Objective-C
Pocket Reference

A Guide to Language Fundamentals

easy computing

Andrew M. Duncan

O'REILLY®
Objective-C is an exciting and dynamic approach to C-based object-oriented programming. It’s the approach adopted by Apple as the foundation for programming under Mac OS X, a Unix-based operating system gaining wide acceptance among programmers and other technologists. Objective-C is easy to learn and has a simple elegance that is a welcome breath of fresh air after the abstruse and confusing C++.

In this pocket reference, Andrew M. Duncan provides a quick and concise introduction to Objective-C for the experienced programmer. In addition to covering the essentials of Objective-C syntax, Andrew also covers important facets of the language such as memory management, the Objective-C runtime, dynamic loading, distributed objects, and exception handling.

“Don’t even think of venturing into the brave, new world of Objective-C without this little tome in your pocket.”

—Jacek Artymiak, freelance author & consultant
Contents

Introduction 1

What Is Objective-C? 3
  Dynamic Dispatch 4
  Dynamic Typing 4
  Dynamic Loading 5
  Which Objective-C? 5
  How Do I Get Started? 6

Elements of the Language 7
  Objects 7
  Classes 8
  Inheritance and Subtyping 13
  Fields 14
  Methods 16
  Categories 26
  Protocols 29
  Declarations 31
  Predefined Types, Constants, and Variables 34

Compiler and Preprocessor Directives 37
  Class Declaration and Definitions 37
  Forward Declaration 38
  Expanding Directives 39
  Preprocessor Symbols 43
Compiler Flags  43

Remote Messaging  44
  Pointer Parameter Qualifiers  45
  Return Value Qualifiers  46
  Object Qualifiers  46

Object Lifecycle  46
  Creating an Object  47
  Copying an Object  53
  Deallocating an Object  57

Runtime Errors  58
  Object Error Handling  58
  Exceptions in Cocoa  60

Runtime Environment  64
  Class Objects  64
  Metaclass Objects  66
  Selectors  68
  Protocol Objects  69

Root Classes  69
  Fields  70
  Methods  70
  The Object Class  71
  The NSObject Class  79

Forwarding Messages  89
  Object Forwarding  90
  NSObject Forwarding  91

Memory Management  94
  Manual Memory Management  94
  Reference Counting  95
  Garbage Collection  100
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Archiving Objects</td>
<td>100</td>
</tr>
<tr>
<td>Archiving Descendants of Object</td>
<td>101</td>
</tr>
<tr>
<td>Archiving Descendants of NSObject</td>
<td>103</td>
</tr>
<tr>
<td>Key-Value Coding</td>
<td>107</td>
</tr>
<tr>
<td>Access Permissions</td>
<td>107</td>
</tr>
<tr>
<td>NSKeyValueCoding Methods</td>
<td>108</td>
</tr>
<tr>
<td>Handling Key Lookup Failures</td>
<td>111</td>
</tr>
<tr>
<td>Optimizing Method Calls</td>
<td>111</td>
</tr>
<tr>
<td>Objective-C++</td>
<td>113</td>
</tr>
<tr>
<td>Objective-C Resources</td>
<td>114</td>
</tr>
<tr>
<td>Index</td>
<td>116</td>
</tr>
</tbody>
</table>
Introduction

The *Objective-C Pocket Reference* is a quick guide to the Objective-C programming language and some of its fundamental support libraries. This reference takes the form of brief explanations interspersed with examples and definitions. If you are experienced with Objective-C, this handbook will supply the definitions and examples you most often need to jog your memory. If you are coming to Objective-C from C++ or Java and prefer to jump right in and write code, this book will give you enough explanation to use the language’s features appropriately.

**NOTE**

You should be familiar with C-style languages in order to read this book. Objective-C uses C syntax. This book focuses only on Objective-C, and assumes that you understand the underlying C code.

This handbook progresses in sequence as much as possible, with later sections building on earlier ones, but some parts are necessarily interrelated. For example, the section on objects needs to refer to classes and vice versa. Both use the terminology of inheritance. Where you see an unfamiliar
term used, check the index: it is probably defined elsewhere in the book.

Although Objective-C is (apart from its C base) a small language, the implications of its modest set of extensions are substantial. To use Objective-C effectively, you should follow tested design patterns and make judicious use of libraries. The intent of this handbook is to provide a quick reference to the most commonly used features and idioms of the language. It should be like a fast cache, the first stop for frequently used data.

