Contents

Preface xiii
   How This Book Is Organized xiii
   About the Sample Code xiv
   Contacting the Author xv

1 Welcome to Modern Swift 1
   Migrating Code 2
      How to Migrate 2
      Migration Lessons 4
   Using Swift 4
      Compiled Applications 4
      Frameworks and Libraries 5
      Scripting 5
      REPL 6
      Playgrounds 7
      Other Swift 8
   Learning Swift 8
   Wrap-up 9

2 Printing and Mirroring 11
   Basic Printing 11
      Printing Multiple Items 12
      Adding Item Separators 13
      String Interpolation 13
      Controlling Line Feeds 14
   Recipe: Printing to Custom Destinations 15
      Printing to Strings 16
      Printing to Custom Output Streams 17
   Recipe: Printing and String Formats 18
      Swift and Format Specifiers 19
      Format Limitations 19
      Conditional Compilation 20
   Debug Printing 21
      Custom Stream Output 21
Recipe: Postfix Printing 23
Quick Looks 24
  Quick Look for Custom Types 24
  Quick Looks for Custom Types in Playgrounds 25
  Playground Quick Looks 25
  Creating Custom Quick Looks for Playgrounds 26
  Built-in Quick Look Types 27
  Third-Party Rendering 28
Using Dump Functions 30
Building Custom Mirrors 31
  Recursive Reflection 32
  Building Basic Mirror Descriptions Using Protocol Conformance 33
Adding Header Docs 35
  Building Header Documentation 35
  Markdown Support 36
  Keywords 37
  Special Swift Concerns 40
  Adding Images to Header Docs 40
Wrap-up 41

3 Optionals?! 43
Optionals 101 43
Unwrapping Optionals 45
  Forced Unwrapping 45
  Conditional Binding 46
  Conditional Binding and Type Casts 46
  Binding Cascades 48
  guard Statements 50
  Implicitly Unwrapped Optionals 51
  Guarding Failable Initializers 52
  Optionals and Sentinel Values 53
  Coalescing 55
  Optional Assignment 55
  Optional Patterns 56
Contents

Optional Chaining 58
  Selector Testing and Optional Chaining 59
Subscripts 60
Optional Mapping 61
  Maps and Chaining 61
  Filtering nil Values with flatMap 62
Unmanaged Wrappers 62
Wrap-up 64

4 Closures and Functions 65
  Building a Function 65
    Parameter Names 66
    Label Conventions 66
    Naming Methods and Functions 68
    External and Local Parameters 69
    Defaults 70
    Constant and Variable Parameters 71
    Modifying Parameters 72
  Closures and Functions 73
    Function Types 73
    Using Tuple Arguments 74
    Using Shorthand Argument Names 74
    Inferred Closure Types 76
    Parameter Type Inference 77
    Declaring Type Aliases 78
    Nested Functions 79
Tuples 80
  From Tuples to Structs 82
    Tuple Return Types 82
Variadic Parameters 83
Capturing Values 85
Autoclosures 87
  Adding Default Closures 90
Currying 91
  Why Curry? 92
Building Libraries 92
Partial Application 93
Contents

Currying Costs 94
Currying and Callbacks 94
Practical Currying 95
Passing Closures as Function Parameters 97
Wrap-up 99

5 Generics and Protocols 101
Expanding to Generics 101
Protocols 102
Tokens 103
Type Constraints 104
Adopting Protocols 104
Declaring Protocols 105
Member Declarations 107
Building a Basic Protocol 109
Adding Protocol Default Implementations 110
Optional Protocol Requirements 111
Swift-Native Optional Protocol Requirements 112
BuildingGenericTypes 113
TypeParameters 113
GenericRequirements 114
ConformanceRequirements 115
Recipe:Same-TypeRequirements 115
GenericBeautification 117
LegalTokens 117
MatchingAliases 118
ProtocolAliasDefaults 119
CollatingAssociatedTypes 119
ExtendingGenericTypes 120
UsingProtocolsasTypes 121
Protocol-BasedCollections 121
SelfRequirements 122
ProtocolObjectsandSelfRequirements 123
LeveragingProtocols 124
Wrap-up 124
6 Errors 125
   Failing Hard 125
      Fatal Errors 126
      Assertions 126
      Preconditions 127
      Aborting and Exiting 128
   Failing Gracefully 128
      The ErrorType Protocol 129
      Choosing Between Optionals and Error Handling 130
   Swift Error Rules 130
      Rule 1: Move Away from nil Sentinels 131
      Rule 2: Use throw to Raise Errors 132
      Rule 3: Use Error Types with Visible Access 133
      Rule 4: Mark All Error-Participating Methods with throws 134
      Rule 5: Use rethrows Consistently 134
      Rule 6: Consume Errors Where They Matter 135
      Rule 7: Terminate Threaded Error Chains 135
   Building Errors 137
      Good Errors 138
      Naming Your Errors 138
   Adding String Descriptions 139
      Adding Reasons 139
      Simplifying Output 140
      Extending String 140
      Type-Specific Errors 141
   Retrieving Context 141
      Contextualizing Strings 142
      Contextualizing Throwing Types 143
      Simplifying Contexts 144
   Calling Throwing Functions 145
      Using try 145
      Error Propagation 147
      Using try! 148
      Using try? 148
Implementing Alternatives to \texttt{try}? 149
  Guarding Results 151
  Building a Printing Version of \texttt{try}? 151
Working with \texttt{guard} and \texttt{defer} 151
Wrap-up 153

