



**LEARNING**

**OBJECTIVE-C 2.0**

A Hands-On Guide to Objective-C for Mac and iOS Developers



**ROBERT CLAIR**

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@pearson.com

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

*Library of Congress Cataloging-in-Publication Data*

Clair, Robert, 1950–

Learning Objective-C 2.0 : a hands-on guide to Objective-C for Mac and iOS developers / Robert Clair.

p. cm.

Includes index.

ISBN-13: 978-0-321-71138-0 (pbk. : alk. paper)

ISBN-10: 0-321-71138-6 (pbk. : alk. paper)

1. Objective-C (Computer program language) 2. Object-oriented programming (Computer science) 3. Macintosh (Computer)—Programming. 4. iPhone (Smartphone)—Programming. I. Title.

QA76.73.O115C58 2011

005.1'17—dc22

2010019360

Copyright © 2011 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. For information regarding permissions, write to:

Pearson Education, Inc.  
Rights and Contracts Department  
501 Boylston Street, Suite 900  
Boston, MA 02116  
Fax: (617) 671-3447

ISBN-13: 978-0-321-71138-0

ISBN-10: 0-321-71138-6

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana. First printing, July 2010

**Editor-in-Chief**  
Karen Gettman

**Senior Acquisitions Editor**  
Chuck Toporek

**Managing Editor**  
John Fuller

**Project Editor**  
Anna Popick

**Copy Editor**  
Kelli Brooks

**Indexer**  
Jack Lewis

**Proofreader**  
Debbie Liehs

**Technical Reviewers**  
Joachim Bean  
Cory Bohon  
Andy Lee  
Matt Long  
Joseph E. Sacco,  
Ph.D.  
Scott D. Yelich

**Editorial Assistant**  
Romny French

**Compositor**  
Rob Mauhar

**Cover Designer**  
Chuti Prasertsith

# Contents at a Glance

Preface	xxiii
Acknowledgments	xxxix
About the Author	xxxiii

## **I: Introduction to Objective-C 1**

<b>1</b>	C, The Foundation of Objective-C	<b>3</b>
<b>2</b>	More About C Variables	<b>41</b>
<b>3</b>	An Introduction to Object-Oriented Programming	<b>55</b>
<b>4</b>	Your First Objective-C Program	<b>73</b>

## **II: Language Basics 91**

<b>5</b>	Messaging	<b>93</b>
<b>6</b>	Classes and Objects	<b>115</b>
<b>7</b>	The Class Object	<b>143</b>
<b>8</b>	Frameworks	<b>159</b>
<b>9</b>	Common Foundation Classes	<b>171</b>
<b>10</b>	Control Structures in Objective-C	<b>191</b>
<b>11</b>	Categories, Extensions, and Security	<b>213</b>
<b>12</b>	Properties	<b>229</b>
<b>13</b>	Protocols	<b>249</b>

## **III: Advanced Concepts 265**

<b>14</b>	Reference Counting	<b>267</b>
<b>15</b>	Garbage Collection	<b>291</b>
<b>16</b>	Blocks	<b>309</b>

**IV: Appendices 335****A** Reserved Words and Compiler Directives **337****B** Toll-Free Bridged Classes **339****C** 32- and 64-Bit **341****D** Runtimes, Old and New **345****E** Resources for Objective-C **349**Index **351**

# Contents

**Preface** xxiii

**Acknowledgments** xxxi

**About the Author** xxxiii

## **I: Introduction to Objective-C** 1

### **1 C, The Foundation of Objective-C** 3

The Structure of a C Program 4

main Routine 4

Formatting 5

Comments 5

Variable and Function Names 6

Naming Conventions 6

Files 7

Variables 8

Integer Types 8

Floating-Point Types 9

Truth Values 9

Initialization 10

Pointers 10

Arrays 12

Strings 13

Structures 14

typedef 15

Enumeration Constants 15

Operators 16

Arithmetic Operators 16

Remainder Operator 16

Increment and Decrement Operators 16

Precedence 17

Negation 18

Comparisons 18

Logical Operators 18

Logical Negation 19

Assignment Operators	<b>19</b>
Conversion and Casting	<b>19</b>
Other Assignment Operators	<b>20</b>
Expressions and Statements	<b>21</b>
Expressions	<b>21</b>
Evaluating Expressions	<b>21</b>
Statements	<b>22</b>
Compound Statements	<b>22</b>
Program Flow	<b>22</b>
if	<b>23</b>
Conditional Expression	<b>24</b>
while	<b>24</b>
do-while	<b>25</b>
for	<b>25</b>
break	<b>26</b>
continue	<b>26</b>
Comma Expression	<b>27</b>
switch	<b>27</b>
goto	<b>28</b>
Functions	<b>29</b>
Declaring Functions	<b>31</b>
Preprocessor	<b>31</b>
Including Files	<b>31</b>
#define	<b>32</b>
Conditional Compilation	<b>32</b>
printf	<b>33</b>
Using gcc and gdb	<b>35</b>
Summary	<b>37</b>
Exercises	<b>37</b>
<b>2 More About C Variables</b>	<b>41</b>
Memory Layout of an Objective-C Program	<b>41</b>
Automatic Variables	<b>42</b>
External Variables	<b>43</b>
Declaration Keywords	<b>44</b>
auto	<b>44</b>
extern	<b>45</b>

static	45
register	46
const	46
volatile	47
Scope	47
The Scope of Automatic Variables	47
Compound Statements and Scope	48
The Scope of External Variables	49
Dynamic Allocation	49
Summary	51
Exercises	52

### **3 An Introduction to Object-Oriented Programming 55**

Object-Oriented Programming	55
Classes and Instances	56
Methods	56
Encapsulation	56
Inheritance	57
Polymorphism	58
What Is the Point of an Object-Oriented Language?	58
An Introduction to Objective-C	58
Defining a Class	59
Class Names as Types	61
Messaging (Invoking a Method)	62
Class Objects and Object Creation	64
Memory Management	65
Objective-C Additions	66
Runtime	66
Names	66
Message Expressions	66
Compiler Directives	67
Literal Strings	67
Objective-C Keywords	67
Cocoa Numeric Types	70
NSLog	70
Summary	71

**4 Your First Objective-C Program 73**

- Building with Xcode **73**
- Objective-C Program Structure **76**
  - Build and Run the Program **78**
- An Object-Oriented Hello World **79**
  - Greeter.h **80**
  - Greeter.m **82**
- HelloObjectiveC.m **86**
  - Build and Run the Program **87**
- Summary **88**
- Exercises **88**

**II: Language Basics 91****5 Messaging 93**

- Methods **93**
  - A Simple Method **93**
  - Methods with Arguments **94**
- Messaging **96**
  - Polymorphism **97**
- Messaging Details **98**
  - Nesting **98**
  - Messaging `nil` **100**
  - Sending Messages to `self` **100**
  - Overriding and Messages to `super` **101**
  - Selectors **103**
    - Method with the Same Name **104**
    - Dynamic and Static Typing **105**
- Under the Hood **106**
- Message Forwarding **108**
- Efficiency **109**
- Introspection and Other Runtime Fun **111**
- Summary **112**
- Exercises **113**

**6 Classes and Objects 115**

- Defining a Class **115**



The Interface Section	<b>115</b>
@class Directive	<b>117</b>
The Implementation Section	<b>117</b>
Imports	<b>118</b>
Subclassing a Class	<b>119</b>
Defining a Subclass	<b>119</b>
An Example of Subclassing	<b>119</b>
Class Hierarchies	<b>123</b>
A Class Hierarchy Example	<b>124</b>
Abstract Classes	<b>125</b>
Creating Objects	<b>126</b>
Object Allocation	<b>126</b>
Object Initialization	<b>127</b>
Destroying Objects	<b>135</b>
Copying Objects	<b>136</b>
Shallow and Deep Copies	<b>137</b>
Mutable and Immutable Copies	<b>138</b>
Implementing Copying in Your Own Classes	<b>139</b>
Summary	<b>141</b>
Exercises	<b>141</b>
<b>7 The Class Object</b>	<b>143</b>
Class Objects	<b>143</b>
The Class Type	<b>144</b>
Class Methods	<b>146</b>
Other Class Methods	<b>147</b>
Convenience Constructors	<b>147</b>
Singletons	<b>149</b>
Initializing Classes	<b>150</b>
Mimicking Class Variables	<b>151</b>
Summary	<b>157</b>
Exercises	<b>157</b>
<b>8 Frameworks</b>	<b>159</b>
What Is a Framework?	<b>159</b>
Using a Framework	<b>160</b>
Cocoa Frameworks	<b>161</b>
iPhone	<b>162</b>

AppKit	<b>162</b>
Core Foundation	<b>163</b>
Memory Management for Core Foundation Objects	<b>164</b>
Toll-Free Bridging	<b>165</b>
Core Graphics	<b>166</b>
Core Animation	<b>167</b>
Other Apple-Supplied Frameworks	<b>167</b>
Third-Party Frameworks	<b>168</b>
Under the Hood	<b>168</b>
Summary	<b>170</b>
<b>9 Common Foundation Classes</b>	<b>171</b>
Immutable and Mutable Classes	<b>171</b>
Class Clusters	<b>172</b>
NSString	<b>173</b>
NSString Examples	<b>174</b>
C String to NSString and Back	<b>176</b>
NSMutableString	<b>176</b>
Literal Strings	<b>177</b>
Collection Classes	<b>177</b>
NSArray	<b>177</b>
NSDictionary	<b>180</b>
NSSet	<b>182</b>
NSNumber	<b>183</b>
NSNull	<b>184</b>
NSData	<b>185</b>
Accessing NSData's Bytes	<b>185</b>
File to NSData and Back	<b>186</b>
NSURL	<b>186</b>
Structures	<b>187</b>
Summary	<b>188</b>
Exercises	<b>189</b>
<b>10 Control Structures in Objective-C</b>	<b>191</b>
if Statements	<b>191</b>
Testing Objects for Equality	<b>193</b>

for Statements and Implicit Loops	<b>195</b>
for Statements	<b>195</b>
Implicit Loops	<b>195</b>
Implicit Loops with Blocks	<b>196</b>
while Statements and NSEnumerator	<b>196</b>
Modifying a Mutable Collection While Enumerating	<b>197</b>
Fast Enumeration	<b>199</b>
An Example Using Fast Enumeration	<b>201</b>
Exceptions	<b>205</b>
Throwing Your Own Exceptions	<b>206</b>
Multiple @catch Blocks	<b>207</b>
Nested Exception Handling	<b>207</b>
Using Exceptions	<b>208</b>
Should You Use Exceptions?	<b>209</b>
Summary	<b>210</b>
Exercises	<b>211</b>
<b>11 Categories, Extensions, and Security</b>	<b>213</b>
Categories	<b>213</b>
Overriding Methods with Categories	<b>216</b>
Other Uses for Categories	<b>217</b>
Extensions	<b>218</b>
Instance Variable Scope (Access Control)	<b>220</b>
Access Control for Methods	<b>221</b>
Namespaces	<b>221</b>
Security	<b>222</b>
Calling C Functions from Objective-C	<b>224</b>
Technical	<b>225</b>
Practical	<b>225</b>
Philosophical	<b>225</b>
Summary	<b>226</b>
Exercises	<b>226</b>
<b>12 Properties</b>	<b>229</b>
Accessing Instance Variables Outside of an Object (Don't Do It)	<b>230</b>

Declaring and Implementing Accessors	<b>231</b>
The Form of Accessors	<b>232</b>
Accessors Using Properties	<b>233</b>
The Instance Variable Name Can Be Different from the Property Name	<b>235</b>
The @property Statement	<b>236</b>
assign, retain, copy	<b>236</b>
readwrite, readonly	<b>237</b>
nonatomic	<b>237</b>
setter=name, getter=name	<b>237</b>
attributes and @dynamic	<b>238</b>
More About @dynamic	<b>238</b>
Properties and Memory Management	<b>240</b>
dealloc	<b>240</b>
Subclassing and Properties	<b>240</b>
Hidden Setters for readonly Properties	<b>242</b>
Properties as Documentation	<b>242</b>
Dot Syntax	<b>243</b>
Dot Syntax and Properties	<b>244</b>
Dot Syntax and C structs	<b>245</b>
Summary	<b>246</b>
Exercises	<b>247</b>
<b>13 Protocols</b>	<b>249</b>
Protocols	<b>249</b>
Using Protocols	<b>250</b>
Declaring a Protocol	<b>250</b>
Adopting a Protocol	<b>251</b>
Protocols as Types	<b>252</b>
Properties and Protocols	<b>252</b>
TablePrinter Example	<b>253</b>
TablePrinterDataSource	<b>253</b>
TablePrinter	<b>254</b>
FruitBasket	<b>256</b>
main	<b>258</b>
A Problem	<b>259</b>
Implement the Optional Methods	<b>260</b>

Protocol Objects and Testing for Conformance	<b>260</b>
Informal Protocols	<b>261</b>
Summary	<b>262</b>
Exercises	<b>263</b>

### **III: Advanced Concepts 265**

#### **14 Reference Counting 267**

The Problem	<b>268</b>
Reference Counting	<b>269</b>
Receiving Objects	<b>271</b>
Ownership	<b>273</b>
Taking Ownership by Copying	<b>274</b>
dealloc	<b>274</b>
Returning Objects	<b>276</b>
Autorelease	<b>277</b>
Autorelease Pools	<b>277</b>
Managing Autorelease Pools	<b>278</b>
Back to Convenience Constructors	<b>280</b>
Autorelease and the iPhone	<b>280</b>
Using Extra Autorelease Pools to Control Memory Usage	<b>280</b>
retainCount	<b>281</b>
Multithreading	<b>282</b>
When Retain Counts Go Bad	<b>283</b>
NSZombie	<b>284</b>
Retain Cycles	<b>285</b>
The Final Goodbye: When Programs Terminate	<b>288</b>
Summary	<b>288</b>
Exercises	<b>289</b>

#### **15 Garbage Collection 291**

Garbage Collection: The Theory	<b>291</b>
Garbage Collection: The Practice	<b>293</b>
Strong and Weak References	<b>293</b>
Using Garbage Collection	<b>294</b>
Controlling When Collections Occur	<b>296</b>

