Swift for Programmers

The New Language for iOS® and OS X® Developers

PAUL DEITEL • HARVEY DEITEL

FREE SAMPLE CHAPTER

SHARE WITH OTHERS
SWIFT™ FOR PROGRAMMERS
Deitel® Developer Series
### Deitel® Developer Series

<table>
<thead>
<tr>
<th>Title</th>
<th>Edition</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android for Programmers: An App-Driven Approach</td>
<td>2/E</td>
<td></td>
</tr>
<tr>
<td>C for Programmers with an Introduction to C11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td>3/E</td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++11 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C# 2012 for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iOS® 8 for Programmers: An App-Driven Approach with Swift®</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java™ for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaScript for Programmers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swift™ for Programmers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### CourseSmart Web Books

- [www.deitel.com/books/CourseSmart/](www.deitel.com/books/CourseSmart/)
  - C++ How to Program, 8/E and 9/E
  - Simply C++: An App-Driven Tutorial Approach

### LiveLessons Video Learning Products

- [www.deitel.com/books/LiveLessons/](www.deitel.com/books/LiveLessons/)
  - Android App Development Fundamentals, 2/e
  - C++ Fundamentals
  - Java™ Fundamentals, 2/e
  - C# 2012 Fundamentals
  - C# 2010 Fundamentals
  - iOS® 8 App Development Fundamentals, 3/e
  - JavaScript Fundamentals
  - Swift™ Fundamentals

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please join the Deitel communities on:

- Facebook®—facebook.com/DeitelFan
- Twitter®—@deitel
- Google+™—google.com/+DeitelFan
- YouTube™—youtube.com/DeitelTV
- LinkedIn®—linkedin.com/company/deitel-&-associates

and register for the free Deitel® Buzz Online e-mail newsletter at:

[www.deitel.com/newsletter/subscribe.html](www.deitel.com/newsletter/subscribe.html)

To communicate with the authors, send e-mail to:

deitel@deitel.com

For information on Dive-Into® Series on-site seminars offered by Deitel & Associates, Inc. worldwide, write to us at deitel@deitel.com or visit:

[www.deitel.com/training/](www.deitel.com/training/)

For continuing updates on Pearson/Deitel publications visit:

- [www.deitel.com](www.deitel.com)
- [www.pearsonhighered.com/deitel/](www.pearsonhighered.com/deitel/)

Visit the Deitel Resource Centers that will help you master programming languages, software development, Android and iOS app development, and Internet- and web-related topics:

In Loving Memory of Aunt Rochelle Deitel:

The most positive person we ever knew.
You brought joy to our lives.

Harvey, Barbara, Paul and Abbey
Trademarks

DEITEL, the double-thumbs-up bug and DIVE-INTO are registered trademarks of Deitel & Associates, Inc.
Apple, iOS, iPhone, iPad, iPod touch, Xcode, Swift, Objective-C, Cocoa and Cocoa Touch are trademarks or registered trademarks of Apple, Inc.
Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Throughout this book, trademarks are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.
Contents

Preface xix
Before You Begin xxvii

I Introduction to Swift and Xcode 6 1
1.1 Introduction 2
1.2 Apple’s OS X® and iOS® Operating Systems: A Brief History 3
1.3 Objective-C 3
1.4 Swift: Apple’s Programming Language of the Future 4
  1.4.1 Key Features of Many Popular Languages 4
  1.4.2 Performance 6
  1.4.3 Error Prevention 6
  1.4.4 Swift Standard Library 6
  1.4.5 Swift Apps and the Cocoa® and Cocoa Touch® Frameworks 7
  1.4.6 Swift and Objective-C Interoperability 9
  1.4.7 Other Apple Swift Resources 9
1.5 Can I Use Swift Exclusively? 9
  1.5.1 Objective-C Programmers Who Are Developing New iOS and OS X Apps 10
  1.5.2 Objective-C Programmers Who Are Enhancing Existing iOS and OS X Apps 10
  1.5.3 Java, C++ and C# Programmers Who Are New to iOS and OS X App Development 10
  1.5.4 Significant Language Changes Expected 10
  1.5.5 A Mixture of Swift and Objective-C 10
1.6 Xcode 6 Integrated Development Environment 10
1.7 Creating Swift Apps with Xcode 6 13
1.8 Web Resources 18

2 Introduction to Swift Programming 20
2.1 Introduction 21
2.2 A First Swift Program: Printing a Line of Text 21
2.3 Modifying Your First Program 23
2.4 Composing Larger Strings with String Interpolation 25
2.5 Another Application: Adding Integers 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6 Arithmetic</td>
<td>28</td>
</tr>
<tr>
<td>2.6.1 Automatic Arithmetic Overflow Checking</td>
<td>29</td>
</tr>
<tr>
<td>2.6.2 Operator Precedence</td>
<td>29</td>
</tr>
<tr>
<td>2.7 Decision Making: The <code>if</code> Conditional Statement and the Comparative Operators</td>
<td>29</td>
</tr>
<tr>
<td>2.8 Wrap-Up</td>
<td>32</td>
</tr>
<tr>
<td>3 Introduction to Classes, Objects, Methods and Functions</td>
<td>33</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>34</td>
</tr>
<tr>
<td>3.2 Account Class</td>
<td>35</td>
</tr>
<tr>
<td>3.2.1 Defining a Class</td>
<td>35</td>
</tr>
<tr>
<td>3.2.2 Defining a Class Attribute as a Stored Property</td>
<td>36</td>
</tr>
<tr>
<td>3.2.3 Defining a <code>public</code> Stored Property with a <code>private</code> Setter</td>
<td>37</td>
</tr>
<tr>
<td>3.2.4 Initializing a Class’s Properties with <code>init</code></td>
<td>37</td>
</tr>
<tr>
<td>3.2.5 Defining a Class’s Behaviors as Methods</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Creating and Using Account Objects</td>
<td>40</td>
</tr>
<tr>
<td>3.3.1 Importing the Foundation Framework</td>
<td>40</td>
</tr>
<tr>
<td>3.3.2 Creating and Configuring an <code>NSNumberFormatter</code> to Format Currency Values</td>
<td>41</td>
</tr>
<tr>
<td>3.3.3 Defining a Function—<code>formatAccountString</code></td>
<td>42</td>
</tr>
<tr>
<td>3.3.4 Creating Objects and Calling an <code>Initializer</code></td>
<td>42</td>
</tr>
<tr>
<td>3.3.5 Calling Methods on Objects—Depositing into Account Objects</td>
<td>43</td>
</tr>
<tr>
<td>3.3.6 Calling Methods on Objects—Withdrawing from Account Objects</td>
<td>44</td>
</tr>
<tr>
<td>3.4 Value Types vs. Reference Types</td>
<td>45</td>
</tr>
<tr>
<td>3.5 Software Engineering with Access Modifiers</td>
<td>46</td>
</tr>
<tr>
<td>3.6 Wrap-Up</td>
<td>47</td>
</tr>
<tr>
<td>4 Control Statements; Assignment, Increment and Logical Operators</td>
<td>48</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>49</td>
</tr>
<tr>
<td>4.2 Control Statements</td>
<td>49</td>
</tr>
<tr>
<td>4.3 <code>if</code> Conditional Statement</td>
<td>50</td>
</tr>
<tr>
<td>4.4 <code>if…else</code> Conditional Statement</td>
<td>50</td>
</tr>
<tr>
<td>4.5 Compound Assignment Operators</td>
<td>52</td>
</tr>
<tr>
<td>4.6 Increment and Decrement Operators</td>
<td>53</td>
</tr>
<tr>
<td>4.7 <code>switch</code> Conditional Statement</td>
<td>55</td>
</tr>
<tr>
<td>4.7.1 Using a <code>switch</code> Statement to Convert Numeric Grades to Letter Grades</td>
<td>55</td>
</tr>
<tr>
<td>4.7.2 Specifying Grade Ranges with the Closed-Range Operator (…)</td>
<td>56</td>
</tr>
<tr>
<td>4.7.3 The <code>default</code> Case</td>
<td>56</td>
</tr>
<tr>
<td>4.7.4 Other Patterns in the <code>case</code> Label</td>
<td>57</td>
</tr>
<tr>
<td>4.7.5 No Automatic Fall Through as in Other C-Based Languages</td>
<td>57</td>
</tr>
</tbody>
</table>
4.8 while Loop Statement 57
4.9 do...while Loop Statement 58
4.10 for...in Loop Statement and the Range Operators 58
  4.10.1 Iterating Over Collections of Values with Closed Ranges, 59
    Half-Open Ranges and the Global stride Function
  4.10.2 Compound-Interest Calculations with for...in 60
  4.10.3 Formatting Strings with Field Widths and Justification 61
  4.10.4 Performing the Interest Calculations 62
  4.10.5 A Warning about Displaying Rounded Values 62
4.11 for Loop Statement 63
  4.11.1 General Format of a for Statement 64
  4.11.2 Scope of a for Statement’s Control Variable 64
  4.11.3 Expressions in a for Statement’s Header Are Optional 64
4.12 break and continue Statements 64
  4.12.1 break Statement Example 64
  4.12.2 continue Statement Example 65
4.13 Logical Operators 66
  4.13.1 Logical AND (&&) Operator 66
  4.13.2 Logical OR (||) Operator 67
  4.13.3 Short-Circuit Evaluation of Complex Conditions 67
  4.13.4 Logical NOT (!) Operator 68
4.14 Wrap-Up 69

5 Functions and Methods: A Deeper Look; enums and Tuples 70
5.1 Introduction 71
5.2 Modules in Swift 72
5.3 Darwin Module—Using Predefined C Functions 73
5.4 Multiple-Parameter Function Definition 74
5.5 Random-Number Generation 76
5.6 Introducing Enumerations and Tuples 77
  5.6.1 Introducing Enumeration (enum) Types 80
  5.6.2 Tuples and Multiple Function Return Values 82
  5.6.3 Tuples as Function Arguments 83
  5.6.4 Accessing the Raw Value of an enum Constant 83
5.7 Scope of Declarations 84
5.8 Function and Method Overloading 86
5.9 External Parameter Names 88
5.10 Default Parameter Values 89
5.11 Passing Arguments by Value or by Reference 90
5.12 Recursion 92
5.13 Nested Functions 93
5.14 Wrap-Up 95
## 6 Arrays and an Introduction to Closures

6.1 Introduction 97
6.2 Arrays 98
6.3 Creating and Initializing Arrays 99
6.4 Iterating through Arrays 101
6.5 Adding and Removing Array Elements 104
6.6 Subscript Expressions with Ranges 107
6.7 Sorting Arrays; Introduction to Closures 108
   6.7.1 Closures and Closure Expressions 108
   6.7.2 Array Methods `sort` and `sorted` 109
   6.7.3 Sorting with Function `ascendingOrder` 111
   6.7.4 Using a Fully Typed Closure Expression 111
   6.7.5 Using a Closure Expression with Inferred Types 111
   6.7.6 Using a Closure Expression with Inferred Types and an Implicit `return` 112
   6.7.7 Using a Closure Expression with Shorthand Argument Names 112
   6.7.8 Using an Operator Function as a Closure Expression 112
   6.7.9 Reversing an Array’s Elements 112
6.8 Array Methods `filter`, `map` and `reduce` 112
   6.8.1 Filtering an Array 114
   6.8.2 Mapping an Array’s Elements to New Values 115
   6.8.3 Reducing an Array’s Elements to a Single Value 115
   6.8.4 Combining Filtering, Mapping and Reducing 116
6.9 Card Shuffling and Dealing Simulation; Computed Properties; Optionals 116
   6.9.1 Class `Card` 116
   6.9.2 Class `DeckOfCards` 117
   6.9.3 `DeckOfCards` Initializer 118
   6.9.4 `DeckOfCards` Method `shuffle` 119
   6.9.5 `DeckOfCards` Method `dealCard` and Optional Return Values 119
   6.9.6 Shuffling and Dealing Cards 119
   6.9.7 Unwrapping Optional Values with Optional Binding and the `if` or `while` Statements 121
6.10 Passing Arrays to Functions 121
   6.10.1 Passing an Entire Array By Value 123
   6.10.2 Passing One Array Element By Value 123
   6.10.3 Passing an Entire Array By Reference 123
   6.10.4 Passing One Array Element By Reference 124
6.11 Notes on Pass-By-Value and Pass-By-Reference 124
6.12 Multidimensional Arrays 124
6.13 Variadic Parameters 128
6.14 Wrap-Up 129

## 7 Dictionary

7.1 Introduction 132
10 Inheritance, Polymorphism and Protocols

10.1 Introduction
  10.1.1 Superclasses and Subclasses 215
  10.1.2 Polymorphism 216
  10.1.3 Implementing for Extensibility 216
  10.1.4 Programming in the Specific 216
  10.1.5 Protocols 217

10.2 Superclasses and Subclasses 217

10.3 An Inheritance Hierarchy: CommunityMembers 218

10.4 Case Study: Using Inheritance to Create Related Employee Types 218
  10.4.1 Superclass CommissionEmployee 220
  10.4.2 Subclass BasePlusCommissionEmployee 221
  10.4.3 Testing the Class Hierarchy 224

10.5 Access Modifiers in Inheritance Hierarchies 226

10.6 Introduction to Polymorphism: A Polymorphic Video Game Discussion 227

10.7 Case Study: Payroll System Class Hierarchy Using Polymorphism 228
  10.7.1 Base Class Employee 229
  10.7.2 Subclass SalariedEmployee 231
  10.7.3 Subclass CommissionEmployee 232
  10.7.4 Indirect Subclass BasePlusCommissionEmployee 233
  10.7.5 Polymorphic Processing 235

10.8 Case Study: Creating and Using Custom Protocols 238
  10.8.1 Protocol Capabilities Must Be Defined in Each Conforming Type 238
  10.8.2 Protocols and Is-a Relationships 238
  10.8.3 Relating Disparate Types Via Protocols 238
  10.8.4 Accounts-Payable Application 239
  10.8.5 Developing a Payable Hierarchy 239
  10.8.6 Declaring Protocol Payable 240
  10.8.7 Creating Class Invoice 241
  10.8.8 Using extensions to Add Printable and Payable Protocol Conformance to Class Employee
    Polymorphically 242
  10.8.9 Using Protocol Payable to Process Invoices and Employees
    Polymorphically 244

10.9 Additional Protocol Features 246
  10.9.1 Protocol Inheritance 246
  10.9.2 Class-Only Protocols 246
  10.9.3 Optional Capabilities in Protocols 246
  10.9.4 Protocol Composition 247
  10.9.5 Common Protocols in Swift 247
12.6.1 Box Type with Custom Subscripts 279
12.6.2 Subscript Syntax 281
12.6.3 Type Box’s Int Subscript and the precondition Function 281
12.6.4 Type Box’s String Subscript 282
12.6.5 Using Type Box’s Subscripts 282
12.7 Custom Operators 283
12.7.1 Precedence andAssociativity 283
12.7.2 Symbols Used in Custom Operators 284
12.7.3 Defining a Custom Exponentiation Operator for Type Int 285
12.8 Custom Generic Operators 286
12.9 Wrap-Up 287

