOCConnectionStateConnected,
};
```

The NS_OPTIONS macro is defined in different ways if compiling as C++ or not. If it's not C++, it's expanded out the same as NS_ENUM. However, if it is C++, it's expanded out slightly differently. Why? The C++ compiler acts differently when two enumeration values are bitwise OR'ed together. This is something, as shown earlier, that is commonly done with the options type of enumeration. When two values are OR'ed together, C++ considers the resulting value to be of the type the enumeration represents: NSUInteger. It also doesn't allow the implicit cast to the enumeration type. To illustrate this, consider what would happen if the EOCPermittedDirection enumeration were expanded out as NS_ENUM:

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

Then consider attempting the following:

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

If the compiler were in C++ mode (or potentially Objective-C++), this would result in the following error:

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

You would be required to put in an explicit cast to the result of the ORing, back to EOCPermittedDirection. So the NS_OPTIONS enumeration is defined differently for C++ such that this does not have to be done. For this reason, you should always use NS_OPTIONS if you are going to be ORing together the enumeration values. If not, you should use NS_ENUM.

An enumeration can be used in many scenarios. Options and states have been shown previously; however, many other scenarios exist. Status codes for errors are a good candidate as well. Instead of using preprocessor defines or constants, enumerations provide a means for grouping together logically similar status codes into one enumeration. Another good candidate is styles. For example, if you have a UI element

that can be created with different styles, an enumeration type is perfect for that situation.

One final extra point about enumerations has to do with using a switch statement. Sometimes, you will want to do the following:

```
typedef NS_ENUM(NSUInteger, EOConnectionState) {
    EOConnectionStateDisconnected,
    EOConnectionStateConnecting,
    EOConnectionStateConnected,
};

switch (_currentState) {
    EOConnectionStateDisconnected:
        // Handle disconnected state
        break;
    EOConnectionStateConnecting:
        // Handle connecting state
        break;
    EOConnectionStateConnected:
        // Handle connected state
        break;
}
```

It is tempting to have a default entry in the switch statement. However, when used for switching on an enumeration that defines a state machine, it is best not to have a default entry. The reason is that if you add a state later on, the compiler will helpfully warn that the newly added state has not been cared for in the switch statement. A default block handles the new state, so the compiler won't warn. The same applies to any other type of enumeration defined using the NS_ENUM macro. For example, if used to define styles of a UI element, you would usually want to make sure that switch statements handled all styles.

Things to Remember

- ◆ Use enumerations to give readable names to values used for the states of a state machine, options passed to methods, or error status codes.
- ◆ If an enumeration type defines options to a method in which multiple options can be used at the same time, define its values as powers of 2 so that multiple values can be bitwise OR'ed together.

- ◆ Use the NS_ENUM and NS_OPTIONS macros to define enumeration types with an explicit type. Doing so means that the type is guaranteed to be the one chosen rather than a type chosen by the compiler.
- ◆ Do not implement a default case in switch statements that handle enumerated types. This helps if you add to the enumeration, because the compiler will warn that the switch does not handle all the values.

Index

A

ABI. *See Application Binary Interface (ABI)*
abstract base class
 hiding implementation detail, 42–47
 immutable and mutable arrays, 45
 subclasses inheriting from class cluster, 46
ACAccountStoreBooleanCompletion Handler type, 196
accessor methods
 automatically writing, 25, 28–29
 autosynthesis, 28–29
 categories, 131
 fine-grained control over, 31
 instance variables, 34, 227
 locking for atomicity, 29–30, 32
 names, 31
 preventing synthesis, 29
 strict naming patterns, 27
 user-implemented, 29
Accounts framework, 196
addFriend: method, 94, 126
addObject: method, 149
addOperationWithBlock: method, 218
alerts, 48–50
alertView parameter, 49
allObjects method, 237
alloc method, 149, 151, 153–154
AND operator, 19
animation, 235
anInstanceMethod method, 190

anonymous objects and protocols, 140–142
APIs
 C-level, 235
 completion callback blocks, 205–206
AppKit framework, 82, 235
Application Binary Interface (ABI), 27
applications, 2
 controlling high memory, 174
 multithreaded, 187
 not linking, 73–74
 terminating when main threads blocked, 197
applicationWillTerminate: method, 163
ARC. *See Automatic Reference Counting (ARC)*
arguments and messages, 50
arrays, 243
 adding number object, 149
 binning objects with hash, 38
 block-based enumeration, 189, 240–241
 common methods for, 45
 defining interface to, 46
 enumerating, 236–238
 immutable, 12, 45
 indexes, 9, 12, 241
 literals, 9–10
 mutable, 45
 nil objects and exceptions, 9–10
 placeholder arrays, 45

- arrays, (*cont'd*)
 - reversing enumeration, 240
 - size of, 243
- arrayWithObjects: method, 10
- assign attribute, 30, 171
- associated objects
 - attaching custom data to classes, 47–50
 - categories, 131
 - keys, 48
 - usage example, 48–50
- associations, 47
- asynchronous dispatch, 210
- asynchronous methods, 197–198
- atomic attribute, 32
- atomic locking and accessor methods, 29–30
- atomic properties, 208–209
- attributes
 - memory-management semantic, 91
 - properties, 29–33
- audio hardware, C-level API for interfacing with, 234–235
- audio playback and recording, 235
- AudioToolbox framework, 74
- autoDictionaryGetter function, 60–61
- autoDictionarySetter function, 60, 61
- Automatic Reference Counting (ARC), xiii, 108
 - backward compatibility, 156
 - cleanup code during deallocation, 160
 - code not compiling, 145, 149
 - destroying objects, 167
 - exceptions, 105
 - extraneous autorelease plus
 - immediate retain, 157–158
 - instance variables, 159, 160–161
 - lower-level C methods, 154
 - memory management, 154–156
 - memory-management rules, 214
 - memory-management semantics
 - of variables, 158–161
 - message dispatch and, 161
 - method-naming rules, 154–158
 - Objective-C objects, 231, 235, 244
 - optimizations, 156, 161
 - ownership of objects, 244
 - reference counting and, 153–161
 - retainCount method, 183–184
 - retains and releases, 154–156
 - runtime component, 156
 - safely handling exceptions, 167
 - variables retaining value, 158–159
 - writing setter, 159
- automilling, 117
- autorelease method, 146, 150–151, 154, 156–157, 184
- autorelease pools, 146, 150–151, 185
 - draining, 151
 - nesting, 174–175
 - overhead, 176
 - reducing high memory waterline, 173–177
 - scope, 174
 - stack, 175
 - temporary objects, 174–175
 - threads, 173