Finalizers	<b>296</b>
malloc and Garbage Collection	<b>297</b>
Core Foundation Objects and Garbage Collection	<b>298</b>
Some Bumps in the Road	<b>299</b>
Opaque Pointer Problems in the AppKit	<b>299</b>
Interior Pointers	<b>302</b>
Falsely Rooted Objects	<b>303</b>
Garbage Collection Pro and Con	<b>303</b>
The Positive	<b>304</b>
The Negative	<b>304</b>
Should You Use Garbage Collection?	<b>304</b>
Summary	<b>305</b>
Exercises	<b>305</b>
<b>16 Blocks</b>	<b>309</b>
Function Pointers	<b>310</b>
Calling a Function with a Function Pointer	<b>311</b>
Using Function Pointers	<b>312</b>
The Trouble with Function Pointers	<b>314</b>
NSInvocation	<b>315</b>
Blocks	<b>317</b>
Block Pointers	<b>318</b>
Access to Variables	<b>319</b>
Block Variables	<b>320</b>
Blocks Are Stack Based	<b>321</b>
Global Blocks	<b>322</b>
Blocks Are Objective-C Objects	<b>322</b>
Copying Blocks	<b>323</b>
Memory Management for Blocks	<b>323</b>
Traps	<b>326</b>
Blocks in Cocoa	<b>327</b>
Style Issues	<b>330</b>
Some Philosophical Reservations	<b>331</b>
Summary	<b>332</b>
Exercises	<b>332</b>

**IV: Appendices 335****A Reserved Words and Compiler Directives 337****B Toll-Free Bridged Classes 339****C 32- and 64-Bit 341**Kernel and User Programs in 64-Bit **342**Coding Differences for 64-Bit Programs **342**Performance **342**Compiling for 64-Bit **343**More Information **343****D Runtimes, Old and New 345**Synthesized Instance Variables **345**    Synthesized Instance Variables and Mac OS X  
    Leopard (v 10.5) **346**The Fragile Base Class Problem—Solved **347****E Resources for Objective-C 349**Apple Resources **349**Internet Resources **350**Groups **350**Books **350****Index 351**

*This page intentionally left blank*



# Preface

Objective-C is an object-oriented extension to C. You could call it “C with Objects.” If you’re reading this book, you’re probably interested in learning Objective-C so that you can write applications for Mac OS X or for iOS. But there’s another reason to learn Objective-C: It’s a fun language and one that is relatively easy to learn. Like anything else in the real world, Objective-C has some rough spots, but on the whole it is a much simpler language than other object-oriented languages, particularly C++. The additions that Objective-C makes to C can be listed on a page or two.

In the Apple world, Objective-C does not work alone. It is used in conjunction with two class libraries called *frameworks*. The Foundation framework contains classes for basic entities, such as strings and arrays, and classes that wrap interactions with the operating system. The AppKit contains classes for windows, views, menus, buttons, and the assorted other widgets needed to build graphical user interfaces. Together, the two frameworks are called Cocoa. On iOS, a different framework called the UIKit replaces the AppKit. Together, Foundation and UIKit are called Cocoa Touch.

Objective-C was initially created by Brad J. Cox in the early 1980s. In 1988, NeXT Computer, the company started by Steve Jobs after he left Apple, licensed Objective-C and made it the basis of the development environment for creating applications to run under NeXT’s NeXTSTEP operating system. The NeXT engineers developed a set of Objective-C libraries for use in building applications. After NeXT withdrew from the hardware business in 1993, it worked with Sun Microsystems to create OpenStep, an open specification for an object-oriented system, based on the NeXTSTEP APIs. Sun eventually lost interest in OpenStep. NeXT continued selling its version of OpenStep until NeXT was purchased by Apple in early 1997. The NeXTSTEP operating system became the basis of Mac OS X. The NeXT Objective-C libraries became the basis of Cocoa.

This book concentrates on the Objective-C language. It will not teach you how to write Cocoa programs or make you an expert Xcode user. It covers and makes use of a small part of the Foundation framework, and mentions the AppKit and UIKit only in passing. The book’s premise is that you will have a much easier time learning Cocoa if you first acquire a good understanding of the language on which Cocoa is based.

## Who Should Read This Book

This book is intended for programmers who want to learn Objective-C in order to write programs for Mac OS X or iOS. (iOS is used for the iPhone, the iPod touch, and the iPad.) Although it is technically possible to write complete OS X programs using

other languages, writing a program that follows the Apple *Human Interface Guidelines*<sup>1</sup> and has a proper Mac “look and feel” requires the use of the Objective-C Cocoa frameworks. Even if you write the core of a Mac application in a different language, such as plain C or C++, your user interface layer should be written in Objective-C. When writing for iOS, there is no choice: An iPhone app’s outer layer and user interface must be written in Objective-C.

The book will also be useful for programmers who want to write Objective-C programs for other platforms using software from the GNUStep project,<sup>2</sup> an open source implementation of the OpenStep libraries.

## What You Need to Know

This book assumes a working knowledge of C. Objective-C is an extension of C; the book concentrates on what Objective-C adds to C. For those whose C is rusty and those who are adept at picking up a new language quickly, Chapters 2 and 3 form a review of the essential parts of C, those that you are likely to need to write an Objective-C program. If you have no experience with C or any C-like language (C++, Java, and C#), you will probably want to read a book on C in conjunction with this book. Previous exposure to an object-oriented language is helpful but not absolutely necessary. The required object-oriented concepts are introduced as the book proceeds.

## New in Objective-C 2.0

If you already know some Objective-C and want to skip to the parts of the language that are new in the 2.0 version, they are covered in these chapters:

- *Fast Enumeration* (Chapter 10) provides a simple (and fast) way to iterate over a collection of objects.
- *Declared properties* (Chapter 12) provide an easy way to specify an object’s instance variables and to have the compiler create methods to access those variables for you.
- *Garbage collection* (Chapter 15) adds automatic memory management to Objective-C.
- *Blocks* (Chapter 16) let you define function-like objects that carry their context with them.

## How This Book Is Organized

This book is organized into three sections: The first section is a review of C, followed by an introduction to object-oriented programming and Objective-C. The second section of the book covers the Objective-C language in detail, and provides an introduction to

1. <http://developer.apple.com/mac/library/documentation/UserExperience/Conceptual/AppleHIGuidelines>

2. [www.gnustep.org](http://www.gnustep.org)

the Foundation framework. The final section of the book covers the two forms of memory management used in Objective-C, and Objective-C 2.0's newly added Blocks feature.

## Part I: Introduction to Objective-C

- Chapter 1, “C, The Foundation of Objective-C,” is a high-speed introduction to the essentials of C. It covers the parts of C that you are most likely to need when writing Objective-C programs.
- Chapter 2, “More About C Variables,” continues the review of C with a discussion of the memory layout of C and Objective-C programs, and the memory location and lifetime of different types of variables. Even if you know C, you may want to read through this chapter. Many practicing C programmers are not completely familiar with the material it contains.
- Chapter 3, “An Introduction to Object-Oriented Programming,” begins with an introduction to the concepts of object-oriented programming and continues with a first look at how these concepts are embodied in Objective-C.
- Chapter 4, “Your First Objective-C Program,” takes you line by line through a simple Objective-C program. It also shows you how to use Xcode to create a project, and then compile and run an Objective-C program. You can then use this knowledge to do the exercises in the remainder of the book.

## Part II: Language Basics

*Objects* are the primary entities of object-oriented programming; they group variables, called *instance variables*, and function-like blocks of code, called *methods*, into a single entity. *Classes* are the specifications for an object. They list the instance variables and methods that make up a given type of object and provide the code that implements those methods. An object is more tangible; it is a region of memory, similar to a C struct, which holds the variables defined by the object's class. A particular object is said to be an *instance* of the class that defines it.

- Chapter 5, “Messaging,” begins the full coverage of the Objective-C language. In Objective-C, you get an object to “do something” by sending it a *message*. The message is the name of a method plus any arguments that the method takes. In response to receiving the message, the object executes the corresponding method. This chapter covers methods, messages, and how the Objective-C messaging system works.
- Chapter 6, “Classes and Objects,” covers defining classes, and creating and copying object instances. It also covers *inheritance*, the process of defining a class by extending an existing class, rather than starting from scratch.

Each class used in executing an Objective-C program is represented by a piece of memory that contains information about the class. This piece of memory is called the class's *class object*. Classes can also define *class methods*, which are methods executed on behalf of the class rather than an instance of the class.

- Chapter 7, “The Class Object,” covers class objects and class methods. Unlike classes in some other object-oriented languages, Objective-C classes do not have class variables, variables that are shared by all instances of the class. The last sections of this chapter show you how to obtain the effect of class variables by using static variables.
- Chapter 8, “Frameworks,” describes Apple’s way of encapsulating dynamic link libraries. It covers the definition and structure of a framework, and takes you on a brief descriptive tour of some of the common frameworks that you will encounter when writing OS X or iOS programs.
- Chapter 9, “Common Foundation Classes,” covers the most commonly used Foundation classes: classes for strings, arrays, dictionaries, sets, and number objects.
- Chapter 10, “Control Structures in Objective-C,” discusses some additional considerations that apply when you use Objective-C constructs with C control structures. It goes on to cover the additional control structures added by Objective-C, including Objective-C 2.0’s new Fast Enumeration construct. The chapter concludes with an explanation of Objective-C’s exception handling system.
- Chapter 11, “Categories, Extensions, and Security,” shows you how to add methods to an existing class without having to subclass it and how to hide the declarations of methods that you consider private. The chapter ends with a discussion of Objective-C security issues.
- Chapter 12, “Properties,” introduces Objective-C 2.0’s new *declared properties* feature. Properties are characteristics of an object. A property is usually modeled by one of the object’s instance variables. Methods that set or get a property are called *accessor methods*. Using the declared properties feature, you can ask the compiler to synthesize a property’s accessor methods for you, saving yourself a considerable amount of effort.
- Chapter 13, “Protocols,” covers a different way to characterize objects. A *protocol* is a defined group of methods that a class can choose to implement. In many cases, what is important is not an object’s class, but whether the object’s class *adopts* a particular protocol by implementing the methods declared in the protocol. (More than one class can adopt a given protocol.) The Java concept of an interface was borrowed from Objective-C protocols.

### Part III: Advanced Concepts

Objective-C provides two different systems for managing object memory: *reference counting* and automatic *garbage collection*.

- Chapter 14, “Reference Counting,” covers Objective-C’s traditional reference counting system. Reference counting is also called *retain counting* or *managed memory*. In a program that uses reference counting, each object keeps a count,

called a *retain count*, of the number of other objects that are using it. When that count falls to zero, the object is deallocated. This chapter covers the rules needed to keep your retain counts in good order.

- Chapter 15, “Garbage Collection,” describes Objective-C 2.0’s new automatic garbage collection system. Using garbage collection, a separate thread called the *garbage collector*, is responsible for determining which objects are no longer needed and freeing them. Garbage collection relieves you of most memory management chores.
- Chapter 16, “Blocks,” discusses Objective-C 2.0’s new Blocks feature. A block is similar to an anonymous function, but in addition, a block carries the values of the variables in its surrounding context with it. Blocks are a central feature of Apple’s Grand Central Dispatch concurrency mechanism.

## Part IV: Appendices

- Appendix A, “Reserved Words and Compiler Directives,” provides a table of names that have special meaning to the compiler, and a list of Objective-C compiler directives. Compiler directives are words that begin with an `@` character; they are instructions to the compiler in various situations.
- Appendix B, “Toll-Free Bridged Classes,” gives a list of Foundation classes whose instances have the same memory layout as, and may be used interchangeably with, corresponding objects from the low-level C language Core Foundation framework.
- Appendix C, “32- and 64-Bit,” provides a brief discussion of Apple’s ongoing move to 64-bit computing.
- Appendix D, “Runtimes, Old and New,” describes the difference between the older “legacy” Objective-C runtime used for 32-bit OS X programs and the newer “modern” runtime used for 64-bit Objective-C programs running on OS X 10.5 or later, and for iOS programs.
- Appendix E, “Resources for Objective-C,” lists books and websites that have useful information for Objective-C developers.

## Compile Time and Run Time

There are two times that are significant when you create programs: compile time, when your source code is translated into machine language and linked together to form an executable program, and run time (also called execution time), when the executable program is run as a process on some computer. One of the characteristics that distinguishes Objective-C from other common languages, especially C++, is that Objective-C is a very dynamic language. “Dynamic” here means that decisions that other languages make at compile time are postponed to run time in Objective-C. The most prominent example

of this is Objective-C's messaging system. The section of code that a program executes when it evaluates a message expression (the equivalent of a method call in other languages) is determined at run time.

Postponing decisions until run time has many advantages, as you will see as you read this book, but it has one important drawback. It limits the amount of checking that the compiler can do. When you code in Objective-C, some errors, which would be caught at the compile stage in other languages, only become apparent at run time.

## About Memory Management

As noted earlier, Objective-C 2.0 offers you the choice between using a manual reference counting system or automatic garbage collection for managing object memory. With the exception of Chapter 15, which covers Objective-C 2.0's garbage collection system, this book uses reference counting from the beginning in all of its examples. It then provides a complete treatment of reference counting in Chapter 14.

The primary reason for this is that garbage collection is not available on iOS. If you want to write programs for the iPhone, the iPod touch, or the iPad, you must learn Objective-C's reference counting system.

Judging from the contents of various Objective-C and Cocoa mailing lists, reference counting is probably the single greatest source of confusion among people learning Objective-C. But if you learn its rules early and apply them uniformly, you will discover that reference counting isn't really difficult.

If, at a later time, you want to use garbage collection for a project, the transition should be relatively painless. Although there are some architectural issues that you need to be aware of when moving from reference counting to garbage collection (which are covered in Chapter 15), much of using garbage collection simply consists of not doing things that you have to do when using reference counting.

## About the Examples

Creating code examples for an introductory text poses a challenge: how to illustrate a point without getting lost in a sea of boilerplate code that might be necessary to set up a working program. In many cases, this book takes the path of using somewhat hypothetical examples that have been thinned to help you concentrate on the point being discussed. Parts of the code that are not relevant are omitted and replaced by an ellipsis (...).

For example:

```
int averageScore = ...
```

The preceding line of code should be taken to mean that `averageScore` is an integer variable whose value is acquired from some other part of the program. The source of `averageScore`'s value isn't relevant to the example; all you need to consider is that it *has* a value.

## About the Code Listings

The examples in this book are a mixture of unnumbered and numbered code listings.

- **Unnumbered Code Listings**

These are primarily short snippets of code that are referenced in the text that immediately precedes or follows the example.