13 iOS 8 App Development: Welcome App 288
13.1 Introduction 289
13.2 Technologies Overview 290
13.2.1 Xcode and Interface Builder 290
13.2.2 Labels and Image Views 290
13.2.3 Asset Catalogs and Image Sets 291
13.2.4 Running the App 291
13.2.5 Accessibility 291
13.2.6 Internationalization 291
13.3 Creating a Universal App Project with Xcode 291
13.3.1 Xcode Projects and App Templates 291
13.3.2 Creating and Configuring a Project 292
13.4 Xcode Workspace Window 293
13.4.1 Navigator Area 294
13.4.2 Editor Area 294
13.4.3 Utilities Area and Inspectors 295
13.4.4 Debug Area 295
13.4.5 Xcode Toolbar 295
13.4.6 Project Navigator 296
13.4.7 Keyboard Shortcuts 296
13.5 Storyboarding the Welcome App’s UI 296
13.5.1 Configuring the App for Portrait and Landscape Orientations 297
13.5.2 Providing an App Icon 297
13.5.3 Creating an Image Set for the App’s Image 299
13.5.4 Overview of the Storyboard and the Xcode Utilities Area 300
13.5.5 Adding an Image View to the UI 302
13.5.6 Using Inspectors to Configure the Image View 302
13.5.7 Adding and Configuring the Label 304
13.5.8 Using Auto Layout to Support Different Screen Sizes and Orientations 306
13.6 Running the Welcome App 308
13.6.1 Testing on the iOS Simulator 308
13.6.2 Testing on a Device (for Paid Apple iOS Developer Program Members Only) 311
13.7 Making Your App Accessible 311
  13.7.1 Enabling Accessibility for the Image View 311
  13.7.2 Confirming Accessibility Text with the Simulator's Accessibility Inspector 312
13.8 Internationalizing Your App 313
  13.8.1 Locking Your UI During Translation 314
  13.8.2 Exporting Your UI’s String Resources 315
  13.8.3 Translating the String Resources 316
  13.8.4 Importing the Translated String Resources 316
  13.8.5 Testing the App in Spanish 317
13.9 Wrap-Up 318

14 iOS 8 App Development: Tip Calculator App 319
14.1 Introduction 320
14.2 Test-Driving the Tip Calculator App in the iPhone and iPad Simulators 321
14.3 Technologies Overview 322
  14.3.1 Swift Programming 322
  14.3.2 Swift Apps and the Cocoa Touch® Frameworks 322
  14.3.3 Using the UIKit and Foundation Frameworks in Swift Code 323
  14.3.4 Creating Labels, a Text Field and a Slider with Interface Builder 324
  14.3.5 View Controllers 324
  14.3.6 Linking UI Components to Your Swift Code 324
  14.3.7 Performing Tasks After a View Loads 325
  14.3.8 Bridging Between Swift and Objective-C Types 325
14.4 Building the App’s UI 325
  14.4.1 Creating the Project 325
  14.4.2 Configuring the Size Classes for Designing a Portrait Orientation iPhone App 327
  14.4.3 Adding the UI Components 327
  14.4.4 Adding the Auto Layout Constraints 334
14.5 Creating Outlets with Interface Builder 337
14.6 Creating Actions with Interface Builder 340
14.7 Class ViewController 341
  14.7.1 import Declarations 342
  14.7.2 ViewController Class Definition 342
  14.7.3 ViewController's @IBOutlet Properties 342
  14.7.4 Other ViewController Properties 343
  14.7.5 Overridden UIViewController method viewDidLoad 344
  14.7.6 ViewController Action Method calculateTip 345
  14.7.7 Global Utility Functions Defined in ViewController.swift 347
14.8 Wrap-Up 349
A Keywords  351

B Operator Precedence Chart  352

C Labeled break and continue Statements  354
  C.1 Introduction  354
  C.2 Labeled break Statement  354
  C.3 Labeled continue Statement  355

Index  357
This page intentionally left blank
Welcome to Apple’s new Swift programming language and *Swift for Programmers*! This book presents leading-edge computing technologies for software developers. It’s designed primarily for three audiences of developers who already know object-oriented programming and are considering using Swift:

- Objective-C programmers who are developing *new* iOS and/or OS X apps and who want to quickly begin using Swift in their apps.
- Objective-C programmers who are enhancing *existing* iOS and/or OS X apps and who want to quickly begin using Swift in their apps.
- Java, C++, and C# programmers who are new to iOS and OS X development and who want to start developing iOS and/or OS X apps in Swift.

Chapters 1 through 12 focus on Swift programming, then Chapters 13 and 14 briefly introduce iOS 8 app development. The iOS 8 chapters are condensed versions of Chapters 2 and 3 of our book, *iOS® 8 for Programmers: An App-Driven Approach with Swift™*, in which we focus on building many complete iPhone® and iPad® apps.¹

We emphasize software engineering best practices. At the heart of the book is the Deitel signature “live-code approach.” Rather than using only code snippets, we present most concepts in the context of complete working Swift programs that run on OS X® and—in the last two chapters—iOS® 8. Each complete code example is accompanied by one or more live sample executions. In the few cases where we use code snippets, we always extract them from compiled, correctly executing, live-code examples. All of the book’s source code is available at

http://www.deitel.com/books/SwiftFP

Some complete live-code programs might appear to be code snippets—this is because Swift eliminates various items that are common in many C-based languages, such as the need for a `main` method. For example, the following is actually a complete Swift program:

```swift
println("Welcome to Swift Programming!")
```

### Swift Programming Language

Swift was a surprise announcement at Apple’s WWDC (Worldwide Developer Conference) in June 2014. Because the language is so new, it’s likely to evolve quickly over the next few years. Here’s some key aspects of Swift:

---

¹ Swift is a young language that’s evolving rapidly. We plan to post bonus content covering important new features as they emerge. See [http://www.deitel.com/books/SwiftFP](http://www.deitel.com/books/SwiftFP) for details.
Apple’s Language of the Future—Apple is the most valuable technology company in the world, and they’ve declared that Swift is their language of the future for app and systems programming.

Popular Language Features—Swift is a contemporary language with simpler syntax than Objective-C. Because Swift is new, its designers were able to include popular features like those in Objective-C, Java, C#, Ruby, Python and many others. These features (which are listed in Fig. 1.1) include type inference, tuples, closures (lambdas), generics, operator overloading, functions with multiple return values, optionals, String interpolation, switch statement enhancements and more. We’ve found it easier and faster to develop iOS and OS X apps in Swift than in Objective-C.

Performance—Swift was designed for better performance than Objective-C. Apple has observed that Swift code is about 1.5 times faster than Objective-C code on today’s multi-core systems.

Error Prevention—Swift eliminates many common programming errors, making your code more robust and secure. Some of these error prevention features (which are listed in Fig. 1.2) include automatic memory management, no pointers, required braces around every control statement’s body, assignment operators that do not return values, requiring initialization of all variables and constants before they’re used, array bounds checking, automatic checking for overflow of integer calculations, and more.

Interoperability with Objective-C—You can combine Swift and Objective-C in the same app. This enables you to enhance existing Objective-C apps without having to rewrite all the code. Your apps will easily be able to interact with the Cocoa/Cocoa Touch frameworks, which are largely written in Objective-C.

Playgrounds—A playground is an Xcode window in which you can enter Swift code that compiles and executes as you type it. This allows you to see and hear your code’s results as you write it, to quickly find and fix errors, and to experiment with features of Swift and the Cocoa/Cocoa Touch frameworks.

Software Used in Swift for Programmers
To execute our Swift examples and write your own Swift code, you must install Xcode 6, which is available free from the Mac App Store. When you open Xcode for the first time, it will download and install additional features required for development. For the latest information about Xcode, visit

https://developer.apple.com/xcode

Swift Fundamentals: Parts I, II and III LiveLessons Video Training
Our Swift Fundamentals: Parts I, II and III LiveLessons video training product shows you what you need to know to start building robust, powerful software with Swift. It includes approximately 20 hours of expert training synchronized with Swift for Programmers. For additional information about Deitel LiveLessons video products, visit

http://www.deitel.com/livelessons

or contact us at deitel@deitel.com.
You also can access our books and LiveLessons videos on Safari Books Online if you have an appropriate subscription. A limited free-trial is available. Safari is popular with large companies, colleges, libraries and individuals who would like access to video training and electronic versions of print publications.

Explosive Growth of the iPhone and iPad Is Creating Opportunity for Developers

iPhone and iPad device sales have been growing exponentially, creating significant opportunities for iOS app developers. The first-generation iPhone, released in June 2007, sold 6.1 million units in its initial five quarters of availability. The iPhone 5s and the iPhone 5c, released simultaneously in September 2013, sold over nine million combined in the first three days of availability. The most recent iPhone 6 and iPhone 6 Plus, announced in September 2014, pre-sold four million combined in just one day—double the number of iPhone 5 pre-sales in its first day of pre-order availability. Apple sold 10 million iPhone 6 and iPhone 6 Plus units combined in their first weekend of availability.

Sales of the iPad are equally impressive. The first generation iPad, launched in April 2010, sold 3 million units in its first 80 days of availability and over 40 million worldwide by September 2011. The iPad mini with Retina display (the second-generation iPad mini) and the iPad Air (the fifth-generation iPad) were released in November 2013. In just the first quarter of 2014, Apple sold a record 26 million iPads.

There are over 1.3 million apps in the App Store and over 75 billion iOS apps have been downloaded. The potential for iOS app developers is enormous. It’s likely that most new iOS and OS X development soon will be done in Swift, so there are great opportunities for Swift programmers.

Our Research Sources

Due to Swift’s similarities with many of today’s popular programming languages, we were able to repurpose and customize examples from many of our other programming textbooks and professional books. Because Swift is new, we performed most of our research using the Apple resources listed on the next page.

Preface

- *The Swift Programming Language*—available in the iBooks store and at:

- *Using Swift with Cocoa and Objective-C*—available in the iBooks store and at:

- The *Swift Standard Library Reference*:

- The Swift Blog:
  https://developer.apple.com/swift/blog/

- World Wide Developers Conference (WWDC) 2014 videos:

**Teaching Approach**

*Swift for Programmers* contains numerous complete working code examples. We stress program clarity and concentrate on building well-engineered, high-performance software.

**Syntax Coloring.** For readability, we syntax color all the Swift code, similar to the syntax coloring in the Xcode 6 integrated-development environment. Our conventions are:

- comments appear in green
- keywords appear in dark blue
- constants and literal values appear in light blue
- all other code appears in black

**Code Highlighting.** We place colored rectangles around key code segments.

**Using Fonts for Emphasis.** We place key terms and the index’s page reference for each term’s defining occurrence in *bold colored* text for easier reference. We emphasize on-screen components in the *bold Helvetica* font (e.g., the *File* menu) and emphasize Swift program text in the *Lucida* font (for example, `println()`).

**Objectives/Outline.** Each chapter begins with a list of objectives and a chapter outline.

**Illustrations/Figures.** Abundant tables, programs and program outputs are included.

**Programming Tips.** We include programming tips to help you focus on important aspects of program development. These tips and practices represent the best we’ve gleaned from a combined eight decades of programming experience.

---

**Good Programming Practices**

*The Good Programming Practices call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

---

**Common Programming Errors**

*Pointing out these Common Programming Errors reduces the likelihood that you’ll make them.*
Index. We’ve included an extensive index. Each key term’s defining occurrence is highlighted with a bold colored page number.

**Academic Bundle iOS® 8 for Programmers and Swift™ for Programmers**
The Academic Bundle iOS® 8 for Programmers and Swift™ for Programmers is designed for professionals, students and instructors interested in learning or teaching iOS 8 app development with a broader and deeper treatment of Swift. You can conveniently order the Academic Bundle from pearsonhighered.com with one ISBN: 0-13-408775-5. The Academic Bundle includes:

- Swift™ for Programmers (print book)
- iOS® 8 for Programmers: An App Driven Approach with Swift™, Volume 1, 3/e (print book)
- Access Code Card for Academic Package to accompany Swift™ for Programmers
- Access Code Card for Academic Package to accompany iOS® 8 for Programmers: An App Driven Approach with Swift™, Volume 1, 3/e

The two Access Code Cards for the Academic Packages (when used together) give you access to the companion websites, which include self-review questions (with answers), short-answer questions, programming exercises, programming projects and selected videos chosen to get you up to speed quickly with Xcode 6, visual programming and basic Swift-based, iOS 8 programming.

**Ordering the Books and Supplements Separately**
The print books and Access Code Cards may be purchased separately from pearsonhighered.com using the following ISBNs (email deitel@deitel.com if you have questions):

- Standalone access code card for Academic Package to accompany Swift™ for Programmers: ISBN 0-13-405818-6
- Standalone access code card for Academic Package to accompany iOS® 8 for Programmers: An App Driven Approach with Swift™, Volume 1, 3/e. ISBN 0-13-405825-9
Instructor Supplements
Instructor supplements are available online at Pearson’s Instructor Resource Center (IRC). The supplements include:

- Solutions Manual with selected solutions to the short-answer exercises.
- Test Item File of multiple-choice examination questions (with answers).
- PowerPoint® slides with the book’s source code and tables.

Please do not write to us requesting access to the Pearson Instructor’s Resource Center. Certified instructors who adopt the book for their courses can obtain password access from their regular Pearson sales representatives (www.pearson.com/repllocator). Solutions are not provided for “project” exercises.

Acknowledgments

Deitel Team
We’d like to thank Abbey Deitel and Barbara Deitel of Deitel & Associates, Inc. for long hours devoted to this project. Abbey co-authored Chapter 1 and this Preface, and she and Barbara painstakingly researched the world of Swift. Our Art Director, Jessica Deitel (age 10) chose the cover color.

Pearson Education Team
We’re fortunate to have worked on this project with the dedicated publishing professionals at Prentice Hall/Pearson. We appreciate the extraordinary efforts and 20-year mentorship of our friend and professional colleague Mark L. Taub, Editor-in-Chief of Pearson Technology Group. Kim Boedigheimer recruited distinguished members of the iOS, OS X and emerging Swift communities to review the manuscript and she managed the review process. We selected the cover art and Chuti Prasertsith designed the cover. John Fuller managed the book’s production.

Reviewers
We wish to acknowledge the efforts of our reviewers. They scrutinized the text and the programs and provided countless suggestions for improving the presentation.

- Scott Bossack, Lead iOS Developer, Thrillist Media Group
- René Cacheaux, iOS Architect, Mutual Mobile
- Ash Furrow, iOS Developer, Artsy
- Rob McGovern, Independent Contractor
- Abizer Nasir, Freelance iOS and OS X Developer, Jungle Candy Software Ltd.
- Rik Watson, Technical Team Lead for HP Enterprise Services (Applications Services)
- Jack Watson-Hamblin, Programming Writer and Teacher, MotionInMotion (https://motioninmotion.tv/)

A Special Thank You to Reviewer Charles Brown
When Swift was announced in June 2014, within days our publisher, Prentice Hall/Pearson, agreed to publish our Swift book, which at the time was just an idea. One key prob-
Preface

—where would we find Swift reviewers when the language was so new? We asked for help from our 75,000 social media and newsletter followers. Charles E. Brown, Independent Contractor affiliated with Apple and Adobe, was the first to respond and became the core member of our review team. He mentored us throughout the project, providing insights, encouragement, answers to our technical questions and appropriate cautions.

Keeping in Touch with the Authors

As you read the book, if you have questions, comments or suggestions, send an e-mail to us at

deitel@deitel.com

and we’ll respond promptly. For updates on this book, visit

http://www.deitel.com/books/SwiftFP

subscribe to the Deitel® Buzz Online newsletter at

http://www.deitel.com/newsletter/subscribe.html

and join the Deitel social networking communities on

- Facebook® (http://facebook.com/DeitelFan)
- Twitter® (@deitel)
- Google+™ (http://google.com/+DeitelFan)
- YouTube® (http://youtube.com/DeitelTV)
- LinkedIn® (http://linkedin.com/company/deitel-&-associates)

Well, there you have it! As you read the book, we’d appreciate your comments, criticisms, corrections and suggestions for improvement. Please address all correspondence to:

deitel@deitel.com

We’ll respond promptly. We hope you enjoy working with Swift for Programmers as much as we enjoyed writing it!