- autoreleasepools, collapsing, 156
- @autorelease syntax, 176
- __autorelease qualifier, 159
- autosynthesis, 28–29
- AVFoundation framework, 235

B

- backward compatibility and Automatic Reference Counting (ARC), 156
- barrier feature, 211
- base class
 - defining, 43
 - inheritance, 42–43, 140–142
 - instances, 45
- beginContentAccess method, 251–252
- bitfield data type, 121–122
- bitwise AND operator, 19
- bitwise OR operator, 19, 22, 23
- block-based enumeration, 240–243
 - changing method signature to limit need for casting, 241–242
 - versus for loops, 236–243
- reversing, 242
- stopping, 241

- __block qualifier, 189
 - blocks, xiii, 187, 217
 - array enumeration, 189
 - asynchronous dispatch, 210
 - callback, 203
 - caret (^) symbol, 188
 - concurrent queues, 229–230
 - copies of captured variables, 191–192
 - copying, 192–193
 - direct use and deadlock, 208
 - error handling, 200–202
 - flexibility, 201
 - function pointers, 191
 - global, 192–193
 - Grand Central Dispatch (GCD), 216
 - handler blocks, 197–203
 - heap, 192–193
 - hiding complicated type, 194
 - inability to cancel, 218
 - inline, 188–189
 - instance methods, 190
 - instance variables, 190
 - memory regions, 191
 - network fetchers, 199–200, 205
 - as objects, 189–190
 - objects owning, 203–207
 - only executed once, 225
 - parameters, 194
 - pointer declaring size of, 191
 - queues to schedule in, 203
 - reference counting, 190, 192
 - registering to observe notification through, 219
 - retain cycles, 160, 203–207
 - returning int, 188
 - return type, 194
 - running on queue, 221
 - runtime component, 188
 - scope, 188, 189, 192
 - self variable, 190
 - splitting up success and failure cases, 200–202
 - stack, 192–193
 - synchronous dispatch, 210
 - timers and, 261–262
 - types, 188, 194–196, 198–199
 - used as function, 188–189
 - used as variable, 188
 - variables, 189, 194, 210
 - Boolean properties
 - methods, 100
 - specifying name of getter, 31
 - BOOL type, 194
 - breakpoints, 88
 - _bridge keyword, 244
 - _bridge_retained keyword, 244
 - _bridge_transfer keyword, 244
 - buffer parameter, 99
 - business logic, decoupling data from, 115
 - buttons, 42
 - buttonWithType: method, 42
- C**
- C++
 - exceptions, 165
 - function names, 96
 - NS_OPTIONS enumeration, 22
 - scope of instance variables, 26
 - system libraries, 137
 - caches
 - automatically removing objects, 251
 - cost limit, 249–250
 - freeing memory, 250–252
 - NSCache class, 248–252
 - NSDictionary class, 248–252
 - order objects evicted, 249
 - pruning contents, 248–249
 - retaining keys, 248
 - thread safe, 248
 - URL retrieved as cache key, 250
 - caching
 - delegate responding to selectors, 122–123
 - images from Internet, 248–252
 - methods delegates implement, 122
 - optimization of, 121
 - CALayer class, 62
 - callback blocks, 203
 - camel casing names, 95
 - cancel method, 218
 - capitalizedString selector, 63–64
 - catch block and exceptions, 166
 - categories, 115
 - accessor methods, 131

- categories, (*cont'd*)
 - associated objects, 131
 - avoiding properties, 130–133
 - breaking class implementation into segments, 123–127
 - inability to synthesize variables, 131
 - loading, 252
 - load method, 253
 - methods, 127–128
 - namespacing name of, 128–129
 - overriding methods, 127–129
 - prefixing names, 74
 - prefixing third-party names, 127–129
- `CFArray` class, 243
- `CFArrayGetCount` function, 243
- `CFArrayRef` class, 243
- `_CFArray` struct, 243
- `CFDictionaryKeyCallBacks` struct, 245
- `CFDictionary` type, 244
- `CFDictionaryValueCallBacks` struct, 245
- `CFEqual` class, 247
- `CFHash` class, 247
- `CFMutableDictionary` type, 244
- `CFNetwork` framework, 234
- `CFString` class, 234
- `CFURLRequest` data structure, 126
- `CGFloat` scalar type, 30
- `CGPoint` data structure, 235
- `CGRect` data structure, 3–4, 235
- `CGSize` data structure, 235
- char type, 17–24
- Clang compiler, 153
 - blocks and, 188
- `class_addMethod` runtime method, 60
- Class Cluster pattern, hiding implementation detail, 42–47
- class clusters, 12, 43
 - adding concrete implementation, 46
 - checking class of instance of, 46
- Cocoa, 45–46
- collection classes as, 45
- Factory pattern, 44
- member objects, 45
- subclasses inheriting from abstract base class, 46
- class-continuation category, 7–8, 30, 130
- classes implementing protocol, 117–118
- data definitions, 135
- hiding implementation detail, 133–140
- implementation file definition, 133–140
- instance variables, 27, 122, 133–134
- methods, 133–134
- mixing Objective-C and C++ code, 135–137
- objects conforming to private protocols, 138–139
- private methods, 103, 138
- properties, 135
 - read-only properties, 137–138
- classes, 25
 - attaching custom data with associated objects, 47–50
 - breaking implementation into segments, 123–127
 - categories adding methods to, 127–128
 - checking instance of class cluster, 46
 - collections, 243
 - comparing for equality, 37
 - copying mutable and immutable variants, 112–113
 - custom, 12
 - defining blocks in instance method, 190
 - delegate protocols, 117–118
 - delegates, 7
 - designated initializers, 78–84
 - documentation, 44
 - dynamically adding method for unknown selector, 55–56
 - equality-checking methods, 36
 - forward declaring, 6, 8
 - Foundation framework, xi, 8
 - header files, 4–5
 - hierarchy, 69–71
 - hooking into forwarding paths, 55

- implementation files, 4–5
- inheritance, 7, 42–43
- initializers, 78
- initializing, 252–257
- instance variables, 26, 28, 130
- internal methods, 102
- introspection, 66–71, 92
- limiting imports to, 6
- literals and, 12
- loading, 252–257
- metadata, 68
- methods, 54, 62, 115
- multiple alerts, 48–49
- multiple network fetchers, 199
- mutable, 40–42, 89
- namespace clashes, 100
- naming, 95, 100–101
- not leaking, 140–142
- not understanding messages, 54–62
- parent class, 68
- prefixing names, 73–74
- properties, 89
- protocols, 120
- referring to each other, 6
- representing rectangle, 78–79
- retrieving methods for selectors, 64
- shared-instance method, 225
- system-supplied, 129
- toll-free bridging, 243–247
- translation unit, 14
- type definition, 196
- `class_getInstanceMethod` method, 64, 66
- class objects, 27, 66–71, 191
- class-specific equality methods, 39–40
- Class type, 68
- C-level APIs, 234–235
- C-level facilities for talking to networks, 234
- `close` method, 163
- Cocoa, xi, xiii, 173, 234
 - class clusters, 45–46
 - protocol language feature, 116
 - two-letter prefixes, 74
 - UI work on main thread, 202
 - zombies, 177–183
- Cocoa Touch, xi, xiii, 173, 234
 - UI work on main thread, 202
- code
 - accessing from multiple threads, 208
 - calculating offset at compile time, 27
 - conventions for generating, 27
 - documenting interface to, 115
 - incompatibility at runtime, 27
 - memory leaks, 105
 - mixing Objective-C and C++ and class-continuation category, 135–137
 - readable, 18
 - reducing separation with handler blocks, 197–203
 - segmenting into functional areas, 126
 - thread-safe single-time execution, 225–226
 - waiting unnecessarily on lock, 208
- collection classes
 - as class clusters, 45