Because of its size, this handbook can present only an outline of the language, its libraries, and conventional patterns of usage. If you are interested in truly understanding the Objective-C way of thinking, you should also look at some of the texts listed in the “Objective-C Resources” section at the end of this book.

For supplementary information and corrections to this handbook, see our web site at http://www.oreilly.com/catalog/objectcpr.

Typographic Conventions

The following typographical conventions are used in this book:

*Italic*

New terms, URLs, and filenames

*Constant Width*

Code examples and names of classes, fields, variables, and methods

*Constant Width Bold*

Words that are reserved in C or Objective-C

*Constant Width Italic*

Text in an example that should be replaced by values you supply
Acknowledgments

Many thanks to my attentive editor, Jonathan Gennick, who considers every word. Greg Parker gave valuable feedback about the Objective-C runtime, Mark Bessey about the compiler, and Nicola Pero about archiving. Paul Kmiec asked the tough questions that made me think about how to explain the answers. Thanks also to Jacek Artymiak, Scott Anguish, and Eric Buck for their very thorough technical review of this book.

What Is Objective-C?

Objective-C is an object-oriented language: it supports hierarchies of substitutable types, message-passing between objects, and code reuse through inheritance. Objective-C adds these features to the familiar C programming language.

Because Objective-C is an extension of C, many properties of an Objective-C program depend on the underlying C development tools. Among these properties are:

- The size of scalar variables such as integers and floating-point numbers
- Allowed placement of scoped declarations
- Implicit type conversion and promotion
- Storage of string literals
- Preprocessor macro expansion
- Compiler warning levels
- Code optimization
- Include and link search paths

For more information about these topics, consult the documentation for your development platform and tools.

Objective-C differs from C++, another object-oriented extension of C, by deferring decisions until runtime that C++
would make at compile time. Objective-C is distinguished by the following key features:

- Dynamic dispatch
- Dynamic typing
- Dynamic loading

**Dynamic Dispatch**

Object-oriented languages replace function calls with messages. The difference is that the same message may trigger different code at runtime, depending on the type of the message receiver. Objective-C decides dynamically—at runtime—what code will handle a message by searching the receiver’s class and parent classes. (The Objective-C runtime caches the search results for better performance.) By contrast, a C++ compiler constructs a dispatch table statically—at compile time.

Because the simple linear search for a receiver used by Objective-C mirrors the way we think about inheritance, it’s easy to understand how an Objective-C program works. Dynamic dispatch can handle changes in the inheritance hierarchy at runtime. A dynamic message-sending model is also more natural for distributed objects than a table-based model.

**Dynamic Typing**

Because message-sending is dynamic, Objective-C lets you send messages to objects whose type has not been declared. The Objective-C environment determines dynamically—at runtime—the class of the message receiver and calls the appropriate code. By comparison, C++ requires the type of the receiver to be declared statically—at compile time—in order to consult dispatch tables.

Static typing allows the compiler to detect some program errors, but type checking is undecidable—that is, no algorithm can infallibly distinguish between programs that have
type errors and those that do not. A compiler must either miss some errors or prohibit some safe operations. Of course, in practice compilers follow the latter course, so some programs that would run correctly will not compile. Dynamic typing admits designs whose correctness is not evident to the compiler.

Objective-C lets you use static type checking where you want it, but dynamic typing where you need it. This represents a move away from the question of What is the receiver’s type at compile time? to What messages does an object respond to at runtime? Since programs run only at runtime, this is a more useful perspective.

**Dynamic Loading**

Because the process of method dispatch is simple and uniform, it’s easy to defer until runtime the linking of separately-compiled code modules. Objective-C programs can be factored into components that have minimal interdependency; these components can be loaded as needed by a running program. This makes it easier to deliver code, as well as content, over a network; design applications or systems that are distributed; or write an application that can be extended by third-party plug-ins.

**Which Objective-C?**

If you are programming in a Unix environment, you probably already have an Objective-C compiler: the gcc compiler, which is part of many Unix installations and is available under the terms of the GNU Public License. Because of the wide availability of this compiler for many software and hardware platforms, this handbook documents the features of the language compiled by Version 3.1 of gcc.