Paul and Harvey Deitel

About the Authors

Paul Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. He holds the Java Certified Programmer and Java Certified Developer designations, and is an Oracle Java Champion. Paul was also named as a Microsoft® Most Valuable Professional (MVP) for C# in 2012–2014. Through Deitel & Associates, Inc., he has delivered hundreds of programming courses worldwide to clients, including Cisco, IBM, Siemens, Sun Microsystems (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world’s best-selling programming-language textbook/professional book/video authors.
Dr. Harvey Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has over 50 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul. The Deitels’ publications have earned international recognition, with translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, academic, government and military clients.

About Deitel® & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in mobile app development, computer programming languages, object technology and Internet and web software technology. The company’s training clients include many of the world’s largest companies, government agencies, branches of the military and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including Swift and iOS app development, Java™, Android app development, C++, C, Visual C#®, Visual Basic®, Python®, object technology, Internet and web programming and a growing list of additional programming and software development courses.

Through its 39-year publishing partnership with Pearson/Prentice Hall, Deitel & Associates, Inc., publishes leading-edge programming textbooks and professional books in print and a wide range of e-book formats, and *LiveLessons* video courses. Deitel & Associates, Inc. and the authors can be reached at:

deitel@deitel.com

To learn more about Deitel’s *Dive-Into® Series* Corporate Training curriculum, visit:

http://www.deitel.com/training

To request a proposal for worldwide on-site, instructor-led training at your organization, e-mail deitel@deitel.com.

Individuals wishing to purchase Deitel books and *LiveLessons* video training can do so through www.deitel.com. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit:

Before You Begin

This section contains information you should review before using this book. Updates to the information presented here will be posted at:

http://www.deitel.com/books/SwiftFP

Conventions

Font and Naming
We use fonts to distinguish between on-screen components (such as menu names and menu items) and Swift code or commands. Our convention is to emphasize on-screen components in a sans-serif bold Helvetica font (for example, File menu) and to emphasize Swift code and commands in a sans-serif Lucida font (for example, println()). When building user interfaces (UIs) using Xcode’s Interface Builder, we also use the bold Helvetica font to refer to property names for UI components (such as a Label’s Text property).

Conventions for Referencing Menu Items in a Menu
We use the > character to indicate selecting a menu item from a menu. The notation File > Open... indicates that you should select the Open... menu item from the File menu.

Software Used in this Book
To execute our Swift examples and write your own Swift code, you must install Xcode 6. You can install the currently released Xcode version for free from the Mac App Store. When you open Xcode for the first time, it will download and install additional features required for development. For the latest information about Xcode, visit

https://developer.apple.com/xcode

A Note Regarding the Xcode 6 Toolbar Icons
We developed this book’s examples with Xcode 6 on OS X Yosemite. If you’re running OS X Mavericks, some Xcode toolbar icons we show in the text may differ on your screen.

Becoming a Registered Apple Developer
Registered developers have access to the online iOS and OS X documentation and other resources. Apple also now makes Xcode pre-release versions (such as the next point release or major version) available to all registered Apple developers. To register, visit:

https://developer.apple.com/register
Before You Begin

To download the next pre-release Xcode version, visit:


Once you download a prerelease DMG (disk image) file, double click it to launch the installer, then follow the on-screen instructions.

Fee-Based iOS Developer Programs

In Chapters 13–14, you’ll build two iOS apps and test them on your Mac using the iOS simulator that’s bundled with Xcode. If you’d like to run iOS apps on actual iOS devices, you must be a member of one of the following iOS developer programs.

iOS Developer Program

The fee-based iOS Developer Program allows you to load your iOS apps onto iOS devices for testing and to submit your apps to the App Store. If you intend to distribute iOS apps, you’ll need to join the fee-based program. You can sign up at

https://developer.apple.com/programs

iOS Developer Enterprise Program

Organizations may register for the iOS Developer Enterprise Program at


which enables developers to deploy proprietary iOS apps to employees within their organization.

iOS Developer University Program

Colleges and universities interested in offering iOS app-development courses can apply to the iOS Developer University Program at


Qualifying schools receive free access to all the developer tools and resources. Students can share their apps with each other and test them on iOS devices.

Adding Your Paid iOS Developer Program Account to Xcode

Xcode can interact with your paid iOS and OS X Developer Program accounts on your behalf so that you can install apps onto your iOS devices for testing. If you have a paid iOS Developer Program account, you can add it to Xcode. To do so:

1. Select Xcode > Preferences....
2. In the Accounts tab, click the + button in the lower left corner and select Add Apple ID....
3. Enter your Apple ID and password, then click Add.

Obtaining the Code Examples

The Swift for Programmers examples are available for download as a ZIP file from

http://www.deitel.com/books/SwiftFP
under the heading Download Code Examples and Other Premium Content. When you click the link to the ZIP file, it will be placed by default in your user account’s Downloads folder. We assume that the examples are located in the SwiftFPExamples folder in your user account’s Documents folder. You can use Finder to move the ZIP file there, then double click the file to extract its contents.

Xcode Playgrounds and Projects for the Code Examples

Playgrounds are a new interactive coding capability in Xcode 6. They execute Swift code as you write it. They’re particularly useful for learning and experimenting with Swift or the Cocoa and Cocoa Touch frameworks that are used to build iOS and OS X apps. Projects, on the other hand, are used to manage all the files for each app that you create.

For each example, we provide one of the following:

- an Xcode playground file with the .playground extension
- an Xcode project for an OS X Command Line Tool app that produces text output (such projects don’t require you to develop a GUI or to run apps in the iOS simulator)
- an Xcode project for an iOS 8 app that runs in the iOS simulator bundled with Xcode.

An Xcode project is stored in a folder with the project’s name. In that folder is a file with a .xcodeproj extension. You can double click a .playground or .xcodeproj file to open it in Xcode. Throughout this book, we use playgrounds for single-source-file examples and projects for multi-source-file examples.

Use Playgrounds for Learning

We recommend that as you learn Swift, you enter each example’s code into an Xcode 6 playground so that you can immediately see the code in action as you write it. Sometimes you might need to restart the IDE if a playground stops working correctly. If you enter any of our multi-source-file examples into a playground, you must define any functions and types before they’re used.

Viewing Output in a Playground

In a playground, the results of any output statements are visible only if the Assistant Editor is displayed. To open it in a playground, select Assistant Editor > Show Assistant Editor from Xcode’s View menu. The Assistant Editor will appear at the playground window’s right side.

Playground and Project Naming Conventions

Each project or playground is named based on its figure number(s) or the concept being presented. The comment in the first line of a source code file contains information to help you identify which playground or project to open from the chapter’s examples folder:

- the project’s or playground’s base name—e.g., fig02-01 and fig03-01-11 correspond to fig02-01.playground and fig03-01-11.xcodeproj, respectively.
- the project’s or playground’s complete name—e.g., CompoundInterest.playground or Inheritance.xcodeproj.
Configuring Xcode to Display Line Numbers

Many programmers find it helpful to display line numbers in the code editor. To do so:

1. Open Xcode and select Preferences... from the Xcode menu.
2. Select the Text Editing tab, then ensure that the Editing subtab is selected.
3. Check the Line Numbers checkbox.

You’re now ready to begin learning Swift with Swift for Programmers. We hope you enjoy the book! If you have any questions, please email us at deitel@deitel.com.
Functions and Methods: A Deeper Look; enums and Tuples

Objectives
In this chapter you’ll:

■ Learn about Swift modules (for software reuse).
■ Define functions with multiple parameters.
■ Use random-number generation to implement a game-playing app.
■ Use enum types to create sets of named constants.
■ Return multiple values from a function via a tuple, pass a tuple to a function and access a tuple’s elements.
■ Learn how an identifier’s scope limits its visibility to specific parts of a program.
■ Create overloaded functions.
■ Learn how local and external parameter names are used in function and method calls.
■ Use default parameter values in function calls.
■ Pass method arguments by value and by reference.
■ Define a recursive function.
■ Define a nested function.
5.1 Introduction

We introduced functions and methods in Chapter 3. The key distinction between a function and a method is that any function defined *in a type* is a method.

In this chapter, we begin by discussing modules, which Swift uses to package related software components for reuse. We introduce Darwin—Apple’s UNIX-based core of OS X and iOS—and import Darwin features (such as a C-based random-number-generation function) for use in apps.

We discuss random-number generation and develop a version of a popular casino dice game. That example demonstrates basic `enum` types for creating named constants that improve the readability of the code. You’ll see that Swift’s `enum` constants can have values, but that’s not required. The example also presents tuples—collections of values of the same or different types. We return multiple values from a function via a tuple, pass a tuple to a function and access a tuple’s elements via both names and indices.

Next, we discuss Swift’s scope rules. Then, we introduce the concept of *overloading*. You’ll frequently see identically named functions or, within a type, identically named methods. This overloading is used to implement functions or methods that perform similar tasks but with different types and/or different numbers of parameters. This chapter demonstrates overloading with functions, and you’ll see examples of method overloading in later chapters.

We discuss the differences between calling functions and methods and present the concepts of local vs. external parameter names. As you’ll see, external parameter names must be used in a function call to label all of the corresponding arguments. This is another distinction between functions and methods—by default, methods require their second and subsequent arguments to be labeled with parameter names. This has to do with the similarities between how methods are named in Objective-C and Swift, which we discuss in Section 5.9. We also mention how to disable this feature when calling methods. Parameter names are always required in initializer calls.

We use a default parameter value that the compiler inserts in a function call if you do not provide the corresponding argument when the function is called. We discuss how value- and reference-type arguments are passed to methods, then demonstrate how to pass...
arguments by reference using the keyword `inout`. You’ll write recursive functions (functions that call themselves) and nested functions.

Many of the features presented as functions in this chapter also apply to methods and initializers in the new types you create. We’ll point out key differences between functions, methods and initializers.

## 5.2 Modules in Swift

Swift apps are written by combining new functions and types, properties, methods, classes, structs (Chapter 9) and enums (introduced in Section 5.6 and discussed in more detail in Chapter 9) with predefined capabilities in the Swift Standard Library, the Cocoa and Cocoa Touch frameworks, and other class libraries. Figure 5.1 overviews some functions, types and protocols (similar to interfaces in other languages) from the Swift Standard Library. You can locate additional information about Swift Standard Library types and functions in the *Swift Standard Library Reference* at [http://bit.ly/SwiftStandardLibrary](http://bit.ly/SwiftStandardLibrary).

At the time of this writing, the *Swift Standard Library Reference* is not yet complete. There are many other built-in free functions (sometimes called global functions), but only a few are currently listed. Similarly, there are other protocols not yet included in the reference, but mentioned in other Swift documentation (e.g., `Hashable` and `DebugPrintable`).

### Fig. 5.1 Some Swift Standard Library features, (Part 1 of 2.)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Types</strong></td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>This type is used to represent arrays—collections of related data items. Type Array provides many initializers, properties, methods and operators for performing common array manipulations. Chapter 6 discusses type Array in detail.</td>
</tr>
<tr>
<td>Dictionary</td>
<td>A Dictionary maps unique keys to values—for example, an employee’s ID number can be mapped to one employee’s information. Type Dictionary provides many initializers, properties, methods and operators for performing common manipulations of key–value pairs. Chapter 7 discusses type Dictionary in detail.</td>
</tr>
<tr>
<td>Boolean and numeric types</td>
<td>As you’ve seen, Swift provides type <code>Bool</code> and integer and floating-point numeric types (Fig. 2.6). These are the equivalent of what many programming languages refer to as the built-in, primitive or fundamental types.</td>
</tr>
<tr>
<td>String</td>
<td>Strings are collections of characters. Type String provides many initializers, properties, methods and operators for performing common String manipulations. We present details of type String throughout the book.</td>
</tr>
<tr>
<td><strong>Protocols</strong></td>
<td></td>
</tr>
<tr>
<td>Comparable</td>
<td>An item that is <code>Comparable</code> can be compared with another item of the same type using the <code>&lt;</code> operator. Strings and all of Swift’s integer and floating-point numeric types are <code>Comparable</code>. We discuss how to make your own types <code>Comparable</code> in Chapter 12, Operator Overloading and Subscripts.</td>
</tr>
</tbody>
</table>
5.3 Darwin Module—Using Predefined C Functions

Related software components in Objective-C are grouped into frameworks (similar to namespaces or packages in other languages) so that they can be reused in Cocoa and Cocoa Touch apps. Swift’s equivalent to a framework is a module. When you create a Swift project, Xcode places all of the project’s Swift code in a module with the same name as your project. If you create a Swift-based Cocoa Framework project or Cocoa Touch Framework project, you can then reuse that framework in Cocoa and Cocoa Touch apps by importing it with the `import` keyword (as you did with the Foundation framework in Fig. 3.6).

### Software Engineering Observation 5.1

Don’t try to “reinvent the wheel.” When possible, reuse capabilities of the Swift Standard Library, the Cocoa and Cocoa Touch frameworks, and other libraries. This reduces app development time, avoids introducing programming errors and contributes to good app performance.

### 5.3 Darwin Module—Using Predefined C Functions

Just as your Swift apps can reuse Cocoa and Cocoa Touch frameworks (written largely in Objective-C), they can also reuse C-based UNIX functions (such as `arc4random_uniform` in Section 5.5) and C Standard Library functions (such as the common C math functions listed in Fig. 5.2) that are built into OS X and iOS. These and many other features of UNIX and C are available via the Darwin module, which provides access to the C libraries in Darwin—Apple’s open-source UNIX-based core on which the OS X and iOS operating systems are built. To import the Darwin module, use the following `import` declaration:

```plaintext
import Darwin
```

The Darwin module is imported by default into several Cocoa and Cocoa Touch frameworks—such as Foundation, AppKit and UIKit—so that various software components in those frameworks can interact with the underlying operating system.
### Chapter 5 Functions and Methods: A Deeper Look; enums and Tuples

#### 5.4 Multiple-Parameter Function Definition

In previous chapters, you called functions, methods and initializers with varying numbers of arguments. You also defined functions and methods with only one parameter. In this section, we define and call a function with multiple parameters.

Figure 5.3 defines a function `maximum` (lines 4–18) that determines and returns the largest of three `Double` values. Lines 21–23 call `maximum` with the largest value (3.3) as the first, second or third argument, respectively, to show that the function always returns the largest of its three arguments.

```swift
// fig05-03: Function maximum with three Double parameters.
// returns the maximum of its three Double parameters
func maximum(x: Double, y: Double, z: Double) -> Double {
    var maximumValue = x // assume x is the largest to start
    if y > maximumValue { maximumValue = y }
    if z > maximumValue { maximumValue = z }
    return maximumValue
}
```

Fig. 5.2 | Some math functions from the C Standard Library.

Fig. 5.3 | Function `maximum` with three `Double` parameters. (Part 1 of 2.)
5.4 Multiple-Parameter Function Definition

Function maximum

Line 4 indicates that maximum requires three Double parameters (x, y, and z) to accomplish its task and returns a Double. There must be one argument in the function call for each parameter in the function definition. Also, each argument must match the type of the corresponding parameter. Parameters are constants by default—if you need to modify a parameter’s value in the function’s body, you must place var before the parameter’s name.

Three Ways to Return Control from a Function

There are three ways to return control to the statement that calls a function. If the function’s return type is Void (that is, it does not return a result), control returns when the function-ending right brace is reached or when the statement

```
return
```

is executed from the function's body. If the function returns a result, the statement

```
return expression
```

evaluates the expression, then returns the result (and control) to the caller (as in line 17).
Swift Function max
Swift provides a max function that can be used to compare two values of the same Comparable type—all of Swift’s numeric types and Strings are Comparable. A second version of max takes a variable number of arguments and is used to compare three or more arguments of the same Comparable type. You’ll create your own functions with variable-length parameter lists in Chapter 6, Arrays and an Introduction to Closures. There is no need for us to define our own maximum function, as we could have replaced the maximum calls in lines 21–23 with:

```swift
max(3.3, 2.2, 1.1)
max(1.1, 3.3, 2.2)
max(2.2, 1.1, 3.3)
```

5.5 Random-Number Generation
We now take a brief diversion into a popular type of programming application—simulation and game playing. In this and the next section, we develop a game-playing program with multiple functions.