 - mutable or immutable properties, 92–94
 - shallow copying, 113
- collections, 234
 - adding objects, 40
 - classes for, 243
 - custom memory-management semantics, 243–247
 - enumerating, 236–243
 - guaranteeing types of objects in, 242
 - looping through, 223–224
 - mutable objects, 89–90
 - retrieving objects from, 69–70
 - toll-free bridging, 243–247
- comparing objects for equality, 36–42
- compiler
 - accessor methods, 25, 28–29
 - deciding which code executes, 1–2
 - instance variables, 28
 - Objective-C++ mode, 167
- compiler flag enabling exception-safe code, 105
- `_completion` function, 76

- completion handler, 203
 - block types as, 198–199
 - instance variables, 203
 - other classes creating, 205
 - completion-handler block, 205–206
 - `completionHandler` property, 205, 207
 - concurrent queues, 210–211
 - blocks, 229–230
 - concurrent threads, 223
 - reads as normal blocks and writes as barrier blocks, 212
 - connections handling error representing what happened, 107
 - `connectionWithIdentifier:` method, 142
 - constants
 - defining, 12–13
 - exposing externally, 14–15
 - global, 17
 - global symbol table, 15
 - header file declaration, 15, 17
 - implementation file definition, 13, 15, 17
 - naming, 13, 15–16
 - naming values, 17–18
 - prefixing class name, 13, 16
 - translation unit, 17
 - types, 12–17
 - `const` qualifier, 14
 - constructors, 96–97
 - containers with mutable classes and equality, 40–42
 - `copy` attribute, 30–31
 - `copyDescription` callback, 247
 - `copy helper`, 191
 - copying
 - deep or shallow, 113–114
 - designated initializer, 110, 112
 - mutable and immutable variants of class, 112–113
 - objects, 109–114
 - `copy message`, 192
 - `copy` method, 109–110, 112–113, 154
 - `copyWithZone:` method, 109–110, 112–114, 247
 - CoreAnimation framework, 137, 235
 - `CALayer` class, 62
 - CoreAudio framework, 234–235
 - CoreData databases, 142
 - CoreData framework, 235
 - accessing properties of `NSManagedObject` class, 55–56
 - anonymous objects, 142
 - CoreFoundation framework, 234, 243
 - dictionaries, 244–245
 - zombie objects, 179
 - CoreFoundation objects
 - cleaning up, 160
 - releasing references, 162
 - CoreGraphics framework, 235
 - `CGRect` variable, 3–4
 - CoreText framework, 235
 - `countByEnumeratingWithState:` method, 239
 - counters for objects, 145–152
 - `count` method, 46
 - `_cppClass` instance variable, 136
 - C programming language
 - accessing stack-allocated struct member, 28
 - calling functions, 50–51
 - memory model, 2
 - Objective-C as superset of, 2, 4
 - prefixing function names, 77
 - seamless casting data structures with Objective-C objects, 234
 - static binding, 50–51
 - custom
 - classes, 12
 - data attaching to classes with associated objects, 47–50
 - memory-management semantics, 243–247
 - `.cxx_destruct` method, 160–161, 162
- D**
- dangling pointer, 149
 - data
 - decoupling from business logic, 115
 - encapsulating, 130
 - equality, 40
 - ownership semantics, 30–31
 - databases, interfaces for persisting objects, 235

data source, 115–123
Data Source pattern, 120–121
 data source protocol, 120
 data structures, 2D rendering, 235
`_dateOfBirth` instance variable, 27
 deadlocks
 dispatch queues, 228
 getters, 227
 synchronization blocks, 208
`dealloc` method, 154, 160–165, 181
`debugDescription` method, 88–89
 debugger, invoking print-object command within, 88–89
 debugging
 breakpoints, 88
 `dispatch_get_current_queue` function, 231
 `load` method, 253–254
 memory-management problems, 177–183
 opaque methods, 62–66
 printing object to inspect during, 84–89
 public and private methods, 102
 retain counts, 185–186
 splitting classes into categories, 126–127
 deep copy, 113–114
 deep equality, 40
`#define` preprocessor directive versus typed constants, 12–17
`#define` preprocessor macros, 20–22
 delegate methods, 49, 119–120
 Delegate pattern, 115–123, 171, 197
 delegate protocols, 7, 117–118, 197–198
 errors, 107
 methods, 118–119
 delegates
 anonymous, 141
 caching methods, 122
 classes implementing, 7
 implementing, 117
 introspection, 119
 list views, 120
 methods, 119
 network fetchers, 119
 nonowning relationship, 117
 objects, 115–123
 obtaining information from, 119–120
 passing errors to, 107
 property definition, 140
 responding to selectors, 121–123
 task completion notification, 197–198
 description message, 84
 description method
 dictionary within, 86–87
 implementing, 84–89
 overriding, 85–86
 piggybacking on `NSDictionary` class's implementation, 86–87
 returning string from, 86–87
 descriptor variable, 191
 designated initializers, 78–84
 `EOCRactangle` class, 81
 `EOCSquare` class, 80–81
 multiple, 81–83
 overriding superclass, 80
 `Rectangle` object, 79
 superclass, 80
 destructor function, 231
 device orientations supported by views, 20
 dictionaries, 243
 block-based enumeration, 241
 CoreFoundation framework, 244–245
 within `description` method, 86–87
 enumerating, 237–238
 fast enumeration, 239–240
 Foundation framework, 244
 immutable variants, 12
 keys, 11–12, 48, 237–238, 241, 245–247
 key-value pairs, 10–11
 literals, 10–11
 memory-management semantics, 140–141
 mutable arrays and, 11
 nil values and exceptions, 11
 objects, 11, 241–242
 retaining keys, 248
 size of, 244–245
 storing downloaded images, 248

dictionaries (*cont'd*)
 subscripting, 11
 user-created pruning code, 248
 values, 238, 241, 245–247
dispatch_after argument, 216
dispatch_apply function, 223–224
dispatch_async argument, 216, 220
dispatch_function_t type, 231
dispatch_get_current_queue function, 226–231
dispatch_group_async function, 220
dispatch_group_enter function, 220
dispatch_group_leave function, 220
dispatch_group_notify function, 221
dispatch groups
 associating tasks with, 220
 platform scaling and, 220–224
 timeout value, 221
 tracking tasks submitted to serial queues, 222–223
dispatch_group_wait function, 221
dispatch_once function, 217, 225–226
dispatch_once_t type, 225
dispatch queues, 91, 187, 217
 deadlock, 228
 versus locks for synchronization, 208–213
dispatch_sync argument, 216
dispose helper, 191
doADaysWork, 45
documentation, 44
doesNotRecognizeSelector: exception, 55, 57
 "Don't Repeat Yourself" (DRY) principle of software development, 219
doSomething: method, 108
doSomethingThatMayThrow object, 166
doSomethingThatUsesItsInternalData method, 256
doSomethingWithInit: method, 174
doSomethingWithObject: method, 186

dot syntax, 25, 27–28
DRY. *See* "Don't Repeat Yourself"
 (DRY) principle of software development
duplicate-symbol error, 74–77
dynamic binding, 2, 4, 51–52
 on top of dynamic binding, 213–214
@dynamic keyword, 29
dynamic libraries, 233
dynamic method resolution, 55–56, 59–62
@dynamic properties, 55–56, 59–62
dispatch_sync function, 227–228

E

else statement, 192
Employee example, 45, 46
encapsulating data, 130
endContentAccess method, 251–252
enumerateKeysAndObjectsUsingBlock: method, 242
enumerateKeysAndObjectsWithOptions: method, 242
enumerateObjectsUsingBlock: method, 189
enumerateObjectsWithOptions: method, 242
enumerating collections
 arrays, 236–237
 block-based enumeration, 240–243
 dictionaries, 237
 fast enumeration, 239–240