7 Types 155
  Language Concepts 155
    Reference and Value Types 155
    Copy-and-Write-Back 156
    Algebraic Data Types 157
    Other Terminology 158
  Enumerations 159
    Basic Enumerations 159
    Using Hash Values to Support Ranges 160
    Raw Value Enumerations 162
    Raw Value Members and Sequences 163
    Associated Values 164
    Indirect Values 165
  Switches 166
    Branching 166
    Breaks 167
    Fallthroughs 167
    Complex Cases 169
    Tuples 169
    Pattern Matching with Value Bindings 169
      \texttt{where} Clauses 170
    Unwrapping Optional Enumerations 171
  Embedding Values by Type 172
  Option Sets 174
    Revisiting \texttt{NS\_OPTIONS} 174
    Building Enumerations 175
    Building Option Sets 176
    Viewing Options 176
Preface

More than a year after its introduction, the Swift programming language is still growing and evolving. While it seems a bit ridiculous writing a book about a language that’s not fixed, final, and polished, that’s exactly what The Swift Developer’s Cookbook does. Swift is not so raw and young that even in its ever-changing state, it doesn’t know what it’s trying to achieve for its audience of Apple developers. A modern type-safe language, Swift already has locked down its fundamentals, even as its specific implementation details resolve over time.

Swift is simply a joy to program in. Its constructs and libraries present you with new ways to craft code, handle data, and perform the endless daily tasks that make up a programmer’s life. From protocol-oriented and functional programming to first-class closures and algebraic data types, Swift offers a fresh and exciting take on programming. The more time you spend developing in Swift, the harder it becomes to return to older languages that don’t offer these powerful features.

This book is not a traditional tutorial. It’s written for programmers both experienced and new who are looking to push existing aptitudes into a new arena. Each concept-focused chapter covers a practical skill set. These chapters guide you to mastery over those essential Swift development tasks. You needn’t read this cookbook from cover to cover (although you are more than welcome to do so). Instead, dive into whatever topic you want to learn about and uncover key insights to bring away with you from each discussion.

The Swift Developer’s Cookbook has been an amazing project to work on. I hope you enjoy reading this book half as much as I’ve enjoyed working on it.

How This Book Is Organized

This book offers a practical survey of key Swift development topics. Here’s a rundown of what you’ll find in this book’s chapters:

**Chapter 1, “Welcome to Modern Swift”—**This chapter explores the kinds of applications you can build using this new modern type-safe programming language. Working in a new and changing language isn’t always smooth sailing. Since Apple introduced Swift, the language has remained in constant flux. From beta to beta, dot release to dot release, new features and updated syntax mean source code that compiles in one release might not in the next. In this chapter, learn what it means to work in an evolving language and how to migrate your code as the language updates.

**Chapter 2, “Printing and Mirroring”—**Although programming focuses on crafting artifacts through code, it’s important to remember that code serves the developer as well as the end user. Code doesn’t just need to compile: It should communicate clearly, coherently, and effectively as well. The features discussed in this chapter cover the gamut of output scenarios that range from user-facing write operations to developer-facing debugging support. This chapter surveys these technologies and explores how to precisely build feedback, documentation, and output to fit your development and debugging needs.
• **Chapter 3, “Optionals?!”—**Unlike in many other languages, in Swift a nil is not a pointer. Using it is a safe and expressive way to represent the potential for both valid and invalid values within a single construct. Learning how to recognize and use optionals is an essential step in mastering the Swift language. This chapter introduces optionals and surveys the supporting constructs you need in order to create, test, and successfully use optionals in your code.