The element of chance can be introduced in a program via the arc4random_uniform function (a C-based UNIX function from the Darwin module), which produces random unsigned 32-bit integers (UInt32; see Fig. 2.6) from 0 up to but not including an upper bound that you specify as an argument. There’s also function arc4random, which takes no arguments and returns a random unsigned 32-bit integer in the range 0 (UInt32.min) to 4,294,967,295 (UInt32.max).

Both functions use the RC4 (also called ARCFOUR) random-number generation algorithm (http://en.wikipedia.org/wiki/RC4) and produce nondeterministic random numbers that cannot be predicted. To use these functions, you must import the Darwin module (Section 5.3).

**Error-Prevention Tip 5.2**
Functions arc4random_uniform and arc4random cannot produce repeatable random-number sequences. If you require repeatability for testing, use the Darwin module’s C function random to obtain the random values and function srandom to seed the random-number generator with the same seed during each program execution. Once you’ve completed testing, use either arc4random_uniform or arc4random to produce random values.

Obtaining a Random Value with arc4random
The following statement generates a random UInt32 value in the range 0 (UInt32.min) to 4,294,967,295 (UInt32.max):

```swift
let randomValue = arc4random()
```

Obtaining a Random Value in a Specific Range with arc4random_uniform
The range of values produced by arc4random generally differs from the range of values required in a particular app. For example, a program that simulates the rolling of a six-sided die might require random integers in the range 1–6. For cases like this, we’ll use the function arc4random_uniform.

To demonstrate arc4random_uniform, let’s develop a program that simulates 20 rolls of a six-sided die and displays the value of each roll. First, we use arc4random_uniform to produce random values in the range 0–5, as follows:
5.6 Introducing Enumerations and Tuples

The argument 6 is the upper bound of the values produced and represents the number of unique values to produce (in this case six—0, 1, 2, 3, 4 and 5).

A six-sided die has the numbers 1–6 on its faces, not 0–5. So we shift the range of numbers produced by adding 1 to our previous result, as in

```
let face = 1 + arc4random_uniform(6)
```

**Rolling a Six-Sided Die 20 Times**

Figure 5.4 shows two sample outputs which confirm that the results of the preceding calculation are integers in the range 1–6, and that each run of the program can produce a different sequence of random numbers. Line 2 imports the Darwin module to allow the program to access function `arc4random_uniform`—the Swift Standard Library does not have its own random-number-generation capabilities. Line 5 executes 20 times in a loop to roll the die.

To run the program multiple times in a playground, simply press *Enter* on a blank line.

```
// fig05-04: Shifted and scaled random integers
import Darwin  // allow program to use C function arc4random_uniform

for i in 1...20 {
    print("(1 + arc4random_uniform(6)) ")
}
```

3 3 3 1 1 2 1 2 4 2 2 3 6 2 5 3 4 6 6 1

6 2 5 1 3 5 2 1 6 5 4 1 6 1 3 3 1 4 3 4

Fig. 5.4 | Shifted and scaled random integers.

### 5.6 Introducing Enumerations and Tuples

One popular game of chance is the dice game known as “craps.” In this section, we implement a simple version of the game and introduce Swift’s `enum` and tuple features.

The rules of the game are straightforward:

*You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called “craps”), you lose (i.e., “the house” wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your “point.” To win, you must continue rolling the dice until you “make your point” (i.e., roll that same point value). You lose by rolling a 7 before making your point.*

The app in Fig. 5.5 simulates the game of craps. Lines 31–74 of the program play the game. The `rollDice` function (lines 19–23) is called to roll the two dice and compute their sum, and the `displayRoll` function (lines 26–28) is called to display the results of a roll.

The four sample outputs show winning on the first roll, losing on the first roll, winning on a subsequent roll and losing on a subsequent roll, respectively.
// fig05-05: Simulating the dice game craps
import Darwin

// enum representing game status constants (no raw type)
enum Status {
    case Continue, Won, Lost
}

// enum with Int constants representing common dice totals
enum DiceNames: Int {
    case SnakeEyes = 2
    case Trey = 3
    case Seven = 7
    case YoLeven = 11
    case BoxCars = 12
}

// function that rolls two dice and returns them and their sum as a tuple
func rollDice() -> (die1: Int, die2: Int, sum: Int) {
    let die1 = Int(1 + arc4random_uniform(6)) // first die roll
    let die2 = Int(1 + arc4random_uniform(6)) // second die roll
    return (die1, die2, die1 + die2)
}

// function to display a roll of the dice
func displayRoll() {
    println("Player rolled \(\(roll.0\)) + \(\(roll.1\)) = \(\(roll.2\))")
}

// play one game of craps
var myPoint = 0 // point if no win or loss on first roll
var gameStatus = Status.Continue // can contain Continue, Won or Lost

var roll = rollDice() // first roll of the dice
displayRoll(roll) // display the two dice and the sum

// determine game status and point based on first roll
switch roll.sum {
    // win on first roll
    case DiceNames.Seven.rawValue, DiceNames.YoLeven.rawValue:
        gameStatus = Status.Won
    // lose on first roll
    case DiceNames.SnakeEyes.rawValue, DiceNames.Trey.rawValue,
        DiceNames.BoxCars.rawValue:
        gameStatus = Status.Lost
    // did not win or lose, so remember point
    default:
        gameStatus = Status.Continue // game is not over
        myPoint = roll.sum // remember the point
        println("Point is \(\(myPoint)\))")
}

Fig. 5.5 | Simulating the dice game craps. (Part 1 of 2.)
// while game is not complete
while gameStatus == Status.Continue {
    roll = rollDice() // first roll of the dice
    displayRoll(roll) // display the two dice and the sum

    // determine game status
    if roll.sum == myPoint { // won by making point
        gameStatus = Status.Won
    } else {
        if (roll.sum == DiceNames.Seven.rawValue) { // lost by rolling 7
            gameStatus = Status.Lost
        }
    }

    // display won or lost message
    if gameStatus == Status.Won {
        println("Player wins")
    } else {
        println("Player loses")
    }
}

Player rolled 2 + 5 = 7
Player wins

Player rolled 2 + 1 = 3
Player loses

Player rolled 2 + 4 = 6
Point is 6
Player rolled 3 + 1 = 4
Player rolled 5 + 5 = 10
Player rolled 6 + 1 = 7
Player loses

Player rolled 4 + 6 = 10
Point is 10
Player rolled 1 + 3 = 4
Player rolled 1 + 3 = 4
Player rolled 2 + 3 = 5
Player rolled 4 + 4 = 8
Player rolled 6 + 6 = 12
Player rolled 4 + 4 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 6 = 8
Player rolled 6 + 6 = 12
Player rolled 6 + 6 = 10
Player wins

Fig. 5.5 Simulating the dice game craps. (Part 2 of 2.)
The Game’s Logic

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Lines 31–74 contain the logic for one complete game of craps. Variable myPoint (line 31) stores the “point” if the player does not win or lose on the first roll. Variable gameStatus (line 32) maintains the game status. Variable roll (created at line 34 and assigned a new value at line 56) stores the most recent roll of the dice. Variable myPoint is initialized to 0 so the program can compile. If you do not initialize myPoint, the compiler issues an error, because myPoint is not assigned a value in every case of the switch statement—thus, the app could try to use myPoint before it’s assigned a value. By contrast, gameStatus does not require initialization because it’s assigned a value in every branch of the switch statement—thus, it’s guaranteed to be initialized before it’s used.

Error-Prevention Tip 5.3

Initialize every variable when it’s defined.

The First Roll

Line 34 calls function rollDice, which picks two random values from 1 to 6 and returns both values and their sum. Line 35 calls function displayRoll to display the value of the first die, the value of the second die and the sum of the dice. We explain the details of rollDice’s return value and displayRoll’s argument in Sections 5.6.2 and 5.6.3, respectively. Next, the program enters the switch statement at lines 38–51, which uses the sum of the dice to determine whether the game has been won or lost, or whether it should continue with another roll.

Additional Rolls of the Dice

If we’re still trying to “make our point” (i.e., the game is continuing from a prior roll), the loop in lines 54–67 executes. Line 56 rolls the dice again. Lines 60–66 determine whether the game was won or lost on the most recent roll—if not, the game continues. When the game completes, lines 70—74 display a message indicating whether the player won or lost, and the app terminates.

5.6.1 Introducing Enumeration (enum) Types

In this section, we introduce basic enumeration features—more details are presented in Chapter 9, Structures, Enumerations and Nested Types.

Status Enumeration

The Status type (lines 5–7) is an enumeration that declares a set of constants represented by identifiers. An enumeration is introduced by the keyword enum and a type name (in this case, Status). As with a class, braces ({ and }) delimit the enum’s body. Inside the braces is a case containing a comma-separated list of enumeration constants. The enum constant names must be unique. Unlike enums in other C-based programming languages, a Swift enum’s constants do not have values by default—the constants themselves are the values. Sometimes it’s useful for each constant to have a so-called raw value, as in the DiceNames enum (lines 10–16) that we discuss momentarily.
Variables and constants of type Status can be assigned only constants defined in the Status enum. When the game is won, the app sets variable gameStatus to Status.Won (lines 41 and 61). When the game is lost, the app sets gameStatus to Status.Lost (lines 45 and 64). Otherwise, the app sets gameStatus to Status.Continue (line 48) to indicate that the dice must be rolled again. If a variable has an enum type, you can assign enum constants to the variable using the shorthand notation:

\[
\text{variableName} = .\text{EnumConstantName}
\]

**Good Programming Practice 5.1**

*enum constant names should begin with a capital letter and use camel-case naming.*

**DiceNames Enumeration**

The sums of the dice that would result in a win or loss on the first roll are declared in the DiceNames enumeration in lines 10–16. These are used in the cases of the switch statement (lines 38–51). The identifier names use casino parlance—such as snake eyes (2) and box cars (12)—for these sums. In DiceNames we explicitly assign a value to each constant’s name. When an enum’s constants require values (known as raw values), you must specify the enum’s **raw type**—that is, the type used to represent each constant’s value. Line 10 indicates that DiceNames’s raw type is Int, so each constant’s type is also Int. The raw type can be any of Swift’s numeric types, type String or type Character.

Constants that are assigned explicit values are typically defined in a separate cases for readability (as in lines 11–15), but this is not required. We could have written the DiceNames enumeration as:

```swift
enum DiceNames: Int {
    case SnakeEyes = 2, Trey = 3, Seven = 7, YoLeven = 11,
    BoxCars = 12
}
```

If an enum type’s constants represent sequential integer values, they can be defined as a comma-separated list in one case, as in:

```swift
enum Months: Int {
    case January = 1, February, March, April, May, June, July,
    August, September, October, November, December
}
```

In Months, each subsequent constant after January has a value one higher than the value of the previous constant, so February is 2, March is 3, etc. So, we could have defined the DiceNames constants SnakeEyes and Trey in one case as:

```swift
case SnakeEyes = 2, Trey
```

The raw values of an enum’s constants must be unique. In an enum with one of the integer numeric types, if the first constant is unassigned, the compiler gives it the value 0.

**Good Programming Practice 5.2**

*Using enumeration constants (like Months.January, Months.February, etc.) rather than literal integer values (such as 1, 2, etc.) makes code easier to read and maintain.*
5.6.2 Tuples and Multiple Function Return Values

In the rules of the game, the player must roll two dice on the first roll and must do the same on all subsequent rolls. Function rollDice (lines 19–23) rolls the dice and computes their sum. Function rollDice is declared once, but it’s called from two places (lines 34 and 56). The function takes no arguments. Each time it’s called, rollDice returns three values (the two die values and the sum of the dice) as a tuple—an arbitrary collection of values that can be of the same or different types. In function rollDice’s return type

(die1: Int, die2: Int, sum: Int)

die1, die2 and sum are names that can be used to access the returned tuple’s elements.

**Good Programming Practice 5.3**
You’re not required to specify names for each element of a tuple, but doing so makes the code more readable.

The sum of the dice can be calculated using the values of the tuple elements die1 and die2. We chose to include sum in the tuple because there are multiple locations in the program where we use the sum of the dice. Rather than recalculating the sum each time, we calculate it once in rollDice, return it as part of the tuple, then simply use the tuple’s sum element as necessary in the rest of the code.

**Composing a Tuple**
To return a tuple containing multiple values from a function, you compose it by wrapping the values in parentheses, as in the return statement (line 22).

**Accessing a Tuple’s Elements**
When a tuple specifies names for its elements, you can access them by name using the dot (.) syntax. Line 34 assigns the tuple returned by rollDice to the variable roll, which is inferred to have the tuple type (Int, Int, Int). The switch statement’s control expression (line 38) uses roll.sum to get the sum of the dice from the returned tuple.

**Decomposing a Tuple**
You can also decompose a tuple into individual variables or constants. For example, the statement

```swift
let (die1, die2, sum) = rollDice()
```

assigns the three values in the tuple to the constants die1, die2 and sum, respectively. When decomposing a tuple, if you need only some of the values, you can ignore individual values with the underscore character (_), as in:

```swift
let (_, _, sum) = rollDice()
```

**Explicit Casts Are Required for Numeric Conversions**
Unlike many other programming languages, Swift does not allow implicit conversions between numeric types. To prevent a compilation error when you use a value of one numeric type where a different numeric type is expected, the compiler requires you to cast the value to the required type to force the conversion. This enables you to “take control” from the compiler. You essentially say, “I know this conversion might lose information, but for my purposes here, that’s fine.”
5.6 Introducing Enumerations and Tuples

Function rollDice returns a tuple containing Int values; however, the random numbers returned by function arc4random_uniform are of type UInt32. To convert these to type Int, you must use an Int cast as shown in line 20:

```swift
let die1 = Int(1 + arc4random_uniform(6)) // first die roll
```

The cast Int(1 + arc4random_uniform(6)) creates a temporary Int copy of the argument in parentheses.

Error-Prevention Tip 5.4

Each numeric type represents a different range of values. Disallowing implicit conversions—thus forcing you to use explicit casts for numeric conversions—prevents unintentional conversions between types. This is another Swift feature that eliminates errors.

Common Programming Error 5.2

Converting a numeric-type value to a value of another numeric type may change the value. For example, converting a Double value to an Int value may introduce truncation errors (loss of the fractional part) in the result.

5.6.3 Tuples as Function Arguments

After each call to rollDice, the program calls function displayRoll (lines 35 and 57) to display the two die values and the sum of the dice. The function (lines 26–28) receives one parameter (roll) which has the tuple type (Int, Int, Int). In this case, we did not specify names for the elements in the tuple, so that we could show accessing a tuple’s members using indices and dot syntax, as in line 27. The first tuple element has index 0, so roll.0 evaluates to the first die’s value, roll.1 evaluates to the second die’s value and roll.2 evaluates to their sum.

5.6.4 Accessing the Raw Value of an enum Constant

The switch statement at lines 38–51 performs its tasks based on the sum of the dice. Swift does not provide implicit conversions between enum constants and numeric types. However, each enum constant has a rawValue property that returns the constant’s raw value. Lines 40, 43 and 44 compare the Int sum of the dice to the raw Int values of several DiceNames constants to determine whether the game was won or lost on the first roll. We use the raw enum constant values in this case because there are several sums (4, 5, 6, 8, 9 and 10) that don’t correspond to the DiceName enum constants.