 for loops, 236–237
 NSEnumerator object, 237–239
 Objective-C 1.0 enumerations, 237–239
 reversing, 238–239
 sets, 237
enumerations
 adding **typedef**, 18
 backward compatibility, 20–22
 behavior of, 242
 combining options, 19–20
 forward declaring types, 18
 grouping error status codes, 22
 member values, 18–19
 naming constant values, 17–18

next object in, 238
 options, 17–24
 readable code, 18
 socket connections, 17–18
 states, 17–24
 status codes, 17–24
 switch statement, 23
 types, 18
 value backing, 18
 enumeration type, 17–24
 error codes, 108–109
 enumerators, reverse, 238–239
 enum type, 17–24
 EOCAutoDictionary class, 16
 EOCAutoDictionary.m file, 14, 16
 EOCAutoDictionary class, 59–60, 62
 EOCAutoDictionary class, 255
 EOCClassA class, 169, 171, 253, 256
 EOCClassB class, 169–171, 253, 256
 EOCClass class, 190, 260, 263
 shared-instance method, 225
 EOCClass.h header file, 136–137
 EOCClass instance, 205
 EOCClass.mm implementation file, 136
 EOCClass object, 180, 182
 EOConnectionStateConnected enumeration, 19
 EOConnectionStateDisconnected enumeration, 19
 EOConnectionState enumeration, 18, 20
 EOCDatabaseConnection protocol, 141–142
 EODataModel object, 116
 EOCDeliver object, 140
 EOCDog class, 37
 EOCDrawable protocol, 7
 EOCEmployer class, 5–6
 EOCEmployer.h header file, 5–6
 EOCTableView class, 101
 EOCLibrary library, 77
 eoc_myLowercaseString method, 65
 EOCHNetworkFetcher class, 117, 119, 122
 EOCHNetworkFetcher object, 116, 207
 EOCHObject class and private methods, 102–103
 EOCHPermittedDirection enumeration, 22
 EOCPerson class, 5, 31–32, 35, 39–40, 88, 93–94, 110–111, 114, 124–125, 130, 138–139, 175
 categories, 125–126
 class-continuation category, 133
 implementation file, 6
 EOCPerson.h header file, 4–6, 126
 EOCPerson objects, 37, 39, 86
 EOCPPointOfInterest class, 90, 91–92
 EOCRRectangle class, 78–79, 80, 96–97
 designated initializers, 81, 82–84
 EOCHReleaseCallback function, 247
 EOCHRetainCallback function, 247
 EOCHSecretDelegate protocol, 138–139
 EOCSmithPerson class, 37
 EOCSomeBlock type, 195
 EOCSoundPlayer class, 74–76
 EOCSoundPlayerCompletion completion handler, 77
 EOCSquare class, 80–81
 EOCSmithPerson subclass, 35
 EOCHStringConstant constant, 15
 EOCHSubClass class, 255
 EOCHSuperSecretClass class, 134–135
 EOCHTheClass class, 73
 EOCHViewClassAnimationDuration constant, 13
 equal callback, 247
 equality
 comparing objects for, 36–42
 data, 40
 deep versus shallow, 40
 mutable classes in containers, 40–42
 objects, 42, 70
 strings, 99
 equality-checking methods, 36–37
 equality methods, 39–40
 equality (==) operator, 36, 70
 error codes as enumeration type, 108–109
 error domains
 defining what error occurred, 106–107
 libraries, 109

error parameter, 108
 errors, 106–107
 error status codes
 grouping, 22
 named constants as, 17–24
`evictsObjectsWithDiscardedContent`
 Content property, 251
 exceptions, 104
 Automatic Reference Counting (ARC), 105
 C++, 165
 catch block, 166
 causing application to exit, 105
 code catching and handling, 165
 fatal errors only, 105–106
 Key-Value Observing (KVO), 165
 memory leaks, 105
 nil objects in arrays, 9–10
 nil values in dictionaries, 11
 object destruction, 166–167
 Objective-C, 165
 throwing in overridden methods, 105–106
 try block, 166
 zombie objects, 178
 exception-safe code, 105
 memory management, 165–168
`extern` keyword, 15

F

factory method, 98
 Factory pattern, 44
 fast enumeration, 236, 239–240
`_fetchedData` instance variable, 205
 fetcher class, 116–117
 file descriptors, 162
`@finally` block, 166, 167
`_firstName` instance variable, 26–27, 28
`-fobjc-arc-exceptions` flag, 105, 167–168
`_foo` instance variable, 150
 foo setter accessor, 150
 for loops, 174, 176
 versus block enumeration, 236–243
 enumerating collections, 236–237
 forward declaring, 8
 classes, 6
 imports, 7

types for enumerations, 18
`_forwarding_` function, 181–182
 forwarding paths, 54–55
`forwardingTargetForSelector:`
 method, 56
`forwardInvocation:method`, 57
 Foundation framework, 89, 233
 classes, xi, 8, 12, 243
 collections, 234, 243
 dictionaries, 244
 helpers defined in, 20–22
 importing entirety, 5
 introspection, 66–71
 loading, 253
 NSCache class, 248
 NS class prefix, 234
 NSTimer class, 258
 string processing, 234
 zombie objects, 179
 Foundation.h header file, 5
 frameworks
 See also system frameworks
 base header file, 5
 protocol language feature, 116
 third-party, 233
 friendship category, 130
`_friends` instance variable, 111–112
 friends property, 94
 full forward mechanism, 57
`fullName` method, 33–34
 function pointers, 191
 functions, 2
 blocks used as, 188–189
 calling, 1–2
 calling versus messaging, 1
 polymorphism, 2
 tail-call optimization, 53–54
 2D rendering, 235
 unknown until runtime, 51

G

Galloway Web site, xii
 garbage collector, 145, 169
 GCD. See Grand Central Dispatch (GCD)
`getCharacters:range:` method, 99
`getter=<name>` attribute, 31
 getters
 `_block` syntax, 210

deadlock, 227
 instance variables, 25, 35–36
 properties, 30
 running concurrently, 210
 selectors, 56, 60
 serial synchronization queue, 210
 specifying name of, 31
 global blocks, 192–193
 global constants, 17
 global symbol table, 15
 global variables
 defining source of error, 106
 name conflicts, 74
 Grand Central Dispatch (GCD), xi,
 xiii, 187
 barrier feature, 211
 blocks, 216, 217
 dispatch groups, 220–224
 dispatch queues, 217
 inability to cancel blocks, 218
 locks, 208–213
 versus `performSelector:` method
 and friends, 213–217
 queue currently being executed,
 226–231
 queue priorities, 218
 queue-specific data functions,
 230–231
 singleton instances, 225–226
 synchronization, 217
 threads, 173, 223
 when to use, 217–220
 graphics, rendering, 235
 grouping tasks, 220–224
 groups, incrementing or decrementing number of tasks, 220

H

handler blocks, reducing code separation with, 197–203
 hash callback, 247
 hashes, 38–40, 42
 hash methods, 37–39, 42, 247
 hash tables, hash as index in, 38
`hasPrefix:` method, 99
 header files, 4–8
 constants, 15, 17
 `extern` keyword, 15
 protocols, 7

superclass, 7
 heap
 allocating and de-allocating memory, 2–4
 copying blocks from stack, 192
 managing memory, 3
 heap-allocated memory, cleaning up, 160
 heap blocks, 192–193
`height` instance variable, 96
`_height` instance variable, 80, 84
 helpers, backward compatibility, 20–22
`.h` file extension, 4
 high memory, reducing waterline, 173–177
 HTTP, obtaining data from server, 126
 HTTP requests to Twitter API, 74
 HTTP-specific methods, 126

I

IDE. *See Integrated Development Environment (IDE)*
 identifier property, 92
`id<EOCDelegate>` type, 140
`id<NSCopying>` type, 141
`id` type, 66–69, 215, 242
`if` statement, 45, 153, 192
 images, caching from Internet, 248–252
 immutable arrays, 45
 immutable collection-class properties, 92–94
`immutableCopy` method, 112–113
 immutable objects, 32, 89–94
 immutable variants, 12
 implementation
 class-continuation category hiding detail, 133–140
 exchanging, 63–64
 hiding with class clusters, 42–47
 instance variables, 27
 implementation files, 4–5
 class-continuation category, 133–140
 constants, 13, 15, 17