- **Numbered Code Listings**

The numbered code listings have captions and are numbered by chapter number and their order of appearance in the chapter (e.g., Listing 4.1 or Listing 8.3). These are primarily larger examples that are referred to in text later in the chapter or in the exercises following the chapter.

In both cases, examples that require a line-by-line explanation are given line numbers so that the explanatory text can refer to a specific line in the code.

## About the Exercises

Most of the chapters in this book have a set of exercises at the end. You are, of course, encouraged to do them. Many of the exercises ask you to write small programs to verify points that were made in the chapter's text. Such exercises might seem redundant, but writing code and seeing the result provides a more vivid learning experience than merely reading. Writing small programs to test your understanding is a valuable habit to acquire; you should write one whenever you are unclear about a point, even if the book has not provided a relevant exercise.

None of the programs suggested by the exercises require a user interface; all of them can be coded, compiled, and run either by writing the code with a text editor and compiling and running them from a command line, as shown before the exercises in Chapter 2, or by using a simple Xcode project, as shown in Chapter 4.

## We'd Like to Hear from You

You can visit our website and register this book at [www.informit.com/title/9780321711380](http://www.informit.com/title/9780321711380).

Be sure to visit the book's website for convenient access to any updates, downloads, or errata that might be available for this book.

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

When you write, please be sure to include this book's title and the name of the author, as well as your name, phone, and/or email address. I will carefully review your comments and share them with the author and others who have worked on this book.

Email: [chuck.toporek@pearson.com](mailto:chuck.toporek@pearson.com)

Mail: Chuck Toporek  
Senior Acquisitions Editor, Addison-Wesley  
Pearson Education, Inc.  
75 Arlington St., Ste. 300  
Boston, MA 02116  
USA

For more information about our books or conferences, see our website at [www.informit.com](http://www.informit.com).



# Blocks

**B**locks provide a way to package up some executable code and a context (various variables) as a single entity so they can be handed off for execution at a later time or on a different thread. In other languages, blocks or similar constructs are sometimes called *closures* or *anonymous functions*. Blocks are an Apple-supplied extension to C, Objective-C 2.0, and C++. Apple has submitted blocks to the C standards working group as a proposed extension to C. At the time of this writing, blocks are only available on Mac OS X Snow Leopard (v 10.6 and on iOS 4). They are not available on earlier versions of Mac OS X or iPhone iOS.

## Note

You can use blocks on iPhone OS 3.x and on Mac OS X Leopard (v 10.5) if you install *Plausible Blocks* (PLBlocks). Plausible Blocks, a reverse-engineered port from Apple-released open-source Darwin OS code, provides the compilers and runtime required to use blocks. You can obtain Plausible Blocks from <http://code.google.com/p/plblocks/>.

Handing off a package of work is useful in many situations, but one of the main driving forces behind the adoption of blocks is Apple's new *Grand Central Dispatch* (GCD) feature. GCD is designed to make concurrency easier to program and more efficient to execute. Essentially, GCD is a thread pool that is managed for you by the operating system. The idea behind GCD is that the operating system has a global view of all the processes running on your Mac, and allocates resources (CPU, GPU, and RAM) as needed to make things run more efficiently. GCD can make better decisions than a user space program can about the number of threads to use and when to schedule them for execution. You use blocks to submit units of work for GCD to execute.

## Note

GCD provides a C interface for submitting blocks. Cocoa provides a higher-level interface to GCD through the classes `NSOperationQueue`, `NSBlockOperation`, and `NSInvocationOperation`.

`NSInvocationOperation` allows you to submit units of work as `NSInvocation` objects instead of blocks, but as you will see in the section `NSInvocation`, `NSInvocation` objects are somewhat difficult to set up. Blocks are much easier to use.

This chapter is an introduction to blocks. You will learn how to define a block, how a block has access to variables in its surrounding context, how to use a block in your own code, and about the somewhat tricky topic of memory management for blocks. The chapter also explores some pitfalls that can befall an unwary user of blocks.

Before looking at blocks in detail, the chapter takes a pair of detours and looks at two earlier ways of packaging up functionality: *function pointers* and the Foundation class `NSInvocation`.

## Function Pointers

When the compiler encounters a function call, it inserts a jump instruction to the code that performs the function. (A jump instruction causes the program execution to jump to the specified code instead of executing the line of code directly after the jump instruction.) To return, the function executes a jump instruction back to the line of code following the original function call. In a normal function call, the landing point of the jump instruction (and hence the function that is called) is static. It is determined at compile time. But a function call can be made dynamic through the use of a *function pointer*.

The following line declares `myFunctionPtr` as a pointer to a function that takes two ints as arguments and returns an int:

```
int (*myFunctionPtr) (int, int);
```

Figure 16.1 shows the anatomy of a function pointer.

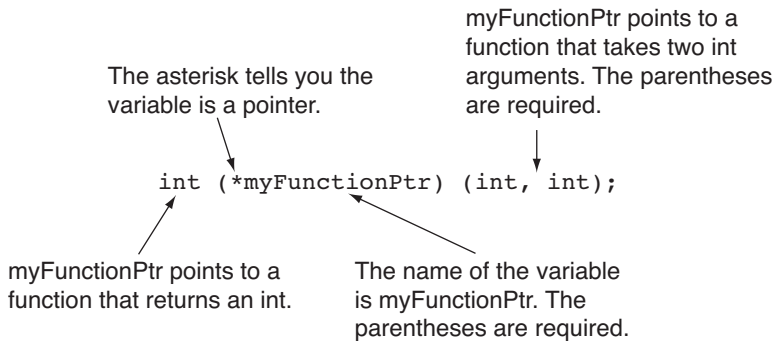


Figure 16.1 The anatomy of a function pointer

The general form of a function pointer is:

```
return_type (*name)(list of argument types);
```

Function pointers are a low point in C syntax. Instead of reading left-to-right or right-to-left, they read from the inside out. More complicated function pointer declarations can quickly turn into puzzles, as you will see in Exercise 16.1.

## A Syntax Quirk in Objective-C

The “from the inside out” declaration style of function pointers doesn’t fit with Objective-C’s syntax for method arguments. Recall that Objective-C requires the type for a method argument to be enclosed in parentheses before the argument name. This conflict is resolved in favor of the syntax for method arguments.

When declaring a method argument that is a function pointer, the name comes outside. For example, a pointer to a function that has no arguments or return value is normally declared as follows:

```
void (*functionPtr)(void);
```

A function that takes a function pointer of preceding type as an argument is declared like this:

```
void functionWithFPArg(void (*fp)(void));
```

But a method with the same argument type is declared like this:

```
-(void) methodWithFFPArg:(float*)(float) fp;
```

Putting the name of the function pointer last in a function pointer declaration only works when declaring the type of a method argument. Putting the name last results in a compiler error in any other situation.

You can also declare arrays of function pointers. The following line declares `fpArray` as an array of 10 pointers to functions. Each function takes a single argument, a pointer to `float`, and returns `void`:

```
void (*fpArray[10])(float*);
```

A function pointer can point to a function that has another function pointer as an argument or a return value:

```
void *(*complicatedFunctionPointer)(void)(void);
```

`complicatedFunctionPointer` is a pointer to a function that takes no arguments and returns a pointer to a function that takes no arguments and returns `void`.

Declarations like the preceding one are ugly, but you can make your code cleaner by hiding the ugliness with a `typedef`:

```
typedef void *(*complicatedFunctionPointer)(void)(void);  
complicatedFunctionPointer fp;
```

## Calling a Function with a Function Pointer

The following example shows how to assign a function to a function pointer and how to call the function using the function pointer:

```
void logInt( int n )  
{  
    NSLog("The integer is: %d", n);  
}
```

```
void (*myFunctionPtr)(int); // Declare a function pointer

myFunctionPtr = logInt;    // Make it point to logInt
myFunctionPtr( 5 );      // Execute the function through the pointer
```

To make the function pointer refer to a function, you simply assign it the name of the function. The function must be defined or visible by a forward declaration at the point it is assigned.

To call a function through a function pointer, you simply add the arguments, enclosed in parentheses, to the function pointer. A function call through a function pointer is just like a normal function call except that you use the name of the function pointer variable instead of the function name, as shown in the previous code snippet.

### Note

There is no need to use the address of operator (&) or the dereferencing operator (\*) with function pointers. The compiler knows which names are functions or function pointers and which names are regular variables. However, if you would like to use them, you may, as shown here:

```
myFunctionPtr = &logInt; // The same as myFunctionPtr = logInt;

(*myFunctionPtr)( 5 ) // The same as myFunctionPtr( 5 );
```

The compiler doesn't care which form you use.

## Using Function Pointers

One of the primary uses of function pointers is for *callbacks*. Callbacks are used in situations where you have a function or method that is going to do some work for you, but you would like the opportunity to insert your own code somewhere in the process. To do this, you pass the working function or method a pointer to a function containing the code you want executed. At the appropriate time, the working function or method will call your function for you.

For example, `NSMutableArray` provides the following method for use in custom sorting:

```
- (void)sortUsingFunction:
    (NSInteger (*)(id, id, void *))compare
    context:(void *)context
```

When you invoke `sortUsingFunction:context:`, the method sorts the contents of the receiver. To perform the sort, `sortUsingFunction:context:` must examine pairs of array elements and decide how they are ordered. To make these decisions, `sortUsingFunction:context:` calls the `compare` function that you passed in by pointer when the method was invoked.

The `compare` function must look at the two objects it receives and decide (based on whatever criterion you require) whether they are ordered `NSOrderedAscending`, `NSOrderedSame`, or `NSOrderedDescending`.

### Note

`NSOrderedAscending`, `NSOrderedSame`, and `NSOrderedDescending` are integer constants defined by the Foundation framework.

`sortUsingFunction:context:` also passes `compare` the `void*` pointer that it received as its `context` argument. This is a pure pass-through; `sortUsingFunction:context:` doesn't look at or modify `context`. `context` may be `NULL` if `compare` doesn't require any additional information.

Listing 16.1 sorts an array containing some `NSNumber` objects. The address of a `BOOL` is passed in to control the direction of a numerical sort.

Listing 16.1 `ArraySortWithFunctionPointer.m`

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

NSInteger numericalSortFn( id obj1, id obj2, void* ascendingFlag )
{
    int value1 = [obj1 intValue];
    int value2 = [obj2 intValue];
    if ( value1 == value2 ) return NSOrderedSame;
    if ( *(BOOL*) ascendingFlag )
    {
        return ( value1 < value2 ) ?
            NSOrderedAscending : NSOrderedDescending;
    }
    else
    {
        return ( value1 < value2 )
            ? NSOrderedDescending : NSOrderedAscending;
    }
}

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // Put some number NSNumber objects in an array
    NSMutableArray *numberArray = [[NSMutableArray alloc] initWithCapacity: 5];
    [numberArray addObject: [NSNumber numberWithInt: 77]];
    [numberArray addObject: [NSNumber numberWithInt: 59]];
    [numberArray addObject: [NSNumber numberWithInt: 86]];
}
```

```

[numberArray addObject: [NSNumber numberWithInt: 68]];
[numberArray addObject: [NSNumber numberWithInt: 51]];

NSLog( @"Before sort: %@", [numberArray description] );

// This flag controls the sort direction.
// Change it to NO to sort in descending order.
    BOOL ascending = YES;

// Sort the array
[numberArray sortUsingFunction: numericalSortFn context: &ascending];

NSLog( @"After sort: %@", [numberArray description] );

[numberArray release];
[pool drain];
return 0;
}

```

---

Notice:

- `ascendingFlag` is passed in as `void*`. It must be cast as `BOOL*` before it can be dereferenced to get the `BOOL` value.
- The name of a function, in this case `numericalSortFn`, can serve as a properly typed pointer to that function. Here, it is used as the argument when invoking `sortUsingFunction:context:` without defining a separate function pointer variable.

## The Trouble with Function Pointers

There is one large inconvenience with using function pointers as callbacks or in any situation where you are trying to hand off some code for execution by another part of the program or another thread. Any context that the function requires must be packed up and submitted as a separate variable or variables.

Most designs using callbacks use the pattern shown in Listing 16.1. The function or method that is passed the callback function accepts a blind pointer as an additional argument and then passes that pointer back to the callback function, as shown on Figure 16.2. This is only a minor inconvenience when the context is a single variable as in the preceding example. However, if your function requires a more complicated context, you must create and load a custom structure to hold the context and then pass the pointer to that structure as the context variable. As an alternative, you could package the context variables in an `NSDictionary` and then pass the dictionary as the context. Either way is awkward if the context involves many variables.

```
void (*callbackFnPtr)(callbackInfo*)=...;
callbackContext* someContext=...;
```

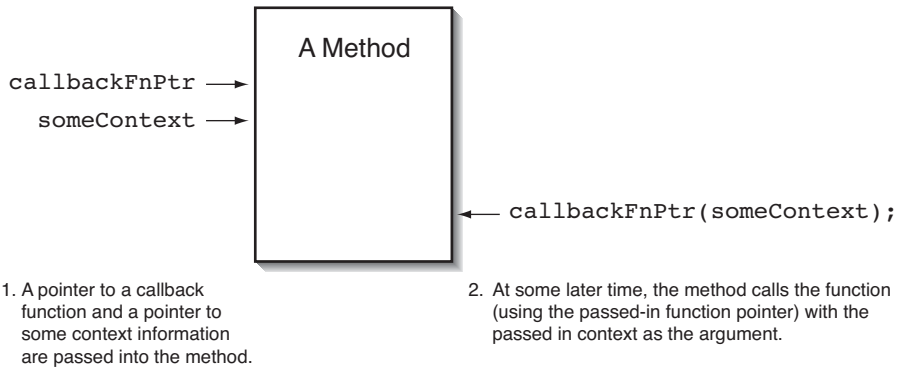


Figure 16.2 Passing a callback function to a method

## NSInvocation

An `NSInvocation` object takes an Objective-C message and turns it into an object. The invocation object stores the message’s receiver (called the *target* in invocation-speak), selector, and arguments. An invocation object can be saved for later execution or passed on to another part of your code. When you send the invocation object an `invoke` message, the invocation sends the target object a message using the stored selector and arguments.

As an example, consider a `LineGraphic` class with a method `drawWithColor:width:` that draws a line with a specified color and line width:

```
LineGraphic *graphic = ...
[graphic drawWithColor: [NSColor redColor] width: 2.0];
```

Listing 16.2 shows how to turn the preceding message into an invocation.

### Listing 16.2 Constructing an NSInvocation

```
LineGraphic *graphic = ...