Converting a Value to an enum Constant

You can use an enum’s initializer to get the enum constant that corresponds to a raw value. For example, using the Months enum discussed in Section 5.6.1, the expression

```swift
Months(rawValue: 2)
```

returns the enum constant Months.February. In a program that receives a month as a value in the range 1–12, you could use the Months enum’s initializer to convert those values to the corresponding Months enum constants for use in a switch’s cases. Because the argument could be invalid, the actual value returned by the initializer is a Months?—an optional value of type Months. We discuss this in more depth in Section 9.3.3.
5.7 Scope of Declarations

You’ve seen declarations of Swift entities, such as classes, methods, properties, variables and parameters. Declarations introduce names that can be used to refer to such Swift entities. The scope of a declaration is the portion of the code that can refer to the declared entity by its unqualified name. Such an entity is said to be “in scope” for that portion of the app. This section introduces several important scope issues. The basic scope rules are:

1. The scope of a parameter is the body of the method in which the declaration appears.
2. The scope of a local variable or constant is from the point at which it’s defined to the closing right brace `}` of the block containing the definition.
3. The scope of a local variable that appears in the initialization section of a for statement’s header is the body of that for statement and the other expressions in the header.
4. The scope of a local variable that receives each value in a for…in statement is the body of that for…in statement.
5. The scope of a method or property of a class is the entire body of the class.
6. A type, function, variable or constant defined outside any other language element has global scope from its point of definition to the end of the file in which the type, function, variable or constant is defined. Types and functions also have module scope—by default, they can be used from other files in the same module or in other apps that import that module, unless they’re declared private.

Any block may contain variable declarations. If a local variable, constant or parameter in a method has the same name as a property of a class, the property is hidden until the block terminates. In Chapter 8, we discuss how to access hidden properties via the keyword `self`. The app in Fig. 5.6 demonstrates the scopes for a global variable, a property of a class and local variables in methods.

```swift
// fig05-06: Demonstrating scopes
var x = 5 // global variable x

class Scope {
    var x = 1 // property hides global variable x in class Scope
    // create and initialize local variable x during each call
    func useLocalVariable() {
        var x = 25 // initialized each time useLocalVariable is called
        println("local x on entering useLocalVariable is \(x)"
        ++x // modifies this method's local variable x
        println("local x before exiting useLocalVariable is \(x)"
    }
}
```

**Fig. 5.6** Demonstrating scopes. (Part 1 of 2.)
5.7 Scope of Declarations

Line 2 defines and initializes the global variable \( x \) to 5. This variable is hidden in any block or method that declares local variable named \( x \) and in any class that defines a property named \( x \). Class Scope (lines 4–23) defines a property \( x \) with the value 1 (line 5). We defined the class after the global variable \( x \) at line 2 to show that the class’s property \( x \) hides the global variable.

Line 25 defines an object of class Scope named \( \text{scope} \). Line 27 outputs the value of global variable \( x \) (whose value is 5). Next, lines 29–32 call Scope methods \( \text{useLocalVariable} \) (lines 8–15) and \( \text{useProperty} \) (lines 18–22) that each take no arguments and do not return results. We call each method twice. Method \( \text{useLocalVariable} \) declares local variable \( x \) (line 10). When \( \text{useLocalVariable} \) is first called (line 29), it creates local variable \( x \) and initializes it to 25 (line 10), outputs the value of \( x \) (line 12), increments \( x \) (line 13) and outputs the value of \( x \) again (line 14). When \( \text{useLocalVariable} \) is called a second time (line 31), it re-creates local variable \( x \) and reinitializes it to 25, so the output of each \( \text{useLocalVariable} \) call is identical.

Method \( \text{useProperty} \) does not declare any local variables. Therefore, when it refers to \( x \), class Scope’s property \( x \) (line 5) is used. When method \( \text{useProperty} \) is first called
(line 30), it outputs the value (1) or property \(x\) (line 19), multiplies the property \(x\) by 10 (line 20) and outputs the value (10) of property \(x\) again (line 21) before returning. The next time method `useProperty` is called (line 32), the property has its modified value, 10, so the method outputs 10, then 100. The app outputs the value of global variable \(x\) again (line 34) to show that none of the method calls modified the global variable \(x\), because the methods all referred to variables or properties named \(x\) in other scopes.

### 5.8 Function and Method Overloading

You can define functions of the same name, as long as they have different sets of parameters (determined by the number, types and order of the parameters). This is called **function overloading** and can be used with a type’s methods and initializers as well. When an overloaded function is called, the Swift compiler selects the appropriate function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions with the same name that perform the same or similar tasks, but on different types or different numbers of arguments. For example, Swift function `max` is overloaded with two versions—one that returns the maximum of two values and one that returns the maximum of three or more values. Our next example demonstrates declaring and invoking overloaded functions. You’ll see examples of overloaded initializers in Chapter 8, Classes: A Deeper Look and Extensions.

#### Declaring Overloaded Functions

In Fig. 5.7, we define overloaded versions of function `square`—one that calculates the square of an `Int` (and returns an `Int`) and one that calculates the square of a `Double` (and returns a `Double`). Although these functions have the same name and similar parameter lists and bodies, you can think of them simply as different methods. It may help to think of the functions names as “square of Int” and “square of Double,” respectively.

```swift
// fig05-07: Overloaded function definitions
func square(value: Int) -> Int
{
    print("Called square with Int argument: \(value)\)
    return value * value
}

func square(value: Double) -> Double
{
    print("Called square with Double argument: \(value)\)
    return value * value
}

// test overloaded square functions
print("Square of Int 7 is \(square(7))\n")
print("Square of Double 7.5 is \(square(7.5))\n")
```

**Fig. 5.7**  Overloaded function definitions. (Part 1 of 2.)
Line 18 invokes method `square` with the argument 7. Literal integer values are treated as type `Int`, so the method call in line 18 invokes the version of `square` at lines 4–8 that specifies an `Int` parameter. Similarly, line 19 invokes `square` with the argument 7.5. Literal floating-point values are treated as type `Double`, so the method call in line 19 invokes the version of `square` at lines 11–15 that specifies a `Double` parameter. Each function first outputs a line of text to prove that the proper function was called in each case.

The overloaded functions in Fig. 5.7 perform the same calculation, but with two different types. Swift’s generics feature provides a mechanism for writing a single “generic function” that can perform the same tasks as an entire set of overloaded functions. We discuss generic functions in Chapter 11, Generics.

### Distinguishing Between Overloaded Functions

The compiler distinguishes overloaded functions by their **signature** —a combination of the function’s name and the number, types and order of its parameters. The signature also includes the way those parameters are passed, which can be modified by the `inout` keyword (discussed in Section 5.11). If the compiler looked only at method names during compilation, the code in Fig. 5.7 would be ambiguous—the compiler would not know how to distinguish between the `square` functions. Internally, the compiler uses signatures to determine whether functions are unique, whether a class’s methods are unique and whether a class’s initializers are unique.

For example, in Fig. 5.7, the compiler will use the function signatures to distinguish between the “square of `Int`” function (the `square` function that specifies an `Int` parameter) and the “square of `Double`” function (the `square` function that specifies a `Double` parameter). If a function `someFunction`’s declaration begins as

```swift
func someFunction(a: Int, b: Double)
```

then that function will have a different signature than the function declared as

```swift
func someFunction(a: Double, b: Int)
```

The order of the parameter types is important—the compiler considers the preceding two functions to be distinct.

### Return Types of Overloaded Functions

In discussing the logical names of functions used by the compiler, we did not mention the return types of the functions. This is because function calls cannot be distinguished by return type. Overloaded functions can have the **same or different** return types if the functions have **different** parameter lists. Also, overloaded functions need not have the same number of parameters.
5.9 External Parameter Names

By default, the parameter names you specify in a function definition are local to that function—they’re used only in the body of that function to access the function’s argument values. You can also define external parameter names that the caller is required to use when a function is called—as is the case for all the arguments to an initializer and any arguments after the first argument in a method call. This can help make the meaning of each argument clear to the programmer calling the function.

For each parameter, you can specify both an external name and a local name by placing the external name before the local name as in:

```
externalName localName: type
```

or you can specify that the local parameter name should also be used as the external parameter name by placing a # before the local parameter name, as we demonstrate in Fig. 5.8 (line 4). The function `power` (lines 4–12) calculates the value of its `base` argument raised to its `exponent` argument. The two calls to `power` (lines 15 and 16) each specify the parameter name before each argument. Once you expose an external parameter name, you must label the corresponding argument in a function call with a parameter name and a colon (:) otherwise, a compilation error occurs.

```
// fig05-08: External parameter names
func power(#base: Int, #exponent: Int) -> Int {
    var result = 1;
    for i in 1...exponent {
        result *= base
    }
    return result
}

// call power with and without default parameter values
println("power(base: 10, exponent: 2) = \(power(base: 10, exponent: 2))")
println("power(base: 2, exponent: 10) = \(power(base:2, exponent: 10))")
```

Fig. 5.8 | External parameter names.

*Changing the Default External Parameter Names for an Initializer or Method*

By default, the names of an initializer’s parameters and the names of a method’s parameters for every parameter after the first are used as their external names. You can customize a method’s or initializer’s external parameter names by specifying your own, using the same syntax we discussed for functions earlier in this section.
Why an External Name Is Not Required for a Method’s First Argument

In Objective-C, method calls read like sentences. The method name refers to the first parameter, and each subsequent parameter has a name that’s specified as part of the method call. In addition, method and parameter names often include prepositions to help make function calls read like sentences.

Apple wants Swift programmers to use similar naming conventions in their methods. Because the method name should refer to the first parameter, Swift provides only a local parameter name for the first method parameter, then provides local and external parameter names for all subsequent parameters. Using this naming convention, we could reimplement the `power` function as

```swift
func raiseBase(base: Int, toExponent: Int) -> Int
```

In this case, we’d call the function as:

```swift
raiseBase(10, toExponent: 2)
```

which reads like the sentence, “Raise the base 10 to the exponent 2.”

Requiring an External Parameter Name for a Method’s First Argument

You can require a method’s caller to provide an external parameter name for the method’s first argument. To do so, simply precede the parameter name with `#` to use the local parameter name as the external parameter name or specify an external parameter name.

Passing Method Arguments Without Parameter Names

You can allow a method to be called without labeling its arguments by using an underscore (_ as each parameter’s external name.

5.10 Default Parameter Values

Methods can have default parameters that allow the caller to vary the number of arguments to pass. A default parameter specifies a default value that’s assigned to the parameter if the corresponding argument is omitted.

You can create functions with one or more default parameters. All default parameters must be placed to the right of the function’s nonoptional parameters—that is, at the end of the parameter list. Each default parameter must specify a default value by using an equal (=) sign followed by the value.

When a parameter has a default value, the caller can optionally pass that particular argument. For example, the function

```swift
func power(base: Int, exponent: Int = 2) -> Int
```

specifies a default second parameter. Any call to `power` must pass at least an argument for the parameter base, or a compilation error occurs. Optionally, a second argument (for the exponent parameter) can be passed to `power`. Consider the following calls to `power`:

```swift
power() // compilation error--first argument is required
power(10) // calls power with 2 as the second argument
power(10, exponent: 3) // explicitly specifying both arguments
```

The first call generates a compilation error because this function requires a minimum of one argument. The second call is valid because the one required argument (10) is being
Chapter 5 Functions and Methods: A Deeper Look; Enums and Tuples

passed explicitly—the optional exponent is not specified in the method call, so 2 is passed by default. The last call is also valid—10 is passed as the required argument and 3 is passed as the optional argument. A function’s default parameter names are automatically external parameter names—when you provide an argument for a default parameter, you must specify the default parameter’s name with that argument in the function call.

Figure 5.9 demonstrates a default parameter. The program reimplements the power function of Fig. 5.8 without external parameter names and with a default value for its second parameter. Lines 15–16 call function power. Line 15 calls it without the second argument. In this case, the compiler provides the second argument, 2, using the default value specified in line 4, which is not visible to you in the call. Notice that the call to power at line 16 requires the parameter name for the second argument.

```
// fig05-09: Default parameter values

// use iteration to calculate power of base raised to the exponent
func power(base: Int, exponent: Int = 2) -> Int {
    var result = 1;
    for i in 1...exponent {
        result *= base
    }
    return result
}

// call power with and without default parameter values
println("power(10) = ")
println("power(2, exponent: 10) = ")

power(10) = 100
power(2, exponent: 10) = 1024
```

Fig. 5.9 | Default parameter values.

5.11 Passing Arguments by Value or by Reference

Swift allows you to pass arguments to functions by value or by reference. When an argument is passed by value (the default for value types in Swift), a copy of its value is made and passed to the called function. Changes to the copy do not affect the original variable’s value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument that’s been passed in the programs in this chapter so far has been passed by value. When an argument is passed by reference, the caller gives the function the ability to access and modify the caller’s original variable.

To pass an object of a class type by reference into a function, simply provide as an argument in the function call the variable that refers to the object. Then, in the function body, reference the object using the parameter name. The parameter refers to the original object in memory, so the called function can access the original object directly.
We’ve considered value types and reference types. A major difference between them is that value-type variables store values, so specifying a value-type variable in a function call passes a copy of that value to the method. Reference-type variables store references to objects, so specifying a reference-type variable as an argument passes the function a copy of the reference that refers to the object. Even though the reference itself is passed by value, the function can still use the reference it receives to interact with—and possibly modify—the original object. Similarly, when returning information from a function via a return statement, the function returns a copy of the value stored in a value-type variable or a copy of the reference stored in a reference-type variable. When a reference is returned, the caller can use that reference to interact with the returned reference-type object.

**inout Parameters**

What if you would like to pass a variable by reference so the called function can modify the variable’s value in the caller? To do this, Swift provides keyword `inout`. Applying `inout` to a parameter declaration allows you to pass a variable to a function by reference—the called function will be able to modify the original variable in the caller. It’s a compilation error to pass a constant to an `inout` parameter. A function can use multiple `inout` parameters as another way to “return” multiple values to a caller. You can also pass a reference-type variable by reference, which allows you to modify it so that it refers to a new object.

**Demonstrating an inout Parameter**

The app in Fig. 5.10 uses the `inout` keyword to allow a function to modify its `Int` argument. Function `square` (lines 4–6) multiplies its parameter value by itself and assigns the result to `value`. The `Int` parameter is preceded with `inout`, which indicates that the argument passed to this method must be an `Int` and that it will be passed by reference. Because the argument is passed by reference, the assignment at line 5 modifies the original argument’s value in the caller.

```
1 // fig05-10: Pass-by-reference with inout parameters
2
3 // square function that modifies its argument in the caller
4 func square(inout value: Int) {
5     value *= value // squares value of caller’s variable
6 }
7
8 // test inout parameter
9 var x = 5
10 println("Original value of x is \(x)")
11 square(&x)
12 println("Value of x after calling square(&x) is \(x)")
```

Fig. 5.10 | Pass-by-reference with `inout` parameters.
**Passing an Argument by Reference**

Line 9 initializes variable `x` to 5. Line 10 displays `x`’s original value. When you pass a variable to a method with a reference parameter, you must precede the argument with an `&` (line 11)—similar to a pointer in languages like Objective-C, C and C++. After line 11 squares `x`’s value, line 12 displays the new value. Notice that `x` is now 25.

**Software Engineering Observation 5.2**

By default, value types are passed by value. Objects of reference types are not passed to methods; rather, references to objects are passed to methods. The references themselves are passed by value. When a method receives a reference to an object, the method can manipulate the object directly, but the reference value cannot be changed to refer to a new object.

### 5.12 Recursion

Swift supports recursion. A recursive function calls itself, either directly or indirectly through another function.