`#import` preprocessor directive, 7
 imports, forward declared, 7

IMPs function pointers, 62–63
`#include` preprocessor directive, 7
 indexes for arrays, 12
 inheritance, hierarchies with equality, 37
initialize method
 called lazily, 254
 inheritance, 255
 lean implementation of, 255–256
 runtime in normal state, 254
 superclass running, 254–255
 thread-safe environment, 254
initializers
 classes, 78
 designated, 78–84
 instance variables, 35
 methods, 35
 naming, 78
 throwing exceptions, 81
init method, 79–81
initWithCoder: method, 82, 84
initWithDimension: method, 81
initWithFormat: method, 151
initWithInt: method, 149
initWithSize: method, 97
initWithTimeIntervalSinceReferenceDate: designated initializer, 78
initWithWidth:andHeight method, 81
in keyword, 239
 inline blocks, 189
instances
 base class, 45
 instance variables, 68
 metadata, 68
 value passed in, 106
instance variables, 25
 accessor methods, 34
 Automatic Reference Counting (ARC) handling, 160–161
 bitfield, 122
 blocks, 190
 class-continuation category, 27, 122, 133–134
 classes, 28, 130
 cleanup code during deallocation, 160
 completion handler, 203
 declaring, 25–26
 directly accessing internal, 33–36
 documenting not exposed publicly, 133–140
 externally accessing, 33
 getters, 25
 inability to be synthesized by category, 131
 instances, 68
 memory-management semantics, 158–160
 naming, 28–29
 network fetchers stored as, 199
 offsets, 27, 92
 only known internally, 134
 pointer to instance, 136
 private, 90
 properties, 33–36
 qualifiers, 159–160
 queue usage, 227
 runtime, 27
 safe setting of, 159
 scope, 26
 setters, 25, 35
 setting to 0 (zero), 79
 as special variables held by class objects, 27
 subclasses, 46
 superclass, 35
 synthesis, 29
integers
 defined at compile time, 257
 parsing strings as, 98
Integrated Development Environment (IDE), 195
interface
 classes implementing protocol, 117
 documenting code, 115
Internet, caching images from, 248–252
introspection, 45, 66–71, 113
 classes, 92
 delegates, 119
 methods, 46, 69–71
 objects, 94
 retrieving objects from collections, 69–70

- `int` type, 98, 194
 - `intValue` method, 98
 - `invoke` variable, 191
 - iOS, xi, xiii
 - blocked UI thread, 187
 - blocks, 188
 - Cocoa Touch, 173, 234
 - main thread, 221
 - mapping functionality, 235
 - method called on application termination, 163
 - operation queues, 217
 - social networking facilities, 235–236
 - UI framework, 78
 - UI work performed on main thread, 226
 - iOS applications
 - main function, 173–174
 - `UIApplication` object, 147
 - iOS 5.0 SDK, 74
 - `isa` instance variable, 181
 - `isa` pointer, 68–69, 191
 - `isCancelled` property, 218
 - `isContentDiscarded` method, 250
 - `isEqual:` method, 36–39, 40, 42, 70, 247
 - `isEqualToArray:` method, 39
 - `isEqualToString:` method, 39
 - `isEqualToString:` method, 36, 39, 99
 - `isFinished` property, 218
 - `isKindOfClass:` method, 69–70
 - island of isolation, 152
 - `isMemberOfClass:` method, 69
- J**
- Java
 - exceptions, 104
 - function names, 96
 - instance variables scope, 26
 - interfaces, 115
- K**
- `kAnimationDuration` constant, 14
 - `kAnimationDuration` global variable, 13
 - keys
 - associated objects, 48
 - dictionaries, 11–12, 48
 - retain and release callbacks, 247
 - static global variables, 48
- Key-Value Coding (KVC), 92
- Key-Value Observing (KVO)
 - exceptions, 165
 - notifications, 34, 137
 - operation properties, 218
 - properties, 165
- KVC. *See* Key-Value Coding (KVC)
- L**
- `_lastName` instance variable, 28
 - `lastObject` method, 46
 - lazy initialization, 35–36
 - `length` method, 98
 - `lengthOfBytesUsingEncoding` method, 98–99
 - libdispatch, xi
 - libraries
 - duplicates, 74
 - error domain, 109
 - Private category, 127
 - retain count, 185
 - list views, 120
 - literal arrays, 9–10
 - literal dictionaries, 10–11
 - literal numbers, 8–9
 - literals, 12
 - LLDB debugger, 88–89
 - `load` method, 252–256
 - `localizedString` method, 100
 - `localizedStringWithFormat:` method, 98
 - local variables, 14
 - breaking retain cycles, 160
 - memory-management semantics, 158–160
 - qualifiers, 159–160
 - locks
 - Grand Central Dispatch (GCD), 208–213
 - recursive, 208
 - threads taking same lock multiple times, 208
 - `lowercaseString` method, 98
 - lowercase strings, 98
 - `lowercaseString` selector, 55, 63, 64–66

M

Mac OS X, xi, xiii
 blocked UI thread, 187
 blocks, 188
 Cocoa, 173, 234
 garbage collector, 145, 169
 main thread, 221, 226
 method called on application termination, 163
 NSApplication object, 147
 operation queues, 217
 social networking facilities, 235–236
 main function, 173–174
 main interface, defining properties, 132
 main thread, 173
 blocked terminating application, 197
 UI work in Cocoa and Cocoa Touch, 202
`malloc()` function, 160
 MapKit framework, 235
`MKLocalSearch` class, 202
 memory
 allocated on stack for blocks, 192
 caches freeing, 250–252
 heap-space allocation for objects, 2–3
 releasing large blocks of, 162
 reusing, 177
 segmenting into zones, 109
 memory leaks, 168, 214
 code safe against, 105
 multiple objects referencing each other cyclically, 152
 object not released, 153
 retain cycles, 169
 memory management
 Automatic Reference Counting (ARC), 155–156
 debugging problems with zombies, 177–183
 exception-safe code, 165–168
 property accessors, 150
 reference counting, 145–152, 183
 memory-management methods, 2
 illegally calling, 154
 overriding, 161
 memory-management model, 168

memory-management semantics, 35
 attributes, 91, 131
 bypassing, 34
 custom, 243–247
 objects, 47
 properties, 30–31
 variables, 158–160
 memory regions, 191
 message dispatch and Automatic Reference Counting (ARC), 161
 message forwarding, 52
 dynamic method resolution, 55–56, 59–62
 flow diagram, 57–59
 full forward mechanism, 57
 receiver, 57, 59
 replacement receiver, 55–57, 59
 selector, 57
`messageName` selector, 52
 message object, 153
 messages
 arguments, 50
 description message, 84
 floating-point values, 53
 forwarding path, 54–55
 method lookup, 103
 names, 50
 object not understanding, 54–62
 objects responding to all, 103
 passing, 50–54
 receivers, 52, 55
 selectors, 50, 52
 structs, 53
 superclass, 53
 unhandled, 57
 values, 50
 messaging, 4, 25
 versus function calling, 1
 runtime and, 1–2
 structure, 1
 metaclass, 68
 metadata, 68
 method dispatch, 34
 method dispatch system, 103
`method_exchangeImplementations`
 method, 64
 method-naming rules and Automatic Reference Counting (ARC), 154–158

- methods
 - arrays, 45
 - Automatic Reference Counting (ARC), 154–158
 - autoreleasing objects, 156–157
 - backtrace, 126–127
 - badly-named, 97
 - Boolean indicating success or failure, 107–108
 - Boolean properties, 100
 - categories, 123–127
 - causing action to happen on object, 100
 - as C function, 53
 - class-continuation category, 133–134
 - classes, 54, 115
 - class methods, 64
 - debugging opaque methods, 62–66
 - delayed execution of, 213
 - delegate methods, 119
 - delegate protocols, 118–119
 - delegates, 122
 - documenting not exposed publicly, 133–140
 - dynamically adding for unknown selector, 55–56
 - dynamic binding, 52
 - equality-checking, 36–37