NSInvocation *drawInvocation =
    [NSInvocation invocationWithMethodSignature:
     [graphic methodSignatureForSelector:
      @selector(drawWithColor:width:)]];

[drawInvocation setTarget: graphic];
[drawInvocation setSelector: @selector(drawWithColor:width:)];

[drawInvocation retainArguments];
```

```

NSColor *color = [NSColor redColor];
float linewidth = 2.0;

[drawInvocation setArgument: &color atIndex: 2];
[drawInvocation setArgument: &linewidth atIndex: 3];

```

---

To set up an invocation, `NSInvocation` needs to know the return type of the message being encapsulated and the types of the message's arguments. The message's selector is just a name and doesn't carry any type information, so you must obtain the type information by calling the target's `methodSignatureForSelector:`. This is a method that all classes inherit from `NSObject`. It returns an `NSMethodSignature` object, which is an encoded representation of the selector's return type and argument types. Finally, you pass the returned `NSMethodSignature` to `NSInvocation`'s `invocationWithMethodSignature:` class method to create the invocation.

### Note

In Listing 16.2, the call to `methodSignatureForSelector:` is nested inside the call to `invocationWithMethodSignature:` so there is no explicit `NSMethodSignature` variable.

Next, you set the invocation's target and selector with `setTarget:` and `setSelector:`.

An `NSInvocation` does not retain its target or any of its arguments by default. If you are going to save an invocation for future execution, you should ask the invocation to retain its target and arguments by sending the invocation a `retainArguments` message. This prevents the target and arguments from being released before the invocation is invoked.

The arguments for the encapsulated message are set with the `setArgument:atIndex:` method:

- You pass the *address* of the variable being used for the argument, not the variable itself. You can't use a value directly in `setArgument:atIndex:` message:
 

```
[drawInvocation setArgument: 2.0 atIndex: 3]; // WRONG!
```
- If the selector has an argument that is not an object (an argument that is a primitive C type such as `int` or `float`), you may use the address of the primitive type directly. There is no need to wrap the `width` argument in an `NSNumber` object.
- The arguments are identified by their position in the message. Notice that indices start at 2. Indices 0 and 1 are reserved for the hidden method arguments `self` and `_cmd`. (For an explanation of a method's hidden arguments, see Chapter 5, "Messaging.")

Now that you have created `drawInvocation`, you can store it or hand it off to other code. When you are ready to draw the line, you simply execute the following line of code:

```
[drawInvocation invoke];
```

An invocation like the preceding one could be used as part of a display list scheme in a drawing program. Each invocation, stored in an array, encapsulates the message required



to draw an element in the final scene. When you are ready to draw the scene, you loop through the array of invocations and invoke each one in turn:

```
NSMutableArray *displayList = ...
for (NSInvocation invocation in displayList)
{
    [invocation invoke];
}
```

Two additional points:

- An `NSInvocation` can be invoked any number of times.
- It is possible to encapsulate a message with a return value in an `NSInvocation`. To get the return value, send the invocation a `getReturnValue:` message, as illustrated here:

```
double result;
[invocationReturningDouble getReturnValue: &result];
```

The argument to `getReturnValue:` must be a pointer to a variable of the same type as the invocation's return type. If you send a `getReturnValue:` message to an invocation object before it has been sent an `invoke` message, the result is undefined. The value is garbage and may cause a program crash if you attempt to it for anything.

`NSInvocation` objects are used in the Cocoa framework to schedule an operation to be performed after a time interval (`NSTimer`), and in the Cocoa undo mechanism (`NSUndoManager`). `NSInvocation` objects solve one of the problems of using function pointers; they carry at least some of their context with them in the form of the arguments to the encapsulated message. That said, `NSInvocation` objects have a major drawback—as you have seen in Listing 16.2, they are difficult to construct.

## Blocks

Blocks are similar in many ways to functions, with the important exception that blocks have direct access to variables in their surrounding context. “Direct access” means that a block can use variables declared in its surrounding context even though those variables are not passed into the block through an argument list. Blocks are declared as follows:

```
^(argument_list){ body };
```

Blocks begin with a caret (^), followed by an argument list enclosed in parentheses and one or more statements enclosed in curly brackets. This expression is called a *block literal*, and is defined as follows:

- Blocks can return a value by using a `return` statement in the body.
- You don't have to specify the return type; the compiler determines the return type by looking at the body of the block.

- If the block doesn't have arguments, the argument list is written as `(void)`.
- Block literals don't have a name; that is why they are sometimes called *anonymous functions*.

The following is a simple block example that takes an integer, doubles it, and returns the doubled value:

```
^(int n){ return n*2; };
```

You can call a block literal directly by appending values for its arguments surrounded in parentheses:

```
int j = ^(int n){ return n*2; }( 9 ); // j is now 18
```

This works, but it is a bit silly; you could get the same result by simply coding the following:

```
int j = 2 * 9;
```

In normal use, a block literal is assigned to a variable typed as pointer to a block, or used as an argument to a function or a method that takes a pointer to block argument.

## Block Pointers

A variable that is a block pointer (a pointer to a block) is declared as follows:

```
return_type (^name)(list of argument types);
```

This should look familiar; it has exactly the same form as the declaration of a function pointer, except that the `*` has been replaced with a `^`.

### Note

Don't refer to a variable that holds a block pointer as a block variable. As you will see later in the chapter, the term *block variable* is reserved for a different entity.

The following example illustrates how to declare a block pointer, assign a block to it, and then call the block through the block pointer:

```
int (^doubler)(int); // doubler is typed as a pointer to a block
```

```
doubler = ^(int n){ return n*2; };
```

```
int j = doubler( 9 ); // j is now 18
```

You can use block pointers as arguments to a function or a method:

```
// someFunction is a function that takes a block pointer as an argument
```

```
void someFunction( int (^blockArg)(int) );
```

```
int (^doubler)(int)= ^(int n){ return n*2; };
```

```
someFunction( doubler );
```

You can also use a block literal directly in a function call or an Objective-C message expression:

```
void someFunction( int (^blockArg)(int) );
```

```
someFunction( ^(int n){ return n*2; } );
```

### Note

Objective-C method declarations have the same quirk with block pointers as they do with function pointers: When declaring a method that takes a block pointer argument, the name comes outside the type declaration:

```
- (void) doSomethingWithBlockPointer:
    (float (^)(float)) blockPointer;
```

## Access to Variables

A block has:

- Read-only access to automatic variables visible in its enclosing scope<sup>1</sup>
- Read/write access to static variables declared in a function and to external variables
- Read/write access to special variables declared as *block variables*

Here is a simple example of a block accessing a local variable in its enclosing scope:

```
int j = 10;
```

```
int (^blockPtr)(int) = ^(int n){ return j+n; };
```

```
int k = blockPtr( 5 ); // k is now 15
```

The value that a block uses for a local variable from its context is the value that local variable has when the flow of execution passes over the block literal, as shown here:

```
1 int j = 10;
```

```
2
```

```
3 int (^blockPtr)(int) = ^(int n){ return j+n; };
```

```
4
```

```
5 j = 20;
```

```
6
```

```
7 int k = blockPtr( 5 ); // k is 15, not 25 as you might expect
```

1. The scope of various classes of variables is discussed in Chapter 2, *More About C Variables*.

In the preceding code:

- Line 1: The local variable `j` is set to 10.
- Line 3: The block `blockPtr` is defined. `blockPtr` has access to the local variable `j`. The value of `j` that `blockPtr` uses is bound to the value that `j` has when the program execution passes over this line. In effect, `blockPtr` now has a private copy of `j` that is set to 10.
- Line 5: To show that `blockPtr`'s value of `j` was bound in Line 3, `j` is reset to 20.
- Line 7: The `blockPtr` is evaluated with an argument of 5 resulting in a return value of 10 (the value of `j` at the time Line 3 is executed) + 5 (the argument) = 15.

A block's access to local variables is read-only. The following code, which attempts to set the value of `j` from inside `blockPtr`, results in a compiler error because `blockPtr`'s access to the local variable `j` is read-only:

```
int j = 10;

void (^blockPtr)(void) = ^(void){ j = 20; };
```

### Note

If a local variable holds a pointer to an object, a block cannot change the variable to point to a different object. But the object that the variable points to can still be modified by the block:

```
NSMutableArray *localArray = ...

void (^shortenArray)(void) = ^(void){ [localArray removeLastObject]; };

// Removes the last object in localArray

shortenArray();
```

The compiler gives blocks access to `static` and external variables by pointer. The value the block sees for this type of variable is the value the variable has when the block is executed, not when the block is defined:

```
static int j = 10;

int (^blockPtr)(int) = ^(int n){ return j+n; };

j = 20;

int k = blockPtr ( 5 ); // k is 25
```

## Block Variables

Variables declared with the new type modifier `__block` are called *block variables*:

```
__block int integerBlockVariable;
```

Block variables are visible to, and are shared and mutable by, any block defined in their scope. For example:

```
1 __block int j = 10;
2
3 void (^blockPtr_1)(void) = ^(void){ j += 15; };
4 void (^blockPtr_2)(void) = ^(void){ j += 25; };
5
6 blockPtr_1(); // j is now 25
7 blockPtr_2(); // j is now 50
```

In the preceding code:

- Line 1: `j` is declared as a `__block` variable and set to 10.
- Line 3: `blockPtr_1` is defined. Because `j` has been declared as a `__block` variable, `blockPtr_1` is permitted to set the value of `j`.
- Line 4: Similarly, `blockPtr_2` is permitted to set the value of `j`. Both `blockPtr_1` and `blockPtr_2` share read-write access to the variable `j`.
- Line 6: `blockPtr_1` is evaluated, incrementing `j` by 15, resulting in a value of 25 for `j`.
- Line 7: `blockPtr_2` is evaluated, incrementing `j` by 25, resulting in a value of 50 for `j`.

Block variables start out on the stack like any other automatic variable. But if you copy a block that references a block variable, the block variable is moved from the stack to the heap along with the block (see the section *Copying Blocks*).

#### Note

The term *block variable* refers to a variable that is declared with the `__block` modifier, as described in the preceding paragraphs. Don't confuse "block variable" and "variable that holds a pointer to a block." A block variable is not a block.

## Blocks Are Stack Based

When you define a block literal inside a function or a method, the compiler creates a structure on the stack that holds the values of any local variables the block references, the addresses of read/write variables it references, and a pointer to block's executable code.

#### Note

The block structure is created on the stack, but the block's executable code is not on the stack. It is in the text portion of the program along with all the other executable code.

Blocks have the same lifetime as automatic variables. When the block literal goes out of scope, it is undefined, just like an automatic variable that has gone out of scope. Scope for a block literal is defined in the same way as scope for an automatic variable (see

Chapter 2, “More About C Variables”): If a block literal is defined inside a compound statement (between a pair of curly brackets), the block literal goes out of scope when the program execution leaves the compound statement. If a block literal is defined in a function, it goes out of scope when the function returns. The following code is incorrect:

```
int (^doubler)(int);
{
    ...
    doubler = ^(int n){ return n*2; };
    ...
}
...
int j = doubler( 9 ); // WRONG! Bug!
```

In the preceding example, `j` is undefined. At the point where `doubler(9)` is executed, the block that the `doubler` variable points to has gone out of scope and the block may have been destroyed.

### Note

If you try the preceding example, it may appear to work correctly. `j` may very well be set to 18. But this would be an accident of the way the compiler has arranged the code in this instance. After the block is out of scope, the compiler is free to reuse the space the block occupied in the stack frame. If the compiler has reused the space, the result of trying to execute the out-of-scope block would be an incorrect value of `j` or, more likely, a crash.

## Global Blocks

You can also assign a block literal to a file-scope block pointer variable:

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

void (^logObject)(id) =
    ^(id obj){ NSLog( @"Object Description: %@", [obj description] ); };

// logObject( someObj ) may be used anywhere in the file.
```

The compiler creates global-scope blocks in low memory like any other file-scope variable. Global blocks never go out of scope.

## Blocks Are Objective-C Objects

It may seem surprising, but blocks are also Objective-C objects. A newly created block is the only example of an Objective-C object that is created on the stack. Blocks are instances of one of several private subclasses of `NSObject`. Apple doesn't provide the header for the block classes so you can't subclass them or do much of anything with them in an Objective-C sense except send them `copy`, `retain`, `release`, and `autorelease` messages. Copying and memory management for blocks are covered in the next sections.

## Copying Blocks

One of the main uses of blocks is to pass a chunk of work (some code plus some context) out of the current scope for processing at a later time. Passing a block to a function or method that you call is safe (as long as that function or method is going to execute on the same thread). But what happens if you want to pass a block to a different thread or pass a block out of the current scope as a return value? When the current function or method returns, its stack frame is destroyed. Any blocks that were defined in its scope become invalid.

To preserve a block, you must copy it. When you copy a block, the copy is created on the heap. The heap-based copy is then safe to return up the stack to the calling function or pass off to another thread.

If you are coding in straight C, you can use the `Block_copy()` function, as follows:

```
int(^doublerCopy)(int) = Block_copy( ^(int n){ return n * 2; } );
```

In Objective-C, you can send a `copy` message to the block:

```
int(^doublerCopy)(int) = [^(int n){ return n * 2; } copy];
```

The two preceding examples are equivalent. In either statement, you could use a block pointer instead of the block literal.

When you copy a block, the new block gets copies of the values of any automatic variables that the block references. (The block accesses automatic variables by value. The value of the variable is copied into the block object when it is created.)

### But What About Block Variables?

Block variables are accessed by reference. It wouldn't be very useful to copy a block and then leave the copied block referring to a variable that is destroyed when the program execution leaves the current scope.

To remedy this, when you copy a block, the compiler also moves any block variables that the block references from the stack to a location on the heap. The compiler then updates any blocks that reference the block variable so they have the variable's new address.

One consequence of the compiler's behavior in this situation is that it is a *very* bad idea to take the address of a block variable and use it for anything. After the copy operation, the original address refers to a memory location that may now be garbage.

## Memory Management for Blocks

If you copy a block with `Block_copy()`, you must eventually balance that call with a call to `Block_release()`. If you use the Objective-C copy message and you are using reference counting, you must balance the `copy` message with a `release` or an `autorelease`:

```
int(^getDoublerBlock()(int)
{
    int(^db)(int) = ^(int n){ return 2*n; };

    // The returned block is autoreleased. This balances the copy
    // and makes getDoublerBlock conform to the naming convention
    // for memory management.
    return [[db copy] autorelease];
}
...
```

```
int(^doubler )(int) = getDoublerBlock(); // Get the block
```

```
int sevenDoubled = doubler(7); // Use the block
```

Don't mix calls to `Block_copy()` and `Block_release()` with the Objective-C's `copy` and `release` messages.