Apple Computer has also adopted gcc as the compiler for its OS X platform, which is based on a Unix variant called
deallocating, 57, 72, 82
reference counting and, 99
error handling, 58–60
forwarding, 90–94
identifying, 72–74, 82–83
inheritance, testing, 74, 83
initializing, 49, 50–52
life cycle of, 46–58
mutable/immutable, 54–57
reference counting, 93–99
releasing memory manually, 94
replacing/returning, 98
testing types of, 74
oneway type qualifier, 34, 46
open_typed_stream_for_file(), 102
optimizing method calls, 111–113
OS X platform and gcc compiler, 5
out type qualifier, 34, 45
overloading methods, 14, 21
overriding methods, 14
avoid using categories, 28
implementing forwarding for classes, 90
for root classes, 71
P
parameters
method declaration and, 16–19
naming collisions and, 20
parent classes, 9
inheritance from, 13
overriding designated initializers, 50–52
super variable and, 24, 37
super_class pointer and, 65
-perform: method, 69, 76
-performSelector: method, 86
-perform:: method, 77
permissions, access, 10, 15, 107
pointer parameter qualifiers, 45
+poseAs: method, 77
+poseAsClass: method, 87
preprocessor directives, 35
preprocessor symbols, 35
@private access keyword, 15
private methods, 21
@protected access keyword, 15
@protocol directive, 35, 39
forward declarations, 38
runtime environment and, 69
starting method declarations, 38
protocol objects, 31, 69
Protocol type, 35
protocols, 29–31
adopting, 54
checking for conformity to, 31
declaring, 29
implementing, 30
informal, 31
NSCopying, 55
NSMutableCopying, 55
@public access keyword, 15
R
raising exceptions in Cocoa, 60–64
-read: method, 79, 101
receivers, 4
deallocating methods, 95
deep copies of, returning, 54
method calls and, 20
mutable/immutable objects and, 54–57
naming collisions and, 20
root class methods and, 70
selectors and, 26
shallow copies of, returning, 54
special, 24–25
reference counting, 95–99
inherited methods returning shallow copies and, 56
methods supporting, 88
reflection, 100
-release method, 57, 89, 96–99
remote messaging, 34, 44–46
-replacementObjectForCoder:
 方法, 88
resources for Objective-C, 114
-respondsTo method, 75
-respondsToSelector: method, 86
restoring objects, 100–106
 retaining method, 57, 88, 96–99
retainCount method, 88, 96
return types
declaring methods and, 16–19
naming collisions and, 20
object qualifiers, 46
return value qualifiers, 46
root classes, 69–89
methods for, 70
NSObject class, 79–89
Object class, 71–79
root objects, 106
runtime environments
Darwin vs. GNU, 5
Objective-C, 64–69
runtime errors, 58–64
exceptions in Cocoa, 60–64
methods that generate, 78
sending messages
to an id type, 32
to nil, 25

S
saving objects, 100–106
SEL type, 25, 33, 68
@selector directive, 35, 39
-self method, 73, 82
self variable, 36
  fields of, 14–16
  implementing methods, 19
  as special receiver, 24
+setVersion: method, 79, 83
shallow copies of objects, 53, 56
-shallowCopy method, 54, 72
-shouldNotImplement method, 78
special receivers, 24–25
static type checking, 5, 32–33
static, declaring variables as, 11
-storedValueForKey: method, 109
+streamVersion: method, 79
"string" directive, 39, 42
subclasses, 13, 71
  vs. categories, 26
  initializing, 50–53
  subclassResponsibility: method, 78
subtyping, 13
super variable, 37
  implementing methods, 19
  NSCoding protocol for, 40
  +poseAs: method, 109
  +poseAsClass: method, 87
  as special receiver, 24
super_class pointer, 65, 67
-superClass method, 73, 82
-superclass method, 83

T
takeStoredValue:forKey: method, 110
takeValue:forKey: method, 109
transmuteClassTo: method, 78
try blocks and Cocoa exceptions, 60–64
type checking, dynamic vs. static, 4, 32
type qualifiers, 34
encodings for, 41
TypedStream type, 79, 101–103
types encodings for, 40
inheritance and subtyping, 13
of objects, testing, 74
pointer qualifiers and, 45
predefined, 34

U
-unableToNilForKey: method, 111
UnknownKeyException exception, 111
+useStoredAccessor method, 110

V
-valueForKey: method, 107, 108
variables
  declaring as static, 11
  special, 36
+version method, 79, 83
vprintf(), 60

W
-Wno-protocol compiler flag, 44
-write: method, 79, 101
-Wselector compiler flag, 44

Y
YES value, 36

Z
-zone method, 89
+allocWithZone: method, 80