 - exchanging implementations, 63–66
 - get prefix, 100
 - HTTP-specific, 126
 - introspection, 46
 - introspection methods, 69–71
 - memory-management, 2
 - mutable arrays, 45
 - naming, 1, 95
 - newly created value, 100
 - nil / 0, 106
 - not reentrant, 227
 - NSObject protocol, 85
 - objects, 50–54, 150–151
 - out-parameter passed to, 107–108
 - owned by caller, 154
 - parameters, 99, 100
 - prefixing names, 74, 102–104
 - return type, 98
 - runtime, 62–66
 - system-supplied class, 129
 - threads, 164–165, 213
 - throwing exceptions, 105–106
 - types, 100
 - verbose naming, 96–100
 - well-named, 97
 - method swizzling, 62–66
 - .m file extension, 4
 - MKLocalSearch class, 202
 - multithreaded applications, 187
 - multithreading, 187
 - mutable arrays, 45
 - dictionaries and, 11
 - methods, 45
 - runtime, 257
 - subscripting, 11
 - mutable classes, 89
 - equality in containers, 40–42
 - mutable collection-class properties, 92–94
 - `mutableCopy` method, 112–113, 154
 - `mutableCopyWithZone:` method, 112
 - mutable dictionaries, 140–141
 - mutable objects, 89–90
 - mutable variants, 12
 - `_myFirstName` instance variable, 28
 - `_myLastName` instance variable, 28
- N**
- name clashes, 74
 - name conflicts, 74
 - named constants as error status codes, 17–24
 - namespace clashes, 73–77, 100
 - naming
 - camel casing, 95
 - classes, 95, 100–101
 - clear and consistent, 95–101
 - delegate protocol, 117
 - methods, 95, 96–100
 - methods rules in Automatic Reference Counting (ARC), 154–158
 - prefixing, 73–77
 - prepositions in, 95
 - private methods, 102–104
 - properties, 98

naming (*cont'd*)
 protocols, 100–101
 superclass, 101
 variables, 95
nesting autorelease pools, 174–175
networkFetcher:didFailWith
 Error: method, 118
networkFetcher:didUpdate
 ProgressTo: method, 121
network fetchers, 198–199
 adding itself to global collection, 206
 blocks, 199–200
 completion handlers, 200, 203
 deallocating, 207
 delegates, 119
 errors, 200–202
 keeping alive, 206–207
 multiple, 199
 retaining block, 205
 stored as instance variables, 199
networks, C-level facilities for
 talking to, 234
new method, 154
newSelector selector, 64
nextObject method, 237–238
NeXTSTEP operating system, 234
NIBs XML format, 82
nil objects, 9–10
nonatomic attribute, 29, 32–33
nonfatal errors indicators, 106
nonfragile Application Binary Inter-
 face (ABI), 27
non-Objective-C objects
 cleaning up, 160
 releasing references, 162
notifications, 15
notify function, 221
NSApplicationDelegate object, 163
NSApplication object, 147
NSArray class, 8, 12, 39–40, 45–46,
 86, 89, 100, 112, 129, 234,
 243
NSArray collection, 236
NSAutoReleasePool object, 176
NSBlockOperation class, 218–219
NSCache class versus NSDictionary
 class, 248–252
NSCalendar class, 132
 _NSCFNumber type, 55
NSCoding protocol, 81–83
NSCopying object, 141
NSCopying protocol, 109–114, 247
NSDate class, 78
NSDictionary class, 8, 12, 39, 140,
 234, 241
 versus NSCache class, 248–252
 piggybacking on description
 method of, 86–87
 retaining keys, 248
NSDictionary collection, 236
NSDictionary object, 48, 231, 246
NSDiscardableContent protocols,
 250
NSEnumerationConcurrent option,
 242
NSEnumerationOptions type, 242
NSEnumerationReverse option, 242
NSEnumerator class, 236
NSEnumerator object, 237–240
NS_ENUM macro, 21–24
NSError class, 106–107
NSError object, 106–109
NSFastEnumeration protocol,
 239–240
 NSFetchedResultsController class,
 142
 NSFetchedResultsSectionInfo pro-
 tocol, 142
NSHTTPURLRequest category, 126
NSInteger scalar type, 30
NSInvocation object, 57, 59
NSLinguisticTagger class, 234
NSLog class, 149, 253
NSManagedObject class, 29
NSManagedObjects class, 55–56
NSMutableArray class, 45, 100, 112
NSMutable arrays, 41
NSMutableCopying protocol, 112
NSMutableDictionary class, 247
NSMutableSet class, 41, 94
NSMutableString subclass, 30
NSMutableURLRequest category, 126
NSMutableURLRequest class, 126
NSNotificationCenter API, 15, 203
NSNotificationCenter class, 162,
 219
NSNumber class, 8–9, 12, 129

- NSNumber object, 174, 185
N
NSObject class, xi, 37, 54–55, 67, 79, 85, 109, 213, 233, 234, 247
NSObject object, 181
NSObject protocol, 36–37, 67, 85, 88, 146, 183
NSOperation class, 218, 219
NSOperationQueue class, 203, 217–218
NS_OPTIONS macro, 22, 24
NSProxy class, 67, 70, 85
NSPurgeableData class, 250
NSPurgeableData objects, 251
NSRecursiveLock class, 208
NSSet class, 94, 113, 241
NSSet collection, 236
NSString class, 8, 12, 36, 38–39, 128–129, 153, 253
 methods, 97–99
 selector table, 63–64
NSString global constant, 108
NSString instance, 2–3, 67
NSString object, 8, 15, 151, 174, 185
NSString type, 30
NSString* type, 2, 3
NSTimeInterval type, 13
NSTimer class, 258–263
NSUInteger type, 22
NSURLConnection class, 107, 234
NSURLConnectionDelegate method, 107
NSErrorDomain domain, 106
NSURLRequest class, 126
`_NSZombie_` class, 181
NSZombieEnabled environment variable, 178, 181
`_NSZombie_EOCClass` object, 180
`_NSZombie_OriginalClass` object, 181
`_NSZombie_` template class, 180
NSZone class, 109
null pointer, 108
number object, 149
numbers and literals, 8–9
- O**
- OBJC_ASSOCIATION_ASSIGN** type, 47
OBJC_ASSOCIATION_COPY_NONATOMIC type, 47
OBJC_ASSOCIATION_COPY type, 47
objc_AssociationPolicy enumeration, 47
OBJC_ASSOCIATION_RETAIN_NONATOMIC type, 47
OBJC_ASSOCIATION_RETAIN type, 47
objc_autoreleaseReturnValue function, 157–158
objc_duplicateClass() function, 181
objc_getAssociatedObject method, 47
objc_msgSend function, 52, 54
objc_msgSend role, 50–54
objc_removeAssociatedObjects method, 48
objc_retainAutoreleaseReturnValue function, 157
objc_retain method, 154
objc_setAssociatedObject method, 47
ObjectA object, 146
objectAtIndex: method, 9, 46
ObjectB object, 146
ObjectC object, 146
object_getClass() runtime function, 179
object graph, 146, 149–150, 168
`_object` instance variable, 158
Objective-C
 error model, 104–109
 exceptions, 165
 garbage collector, 145
 messages and zombie objects, 179–180
 messaging structure, 1
 roots, 1–4
 runtime component, 2
 square brackets, 1
 as superset of C, 2, 4
 verbose syntax, 1, 8
Objective-C 1.0 enumerations, 237–239
Objective-C++ mode, 167
Objective-C objects
 Automatic Reference Counting (ARC), 235
 seamless casting with C data structures, 234
 what to do with, 244

objects, 25
 accessing data encapsulated by, 27
 anonymous, 140–142
 associated with objects, 47
 autoreleasing, 108, 155–157
 becoming related to each other, 146
 binning into arrays with hash, 38
 blocks as, 189–190
 calling methods, 50–54
 classes, 68, 69, 120
 cleaning up observation state, 162–165
 collections, 69–70, 242
 comparing for equality, 36–42
 conforming to private protocols, 138–139
 copying, 109–114
 counters, 145–152, 183
 deallocating, 146, 149, 162, 169
 declaring, 2
 delegates, 115–123
 description message, 84
 designated initializer, 78–84
 destruction and exceptions, 166–167
 dictionaries, 11
 equality, 70
 equality-checking methods, 36–37
 exceptions, 105
 heap-space allocated memory, 2–3
`id` type, 66–67
 immutable, 32, 89–94
 inconsistent internal data, 81
 instance variables, 25
 introspection, 94
 invalid references to, 146
 memory, 2
 memory-management semantics, 47