If a block references a variable that holds an object, that object is retained when the block is copied and released when the block is released.

#### Note

An object held in `__block` variable is *not* retained when a block that references it is copied.

When copying a block inside a method body, the rules are slightly more complicated:

- A direct reference to `self` in a block that is being copied causes `self` to be retained.
- A reference to an object's instance variable (either directly or through an accessor method) in a block that is being copied causes `self` to be retained.
- A reference to an object held in a local variable in a method causes that object, but *not* `self`, to be retained.

You should be careful when copying a block. If the code that copies the block is inside a method and the block refers to any of the object's instance variables, the copy causes `self` to be retained. It is easy to set up a retain cycle that prevents the object from ever being deallocated,

Listing 16.3 shows the interface section for a class that has an instance variable `name` to store a name, and a method `logMyName` to log that name. `logMyName` uses a block stored in the instance variable `loggingBlock` to do the actual logging.

#### Listing 16.3 `ObjectWithName.h`

---

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

@interface ObjectWithName : NSObject
{
```



```

    NSString *name;
    void (^loggingBlock)(void);
}

- (void) logMyName;
- (id) initWithName:(NSString*) inName;

@end

```

---

Listing 16.4 shows the corresponding implementation file.

#### Listing 16.4 **ObjectWithName.m**

---

```

1 #import "ObjectWithName.h"
2
3 @implementation ObjectWithName
4
5 - (id) initWithName:(NSString*) inputName
6 {
7     if (self = [super init] )
8     {
9         name = [inputName copy];
10        loggingBlock = [^(void){ NSLog( @"%@", name ); } copy];
11    }
12    return self;
13 }
14
15 - (void) logMyName
16 {
17     loggingBlock();
18 }
19
20 - (void) dealloc
21 {
22     [loggingBlock release];
23     [name release];
24     [super dealloc];
25 }

```

---

`ObjectWithName` is a very simple class. However, this version of `ObjectWithName` has a retain cycle. If you create an `ObjectWithName` object, it won't be deallocated when you release it.

The problem is Line 10 of Listing 16.4:

```
loggingBlock = [^(void){ NSLog( @"%@", name ); } copy];
```

To store the block in the instance variable `loggingBlock`, you must copy the block literal and assign the copy to the instance variable. This is because the block literal goes out of scope when `initWithName:` returns. Copying the block puts the copy on the heap (like a normal Objective-C object). However, the block literal references the instance variable `name`, so the `copy` causes `self` to be retained, setting up a retain cycle. The block now has ownership of the object and the object has ownership of the block (because it has copied the block). The object's reference count never goes to zero and its `dealloc` method is never called.

You can fix this problem by changing Line 10 of Listing 16.4 so it reads as follows:

```
loggingBlock = [^(void){ NSLog( @"%@", inputName ); } copy];
```

With this change, the block copying operation retains the input argument `inputName` rather than the instance variable `name`. Because the block no longer references any of the object's instance variables, `self` is not retained and there is no retain cycle. The object will still have the same behavior because `name` and `inputName` have the same content.

### Note

The preceding rules are presented separately for blocks copied inside a method to emphasize the consequences of a block accessing an object's instance variable. But there is really no significant difference between copying a block inside or outside a method. The only difference is that, outside of a method, there is no way to reference an object's instance variables without referencing the object itself.

## Traps

Because blocks are stack-based objects, they present some traps for the unwary programmer. The following snippet of code is incorrect:

```
void(^loggingBlock)(void);

BOOL canWeDoIt = ...

// WRONG

if ( canWeDoIt )
    loggingBlock = ^(void){ NSLog( @"YES" ); };
else
    loggingBlock = ^(void){ NSLog( @"NO" ); };

// Possible crash
loggingBlock();
```

At the end of this snippet, `loggingBlock` is undefined. The `if` and `else` clauses of an `if` statement and the bodies of loops are separate lexical scopes, *even if they are single statements and not compound statements*. When the program execution leaves the scope, the compiler is free to destroy the block and leave `loggingBlock` pointing at garbage.

To fix this code, you must `copy` the block, and then remember to release it when you are finished:

```
void(^loggingBlock)(void);

BOOL canWeDoIt = ...

if ( canWeDoIt )
    loggingBlock = [^(void){ NSLog( @"YES" ); } copy];
else
    loggingBlock = [^(void){ NSLog( @"NO" ); } copy];

// Remember to release loggingBlock when you are finished
```

This example is also incorrect:

```
NSMutableArray *array = ...

// WRONG!

[array addObject: ^(void){ doSomething; }];
return array; //
```

Recall that objects added to collection objects receive a `retain` message; however in this case, the `retain` doesn't help because `retain` is a no-op for a stack-based block. Again, to fix the problem, you must `copy` the block:

```
NSMutableArray *array = ...
[array addObject: [^(void){ doSomething; } copy] autorelease];
return array;
```

In the preceding code snippet, the `copy` message puts a copy of the block on the heap. The `autorelease` message balances the `copy`. The `retain` message that the copied block receives when it is placed in the array is balanced by a `release` message when the block is later removed from the array or when the array is deallocated.

## Blocks in Cocoa

Beginning with Mac OS X Snow Leopard (v 10.6), Apple has started deploying blocks throughout the Cocoa frameworks. This section briefly describes three areas where Apple has added features that use blocks.

### Concurrency with `NSOperationQueue`

Concurrent (multithreaded) programming is very difficult to do correctly. To make it easier for programmers to write error-free multithreaded programs, Apple has introduced Grand Central Dispatch (GCD). GCD implements concurrency by creating and managing a *thread pool*. A thread pool is a group of threads that can be assigned to various tasks and reused when the task is finished. GCD hides the details of managing the thread pool and presents a relatively simple interface to programmers.

The Cocoa class `NSOperationQueue` provides a high-level interface to GCD. The idea is simple: You create an `NSOperationQueue` and add units of work, in the form of blocks, for the queue to execute. Underneath `NSOperationQueue`, GCD arranges to execute the block on a separate thread:

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];

[queue addOperationWithBlock: ^(void){ doSomething; } ];

// doSomething will now execute on a separate thread
```

- A block passed to GCD (either through `NSOperationQueue` or through the low-level C interface) must have the form:

```
void (^block)(void)
```

It must not take arguments or return a value.

- The GCD mechanism takes care of copying blocks submitted to it and releases them when no longer needed.

#### Note

Programming concurrency is a complex topic. For a complete discussion of `NSOperationQueue` and GCD, see Apple's *Concurrency Programming Guide*.<sup>2</sup>

### Collection Classes

The Foundation collection classes now have methods that enable you to apply a block to every object in the collection. `NSArray` has the following method:

```
- (void)enumerateObjectsUsingBlock:
    (void (^)(id obj, NSUInteger idx, BOOL *stop))block
```

This method calls `block` once for each object in the array; the arguments to `block` are:

- `obj`, a pointer to the current object.
- `idx`, the index of the current object (`idx` is the equivalent of the loop index in an ordinary `for` loop).
- `stop`, a pointer to a `BOOL`. If the block sets `stop` to `YES`, `-enumerateObjectsUsingBlock:` terminates when the block returns. It is the equivalent of a `break` statement in an ordinary C loop.

Listing 16.5 uses `-enumerateObjectsUsingBlock:` to log a description of every object in an array.

2. <http://developer.apple.com/mac/library/documentation/General/Conceptual/ConcurrencyProgrammingGuide>

## Listing 16.5 DescribeArrayContents.m

---

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

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSArray *array =
        [NSArray arrayWithObjects: @"dagger", @"candlestick",
         @"wrench", @"rope", nil];

    void (^loggingBlock)(id obj, NSUInteger idx, BOOL *stop) =
        ^(id obj, NSUInteger idx, BOOL *stop)
        { NSLog( @"Object number %d is a %@",
                 idx, [obj description] ); };

    [array enumerateObjectsUsingBlock: loggingBlock];

    [pool drain];
    return 0;
}
```

---

If you build and run this program, you should see the following result:

```
DescribeArrayContents [50642:a0b] Object number 0 is a dagger
DescribeArrayContents [50642:a0b] Object number 1 is a candlestick
DescribeArrayContents [50642:a0b] Object number 2 is a wrench
DescribeArrayContents [50642:a0b] Object number 3 is a rope
```

### Did-End Callbacks

I haven't said much about AppKit in this book, but I'll assume that you are familiar with saving files on Mac OS X. You select **File > Save** in an app, and if this is the first time the file is saved, a save sheet appears so you can name the file and select the location where it will be saved. You make your choices and click Save, or if you've changed your mind, you can click Cancel. After clicking one of the buttons, the sheet slides up and disappears.

When you invoke the method that begins the sheet, Cocoa gives you the chance to register some code to be executed when the user dismisses the sheet. (This is where you put the code that actually saved the file to disk.)

Prior to Mac OS X Snow Leopard (v 10.6), a Save sheet was started with this rather formidable method:

```
- (void)beginSheetForDirectory:(NSString *)path
    file:(NSString *)name
    modalForWindow:(NSWindow *)docWindow
```

```
modalDelegate:(id)modalDelegate
didEndSelector:(SEL)didEndSelector
contextInfo:(void *)contextInfo
```

When the user dismisses the sheet, the sheet sends the object registered as `modalDelegate`, the message represented by the selector `didEndSelector`. Typically, the `modalDelegate` is the object that initiates the panel. `didEndSelector` has the form:

```
- (void)savePanelDidEnd:(NSSavePanel *)sheet
    returnValue:(int)returnCode
    contextInfo:(void *)contextInfo;
```

- `sheet` is a pointer to the `NSSavePanel` object itself.
- `returnValue` is an integer that specifies which button the user clicked on.
- `contextInfo` is a blind pointer to the information passed to `beginSheetForDirectory: ...` when it was invoked. This is how you pass information from the object that invoked the sheet to the code responsible for acting on the user's input.

For Mac OS X Snow Leopard and beyond, the preceding method has been deprecated<sup>3</sup> and replaced with the following method:

```
- (void)beginSheetModalForWindow:(NSWindow *)window
    completionHandler:(void (^)(NSInteger result))handler
```

You simply pass in a block to be executed when the sheet is dismissed. The block can capture any required context so the blind `contextInfo` pointer is not required.

#### Note

The `file` and `directoryPath` arguments were removed as part of a separate cleanup that doesn't involve blocks.

## Style Issues

Placing the statements of a block literal on a single line makes debugging difficult; for example:

```
^(void){doStuff; doMoreStuff; evenMore; keepGoing; lastStatement;}
```

You can set a debugger breakpoint on `doStuff`; but there is no way to step through or set a breakpoint on any of the other statements in the block. If you stop on `doStuff`; and try to step, the debugger jumps to the line following the block literal—making it

3. A method whose status is changed to deprecated in a given major OS release is still available in that release but may be withdrawn in a future major OS release. For example, a method marked as deprecated in Mac OS X 10.6 may not be available in Mac OS X 10.7.

impossible to debug the subsequent lines in the literal. If your block literal is non-trivial and may require debugging, you should put the statements in the block's body on separate lines, as follows:

```
^(void){doStuff;
    doMoreStuff;
    evenMore;
    keepGoing;
    lastStatement;}
```

As noted earlier, you can place a block literal directly in a function or method call:

```
someFunction( otherArgs, ^(void){ doStuff;
    doMoreStuff;
    evenMore;
    keepGoing;
    lastStatement;} );
```

You could also assign the block to a block pointer variable and use the block pointer as the argument. Which you choose is a matter of preference: Some people are annoyed at creating an extra variable (which the compiler will probably optimize away), whereas others find that putting the block literal inside the function call makes the code hard to read.

## Some Philosophical Reservations

Blocks are very versatile and they are clearly an important part of Apple's plans for the future of Objective-C and Mac OS X. However, blocks come with a few issues that are worth a moment or two of thought:

- The term “block” was already in use. It is used as interchangeable with “compound statement” in almost every book on the C language. This might cause confusion in some circumstances.
- Blocks are *function oriented* and not very *object oriented*. This may be an issue if you are strongly attached to an ideal of object-oriented purity.
- Blocks completely break encapsulation. A block's access to variables that are not accessed through an argument list or an accessor method presents many of the same issues as using global variables.

Using `__block` variables and copying blocks can result in entangled objects: You can create separate objects (potentially belonging to different classes) communicating via a variable on the heap that is not visible to anything else.

- As with operator overloading in C++, blocks can be used in ways that lead to *Design Your Own Language Syndrome*, code that is very terse but very difficult for others (or yourself, several months later) to read and understand. This may not be an issue for independent developers, but it can be a problem if you are part of a programming team.

## Summary

This chapter looked at several ways of packaging functionality to be executed at a later time or on a different thread. Function pointers let you hand off functions but require that you provide an extra variable to go with the function pointer if you need to pass some context to go along with the function. `NSInvocation` objects wrap the target, the selector, and the arguments of an Objective-C message expression in a single object that can then be stored or handed off for later execution. They are easy to use but difficult to construct.

Blocks, an Apple-added extension to C, Objective-C 2.0, and C++, wrap a series of statements and the variables in their surrounding context in a single entity. Grand Central Dispatch, Apple's system for managing concurrency, uses blocks as the medium for submitting tasks to be executed on other threads. Beginning with Mac OS X Snow Leopard (v 10.6), Apple is deploying blocks throughout the Cocoa frameworks to replace older methods that used `NSInvocation` objects or required separate target, selector, and context arguments for callbacks.

## Exercises

1. This is more of a puzzle than anything else, but it will test your understanding of function pointer (and, by extension, block pointer) declarations. Consider the following declaration:
 

```
int (*(myFunctionPointer)(int (*)(int))) (int);
```

 What (in words) is `myFunctionPointer`?
2. Rewrite the `HelloObjectiveC` program from Chapter 4, “Your First Objective-C Program,” to use an `NSInvocation`:
 

Instead of passing the `Greeter` the greeting text as an `NSString`, create a `Greeting` class that encapsulates the greeting and a method that issues the greeting. (The method should take the greeting string as an argument and log it.)

Package up issuing the greeting as an `NSInvocation` and pass it to the `Greeter`. The `Greeter` should then issue the greeting by sending the invocation object the `-invoke` message.
3. Write a program that uses some simple blocks and verify for yourself that:
  - The value for an ordinary automatic value that a block sees is fixed when execution passes over the block literal, and is unchanged if the value of the variable is changed later in the code.
  - A block cannot modify the value of an ordinary automatic variable in its scope.
  - A block can both read and set a variable declared with the `__block` type modifier.



4. Rewrite the program in Listing 16.1 to use a block instead of a function. Use the `NSMutableArray` method:

```
- (void)sortUsingComparator:(NSComparator)cmptr
```

`NSComparator` is a typedef for a pointer to a block that takes two object arguments and returns the same integer constants as the function in Listing 16.1.

5. Write a program that looks for a name in an array of names and reports back the name's index in the array:
- Create an `NSArray` with some names (use `NSString` objects).
  - Create a local `NSString` variable to hold the name you are searching for and an integer block variable to report back at what index the name was found.
  - Search the array using `-enumerateObjectsUsingBlock:`.
  - Make sure your block uses the `stop` argument to stop looking when the name is found.
  - If the name you are looking for isn't in the array, the block variable holding the index should have a value of `-1`.
6. Write a program that uses the `ObjectWithName` class (see Listings 16.3 and 16.4):
- Add a logging statement to `ObjectWithName`'s `dealloc` routine.
  - In your main program, allocate an instance of `ObjectWithName`.
  - Release the object and verify that it is never deallocated.
  - Make the fix suggested in the text and verify that the object now deallocates.

# Index

---

## Symbols

---

- (negation operator), 18
- // (forward slashes), comment syntax, 5
- /\*...\*/ , comment syntax, 5–6
- ++..., ...++ (increment operators), 16–17
- ..., ...-- (decrement operators), 16–17
- ! (logical negation) operator, 19
- % (modulus) operator, 16
- @ (at character), used for compiler directives, 67
- [] (brackets)
  - in pointer syntax, 12
  - used for message expression, 96
- ^ (caret), in block syntax, 317
- <> (angle brackets), in protocol syntax, 250

---

## Numbers

---

- 2D graphics, in Core Graphics, 166–167**
- 32-bit vs. 64-bit computing**
  - coding for, 342
  - compiling, 343
  - kernel and user programs and, 342
  - overview of, 342
  - performance and, 342–343

---

## A

---

- Abstract classes**
  - class clusters and, 129
  - lack of explicit syntax for, 126
  - overview of, 125–126

**Access control**

- controlling block access to variables, 319–320
- to instance variables, 220–221
- for methods, 221
- security and, 222–224

**Accessors**

- accessing instances variables outside objects (not recommended), 230–231
- assign, retain, copy attributes, 236
- declaring/implementing, 231–232
- dot syntax for calling, 243–244
- form of, 232
- in Greeter class, 82
- multithreading and, 282–283
- properties using, 233–234
- side effects of, 101

**Address space, translation between virtual and physical, 41–42****Adopting protocols, 251–252****Aliases, variable, 15****alloc**

- as class method, 143
- combining with init, 129
- object allocation, 126–127
- release message balancing, 270
- syntax for declaring, 146

**Allocation**

- dynamic allocation of variables, 49–51
- object allocation, 126–127

**allocWithZone method, 127****Ancestor classes, 123****AND logical operator, 18–19****Angle brackets (<>), in protocol syntax, 250****Animation, Core Animation, 167****Anonymous functions, 309, 318. See also blocks****AppKit framework**

- autorelease pools and, 278
- function of, 161
- garbage collection threads in, 295
- graphics capabilities in, 166
- mechanism for program termination, 288
- problems with opaque pointers, 299–302
- prominent classes in, 162–163

**Apple. See also Mac OS X**

- 32 bit vs. 64 bit computing, 341
- blocks as extension to C, Objective-C, and C++, 309
- documentation for 64-bit Macs, 343
- frameworks supplied by, 167
- IDE, 73
- resources for Objective-C, 349

**Arguments**

- adding to function pointers, 312
- initializers with, 129–131
- methods with, 94–95

**Arithmetic operators, in C language, 16–17****Arrays**

- declaring in C, 12–13
- of function pointers, 311
- iterative loops over, 195
- multidimensional, 13
- mutable, 163–164
- NSArray and NSMutableArray, 177–180
- strings as, 13–14

**assign attribute, 236****Assignment operators, in C language, 19–21****Associative arrays, 180****auto keyword, declaring automatic variables, 44****Auto zone, of garbage collection heap, 297**

**Automatic (local) variables**

- declaring, 44
- overview of, 42–43
- scope of, 47–48

**autorelease**

- convenience constructors and, 277
- memory management for blocks, 323–324, 327

**Autorelease pools**

- adding objects to, 147
- controlling memory usage with extra pools, 280–281
- in HelloObjectiveC.m, 77
- managing, 278–279
- in reference counting, 277–278
- scheduling release, 148
- troubleshooting, 283

---

## B

**Base class, fragile base class problem, 347–348****Bitmapped images, drawing with AppKit, 163****Block literals**

- overview of, 317–318
- scope of, 321–322

**Block pointers, 318–319****Block variables**

- copying blocks and, 323
- overview of, 320–321
- read/write access to, 319

**Blocks**

- block literals, 317–318
- block pointers, 318–319
- block variables, 320–321
- in Cocoa frameworks, 327
- collection classes for applying, 328–329

- compared with functions, 317
- concurrency (multithreading) and, 327–328
- copying, 323
- Did-End callbacks, 329–330
- global, 322
- implicit loops with, 196
- issues with use of, 331
- memory management for, 323–326
- as objects, 322
- overview of, 309–310
- stack basis of, 321–322
- style issues, 331–332
- summary and exercises, 332–333
- traps in use of, 326–327
- variable access from, 319–320

**Blocks of bytes, NSData for working with, 185****Books resources, for Objective-C, 350****BOOL**

- Objective-C keywords, 68
- using BOOL type in control statements, 191

**Boolean values, 9****Brackets []**

- in pointer syntax, 12
- in messaging expressions, 96

**break statements, in C language, 26****Building and running HelloObjectiveC.m program, 87****Bytes**

- 32 bit vs. 64 bit computing, 341
- NSData for working with blocks of, 185

---

## C

**C: A Reference Manual (Harbison and Steele), 4****.c filename extension, for files in C, 7**

**C language**

- additions to in Objective-C, 66–71
- arithmetic operators, 16–17
- arrays, 12–13
- assignment operators, 19–21
- blocks as extension to, 309
- break** statements, 26
- calling functions from Objective-C, 224–225
- comma expressions, 27
- comments, 5–6
- comparison operators, 18
- compiling using gcc, 35–36
- conditional compilation, 32–33
- conditional expressions, 24
- continue** statements, 26–27
- debugging using gdb, 35–36
- #define** directive, 32
- do-while** statements, 25
- enumeration constants, 15–16
- expressions, 21–22
- files, 7–8
- floating-point types, 9
- formatting in, 5
- functions in, 29–31
- goto** statements, 28–29
- if** statements, 23–24
- #include** directive, 31–32
- initialization, 10
- integer types, 8–9
- logical operators, 18–19
- main** routine, 4–5
- naming conventions, 6–7
- operator precedence, 17–18
- operators, 16
- overview of, 3–4

- pointers, 10–12
- preprocessor, 31
- printf** function, 33–35
- program flow, 22–23
- program structure, 4
- reserved words in, 337
- for** statements, 25–26
- statements in, 22
- strings, 13–14
- structs**, 245–246
- structures in, 14–15
- summary and exercises, 37–39
- switch** statements, 27–28
- truth values, 9–10
- type conversion and casting, 19–20
- typedef** declarations, 15
- variable and function names, 6
- variables, 8
- while** statements, 24–25

***The C Programming Language* (Kernighan and Ritchie), 3****C++ language**

- blocks as extension to, 309
- frameworks compared with C++ class libraries, 159
- mixing with Objective C, 8

**C99 standard, 3****Callbacks**

- problems with function pointers, 314–315
- using function pointers for, 312

**Calling functions**

- arguments are call by value, 30
- C functions, 224–225
- function calls compared with method invocation, 93, 96
- with function pointers, 311–312

**CamelCase**

- naming conventions in C, 7
- strings, 213–216

**Caret (^), in block syntax, 317****Case sensitivity, of names in Objective-C, 66****Casting types, 19–20****@catch**

- exception handling directive, 205–206
- multiple @catch blocks in exception handling, 207
- nested exception handling, 207–208
- throwing own exceptions, 206–207

**Categories**

- extensions as anonymous categories, 219
- overriding methods with, 216–217
- overview of, 213–216
- security and, 222–224
- uses of, 217–218
- when to use, 216–217

**CFMakeCollectable, garbage collection and, 299****CFRelease, garbage collection and, 299, 301–302****CFRetain, garbage collection and, 301****CGFloat, Cocoa numeric types, 70****char**

- integer types, 8–9
- strings and, 13–14

**@class, in class definition, 117****Class clusters**

- consequences of implementing classes as, 172–173
- as design pattern, 129
- in Foundation framework, 172

**Class definition**

- @class, 117
- implementation file importing header file, 118

implementation section, 117–118

interface section, 115–116

in Objective-C, 59–61

overview of, 115

subclasses, 119

**Class hierarchies**

- example, 124–125
- subclasses and, 123–124

**Class keyword, in Objective-C, 69****Class libraries. See frameworks****Class methods**

- alloc. *See* alloc
- convenience constructors, 147–149
- overview of, 146
- singletons, 149–150

**Class objects**

- autorelease messages, 147
- class methods, 146
- Class type, 144–145
- convenience constructors, 147–149
- initialization of, 150–151
- overview of, 64–65, 143–144
- singletons, 149–150
- static variables mimicking class variables, 151–157
- summary and exercises, 157–158
- type introspection, 145–146

**Class type, 144–145****Class variables**

- mimicking, 151–157
- overview of, 144
- as receiver in message expression, 145

**class-dump, 224****Classes**

- abstract. *See* abstract classes
- adopting protocols, 251–252
- command line tool for dumping a class, 224

**Classes (continued)**

- defining. *See* class definition
- determining object class at runtime, 111
- implementing copying in, 139–141
- inheritance in Objective-C, 61
- inheritance in object-oriented programming, 57
- instances of. *See* objects
- names as types in Objective-C, 61–62
- in object-oriented programming, 56
- objects and object creation, 64–65
- overview of, 115
- subclasses. *See* subclasses
- summary and exercises, 141–142
- superclasses. *See* superclasses

**Classes, in Foundation framework**

- class clusters, 172–173
- collection classes, 177
- literal strings, 177
- mutable and immutable, 171–172
- `NSArray`, 177–180
- `NSData`, 185–186
- `NSDictionary`, 180–182
- `NSMutableString`, 176
- `NSNull`, 184–185
- `NSNumber`, 183–184
- `NSSet`, 182
- `NSString`, 173–176
- `NSURL`, 186–187
- overview of, 171
- structures, 187–188
- summary and exercises, 188–190

**Closures, 309.** *See also* blocks**Cocoa Design Patterns (Buck and Yacktman), 150****Cocoa frameworks**

- UIKit and, 162–163

- blocks in, 327
- convenience constructors in, 147–149
- documentation for 64-bit Macs, 343
- Internet resources supporting, 350
- iPhone and, 162
- `NSInvocation` objects in, 317
- overview of, 161–162
- singletons in, 149–150

**Cocoa Programming for Mac OS X (Hillebrand), 162****Cocoa Touch, 162****CocoaDev, 350****CocoaHeads, 350****Coding 32-bit vs. 64-bit programs, 342****`collectIfNeeded` method, in garbage collection, 296****Collection classes**

- applying blocks to collections, 328–329
- `NSArray`, 177–180
- `NSData`, 185–186
- `NSDictionary`, 180–182
- `NSNull`, 184–185
- `NSNumber`, 183–184
- `NSSet`, 182
- `NSURL`, 186–187
- overview of, 177

**Comma expressions, in C language, 27****Comments**

- types supported in C, 5–6
- in Xcode, 80

**Comparison operators, in C language, 18****Compiler directives**

- additions Objective-C makes to C language, 67
- `@class`, 117
- controlling access to instance variables, 220–221

for exception handling. *See* exceptions  
in HelloObjectiveC.m, 81, 83  
`@implementation`, 117–118  
`@interface`, 116  
in Objective-C, 338  
`@selector`, 103

### Compiling

32-bit vs. 64-bit programs, 343  
gcc for, 35–36

### Composition, extending classes by, 217

### Compound statements

overview of, 22  
variable scope and, 48–49

### Concurrency (multithreading), with NSOperationQueue, 327–328

### Conditional compilation, preprocessor allow- ing for, 32–33

### Conditional expression, 24

### Configuring Xcode, 75

### Conformance testing, for protocol objects, 260–261

`const` keyword, declaring variables and,  
46–47

### Constants, `#define` directive, 32

### Constructors. *See* convenience constructors

### continue statements, 26–27

### Control structures

`for...in` construction for fast enumer-  
ation, 199–201  
example of fast enumeration, 201–204  
exceptions, 205–206  
`if` statements, 191–195  
implicit loops, 195–196  
modifying mutable collections while  
enumerating, 197–199  
multiple `@catch` blocks, 207  
nested exception handling, 207–208

overview of, 191  
for program flow, 22–23  
for statements, 195  
summary and exercises, 210–212  
throwing own exceptions, 206–207  
using exceptions, 208–210  
`while` statements and `NSEnumerator`,  
196–197

### Convenience constructors

creating, 148–149  
memory management and, 276–277,  
280  
overview of, 147–148  
reference counting and, 280

### Conversion, type conversion, 19–20

### copy

copying objects, 136–137  
form of a setter using, 232–233  
properties, 236  
taking ownership by copying, 274

### Copying blocks, 323

### Copying objects

implementing copying in classes,  
139–141  
mutable and immutable copies, 138  
overview of, 136–137  
shallow and deep copies, 137–138  
taking ownership by, 274

### copyWithZone:

copying objects, 137, 139–141  
form of a setter using, 233

### Core Animation, 167

### Core Animation: Simplified Animation Techniques for Mac and iPhone Development (Zarra and Long), 167

### Core Audio, 167

### Core Data, 161



**Core Foundation**

- garbage collection and, 298–299
- memory management for objects in, 164–165
- overview of, 163–164
- toll-free bridging, 165–166, 339–340

**Core Graphics, 166–167, 188****Core Image, 167**


---

## D

**dealloc**

- destroying objects and, 133, 135–136
- HelloObjectiveC.m, 85–86
- not confusing with deallocation, 297
- program termination and, 288
- properties, 240
- releasing objects in reference counting, 274–275

**Deallocation, 297****Debugging, with gdb, 35–36****Declared properties, 229. See also properties****Declaring**

- accessors, 231–232
- arrays of function pointers, 311
- class methods, 146
- properties, 236
- protocols, 250–251