**Recursive Factorial Calculations**

Figure 5.11 uses recursion to calculate and display the factorials of the integers from 0 to 10. The recursive function `factorial` (lines 4–11) first tests to determine whether a terminating condition (line 6) is true. If `number` is less than or equal to 1 (the base case), `factorial` returns 1, no further recursion is necessary and the function returns. If `number` is greater than 1, line 9 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`, which is a slightly simpler problem than the original calculation, `factorial(number)`.

```swift
1 // fig05-12: Recursive factorial function
2
3 // recursive factorial function
4 func factorial(number: Int64) -> Int64 {
5     // base case
6     if number <= 1 {
7         return 1
8     } else { // recursion step
9         return number * factorial(number - 1)
10     }
11 }
12
13 // calculate the factorials of 0 through 10
14 for counter in 0...10 {
15     println("\(counter)! = \(factorial(Int64(counter)))")
16 }
```

0! = 1
1! = 1
2! = 2

**Fig. 5.11** | Recursive factorial function. (Part 1 of 2.)
5.13 Nested Functions

You can nest function definitions in other function definitions. This can be useful for organizing complex functions. Rather than defining at global scope a utility (helper) function that’s called by only one other function, you can nest the utility function’s definition in the scope of the function that uses it. This hides it from the rest of your code. For example, an array-sorting function could define a nested swap function for swapping elements into sorted order.

If necessary, an enclosing function can return a nested function so that it can be called from other scopes—for example, you could define a function that returns a nested function based on a value passed to the enclosing function (as we do in this section’s example). A nested function also has access to the local variables and constants in its enclosing function’s scope, including the enclosing function’s parameters.

Figure 5.12 contains a mechanical nested-functions example. Function sortOrder (lines 4–16), based on the Bool parameter increasingOrder’s value, returns either the nested function ascending (defined at lines 6–8) or the nested function descending (defined at lines 11–13). To make the purpose of sortOrder’s argument clear, we specified that its parameter name (increasingOrder) should also be its external parameter name—thus, each call to sortOrder (lines 19 and 28) labels its argument with increasingOrder.

Fig. 5.11 | Recursive factorial function. (Part 2 of 2.)

Function factorial receives a parameter of type Int64 and returns a result of type Int64. As you can see in Fig. 5.11, factorial values become large quickly. We chose Int64 (which can represent relatively large integers) so that the app could calculate factorials up to 20!. Unfortunately, the function produces large values so quickly that 21! exceeds the maximum value that can be stored in an Int64 variable, causing an overflow. Due to the restrictions on the integral types, variables of type Float or Double might ultimately be needed to calculate factorials of larger numbers.

A strength of object-oriented programming languages like Swift is that they can be extended with new types to meet your applications’ needs. For example, you could create a type (e.g., HugeInt) that supports arbitrarily large integers for use in large-number factorial calculations.

3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

Common Programming Error 5.3

Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause infinite recursion, eventually exhausting memory. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

Fig. 5.12 | Mechanical nested-functions example.
Chapter 5  Functions and Methods: A Deeper Look; enums and Tuples

Every Function Has a Type and Can Be Treated as Data

Each function you define has a type that’s determined by the types of its parameters and by its return type. The return type of function `sortOrder` is specified as

```plaintext
(Int, Int) -> Bool
```

A function type consists of parentheses containing the parameter types, followed by `->` and the return type. The preceding type indicates that the value returned by `sortOrder` is a function type for a function that receives two `Int` parameters and returns a `Bool`. Functions `ascending` and `descending` meet these requirements.

Because every function has a type, you can assign functions to variables, pass them to functions and methods, and return them from functions and methods. We’ll discuss functions as data in more detail in Section 6.7.
Assigning a Function to a Variable and Using the Variable to Call the Function

Line 19 calls function `sortOrder` with the argument `true` to indicate that `sortOrder` should return the function that determines whether two `Int` s are in ascending order. The returned function is assigned to the variable `order`, which is inferred to have the type `(Int, Int) -> Bool`

Once you’ve assigned a function to a variable, you can use the variable to call the function, as shown in line 21. Line 28 calls `sortOrder` with the argument `false` to get the function that determines whether two `Int`s are in descending order, then line 30 calls that function.

5.14 Wrap-Up

In this chapter, we continued our discussion of functions and methods. We discussed that Swift automatically creates modules for packaging reusable software components. We introduced Darwin—Apple’s UNIX-based core of OS X and iOS—and imported the Darwin module so we could use the random-number generator.

We used `enum` types to create sets of named constants with and without values for the constants. You returned multiple values from a function via a tuple, passed a tuple to a function and accessed a tuple’s elements via both names and indices.

We discussed the scope of identifiers. We used overloading to define multiple functions with the same name that performed similar tasks but with different types and/or different numbers of parameters.

We discussed differences in how functions and methods are called, and we presented the concepts of local parameter names vs. external parameter names. You saw that, when external parameter names are provided in a function definition, they must be used in the function call to label the corresponding arguments. You used `#` to expose a local parameter name as the external parameter name. We also showed how to disable this feature in methods—by placing an underscore (_) before the parameter’s name—so that parameter names are not required in a method call.

You specified a default parameter value and saw that the compiler supplied that value in a function call when you did not explicitly provide an argument for that parameter.

We discussed how value-type and reference-type arguments are passed to methods and demonstrated how to pass arguments by reference by declaring the parameter as `inout` and providing an ampersand (`&`) before the corresponding argument in a function call. We demonstrated that Swift supports recursive functions and nested functions.

In this chapter, we continued our discussion of functions and methods. We discussed that Swift automatically creates modules for packaging reusable software components. We introduced Darwin—Apple’s UNIX-based core of OS X and iOS—and imported the Darwin module so we could use the random-number generator.

We used `enum` types to create sets of named constants with and without values for the constants. You returned multiple values from a function via a tuple, passed a tuple to a function and accessed a tuple’s elements via both names and indices.

We discussed the scope of identifiers. We used overloading to define multiple functions with the same name that performed similar tasks but with different types and/or different numbers of parameters.

We discussed differences in how functions and methods are called, and we presented the concepts of local parameter names vs. external parameter names. You saw that, when external parameter names are provided in a function definition, they must be used in the function call to label the corresponding arguments. You used `#` to expose a local parameter name as the external parameter name. We also showed how to disable this feature in methods—by placing an underscore (_) before the parameter’s name—so that parameter names are not required in a method call.

You specified a default parameter value and saw that the compiler supplied that value in a function call when you did not explicitly provide an argument for that parameter.

We discussed how value-type and reference-type arguments are passed to methods and demonstrated how to pass arguments by reference by declaring the parameter as `inout` and providing an ampersand (`&`) before the corresponding argument in a function call. We demonstrated that Swift supports recursive functions and nested functions.

In Chapter 6, you’ll use `Array`s to maintain lists and tables of data. You’ll also create functions with variable-length argument lists. We’ll continue our discussion of functions in Chapter 6 which also presents closures—anonymous functions that are typically defined in the scope of a function or method and commonly passed as arguments to other functions or methods. As you’ll see, a function is a closure with a name. Swift’s `Array` type provides methods `filter`, `map` and `reduce` that receive closures as arguments—which enable you to express complex operations in a more concise and elegant manner than with full function definitions. We’ll present additional method and initializer concepts in Chapters 8–10.
Symbols

_ , underscore for ignoring a tuple value during decomposition 82
-- , predecrement/postdecrement operators 53
- , subtraction operator 28, 29
: notation for inheritance 221
! , logical NOT operator 68
   truth table 68
! = , not equals operator 30
! == , not identical to operator 30, 45
? : , ternary conditional operator 52
. , dot syntax 36, 41
.. , closed range operator 56, 59, 107
.. < , half-open range operator 56, 59, 103, 107, 108
{ , left brace 35
}, right brace 35
* , multiplication operator 28, 29
*= , multiplication assignment operator 53
/ , division operator 28, 29
/* */ , multiline comment 22
// , end-of-line comment 22
/= , division assignment operator 53
\ , backslash special character 24
\' , single-quote special character 24
\" , double-quote special character 24
\0 , null special character 24
\n , line feed special character 24
\r , carriage-return special character 24
\t , tab special character 24
&= , overflow subtraction operator 29
&* , overflow multiplication operator 29
&/ , overflow division operator 29
& & , logical AND operator 66, 67
   truth table 67
&% , overflow remainder operator 29
&+ , overflow addition operator 29
% = , remainder assignment operator 53
+ for concatenating Arrays 101
- (minus sign) formatting flag 62
+, addition operator 28, 29
++, preincrement/postincrement operators 53
+= operator for Arrays 104, 106, 107
+= , addition assignment operator 52
< , less than operator 30
<= , less than or equal operator 30
-= , subtraction assignment operator 53
== , is equal to operator 30
=== identical to operator 30, 45
> , greater than operator 30
>= , greater than or equal to operator 30
| | , logical OR operator 66, 67
   truth table 67

Numerics

0 , format flag 163
A
abbreviating assignment expressions 52
abs C function 74
abstract class 228, 230, 231
abstract method 230
abstract property 230
access modifier 35
   internal 35, 37, 46, 226
   private 35, 46, 226
   public 35, 37, 46, 226
access modifiers public 160
Accessibility 291, 311
accessibility strings 289
Accessibility Inspector 312
UIApplication 312
action 50, 324
create 340
action (event handler) 340
actual types in generics 254
adaptive design 11
Add Missing Constraints 334
addition 28, 29
addition compound assignment operator, += 52
adopt a protocol 238, 241
advance global function 270
Alignment attribute of a Label 332
anonymous function 4, 108
Anonymous function type 279, 345
Apple Inc. 3
App Store xxxii
app icons 298
app name 293
Apple Macintosh 3
Apple TV 3
Apple World Wide Developer Conference (WWDC) xix, 3
arc4random UNIX function 76
arc4random_uniform UNIX function 76, 77
arithmetic compound assignment operators 52
arithmetic operators 28
-, subtraction 28, 29
*, multiplication 28, 29
/, division 28, 29
&, overflow subtraction 29
&*, overflow multiplication 29
&/, overflow division 29
&%, overflow remainder 29
&+, overflow addition 29
&%, remainder 28, 29
+, addition 28, 29
Array
  count property 98
array bounds checking 5
Array literal 100, 101
  for multidimensional Array 125
  nested 125
Array of one-dimensional Arrays 125
Array Swift Standard
Library type 72, 97, 153, 278, 325
  + operator for concatenating Arrays 101
  += operator 104, 106, 107
  append method 104, 106, 106
  delete a subset of the elements 108
description property 99, 100
element type 99
filter method 113
initializer with no parameters 100
initializer with two parameters 100
insert method 104, 107
isEmpty property 105

Array Swift Standard
Library type (cont.)
last property 261
map method 113, 115
pass by reference 123
reduce method 114, 115, 128, 142
removeAt1 method 104, 107
removeAtIndex method 104, 107
removeLast method 104, 107
replace a subset 108
reverse method 112
select a subset 108
sort method 109
sorted method 109, 111
two-dimensional 124
zero-based counting 98
as cast operator 278, 279
as operator 238
as? operator 237
aspect ratio 303
assert function 282
asset catalog 291, 298
associated type for a protocol 263
associated value for an enum constant 210
associative array 132
associativity
default value none 284
left 284
none 284
of a new operator 284
right 284
associativity context-sensitive keyword 285
associativity of operators 29, 32
left to right 29
right to left 29
Attributes inspector 303
auto layout 11, 290, 301, 306, 334
auto layout constraints adding 334
Equal Widths 335
missing 334
automatic reference counting (ARC) 159, 184, 193, 343
Autoshrink 305, 332
B
Background attribute of a GUI component 328
backslash (\) 24
base class 248, 342
topmost superclass in a class hierarchy 220, 248
base internationalization 314
base language (internationalization) 314, 315
becomeFirstResponder method of a GUI component 325, 344
binary operator 28
body of a class definition 35
of an if statement 29
Bool expression 52
Bool primitive type 52
Bool type 26, 27
Boolean and numeric Swift Standard Library types 72
braces ({} and )
not required 56
branch statement 29
break statement 57, 64
Breakpoint navigator 16, 294
bridging between Objective-C and Swift 153, 279, 325, 348
Apple’s Using Swift with Cocoa and Objective-C guide 153, 279
downcast 279
bullseye symbol for an outlet or action 339
bundle identifier (bundle ID) 15, 293
C
C Standard Library 73
camel case naming 27
card face 204
card games 116
carriage return 24
case in an enum 80
case keyword 56
case sensitive 27
cast 82
categories in Objective-C 159, 174
ceil C function 74
chain of initializer calls 248
class 72
camel case naming 35
class-only protocol 246
class variable 187
class method in other object-oriented languages 187
closure 4, 142
anonymous function 108
closure expression 109
empty parameter list 111
fully typed 111
inferred types 111
inferred types and implicit return 112
operator function 112
shorthand argument names 112
Cocoa 7
Cocoa Touch 7, 290, 301, 322, 323
code-completion suggestions 22
code reuse 215
Code Snippet library 301
collision in a hashtable 154
color opacity 328
column 124
columns of a two-dimensional Array 124
comment
multiline, /*...*/ 22
single-line, // 22
company identifier 293, 326
Comparable protocol 247, 259
String 268
Comparable Swift Standard Library protocol 72
comparative operators 30, 247, 259
compile-time type safety 254
componentsSeparatedByString method of class NSString 152
compose a tuple 82
composition 181
compound assignment operators 52
compound interest 60
computed property 37, 117, 160, 163, 174, 230
get accessor 117, 178, 179
read-only 163, 178
set accessor 117, 178, 179
shorthand read-only 163
struct 198
computed type property 187, 189
struct 198
concatenate Strings 22, 52
concrete class in other object-oriented programming languages 230
condition 29
conditional statement 49, 49
if...else 50
Conditional Statements if 29
conform to a protocol 238
connect a GUI control to a corresponding 324
Connection type 339
Connections inspector 303
constant property 36, 324
constant value-type object 196
constructor in other object-oriented programming languages 248
count property of type Array 98, 104
count property of type Dictionary 141
currency format 338
currency formatting 322
currencyStyle constant of the NSNumberFormatterStyle enumeration 186, 219
dangling-else problem 52
Darwin module 73, 76
dateFormat property of class NSDateFormatter 184
dateStyle property of class NSDateFormatter 182
dealing a deck of cards 116
deallocate an object 185
debug area (Xcode) 16, 17, 293, 295
debug navigator 16, 294
debugger 12
decimal separator 219
decimalNumberByAdding method of class NSDecimalNumber 187, 276, 349
decimalNumberByDividingBy method of class NSDecimalNumber 187, 220, 275, 349
decimalNumberByMultiplyingBy method of class NSDecimalNumber 187, 220, 275, 349
declaration import 41, 342
declaration modifier 285
declare a protocol 238
declare a GUI control to a corresponding 324
Connection type 339
Connections inspector 303
constant property 36, 324
constant value-type object 196
constructor in other object-oriented programming languages 248
count property of type Array 98, 104
count property of type Dictionary 141
currency format 338
currency formatting 322
CurrencyStyle constant from the NSNumberFormatterStyle enumeration 186, 219
currencyStyle constant of the NSNumberFormatter-Style enumeration 348
custom name for the constant in a property observer 162
deinit keyword 185
inheritance 251
deinitializer 185, 189, 251
not allowed in value types 198
delegate protocol 246
delete a subset of an Array 108
dependent condition 68
Deployment Info 297
derived class 342
description property
Array 99, 100
Dictionary 134, 136
designated initializer 167, 172, 222, 223, 248, 249, 250
protocol conformance 241
designated initializer in a base class 248
Device Orientation 297
Devices project setting 293
Devices window (Xcode 6) 22
dice game 77
Dictionary Swift Standard Library type (cont.)
modifying the value associated with a key 145, 147
mutable 134
removeAll method 151
removeValueForKey method 151
removing key–value pairs 145, 147
shorthand notation 133
subscripting operations 145
type inference 136
updateValue method 150, 151
value 132
values property 145
Dictionary Swift Standard Library type 72
didSet property observer 162
direct superclass 218
disabilities 291, 311
display a line of text 22
division 28, 29
division compound assignment operator, /= 53
do...while loop statement 50, 58
document outline window 307
dot (.) syntax 36, 41
double-precision floating-point number 34
double quotes, " 22, 24
double quotes, ” 22, 24
download type 26, 27, 34, 34
downcast 237, 238, 279
duplicate existing GUI components 324
dynamic binding 237
dynamically typed 345