 memory regions, 191
 messaging, 25
 methods, 100, 142, 150–151
 methods as C functions, 53
 multiple referencing each other cyclically, 152
 mutable, 30–31
 nonvalid, 149
 notifications, 15, 162
 not understanding message, 54–62
 object graph, 150
 operation queues as, 217
 overhead, 4
 owning other objects, 146, 148
 pointers, 2, 67
 printing to inspect while debugging, 84–89
 properties, 25–33
 proxy, 70–71
 reference counting, 173
 releasing, 156, 173–177
 releasing references, 162–165
 responding to messages, 103
 retain count, 146, 173, 183
 retaining, 156
 runtime, 67–68
 with same hash, 42
 serializing with NIBs XML format, 82
 sets, 237
 stack-allocated memory for, 2, 3
 storage policies, 47
 synchronized against self variable, 208
 talking to each other, 115–123
 tasks, 221–222
 types, 4, 66–71, 242
 variable of Class type, 68
 variables, 2
 when application terminates, 163
 zombie objects, 179
`object1` variable, 10
`object2` variable, 10
`object3` variable, 10
`obj_msgSend_fpRet` function, 53
`obj_msgSend_stret` function, 53
`obj_msgSendSuper` function, 53
 observation state, 162–165
 opaque methods, debugging, 62–66
`opaqueObject` property, 60
`open:` method, 163
 operation queues, 217–220
 operations, 218–219
`@optional` keyword, 118

- options
 - combining, 17–24, 19–20
 - enumerations, 17–24
- OR operator, 19, 22, 23
- out-parameter, 99, 107–108
- overriding methods, 129

- P**
- parameters
 - methods, 99–100
 - pointers to structures, 245
- parsing strings, 234
- passing messages, 50–54
- performSelector: method versus Grand Central Dispatch (GCD), 213–217
- performSelector:withObject:with Object: method, 215
- personWithName: method, 156
- placeholder arrays, 45
- platform scaling and dispatch groups, 220–224
- po command, 88
- pointers
 - comparing for equality, 37
 - dangling pointer, 149
 - denoting objects, 2
 - niling out, 149
 - to struct, 243
- pointerVariable variable, 67
- polymorphism, 2
- prefixing
 - category third-party names, 127–129
 - names, 73–77
- preprocessor directive, 13
- printGoodbye() function, 51
- printHello() function, 51
- print-object command invoking in debugger, 88–89
- Private category, 127
- private instance variables, 90
- private methods, 102–104, 138
- programmer error, 164
- properties, 25–33
 - atomic, 208–209
 - attributes, 29–33, 131
 - avoiding in categories, 130–133
 - class-continuation category, 135

- classes, 89
- collection-class mutable or immutable, 92–94
- comparing for equality, 37
- custom initializer, 31–32
- in definition of object, 27
- dot syntax, 27–28
- encapsulating data, 130
- getter methods, 30
- instance variables, 33–36, 35, 92
- Key-Value Coding (KVC), 92
- Key-Value Observation (KVO), 165
- lazy initialization, 35–36
- main interface definition, 132
- memory-management semantics, 30–31, 34, 91, 131
- naming, 98
- nonowning relationship, 30
- operations, 218
- owning relationship, 30
- read-only, 30, 32, 89, 90–91
- read-write, 30, 89
- setter methods, 30
- single words as, 98
- synchronization, 208–210
- value potentially unsafe, 171
- weak, 117
- property accessors, 150, 165
- @property attribute, 47
- property dot syntax, 34
- @property syntax, 27
- protocol language feature, 116
- protocols, 115
 - acquiring data for class, 120
 - anonymous objects, 140–142
 - declaring conformance, 8
 - Delegate pattern, 115–123
 - header files, 7
 - hiding implementation detail in API, 141
 - methods for database connections, 141–142
 - namespace clashes, 100
 - naming, 100–101
 - objects conforming to private, 138–139
- proxy, 70–71
- public API, 92–94

public interface declaring instance variables, 25–26
pure C functions name conflicts, 74

Q

QuartzCore framework, 235
querying structures, 122
`_queueA` variable, 228
queue barrier, 211
`_queueB` variable, 228
queues
 associating data as key-value pair, 230–231
 currently being executed, 226–231
 hierarchy, 229–230
 instance variables, 227
 many reads and single write, 211–212
 priorities, 218
 reading and writing to same, 209–210
 scheduling callback blocks, 230
 synchronization of property, 229

R

`readonly` attribute, 30, 90–91
read-only properties, 30, 32, 89, 90–91
 class-continuation category and setting internally, 137–138
`readonly` property, 91–92
`readwrite` attribute, 30
`read-write` property, 89
receiver, 55, 57, 59
rectangle class, 7
Rectangle object, 79
rectangles, 78–79
recursive locks, 208
reference-counted objects, 192
reference counting, xiii, 2–3, 3, 145–152, 173, 220–221
 automatic, 153–161
 Automatic Reference Counting (ARC) and, 153–161
 autorelease pools, 150–151, 173–177
 blocks, 190, 192
 catching exceptions, 168

destruction of objects, 166–167
indicating problem areas, 153–161
manually releasing objects, 162
memory management, 183
operation of, 146–150
refactoring, 168
retain cycles, 152
zombie objects, 179
reference-counting architecture, 168
references, releasing, 162–165
`release` method, 146, 154, 159, 184
`removeFriend:` method, 94
`replace` function, 95
replacement receiver, 55–57, 59
`_resetViewController` method, 104
`resolveClassMethod:` method, 55
`resolveInstanceMethod:` method, 56
`respondsToSelector:` method, 119
results controller, 142
retain count, 146, 173, 183, 185–186
 autorelease pool, 146
 balancing, 151
 changing, 184
 current, 183
 at given time, 184
 incrementing and decrementing, 146, 150
 inspecting, 146
 at least 1, 149
 very large value, 184–185
`retainCount` method, 146
 Automatic Reference Counting (ARC), 183–184
 avoiding use of, 183–187
retain cycles, 50, 152, 205–207
 avoiding, 168–172
 blocks, 203–207
 detecting, 169
 memory leaks, 169
 objects referencing each other, 168–169
`retain` method, 146, 154, 156–157, 159, 184
`ret` object, 214–215
reverse enumerator, 238–239
root classes, 85
run loops and timers, 258

- runtime, 25
dealloc method, 162
 deciding which code executes, 1
 as dynamic library, 2
 functions, 2, 51
 instance variables, 27
 lookup, 2
 memory-management methods, 2
 methods, 54, 62–66
 object types, 4, 66–71
 reference counting, 3
 selectors, 213
 structures, 2
 updating, 2
 zombie objects, 179
- S**
- scalar types, 30
scheduledTimerWithTimeInterval:
 method, 258
sectionInfo object, 142
sections property, 142
 selectors, 57
 calling directly, 213
 delegates, 121–123
 getters, 56, 60
 IMPs function pointers, 62–63
 messages, 50
 methods, 55–56, 62–66, 64
 parameters, 215
 running after delay, 215–216
 runtime, 213
 setters, 56, 60
 storing to perform after event, 214
 threads, 215–216
self variable, 190, 262
 completion-handler block refer-
 encing, 205
 objects synchronized against, 208
 serial synchronization queue,
 209–213
setFirstName: method, 138
setFullName: method, 33–34
setIdentifier: method, 92
setLastName: method, 138
set prefix, 56
 sets, 243
 block-based enumeration, 241
 enumerating, 237–238
- fast enumeration, 239–240
 objects, 237
 setter accessor, 150
setter=<name> attribute, 31
 setters
 Automatic Reference Counting
 (ARC), 159
 barrier block, 211
 instance variables, 25, 35
 naming, 31
 properties, 30
 scalar types, 30
 selectors, 56, 60
 serial synchronization queue, 210
 subclasses overriding, 35