- variables, 44–47

**Decrement operators (a– –a), 16–17****Deep copies, objects, 137–138****#define directive, in C language, 32****Defining classes. See class definition****Design patterns**

- convenience constructors, 147–149
- extending classes by composition, 217
- singletons, 149–150

**Design Your Own Language Syndrome, blocks and, 331****Designated initializers, 131–132****Destroying objects, 135–136****Dictionaries, 180****Did-End callbacks, blocks, 329–330****Directives**

- compiler. *See* compiler directives
- preprocessor directives. *See* preprocessor directives

**Directories, for frameworks, 169****Display lists, in drawing programs, 97****Documentation**

- for 64-bit Macs, 343
- properties as, 242–243

**Dot syntax**

- C structs and, 245–246
- for calling accessor methods, 243–244
- properties and, 244–245

**double, floating-point type, 9****do-while statements, 25****drain, NSAutoreleasePool, 279****@dynamic**

- implementing accessors, 234
- properties and, 238–239

**Dynamic allocation, of variables, 49–51****Dynamic libraries, frameworks as, 159–160****Dynamic typing**

- in messaging (method invocation), 105–106
- methods with same names and, 104–105

---

## E

**Efficiency, in messaging (method invocation), 109–110****Encapsulation**

- blocks breaking, 331

in `HelloObjectiveC.m`, 82  
 object-oriented programming and,  
 56–57, 231

### Enumeration

`for...in` construction for fast enumeration, 199–201  
 example of fast enumeration, 201–204  
 modifying mutable collections while enumerating, 197–199

**Enumeration constants, in C language, 15–16**

**Enumeration tags, 15**

**Event loops, in GUI programs, 278–279**

### Exceptions

multiple `@catch` blocks, 207  
 nested exception handling, 207–208  
 overview of, 205–206  
 rarely using in Objective-C, 209–210  
 throwing own, 206–207  
 using, 208–209

**Expressions. See also statements**

comma expressions, 27  
 conditional expression, 24  
 evaluating, 21–22  
 message. *See* message expressions  
 overview of, 21

**Extensions, 218–219**

**extern keyword, 45**

### External variables

declaring, 45  
 overview of, 43–44  
 scope of, 49

## F

**Failed initialization, 132–135**

**Falsely rooted objects, garbage collection and, 303**

### Fast Enumeration

`for...in` construction for, 199–201  
 example of, 201–204  
`NSDictionary` and, 201

**File name extensions. See extensions**

### Files

C language, 7–8  
 moving to/from `NSData`, 185–186  
 saving, 329

**finalize method, 297**

**Finalizers, garbage collection, 296–297**

### @finally

compiler directive for exception handling, 205–206  
 multiple `@catch` blocks in exception handling, 207  
 nested exception handling, 207–208  
 throwing own exceptions, 206–207

**Flags, in garbage collection, 294**

**float, floating-point types, 9**

**Floating-point types, in C language, 9**

### for statements

iterative loop over arrays, 195  
 overview of, 25–26

**for...in construction, for fast enumeration, 199–201**

**Formatting, in C language, 5**

**Forward declaration, @class, 117**

**Forwarding, in messaging (method invocation), 108–109**

**forwardInvocation, NSObject, 108**

### Foundation framework

classes. *See* classes, in Foundation framework  
 Cocoa Touch and, 162  
 garbage collection threads in, 295  
`NSGarbageCollector`, 296  
 overview of, 161

**Foundation framework** (continued)

- singletons, 150
- toll-free bridging, 165–166, 339–340

**Fragile base class problem, 347–348****Frameworks**

- Apple-supplied, 167
- Cocoa. *See* Cocoa frameworks
- Core Animation, 167
- Core Foundation. *See* Core Foundation
- Core Graphics, 166–167
- directories for, 169
- overview of, 159–160
- summary, 169
- third-party, 168
- umbrella frameworks, 169
- using, 160–161
  - as versioned bundles, 168–169

**free function, dynamic allocation of variables, 50–51****Function parameters, automatic variables and, 43****Function pointers**

- calling functions with, 311–312
- overview of, 310–311
- problems with, 314–315
- using, 312–314

**Functions**

- arguments are call by value, 30
- blocks compared with, 317
- calling C functions from Objective-C, 224–225
- calling with function pointers, 311–312
- declaring, 31
- function calls compared with method execution, 93, 96
- names in C, 6
- overview of, 29–30

**G****Garbage collection**

- considerations in whether to use, 304–305
- controlling when occurs, 296
- Core Foundation objects and, 298–299
- falsely rooted objects and, 303
- finalizers, 296–297
- interior pointers and, 302–303
- `malloc`, 297–298
- memory management and, 267–268
- opaque pointers problems in AppKit, 299–302
- overview of, 291
- in practice, 293
- pros/cons of, 303–304
- strong and weak references and, 293–294
- summary and exercises, 305–308
- theory behind, 291–292
- using, 294–295

**GB (gigabytes), 341****gcc (GNU compiler)**

- compiling using, 35–36
- listing multiple frameworks on command line, 160
- using exceptions and, 208

**GCD (Grand Central Dispatch)**

- as driving force behind blocks, 309
- thread pools for concurrency, 327–328

**gdb (GNU debugger), 35–36****Generational garbage collection, 293****Generic pointers, 11–12****Getters. *See also* accessors**

- form of, 232
- `getter=name`, 237

instance variables, 229

multithreading and, 282–283

**Gigabytes (GB), 341**

**Global blocks, 322**

**GNU compiler.** See **gcc (GNU compiler)**

**GNU debugger (gdb), 35–36**

**GNUStep, 350**

**goto statements, 28–29**

**Grand Central Dispatch (GCD)**

as driving force behind blocks, 309

thread pools for concurrency,  
327–328

**Graphical user interface (GUI)**

event loops in GUI programs,  
278–279

writing GUI app using AppKit  
framework, 162–163

**Graphics, Core Graphics, 166–167**

**Greeter class**

in HelloObjectiveC.m, 80–83

putting Greeter class to work, 86–87

**greetingText method, in**

**HelloObjectiveC.m, 83**

**Group resources, for Objective-C, 350**

**GUI (graphical user interface)**

event loops in GUI programs,  
278–279

writing GUI app using AppKit  
framework, 162–163

---

## H

---

**Header files**

C programs using, 7

declaring informal protocols with  
category headers, 218

in TablePrinter class, 254–255

**HelloObjectiveC.m program**

building and running, 78, 87

**dealloc method, 85–86**

**Greeter.h class, 80–82**

**Greeter.m class, 82–83**

**greetingText method, 83**

**issueGreeting method, 85**

overview of, 79–80

program structure, 76–77

putting Greeter class to work, 86–87

**setGreetingText method, 84–85**

summary and exercises, 88–89

---

## I

---

**id**

convenience constructors typed as, 149

dynamic typing and, 104–106

Objective-C keywords, 67–68

**IDE (Integrated Development Environment),  
Xcode, 73**

**if statements**

compound conditions, 191–192

condition part of, 191

equality tests with, 193–195

explicit comparisons and, 192–193

overview of, 23–24

**ImageIO framework, from Apple, 167**

**Immutable classes**

creating separate mutable and  
immutable classes for containers,  
171–172

**NSString, 176**

**Immutable copies, objects, 138**

**IMP**

Objective-C keywords, 69

**typedef** declaration of, 110

**@implementation, 117–118, 219**

**Implementation section, of class definition**

in HelloObjectiveC.m, 82–83

**Implementation section, of class definition**  
(continued)

- introduction to Objective-C, 60–61
- overriding methods, 101
- messages to `super`, 102
- overview of, 117–118
- in separate file from interface section, 118–119
- for `TablePrinter` class, 255–256

**Implicit loops**

- with blocks, 196
- overview of, 195–196

**#import directive, 32****Imports, implementation file importing header file, 118****Include files, C language, 31–32****Increment operators (a++ ++a), 16–17****Informal protocols, 261–262****Information hiding. See encapsulation****Inheritance**

- class hierarchies and, 125
- convenience constructors inherited as subclasses, 149
- in Objective-C, 61
- in object-oriented programming, 57
- protocols compared with subclasses, 250
- subclasses and, 121

**init**

- class clusters and, 172
- combining with `alloc`, 129
- form of, 128
- overview of, 127
- return type of, 128–129

**Initialization, of objects**

- in C language, 10
- class objects, 150–151
- designated initializers, 131–132

- failed initialization, 132–135
- initializers with arguments, 129–131
- overview of, 127–129

**initialize message, 150–151****Input/output (I/O), 33****Instance variables**

- access control to, 220–221
- accessing outside objects (not recommended), 230–231
- class objects not using, 144
- classes and, 56
- in `HelloObjectiveC.m`, 81
- names can be different than property name, 235
- subclasses and, 217
- synthesized, 345–347

**Instances**

- class objects, 146
- of classes, 56

**int, 8–9****Integer types, C language, 8–9****Integrated Development Environment (IDE), Xcode, 73****@interface, 116****Interface section, of class definition**

- introduction to Objective-C, 59–60
- overview of, 115–116
- in separate file from implementation section, 118–119

**Interior pointers, garbage collection and, 302–303****Internet resources, for Objective-C, 350****Introspection**

- at runtime, 111–112
- type introspection, 145–146

**Invocation. See NSInvocation****I/O (input/output), 33**

**iPhone**

- autorelease pools and, 280
- block availability, 309
- Cocoa frameworks and, 162
- Core Animation, 167
- Core Graphics, 166
- garbage collection and, 291
- mechanism for program termination, 288
- memory management, 267
- SDK for, 349
- synthesized instance variables, 347

**iPhone Developer Program, 349**

**isa variable, object instances and, 107**

**issueGreeting method, in HelloObjectiveC.m, 85**

---

## J

**Java class libraries, 159**

---

## K

**Kernel, 32-bit vs. 64-bit computing, 342**

**Key-value coding, getting/setting instance variables, 229**

**Keywords**

- additions Objective-C makes to C language, 67–69
- for declaring variables, 44–47

---

## L

**Lazy instantiation, 150**

**Literal strings**

- additions Objective-C makes to C language, 67
- in Foundation framework, 177

**Local variables. See automatic (local) variables**

**Logical operators**

- in C language, 18–19
- logical negation (!), 19
- types of, 18

**long double floating-point type, 9**

**long integer type, 8–9**

**long long integer type, 8–9**

**Loops**

- for...in** construction for, 199–201
- implicit, 195–196
- iterative loop over arrays, 195
- overview of, 23
- while** statements, 196–197

**ls command, Unix, 201–202**

**lvalues, assignment to, 19**

---

## M

**Mac Developer Program, 349**

**Mac OS X**

- 32 bit vs. 64 bit computing, 341
- block availability, 309
- garbage collection and, 304
- mechanism for program termination, 288
- memory management, 267
- resources for Objective-C, 349
- saving files, 329
- toll-free bridging and, 339–340
- use of frameworks in, 159–160

**Mac OS X Leopard**

- 32-bit vs. 64-bit computing, 341
- blocks for iterating over Foundation collection objects, 196, 309, 327
- Core Animation, 167
- garbage collection and, 291
- protocol objects, 262
- reserved keywords, 337

**Mac OS X Leopard (continued)**

- save sheets, 329–330
- synthesized instance variables, 346

**main routine**

- in `HelloObjectiveC.m`, 77
- structural aspects of C, 4–5
- in `TablePrinter` class, 258–259

**makeObjectsPerformSelector method, 195–196****malloc**

- divisions of garbage collection heap, 297–298
- dynamic allocation of variables, 50–51

**Managed memory. See reference counting****Mathematical sets, 182****Memory layout, of Objective-C programs, 41–42****Memory leaks, 269****Memory management. See also reference counting, garbage collection**

- for blocks, 323–326
- Core Foundation, 164–165
- default for, 268–269
- importance of, 267
- in Objective-C, 65
- properties and, 240

**Memory Management Unit (MMU), 41–42****Message expressions**

- additions Objective-C makes to C language, 66–67
- form of, 96
- in `HelloObjectiveC.m`, 84
- nesting, 98–99

**Messages**

- `NSInvocation` for turning messages into objects, 315–317
- in object-oriented programming, 55

**Messaging (method invocation)**

- comparing methods in Objective-C with C, 93
- dynamic and static typing, 105–106
- efficiency, 109–110
- forwarding, 108–109
- how it works, 106–108
- introspection at runtime, 111–112
- messaging process, 96
- method naming conventions, 95–96
- methods with arguments, 94–95
- methods with same name, 104–105
- nesting, 98–99
- `nil`, 100
- in Objective-C, 62–63
- overriding messages to `super`, 101–103
- overview of, 93
- polymorphism in, 97–98
- selectors, 103–104
- sending messages to `self`, 100–101
- simple method example, 93–94
- summary and exercises, 112–113
- who is the sender in, 98

**Method overloading, not allowed in Objective-C, 105****Method selector. See selectors****Methods**

- access control for, 221
- AppKit classes, 163
- with arguments, 94–95
- class methods. *See* class methods
- comparing Objective-C with C, 93
- defined, 55
- implementing optional protocol methods, 260
- messaging (method invocation), 62–63
- naming conventions, 95–96

- no explicit syntax for, 126
- in object-oriented programming, 56
- overriding, 119
- with same name, 104–105
- simple method example, 93–94

**Methods, in HelloObjectiveC.m program, 85–86**

- additions Objective-C makes to C language, 70–71
- dealloc method, 85–86
- Greeter class, 82–83
- greetingText method, 83
- issueGreeting method, 85
- setGreetingText method, 84–85

**MMU (Memory Management Unit), 41–42**

**Modulus (%) operator, 16**

**Multidimensional arrays, declaring in C, 13**

**Multiple @catch blocks, 207**

**Multithreading**

- receiving objects and, 271
- reference counting and, 282–283

**Mutable arrays, 163–164**

**Mutable classes**

- creating separate mutable and immutable classes for containers, 171–172
- NSMutableString, 176

**Mutable collections, modifying while enumerating, 197–199**

**Mutable copies, of objects, 138**

---

## N

### Names

- additions Objective-C makes to C language, 66
- class names as types in Objective-C, 61–62
- setter=name, getter=name, 237

### Namespaces, not used in Objective-C, 221

#### Naming conventions

- C language, 6–7
- CamelCase, 213–216
- methods, 95–96
- object ownership and, 273–274

#### Negation operator (-), 18

#### Nesting, message expressions, 98–99

#### new method, object allocation, 127

#### New Project window, in Xcode, 74

#### nil

- Objective-C keywords, 68