E
Editing Changed event for a Text Field 340, 345
Editor area (Xcode) 16, 293, 294
element of chance 76
element type of an Array 99
ellipses (…) in a method parameter list 128
else keyword 51
emojis in Swift code 26
empty Dictionary literal [:] 135, 141
empty Dictionary object 136
empty statement (in C-based languages) 31
empty String 347
encapsulation 46
endIndex property of a String 270
enum 80, 80
case 80
constants in switch statement cases 205
failable initializer 209
initializer 209
keyword 195
raw type 81
unique raw values for constants 202
tenum constant 80
associated values 210
raw value 81
rawValue 204
rawValue property 83
tenum constants in cases 205
tenum keyword 202
tenum type 80
enumerate function 103
enumeration 195
Equal Widths constraint 335
equality operators 140
equality operators 140
equality operators 140
equality operators 140
equality operators 140
equality operators 140
Dicitonary Swift Standard Library type 132, 136, 153, 278, 325
Arrays as values 142
count property 141
description property 134, 136
empty Dictionary 136
empty literal, [:] 135, 141
equality operators 140
generic type 133
immutable 134
inserting new key–value pairs 145, 147
isEmpty property 141
key 132
keys property 143
literal 135
E
Index

Equatable Swift Standard Library protocol 73
Error-Prevention Tips overview xxi
escape character 24
event handler 340
event-handling method 324
Events
  Editing Changed event for a Text Field 340, 345
  Value Changed event for a Slider 340, 345
exception handling 163
exclamation point (!) to unwrap an optional 145
exhausting memory 93
exit point of a control statement 50
  exiting a for statement 65
exp C function 74
explicit cast 82
exponentiation operator 285
expression 28
extended graphene cluster 268
extensible 227
extension 199
  computed property 174
  convenience initializer 174
  keyword 159, 174
  method 174
extension for conforming to a protocol 238, 239, 242
external iteration 113
external parameter name
  # to use local parameter name 88
  for a function parameter 88
F
  face of a card 204
  factorial 92
failable 172
  failable initializer 159, 170, 173, 220, 223, 230, 231
    convenience initializer 172
    designated initializer 172
  init? 172
  failable initializer for an enum 209
  failable initializer in a protocol 241
false keyword 29, 52
fatalError function 231
  field width 62
  file in the Project navigator 18, 296
File inspector 16, 295
File Template library 301
  filter elements of an Array 114
  filter method of Array 113
final
  class 248
  method 248
  property 248
  subscript 248
Find navigator 16, 294
Finder window 321
  first responder 325
Fisher-Yates shuffle algorithm 119
fixed text in a format string 61
Fix-it 12
Float type 26, 27, 34, 34
  floating-point literal
    Double by default 34
  floating-point number 34
    double precision 34
    Double type 34
    Float type 34
    single precision 34
  floor C function 74
  for loop statement 50, 63
  nested 126, 128
  force unwrap an optional with ! 145, 148
format specifiers 61
  @ 61, 62
formatted output
  – (minus sign) formatting flag 62
  0 flag 163
  field width 62
  left justify 62
  minus sign (–) formatting flag 62
  formatted text
  right align 62
Foundation framework 7, 41, 322, 323, 348
  NSNumberFormatter class 41
frameworks 40
  Foundation 41, 323
  UIKit 301, 323, 324, 342
free (global) function 22
free function 40, 72
  fully typed closure expression 111
func keyword 40
function 40
  definition 75
  external parameter name 88
  generic declaration 140
  multiple return values 82
  overloading 86
functions
  advance 270
  assert 282
  countElements 61, 119
  enumerate 103
  fatalError 231
  max 76
  precondition 281
  print 23, 23, 73
  println 22, 23, 73
  sort 73
  sorted 73
functions (cont.)
  stride 59, 60, 103
  with multiple return
  values 5

G
  game playing 76
  Game template 292
  generic function 254
  maximum 258
  generic function declaration
  140
  generic overloaded operator
  function 286
  generic parameter clause 256
  generic type 99, 133, 254
  generic type constraint 134
  generics
    actual types 254
    associated type for a
    protocol 263
    function overload 259
    generic parameter clause
    256
    method 256
    placeholder for a type 254
    specialization 257
    type 259
    type argument 257
    type constraint 259
    type parameter 140, 256
    where clause 259
  get accessor of a computed
  property 117, 178, 179
  getter for a property 36
  Git 293
  global (free) function 22
  global function 40, 72
  global variable
  private 189
  Grand Central Dispatch
  (GCD) 113
  Graphical User Interface
  (GUI) 3
  group in the Project
  navigator 18, 296
  grouping related software
  components (module) 46,
  73
  guard condition in a case 57
  GUI Components
    Image View 290
    Label 290, 327, 328
    naming convention 337
    Slider 320
    guide lines 302
  H
  half-open range operator
  (...<) 56, 59, 103, 107,
  108
  has-a relationship 181
  hash table 154
  Hashable protocol 134,
  217, 248, 259
  hashing 154
  hash-table collisions 155
  hasPrefix method of a
  String 270
  hasSuffix method of a
  String 270
  height or a GUI component
  329
  “hidden” fields 84
  HIG (Human Interface
  Guidelines) 302
  horizontal tab 24
  Human Interface Guidelines
  (HIG) 302, 327
  I
  @IBAction 345
  @IBAction event-handling
  method 324
  @IBOutlet property 324,
  343
  iCloud 7
  IDE (integrated
  development
  environment) 10
  identical to (===) operator
  30, 45
  identifier 26
  identifier naming 26
  identifiers
    camel case naming 35
  Identity inspector 303
  identity operators 140
  identity value in a reduction
  operation 115
  if conditional statement 29
  if...else conditional
  statement 50
  Image attribute 303
  image set 291, 298
  Image View 290, 302, 303
  Images.xcassets 298
  immutable 99, 113
  by default 195
  Dictionary 134
  implementation of a
  function 230
  implicitly unwrapped
  failable initializer 173
  implicitly unwrapped
  optional 220, 222, 343
  import declaration 41, 342
  #import preprocessor
  directive 323
  in keyword
    introduce a closure’s body
    111
increment
  expression 65
  operator, ++ 53
  increment and decrement
  operators 53
  indentation 51
  index (subscript) 98
  indirect superclass 218
  infer enum constant’s type in
  a switch 205
  infinite loop 64, 93
  infinite recursion 93
  inheritance 215, 221, 344
  : notation 221
  base class (topmost
  superclass) 220, 248
inheritance (cont.)
deinitializers 251
examples 217
hierarchy 218
hierarchy for university CommunityMembers 218
initialization 248
initializers 249
two-phase initialization process 249, 249, 250
inherits 342
init keyword 38
init! failable initializer 173
init? failable initializer 172
initialization

two-phase initialization process in class hierarchies 249, 249, 250
initializer 37, 38, 42, 136, 173
call another initializer of the same class using self 168
convenience 167, 168
delegation 168
designated 167, 249
enum 209
failable 159, 170
failable convenience initializer 172
failable designated initializer 172
inherit 249
init! 173
init? 172
no-argument 168
overloaded 166
override a designated initializer 250
struct 198, 199
initializers cannot specify a return type 40
initializing two-dimensional Arrays in declarations 126

inout keyword 91, 195, 262
in an overloaded assignment operator function 272
inout parameter passing an Array by reference 123
insert method of a String 270
insert method of Array 104, 107
inspector 16, 295, 302
Attributes 303
Connections 303
File 16, 295
Identity 303
Quick Help 16, 41, 295
Size 303
instance method 188, 240
instance property 188, 240
Instruments 12
Int cast 83
Int primitive type 53
Int type 25, 27
Int16 type 25
Int32 type 25
Int64 type 25
Int8 type 25
integer 27
quotient 28
value 27
integer division 28
integrated development environment (IDE) 10
Interface Builder 3, 11, 12, 12, 289, 290, 290, 290
duplicate existing GUI components 324
Pin tools 335
interface in other object-oriented programming languages 238
internal
access modifier 35, 37, 46, 226
internal iteration 113
Internationalization 291, 308, 313
base language 314, 315
lock your components for localization 314
Internationalization and Localization Guide 314
iOS 8 3
iOS 8 for Programmers: An App-Driven Approach with Swift 7
iOS app templates 292
iOS Developer Enterprise Program xxxii
iOS Developer Library Reference 7, 323
iOS Developer Program xxxii, 308
iOS Developer University Program xxxii
iOS simulator 289, 291, 308
iPad 3
iPhone 3
iPod Touch 3
is-a relationship 217
is operator 346
isEmpty property
Array 105
Dictionary 141
String 269, 347
Issue navigator 16, 294
iteration of a loop 65
iteration statements 50
J
Jobs, Steve 3
join method of a String 271
jump bar in the Assistant editor 338
Jump To Definition 41
K
key in a Dictionary 132
keyboard
how to display 325, 344
keyboard shortcuts 18, 296

**Keyboard Type** attribute of a Text Field 334

keys property of type Dictionary 143

key–value pair 5, 132, 135, 155

keywords 50
@objc 238
as 238
as? 237
associativity 285

Bool 52
break 57
case 56
class 35
continue 65
convenience 168
default 56
deq 185
didSet 162
do 50, 58
e else 50
enum 80, 202
extension 159, 174
false 52
for 50, 63
func 40
if 50
import 41, 342
init 38
inout 91, 195
internal 35, 37, 46, 226
lazy 160, 191, 193
let 25, 324
mutating 199, 201, 205
operator 285
optional 247
override 224, 344
postfix 269, 278
precedence 285
prefix 269, 278
private 35, 46, 226
protocol 240
public 35, 37, 46, 226

keywords (cont.)
return 40, 42
self 39, 163
static 204, 240
subscript 281
super 223, 344, 344
switch 50
table of Keywords and reserved words 351
table of keywords and reserved words 351
typealias 263
var 25, 324, 343
weak 184
while 50, 58
willSet 162

L

**Label** 290, 304, 329
Alignment attribute 305, 332
Font attribute 305
Lines attribute 305
Text attribute 305
label 354
label in a switch 56
labeled break statement 354
exiting a nested loop 355
labeled continue statement 355
terminating a single iteration of a labeled loop 355
labeled statement 354
lambda 4
landscape orientation 297, 300
last-in, first-out (LIFO) order 263

Last property of Array 261
launch images 298
lazy stored property 160, 191, 192, 193
cannot have property observers 192

leading edge of a view 308
left associativity value 284
left brace, { 35
left justified 62
let keyword 25, 134, 324
Library window 302
LIFO (last-in, first-out) order 263
line break 22, 23
line-feed character, \n 24, 24
LLVM Compiler 12
load factor 155
locale 39, 39, 84
locale-specific property of class NSDateFormatter 182
locale-specific currency string 41, 322, 348
locale-specific date String 182
locale-specific percentage string 346, 347
localization 313
lock GUI components 314
localize 289
localizedStringFromNumber method of class NSNumberFormatter 182, 219, 347, 348
location simulation 12
lock your components for localization 314
All Properties 314
total storyboard 314
Localizable Properties 314
Non-localizable Properties 314

Nothing 314
log C function 74
logical AND, && 66, 68
truth table 67
logical negation, or logical NOT (!) operator truth table 68
logical operators 66
logical OR, || 66, 67
truth table 67
logical statement
switch 55
Look-and-Feel Observations
overview xxii
loop body 58
loop-continuation condition
50, 58, 63, 64, 65
loop statement 50, 57
do...while 50, 58
for 50
for...in 50, 58
while 50, 57
looping terminates 58
M
Mac Developer Library
Reference 7
Mac OS X 3
Macintosh 3
main.swift 21
make your point (game of
craps) 77
map Array elements to new
values 115
map method of an Optional
211
map method of Array 113,
115
Master-Detail Application
template 292
max C function 74
max property of an integer
type 25
max Swift Standard Library
function 76
maximum generic function
258
Media library 301
memberwise copy a value-
type object 199
memberwise initializer for a
struct 198, 199, 200
memory leaks 17, 296
memory-space/execution-
time trade-off 155
memory utilization 155
message 340
Metal 292
method 342
local variable 39
struct 198
method names
came case naming 35
min C function 74
min property of an integer
type 25
Minimum Font Scale 305, 332
minus sign (–) formatting
flag 62
missing auto layout
constraints 334
Mode attribute 303
module 46, 73
module (grouping related
software components) 46,
73, 227
monetary calculations 63, 185
monetary values 322
multidimensional Array
124, 126
multiline comment 22
nested 22
multiple function return
values 82
multiplication
* 28, 29
assignment operator, *=
53, 276
mutable Dictionary 134
mutate 113
mutating method in a
protocol 240
mutating method of a
struct 199, 201, 205
N
namespace 73
naming convention
GUI components 337
naming identifiers 26
Navigator area (Xcode) 16,
17, 293, 294, 295
Navigators 294
Breakpoint 16, 294
Debug 16, 294
Issue 16, 294
Log 16, 294
Project 16, 18, 294, 296
Search 16, 294
Symbol 16, 294
nested
Array initializers 125
for statement 126, 128
for...in loops 126, 128
functions 5
if...else conditional
statement 51
multiline comment 22
type 5, 202
newValue constant received
by willSet property
observer 162
NeXT Inc. 3
NeXT Interface Builder 12
NeXTSTEP operating
system 3, 323
nib file 12
no-argument initializer 168
non-deterministic random
numbers 76
none associativity value 284
not identical to (!==)
operator 30, 45
NSArray class 153, 279, 325
NSDate class 181, 182
NSDateFormatter class 182,
184
dateFormat property
184
dateStyle property 182
locale property 182
stringFromDate method
182, 184
timeStyle property 182
**NSDecimalNumber** class 34, 63, 185, 274, 323
decimalNumberBy-Adding method 187, 276, 349
decimalNumberBy-DividingBy method 349
decimalNumberBy-MultiplyingBy method 187, 220, 275, 349
initializer 186
one method 187
overloaded operators 275, 276
zero method 187
**NSDictionary** class 153, 279, 325
**NSMutableArray** class 153, 279, 325
**NSMutableDictionary** class 153, 279, 325
**NSMutableString** class 279
**NSNumber** class 347, 348
**NSNumberFormatter** class 41, 323, 347
localizedStringFromNumber method 186, 219, 347, 348
numberStyle property 41
stringFromNumber method 42, 219
**NSNumberFormatterStyle** enum 347
CurrencyStyle constant 186, 219
PercentStyle constant 219
**NSString** class 153, 279, 325
componentsSeparatedByString method 152
numberStyle property of class **NSDecimalNumber** 41
numeric keypad 320, 325, 327
display 344
numeric types in Swift 153, 278, 325