 synchronous dispatch to asyn-
 chronous dispatch, 210
setup method, 158
setValue:forKey: method, 92
 shallow copy, 113–114
 shallow equality, 40
 shape class, 7
sharedInstance method, 225–226
 Singleton design pattern, 225
 singleton objects, 185
 Smalltalk, 1
 Social framework, 235–236
 social networking facilities, 235–236
 sockets, 162
 connections and enumerations,
 17–18
someBlock variable, 188
SomeClass class, 68
SomeCPPClass.h header file, 136
someObject receiver, 52
someString variable, 2
 sound file, playing, 74–76
 stack
 autorelease pools, 175
 blocks, 192–193
 copying blocks to heap, 192
 memory allocated for blocks, 192
 memory allocated for objects, 2, 3
 variables, 3–4
 standard view, 48
start method, 195
 states and enumerations, 17–24
 static analyzer, 153
 static global variables, 48

- static library, 233
 - static qualifier, 14
 - status codes and enumerations, 17–24
 - `stopPolling` method, 260–261
 - `stop` variable, 241
 - storage policies, 47
 - string literal, 8
 - `string` method, 98
 - strings
 - characters within range of, 99
 - converting to lowercase, 98
 - custom classes, 12
 - equality, 99
 - factory method, 98
 - immutable variants, 12
 - length of, 98–99
 - parsing, 98, 234
 - prefixed by another string, 99
 - processing, 234
 - returning hash of, 39
 - `stringValue` method, 151
 - `stringWithString:` method, 98
 - `str` object, 151, 179
 - `strong` attribute, 30
 - `__strong` qualifier, 159
 - structs, 53
 - structures, 2
 - delegates, 122
 - fields sized, 121–122
 - instance variables, 122
 - parameters pointers to, 245
 - pointers, 191
 - querying, 122
 - styles and enumeration types, 22–23
 - subclasses, 46
 - overriding setter, 35
 - throwing exceptions, 105–106
 - Subnet Calc app, xii
 - scripting, 9, 11–12
 - superclass
 - designated initializers, 80
 - header files defining, 7
 - inheritance, 7
 - instance variables, 35
 - prefixing names, 101
 - sending messages to, 53
 - subclasses overriding methods of, 46
 - `super_class` pointer, 68, 69
 - `super_class` variable, 68
 - `supportedInterfaceOrientations` method, 20
 - switch statements, 23–24
 - synchronization locks versus dispatch queues, 208–213
 - synchronization
 - Grand Central Dispatch (GCD), 217
 - properties, 208–209
 - synchronization blocks, 208
 - `@synchronized` approach, 226
 - `synchronizedMethod` method, 208
 - synchronous dispatch, 210
 - `@synthesize` syntax, 28
 - system frameworks
 - AVFoundation framework, 235
 - CFNetwork framework, 234
 - class clusters, 45
 - CoreAudio framework, 234–235
 - CoreData framework, 235
 - CoreFoundation framework, 234
 - CoreText framework, 235
 - dynamic libraries, 233
 - Foundation framework, 233, 234
 - UI frameworks, 235
 - system libraries
 - catching and handling exceptions, 165
 - C++ in, 137
 - system resources
 - deterministic life cycles, 163
 - exceptions thrown, 105
 - file descriptors, 162
 - memory, 162
 - performing tasks on, 223
 - sockets, 162
 - system-supplied class, 129
- T**
- table view, 101
 - tagged pointers, 185
 - tail-call optimization, 53–54
 - tasks
 - arrays, 221–222
 - asynchronously performing, 197
 - decrementing number of, 220
 - delegates, 197–198

- dispatch groups, 220
 - grouping, 220–224
 - priorities, 222
 - system resources, 223
 - waiting until finished, 221–222
 - temporary objects, 174–175
 - text, typesetting and rendering, 235
 - third-party classes, 127–129
 - third-party frameworks, 233
 - third-party libraries
 - catching and handling exceptions, 165
 - duplicate-symbol error, 77
 - threads
 - implicit autorelease pool, 173
 - methods, 164–165
 - multiple accessing code, 208
 - priorities, 219
 - safety of, 225–226
 - unblocking when long-running tasks occur, 197
 - thread-safe single-time code execution, 225–226
 - timers
 - blocks and, 261–262
 - deallocating, 260, 263
 - invalidating, 258–259
 - repeating, 258, 259
 - retain cycle, 262
 - retain-cycle with repeating, 259–260
 - retaining target, 258–263
 - run loops and, 258
 - scheduling, 258
 - weak references, 262
 - title:** attribute, 42
 - toll-free bridging, 55, 234, 243–247
 - translation unit, 14, 17
 - @try block, 166–167
 - try block exceptions, 166
 - Twitter API, HTTP requests to, 74
 - Twitter framework
 - TW prefix, 74
 - TWRequest class, 202
 - TWRequest object, 207
 - 2D rendering data structures and functions, 235
 - TW prefix, 74
 - TWRequest class, 74, 202
 - TWRequest object, 207
 - types
 - alias for another type, 194–196
 - blocks, 188, 194–196
 - compiler-dependent, 18
 - constants, 13, 14
 - enumerations, 18
 - literal numbers, 9
 - methods, 100
 - objects, 66–71
 - runtime, 4
 - variable name inside, 194
- U**
- UIAlertView class, 48–50
 - UIApplicationDelegate object, 163
 - UIApplicationDidEnterBackground Notification constant, 16
 - UIApplicationMain function, 174
 - UIApplication object, 147
 - UIApplicationWillEnterForegroundNotification constant, 16
 - UIButton class, 42
 - UI frameworks, 235
 - UIInterfaceOrientationMask enumerated type, 20
 - UIKit framework, 13, 16, 19–20, 42, 78, 82, 235
 - UIKit.h header file, 5
 - UIKit iOS UI library, 100
 - UISwitch class, 31
 - UITableViewCell object, 78
 - UITableView class, 101
 - UITableViewController class, 101
 - UITableViewDelegate protocol, 101
 - UI thread, 187
 - UIViewController class, 103–104
 - UIViewAutoresizing enumeration, 20
 - UIViewAutoresizingFlexibleHeight option, 19
 - UIViewAutoresizingFlexibleWidth option, 19
 - UIView class, 100–101, 235

`UIViewController` class, 5, 100
`UIView` subclass, 13
`unsafe_unretained` attribute, 30, 117, 169–171
`__unsafe_unretained` qualifier, 159
`uppercaseString` selector, 63, 64–65
`urlEncodedString` method, 128–129
URLs
 downloading data from, 234
 errors from parsing or obtaining data from, 106
 obtaining data from, 126
 retrieved as cache key, 250
user-developed equality methods, 39
`userInfo` parameter, 262
user interface
 unresponsive while task occurs, 197
 views, 100

V

`value` property, 215
values
 retain and release callbacks, 247
 tagged pointers, 185
variables
 `_block` qualifier, 189
 blocks, 188, 189, 210
 Class type, 68
 `const` qualifier, 14
 copies of captured, 191–192
 declaring, 2
 forward declaring types for enumerations, 18
 holding memory address, 67
 `isa` pointer, 68
 local, 14
 memory-management semantics, 158–161

naming, 95, 194
Objective-C objects, 3–4
pointing to same object, 2–3
retaining value, 158–159
stack, 3–4
`static` qualifier, 14
strong references to objects, 158 without asterisk (*), 3–4
video playback and recording, 235
views, 20, 100–101
virtual tables, 2
void return type, 215
`void*` variable, 191

W

weak attribute, 30, 117
weak properties, 117, 168–172
`__weak` qualifier, 159–160
weak references and timers, 262
WebKit web browser framework, 137
web services, 90
while loop, 184
width instance variable, 96
`_width` instance variable, 80, 84

X

Xcode and zombie objects, 178

Z

zombie classes, 180
zombie objects
 debugging memory-management problems, 177–183
 detecting, 182
 determining original class from, 181
 exceptions, 178
 Objective-C messages, 179–180
 turning objects into, 179
zones, segmenting memory into, 109