- sending messages to nil receivers, 100

#### nonatomic attribute, 237

#### NSArray

- bounds checking, 178
- length of, 178
- makeObjectsPerformSelector method and, 195
- memory management for objects in, 179–180
- overview of, 177–178

#### NSAutoreleasePool, 277, 279

#### NSData

- accessing bytes, 185–186
- collection class for working with blocks of bytes, 185
- moving files to/from, 185–186

#### NSDictionary

- collection class for associative arrays, 180–182
- Fast Enumeration, 201

#### NSEnumerator, 196–197

#### NSError, 210

#### NSException, 206

#### NSGarbageCollector, 296

#### NSInteger, 70

#### NSInvocation, 108–109, 315–317



**NSLog**  
 overview of, 70–71  
 printf function compared with, 35

**NSMutableArray**  
 adding objects to, 179  
 for custom sorting, 312–314  
 overview of, 177

**NSMutableDictionary, 180–182**

**NSMutableSet, 182**

**NSMutableString, 176**

**NSNull, 184–185**

**NSNumber, 183–184**

**NSObject**  
 blocks as objects, 322  
 forwardInvocation and, 108  
 inheriting from, 124  
 methods defined by, 111  
 <NSObject> compared with, 250–251  
 retainCount method, 281

**<NSObject>, 250–251**

**NSOperationQueue, 327–328**

**NSPoint, 187–188**

**NSProxy, 123–124**

**NSRange, 187–188**

**NSRect, 187–188**

**NSSet**  
 collection class for mathematical sets, 182  
 makeObjectsPerformSelector  
 method and, 196

**NSString, 187–188**  
 appending to another NSString, 175  
 breaking sentences into individual words, 175–176  
 CamelCase method, 213–216  
 converting C strings to/from, 176  
 examples of, 174

finding length of, 174  
 literal NSString, 177  
 overview of, 173–174  
 uppercase version of, 174–175

**NSInteger, 70**

**NSURL, 186–187**

**NSView, 126**

**NSZombie, 284–285**

**null object, 184**

**Numeric types**

NSNumber, 183–184

testing for equality, 193–194

---

## O

**Object allocation, 126–127**

**Objective-C, introduction to**

additions Objective-C makes to C language, 66–71  
 class definition in, 59  
 class names as types in, 61–62  
 class objects and object creation, 64–65  
 implementation section of class definition, 60–61  
 inheritance in, 61  
 interface section of class definition, 59–60  
 memory management, 65  
 messaging (method invocation), 62–63  
 overview of, 58–59  
 polymorphism in, 63–64  
 summary, 71

**Objective-C 2.0 Runtime Reference, 111**

**Object-oriented programming**

classes and instances, 56  
 encapsulation, 56–57  
 inheritance, 57  
 methods, 56

overview of, 55  
 polymorphism, 58  
 summary, 71

## Objects

allocation of, 126–127  
 blocks as, 322  
 classes, 64–65  
 copying, 136–137  
 in Core Foundation, 163–164  
 designated initializers, 131–132  
 destroying, 135–136  
 failed initialization, 132–135  
 implementing copying in classes, 139–141  
 initialization of, 127–129  
 initializers with arguments, 129–131  
 mutable and immutable copies, 138  
`NSInvocation` for turning messages into objects, 315–317  
 overview of, 126  
 protocol objects, 260–261  
 root set, 291–292  
 shallow and deep copies, 137–138  
 summary and exercises, 141–142  
 taking ownership by copying, 274  
 testing for equality, 193–195  
 typing by class, 56

## Omni Group, The 350

### Opaque pointers

for accessing Core Foundation objects, 163–164  
 problems in AppKit, 299–302

### OpenAL framework, 167

### OpenGL framework, 167

### OpenGL ES framework, 167

### Operator precedence, in C language, 17–18

### Operators

arithmetic operators, 16–17

assignment operators, 19–21  
 comparison operators, 18  
 logical operators, 18–19  
 overview of, 16

### `@optional`, protocol methods, 250–251

### OR logical operator, 18–19

### otool, command line tool for dumping a class, 224

### Overriding methods

with categories, 216–217  
 defining subclasses and, 119

### Ownership, in reference counting, 273–274

---

## P

### PDF graphics, 166–167

### Performance, 32-bit vs. 64-bit computing, 342–343

### Pointers

block pointers, 318–319  
 to class objects, 144  
 declaring pointer variables, 10–11  
 to/from Foundation objects to corresponding Core Foundation objects, 165–166  
 function pointers, 311–312  
 generic pointers, 11–12  
 interior pointers, 302–303  
 opaque pointers, 163–164

### Polymorphism

in Objective-C, 63–64  
 in object-oriented programming, 58  
 overview of, 97–98

### Precedence, operator precedence, 17–18

### Preprocessor directives

conditional compilation, 32–33  
`#define` directive, 32  
`#include` directive, 31–32  
 overview of, 31

**printf function, in C language, 33–35**

**@private, controlling access to instance variables, 220–221**

**Procedural languages**

examples of, 55

object-oriented programming vs., 58

**Program flow, in C language, 22–23**

**Program termination, reference counting and, 288**

***Programming in Quartz and PDF Graphics in Mac OS X* (Gelphman and Laden), 166**

**Properties**

accessing instances variables outside objects (not recommended), 230–231

accessors using, 233–234

assign, retain, copy attributes, 236

dealloc, 240

declaring/implementing accessors, 231–232

as documentation, 242–243

dot syntax and, 244–245

dot syntax and C structs, 245–246

dot syntax for calling accessor methods, 243–244

@dynamic, 238–239

form of accessors, 232

hidden setters for readonly, 242

instance variable names can be different than property name, 235

memory management and, 240

nonatomic attribute, 237

overview of, 229–230

@property, 236

protocols and, 252–253

readwrite, readonly attributes, 237

setter=name, getter=name, 237

subclasses and, 240–242

summary and exercises, 246–247

@property

declared properties and, 229

declaring accessors, 233

form of property declaration, 236

**@protected, controlling access to instance variables, 220–221, 231**

@protocol

declaring protocols, 250–251

obtaining protocol objects, 260–261

**Protocol objects, and conformance testing, 260–261**

**Protocols**

adopting, 251–252

declaring, 250–251

informal, 261–262

overview of, 249–250

properties and, 252–253

protocol objects and conformance testing, 260–261

summary and exercises, 262–263

TablePrinter class example, 253–260

as types, 252

**@public, controlling access to instance variables, 220–221**

---

## Q

**Quartz, 166–167**

---

## R

**Raising exceptions. See throwing exceptions**  
**readonly**

controlling block access to variables, 319

hidden setters for, 242

properties, 237

**readwrite**

controlling block access to variables, 319

properties, 237

**Receiving objects, reference counting and, 271–273**

**Reference count, 269**

**Reference counting**

autorelease messages and, 147

autorelease pools, 277–278

controlling memory usage with extra autorelease pools, 280–281

convenience constructors and, 147–148, 280

for Core Foundation objects, 164–165

`dealloc`, 274–275

as default for memory management, 268–269

disadvantages of converting to garbage collection, 304

in `HelloObjectiveC.m`, 84

how it works, 269–270

managing autorelease pools, 278–279

memory management in Objective-C, 65

multithreading and, 282–283

`NSZombie` for over-release problems, 284–285

overview of, 267–268

ownership in, 273–274

program termination and, 288

receiving objects and, 271–273

retain cycles, 285–287

`retainCount` method, 281

returning objects, 276–277

as solution to returning memory allocated to objects, 268

summary and exercises, 288–289

troubleshooting errors in, 283

**register keyword, declaring variables and, 46**

**release**

memory management for blocks, 323–324, 327

`NSAutoreleasePool`, 279

in reference counting, 269–270

troubleshooting, 283

**Remainder operator (%), 16**

**@required, protocol methods, 250–251**

**Reserved words**

in C language, 337

in Objective-C, 337–338

**Resources for Objective-C, 349–350**

**respondsToSelector method, 111**

**retain**

memory management for blocks, 327

properties, 236

in reference counting, 272–273

taking ownership by copying, 274

troubleshooting, 283

**Retain count, 269**

**Retain counting. See reference counting**

**Retain cycles**

garbage collection and, 292

reference counting and, 285–287

**retainCount method, NSObject class, 281**

**return statement, 317**

**Returning objects, reference counting, 276–277**

**Root classes, 123**

**Root set**

falsely rooted objects and, 303

garbage collector starting with, 291–292

**Runtime**

additions Objective-C makes to C language, 66

**Runtime** (continued)

- changes in Objective-C 2.0, 108
- fragile base class problem, 347–348
- introspection at, 111–112
- legacy vs. modern, 345
- library of C functions, 97
- synthesized instance variables and, 345–347

**Runtime Reference, Objective-C 2.0, 111**


---

## S

---

**Scope, variable**

- of automatic variables, 47–48
- compound statements and, 48–49
- of external variables, 49
- overview of, 47

**Security, access control and, 222–224****SEL**

- Objective-C keywords, 69
- for representation of method selectors, 103–104

**Selectors, method names as, 103–104****self**

- class methods and, 146
- sending messages to, 100–101
- super variable compared with, 102–103

**setGreetingText method, in HelloObjectiveC.m, 84–85****Sets, mathematical, 182****Setters.** *See also* **accessors**

- form of accessors, 232
- hidden setters for `readonly`, 242
- instance variables, 229
- multithreading and, 282–283
- `setter=name`, 237

**Shallow copies, objects, 137–138****short integer type, 8–9****Side effects**

- of accessor methods, 101
- of expressions, 22

**Singletons, 149–150****Sorting, NSMutableArray for custom sorting, 312–314****Stack frame, calling functions and, 42****Stacks, block based on, 321–322****Statements.** *See also* **control structures; expressions**

- break statements, 26
- compound statements, 48–49
- continue statements, 26–27
- do-while statements, 25
- goto statements, 28–29
- if statements, 23–24
- overview of, 22
- for program flow, 22–23
- return statement, 317
- for statements, 25–26
- switch statements, 27–28
- using different names for properties and their instance variables, 235
- while statements, 24–25

**Static allocation, of variables, 49****static keyword, 45–46****Static typing, in messaging (method invocation), 105–106****Static variables, mimicking class variables, 151–157****Strings**

- in C language, 13–14
- CamelCase, 213–216
- literal strings, 67

**Strings, in Foundation framework**

- NSString, 176
- NSString, 173–176

**Strong references**

garbage collection and, 293–294  
reference counting and, 287

**structs, dot syntax and, 245–246**

**Structures**

as collection of related variables,  
14–15  
in Foundation framework, 187–188  
program structure in  
HelloObjectiveC.m, 76–77

**Style, blocks, 331–332**

**Subclasses**

abstract classes and, 125  
categories compared with, 213–214  
class clusters and, 129, 173  
class hierarchies and, 123–125  
class methods, 146  
convenience constructors inherited as,  
149  
defining, 119  
example of, 119–123  
of immutable classes, 171  
overview of, 119  
properties and, 240–242  
protocols compared with, 250  
when to use categories instead of,  
216–217

**Subviews, NSView class, 126**

**super variable, 101–103**

**Superclasses**

class objects, 144–145  
defined, 119  
determining object superclass at run-  
time, 111

**switch statements, 27–28**

**Syntactic sugar, 243**

**@synthesize**

declared properties and, 229

**@dynamic** compared with, 238–239

implementing accessors, 233–234

synthesized instance variables,  
345–347

using different names for properties  
and their instance variables, 235

**Synthesized instance variables, 345–347**

---

**T**


---

**TablePrinter class, protocol example,  
253–260**

FruitBasket class, 256–258

header files, 254–255

implementation file for, 255–256

implementing optional methods, 260

main routine, 258–259

overview of, 253

problem in, 259–260

TablePrinterDataSource, 253–254

**TablePrinterDataSource, 253–254**

**testMethod, in messaging (method invo-  
cation), 109–110**

**TextEdit program, 163**

**Third-party frameworks, 168**

**Thread pools, for concurrency, 327**

**@throw, 206**

**Throwing exceptions, 205–207**

**Toll-free bridging, Core Foundation types to  
Foundation classes, 165–166, 339–340**

**Truth values, in C language, 9–10**

**@try**

compiler directive for exception han-  
dling, 205–206

multiple @catch blocks in exception  
handling, 207

nested exception handling, 207–208

throwing own exceptions, 206–207

**Type conversion and casting, in C language,  
19–20**

**Type introspection, class objects, 145–146****typedef declaration**

- for creating aliases of variable types, 15
- of `IMP`, 110

---

## U

**UIKit framework**

- autorelease pools and, 278
- for iPhone, 162
- mechanism for program termination, 288

**Umbrella frameworks. See also Cocoa frameworks**

- indirection in, 169
- overview of, 160

**Unix, `ls` command, 201–202****unsigned integers, 9****URLs, collection class for working with, 185–186**


---

## V

**Variables**

- automatic or local, 42–43
- block pointers, 318–319
- block variables, 320–321
- class objects and, 144
- controlling block access to variables, 319–320
- declaring in C, 8
- declaring in Objective C, 41
- dynamic allocation of, 49–51
- external, 43–44
- initialization of, 10
- keywords for declaring, 44–47
- names in C, 6
- pointers, 10–11
- scope, 47–49

- static variables mimicking class variables, 151–157

- structures as groups of, 14–15

- summary and exercises, 51–53

- typedef** declarations for creating aliases, 15

**Vector graphics, drawing with `AppKit`, 163****Versioned bundles, frameworks as, 168–169****volatile keyword, 47**


---

## W

**Weak references**

- `assign` attribute and, 236
- garbage collection and, 293–294
- reference counting and, 287

**WebKit framework, from Apple, 167****while statements**

- overview of, 24–25
- using in conjunction with `NSEnumerator`, 196–197

**Write barrier, in garbage collection, 293**


---

## X

**Xcode**

- adding frameworks to projects, 160–161
- building and running programs, 87
- compiling 64-bit projects, 343–344
- configuring, 75
- enabling exceptions in, 208–209
- enabling `NSZombie` in, 284–285
- HelloObjectiveC.m. *See* HelloObjectiveC.m program
- Internet resources supporting, 350
- opening and starting new project, 74–76
- overview of, 73