**O**
@objc keyword 238, 247
Object library 301
object-oriented programming (OOP) 215
Objective-C 3, 4, 10
parameter type 345
property 324
subclass 342
superclass 342
Objective-C protocols 246
oldValue constant received by didSet property observer 162
one method of class **NSDecimalNumber** 187
opacity of a color 328
OpenGL ES 12, 292
OpenStep 7
operating system 3
operator keyword 285
operator overloading 5, 265
operator precedence 29
Operator Precedence Chart Appendix 352
rules 29
operators
-->, predecrement/postdecrement 53
!, logical NOT 68
?:, ternary conditional operator 52
... (closed range) 56, 59, 107
..< (half-open range) 56, 59, 103, 107, 108
*=, multiplication assignment operator 53
/=, division assignment operator 53
&&, logical AND 66, 67
%=, remainder assignment operator 53
++, preincrement/postincrement 53
+=, addition assignment operator 52
-=, subtraction assignment operator 53
||, logical OR 66, 67
arithmetic 28
binary 28
compound assignment 52
decrement operator, -- 53, 54
increment and decrement 53
increment, ++ 53
logical AND, && 66, 68
logical operators 66
logical OR, || 66, 67
multiplication, * 28
postfix decrement 53
postfix increment 53
prefix decrement 53
prefix increment 53
remainder, % 28, 29
subtraction, - 29
ternary conditional operator, ?: 52
optional 4
  force unwrap with ! 145, 148
return value 119
returned from
  Dictionary subscript 145
unwrap 148
optional binding 121, 148, 153, 172, 209, 211, 224, 226, 238
  in a while statement 263
optional chaining 172, 211, 247
optional keyword 247
optional parameter
default value 89
Optional type 121, 210
optional value 42, 119
  unwrapping 119, 121
OS X 3
  Yosemite 3
outlet 324
  create 337, 338
outlet property name 337
output 22
overflow checking 5
overflow checking arithmetic operators 29
overflow operators 29
  &-, overflow subtraction 29
  &*, overflow multiplication 29
  &/, overflow division 29
  &%, overflow remainder 29
  &+, overflow addition 29
overloaded constructors 166
overloaded division operator 346
overloaded function 86, 254
overloaded methods 86
overloaded operator function 269
overloaded operators for complex numbers 273
overloading
  inout parameter in an overloaded assignment
  operator function 272
  unary ! operator 266
overloading generic methods 259
override a superclass member 250
override keyword 224, 344
predecrement 53
postdecrement 53
postfix declaration modifier 269, 278
postfix decrement operator 53
postfix increment operator 53
postincrement 53
pow C function 74
power of 2 larger than 100 57
precedence 29, 32
  arithmetic operators 29
default value 100 for a new operator 284
do a custom operator 284
of built-in operators 284
Precedence Chart Appendix 352
precedence context-sensitive keyword 285
precondition keyword 281
predecrement 53
prefix declaration modifier 269, 278
prefix decrement operator 53
prefix increment operator 53
preincrement 53
Preincrementing and postincrementing 54
principle 195
principle of least privilege 195
print a line of text 22
print function 23, 23
print Swift Standard Library function 73
Printable protocol 99, 136, 217, 248
  conform to 239
Printable Swift Standard Library protocol 73
println function 22, 23
println Swift Standard Library function 73
private
  access modifier 35, 46, 226
global variable 189
set for a property 37
private(set) 37
program in the general 216
program in the specific 216
programmatically select a
  component 325, 344
programming languages
  Objective-C 7
project 14, 291
project name 293
Project navigator 16, 18,
  294, 296
Project Structure group 18,
  296
property 342
  computed 37, 163, 324
  constant 36, 324
didSet observer 162
private setter 37
read-only in a protocol
  240
read-write property in a
  protocol 240
stored 36
stored properties must be
  initialized 248
variable 36, 324
willSet observer 162
property attribute 184, 343
  unowned 185
weak 184, 343
property declaration 339
property observer 160, 162,
  192
didSet 162
struct 198
validation 162
willSet 162
protocol 217, 228, 231, 238
  adopt 238, 241
  adopt more than one 241
class only 246
protocol (cont.)
conform 238
  convenience initializer in a
  conforming class 241
definition 240
designated initializer in a
  conforming class 241
failable initializer 241
mutating method 240
Objective-C 246
read-only property 240
read-write property 240
required initializer in a
  conforming class 240
requirements 238
with optional capabilities
  246
protocol composition 246,
  247, 259
protocol inheritance 246
protocol keyword 240
protocols
  Comparable 72, 247, 259
  Equatable 73, 247
  Hashable 134, 217, 248,
  259
  Printable 73, 136, 217,
  248
  SignedIntegerType 286
public
  access modifier 35, 37,
    46, 160, 226
  service 160
Q
Quick Help inspector 16, 41,
  295
R
  random numbers
    arc4random UNIX
    function 76
    arc4random_uniform
      UNIX function 76, 77
      element of chance 76
generation 116
range operators
  \ldots (closed range) 107
  \ldots (half-open range) 103, 107, 108
raw type of an enum type 81
raw value of an enum type
  constant 81
rawValue of an enum
  constant 204
rawValue property of an
  enum constant 83
Read-Eval-Print-Loop
  (REPL) 11
read-only computed
  property 160, 163
read-only property in a
  protocol 240
read-write property in a
  protocol 240
real number 27
recent projects 13
recursive factorial 92
recursive function 92
reduce method of Array
  114, 115, 128, 142
refer to an object 45
reference 45
reference count 184, 343
reference type 45, 91, 181
reinventing the wheel 40,
  322
remainder 28
remainder compound
  assignment operator, \%=
  53
remainder operator, \%
  28, 29
removeAll method of type
  Array 104, 107
removeAll method of type
  Dictionary 151
removeAll method of type
  String 270
removeAtIndex method of
type Array 104, 107
removeAtIndex method of
type String 270
removeLast method of type Array 104, 107
removeValueForKey method of type Dictionary 151
REPL (Read-Eval-Print-Loop) 11
replace a subset of an Array 108
Report navigator 16, 294
required initializer
in a class that conforms to a protocol 240
required keyword 251
requirements of a protocol 238
Resolve Auto Layout Issues 334
responder chain 325
return keyword 40, 42
return multiple values from a function 82
return type of a method or function 40
reuse 40, 322
reverse method of Array 112
Rhapsody 7
right align formatted text 62
right associativity value 284
right brace, } 35
rolling two dice 82
rounding a number 28, 62
rows of a two-dimensional Array 124
rule of thumb (heuristic) 66
rules of operator precedence 29
Run button (Xcode) 35, 321, 322
S
savings account 60
scene 300
SceneKit 292
Scheme selector (Xcode) 321
SCM (source-code management) repository 13
scope of a declaration 84
screen cursor 24
screen-manager program 227
select a component programmatically 325, 344
select a subset of an Array 108
selecting multiple GUI components 332
self
to call another initializer of the same class 168
self keyword 39, 163
sender of an event 345
set accessor of a computed property 117, 178, 179
setter for a property 36
shadow a property 39
sheet 291, 292
short-circuit evaluation 68
shorthand notation for read-only computed properties 163
shorthand type annotation Dictionary 136
shuffle a deck of cards 116
side effect 90
signature of a function 87
SignedIntegerType protocol 286
simulation 76
simulator 289, 291, 308
sin C function 74
double-entry/single-exit control statements 50
single-line comment, // 22
single-precision floating-point number 34
Single View Application project 325
Single View Application template 292
size class 300
Any 327
Compact Width 327
Regular Height 327
Size inspector 303, 336
Slice type 108
Slider 320
thumb 320, 330, 331
thumb position 346
Value Changed event 340, 345
Software Engineering
Observations overview xxii
software reuse 215
sort method of Array 109
sort Swift Standard Library function 73
sorted method of Array 109, 111
sorted Swift Standard Library function 73
source-code control system 293
source-code management (SCM) repository 13
special characters 24
\, backslash 24
`, single-quote 24
" , double-quote 24
\0 , null character 24
\n , line feed 24
\t , tab 24
in String literals 24
specialization (generics) 257
SpriteKit 292
sqrt C function 74
square brackets, [ ] 98
stack 259
StackTrace generic type 259
Standard editor (Xcode) 16, 294
Standard Library
class string 266
standard output 22
standard time format 165
startIndex property of a String 270
String type 22
startIndex property 23
Statements
break 57, 64, 65
conditional 49, 49
continue 65, 355
control-statement nesting 50
control-statement stacking 50
do...while 50, 58
for 50, 63
for...in 50, 58
if 29
if...else 50
labeled break 354, 354
labeled continue 355
loop 50, 57
nested if...else 51
switch 55
while 50, 57
static 240
static keyword 204, 240
type property or type method in a structure
or enumeration 190
static in other object-oriented languages 187
StepStone 3
stored property 36, 37, 117, 160, 223, 230
Lazy 160, 191, 193
must be initialized 248
struct 198
stored type property 187, 204
in a struct 198
storyboard 11, 290, 300
stride global function 59, 60, 103
closed-range 59
half-open range 60
String concatenation 42
String interpolation 5, 160, 238
String Swift Standard Library type 72
String type 22
append method 270
concatenation 22
conforms to Comparable 268
conforms to Equatable 268
endIndex property 270
hasPrefix method 270
hasSuffix method 270
insert method 270
interpolation 25, 26
isEmpty 347
isEmpty property 269
join method 271
literal 22
literals have type String 22
removeAll method 270
removeAtIndex method 270
startIndex property 270
String type in Swift 153, 278, 325
stringFromDate method of class NSDateFormatter 182, 184
stringFromNumber method of class NSNumberFormatter 42, 219
strong reference 184, 185, 343
strong reference cycle 185
struct 198
default initializer 199
initializers in an extension 198, 199
keyword 195
memberwise initializer 198, 199, 200
mutating method 199, 201, 205
structured programming 66
subclass 215, 223, 342
initializer 223
subscript
struct 198
subscript keyword 281
subtraction 28
operator, - 29
subtraction compound assignment operator, -= 53
super keyword 223, 344, 344
superclass 215, 342
direct 218
indirect 218
Swift 4, 7, 320
AnyObject type 279
Apple publications 9
sample code 9
Swift Blog 9
Swift filename extension (.swift) 21
Swift Keywords 351
Swift Programming Language book 9, 263
Swift Resource Center 18
Swift Standard Library 6
Array type 72
Boolean and numeric types 72
Comparable protocol 72
Dictionary type 72
Equatable protocol 73
max 76
print function 73
Printable protocol 73
println function 73
sort function 73
sorted function 73
String type 72
Swift Standard Library Reference 7
switch conditional statement 55
  case label 56, 57
  default case 56
  where clause in a case 57
switch logic 57
switch statement 205
  infer enum constant’s type 205
Symbol navigator 16, 294
T
  tab bar 292
  tab character, \t 24
  tab stops 24
Tabbed Application template 292
tan C function 74
target-language attribute (XLIFF) 316
template 291
termination housekeeping 185
ternary conditional operator, ?: 52
Test navigator 16, 294
Text Field 327, 328
  Editing Changed event 340, 345
  Keyboard Type attribute 334
Text property 329
Text property of a Label 329
text property of a UILabel 347
text property of a UITextField 346
  thumb of a Slider 320, 330, 331
  thumb position of a Slider 346
timeStyle property of class
  NSDateFormatter 182
tokenize a String 152
topmost superclass (base class) 220, 248
trailing closure 111
  trailing edge of a scene 334
  trailing edge of a view 308
true 29, 50, 52
  truncate 28
truth tables
  for operator ! 68
  for operator && 67
  for operator || 67
tuple 4, 40, 55, 82, 104, 210
  compose 82
  decompose 82, 142
  ignoring a value during decomposition 82
  returning multiple values from a function 82
two-dimensional Array 124
two-phase initialization process 223, 249, 250
  type annotation 25, 98
  parameter 38
type argument 257, 262
  type checking 254
  type constraint 134, 258, 259, 286
  type inference 4, 25, 136
  type method 187, 189, 190, 240
    accessing from a class’s other members 190
    struct 198
    type parameter 140, 256, 258, 260
    section 256, 260
  type property 187, 190, 240
    accessing from a class’s other members 190
    computed 187, 189
    stored 187, 204
    stored in a struct 198
    type safe 99, 237, 256, 257
    type scope 188
typealias keyword 263
types
  Array 72, 153, 278, 325
  Bool 26, 27
types (cont.)
  Boolean and numeric types 72
  Dictionary 72, 153, 278, 325
  Double 26, 27, 34
  Float 26, 27, 34, 34
  Int 25, 27
  Int16 25
  Int32 25
  Int64 25
  Int8 25
  max property of an integer type 25
  min property of an integer type 25
  numeric 153, 278, 325
  Optional 121
  Slice 108
  String 22, 72, 153, 278, 325
 _UINT16 26
 _UINT32 26
 _UINT64 26
 _UINT8 26
U
UIAccessibility protocol 312
UIImageView class 290, 302
UIKit 7, 323
UIKit framework 301, 323, 324, 342
UILabel 324
UISlider 324
UITextField 324
UIViewController 324
UILabel class 290, 324
  text property 347
_UINT16 type 26
_UINT32 type 26
_UINT64 type 26
_UINT8 type 26
UISlider class 324
  value property 346
UITextField class 324
   text property 346
UIViewController class 324, 344
UML (Unified Modeling Language) 218
Unicode Technical Standard #35 for locale-specific formatting 184
Unified Modeling Language (UML) 218
unique ID of a GUI component 314
unique raw values for constants in enum types 202
unit tests 17, 296
universal app 289, 291, 293, 296
universal-time format 160, 165
unowned property attribute 185
unqualified name 84
unspecified number of arguments 128
unwrap an optional 148
rewrapping an optional value 119, 121
updateValue method of type Dictionary 150, 151
uppercase letter 27
user interface (UI) 290
user interface events 340
Using Swift with Cocoa and Objective-C 9, 10, 153, 279
Utilities area (Xcode) 16, 17, 293, 295

V
validate a property with a property observer 162
value binding 57
Value Changed event for a Slider 340, 345
value in a Dictionary 132
value property of a UISlider 346
value type 5, 45, 91, 133, 181
value vs. reference types blog post 210
values property of type Dictionary 145
value-type
   memberwise copy 199
var keyword 25, 324, 343
var parameters 123
variable
   reference type 45
   variable names
   camel case naming 35
   variable number of arguments 128
   variable property 36, 324
   variadic parameter 128
Version editor (Xcode) 12, 16, 295
view controller 324
view debugger 12
view device logs 22
viewDidLoad message 325
VoiceOver enable/disable 291, 311, 313
W
weak keyword 184
weak property attribute 343
Welcome to Xcode window 13
where clause 259
where clause in a case 57
while loop statement 50, 57
while statement
   optional binding 263
   white space 22
willSet property observer 162
workspace window 16
Wozniak, Steve 3
WWDC (Apple World Wide Developer Conference) xix, 3
X
Xcode 290, 321
   Assistant editor 16, 295, 338, 340
code-completion suggestions 22
Debug area 16, 17, 293, 295
Editor area 16, 293, 294
Navigator area 16, 17, 293, 294
Single View Application project 325
Standard editor 16, 294
Utilities area 16, 17, 293, 295
Version editor 16, 295
Xcode 6 10
Xcode Groups
   Project Structure 18
   project structure 296
Xcode IDE 289
Xcode Libraries
   Code Snippet 301
   File Template 301
   Media 301
   Object 301
Xcode navigators
   Breakpoint 16, 294
   Debug 16, 294
   Find 16, 294
   Issue 16, 294
   Project 16, 18, 294, 296
   Report 16, 294
   Symbol 16, 294
   Test 16, 294
Xcode playground 22
Xcode project 22
Xcode toolbar 17, 295
Xcode Windows
  Library 302
  Welcome to Xcode 13
XCTest 12
Xerox PARC (Palo Alto Research Center) 3

XLIFF
  XML Localization Interchange File Format 314, 315
XML Localization Interchange File Format (XLIFF) 314, 315

Y
  Yellow Box API 7

Yosemite (OS X) 3

Z
  zero method of class NSDecimalNumber 187
  zero-based counting 98