• **Chapter 4, “Closures and Functions”—**Lexical closures provide the basis for methods, functions, and “block” parameters, all of which power the Swift applications you develop. By encapsulating state and functionality, they promote behavior to first-class constructs. This chapter explores closures, showing how they work in Swift and how to best incorporate them into your applications.

• **Chapter 5, “Generics and Protocols”—**Generics help you build robust code to expand functionality beyond single types. A generic implementation services an entire group of types instead of just one specific version. A combination of generic types and protocols (behavior contracts) establishes a powerful and flexible programming synergy. This chapter introduces these concepts and explores how you can navigate this often-confusing development arena.

• **Chapter 6, “Errors”—**In Swift, as in any other programming language, things fail. In daily development tasks, you encounter both logical errors—that is, things that compile but don’t work the way you expect them to—and runtime errors that arise from real-world conditions such as missing resources or inaccessible services. Swift’s response mechanisms range from assertions that fail fatally to error types that support recovery, enabling you to track what went wrong, and offer runtime workarounds. This chapter introduces errors and helps you understand how to handle many kinds of failure.

• **Chapter 7, “Types”—**When it comes to types, Swift offers three distinct families. Its type system includes classes, which provide reference types, and enumerations and structures, which are both algebraic value types. Each provides unique strengths and features to support your development. This chapter surveys some of the key concepts used in the Swift language and explores how its types work in your applications.

• **Chapter 8, “Miscellany”—**Swift is a vibrant and evolving language with many features that don’t always fit tidily under a single umbrella. This chapter introduces an assortment of topics that could not find proper homes elsewhere in this book but that still deserve your attention.

**About the Sample Code**

You’ll find the source code for this book at https://github.com/erica/SwiftCookbook on the open-source GitHub hosting site. There, you’ll find a chapter-by-chapter collection of source code that provides examples of the material covered in this book.

Retrieve sample code either by using git tools to clone the repository or by clicking GitHub’s Download button. It was at the right center of the page when I wrote this book. It enables you to retrieve the entire repository as a ZIP archive or tarball.
Contribute!
Sample code is never a fixed target. It continues to evolve as Apple updates its Swift language. Get involved. Pitch in by suggesting bug fixes and corrections and by expanding the code that’s on offer. GitHub allows you to fork repositories and grow them with your own tweaks and features and then share them back to the main repository. If you come up with a new idea or approach, let me know.

Getting GitHub
GitHub (http://github.com) is the largest git-hosting site, with more than 150,000 public repositories. It provides both free hosting for public projects and paid options for private projects. With a custom Web interface that includes wiki hosting, issue tracking, and an emphasis on social networking of project developers, it’s a great place to find new code or collaborate on existing libraries. Sign up for a free account at the GitHub website, which then allows you to copy and modify this repository or create your own open-source iOS projects to share with others.

Contacting the Author
If you have any comments or questions about this book, please drop me an email message at erica@ericasadun.com or stop by the GitHub repository and contact me there.
Acknowledgments

My sincere thanks go out to Trina MacDonald, Chris Zahn, and Olivia Basegio, along with the entire Addison-Wesley/Pearson production team, specifically Betsy Gratner, Kitty Wilson, and Nonie Ratcliff, and my amazing team of technical editors, Kevin Ballard, Alex Kempgen, and Sebastian Celis.

My gratitude extends to everyone who helped read through drafts and provided feedback. Specific thanks go out to Ken Ferry, Jeremy Dowell, Remy Demarest, August Joki, Mike Shields, Phil Holland, Mike Ash, Nate Cook, Josh Weinberg, Davide De Franceschi, Matthias Neeracher, Tom Davie, Steve Hayman, Nate Heagy, Chris Lattner, Jack Lawrence, Jordan Rose, Joe Groff, Stephen Celis, Cassie Murray, Kelly Gerber, Norio Nomura, “Eiam,” Wess Cope, and everyone else who contributed to this effort. If I have omitted your name here, please accept my apologies.

Special thanks also go to my husband and kids. You guys are the best.
About the Author

**Erica Sadun** is a bestselling author, coauthor, and contributor to several dozen books on programming and other digital topics. She has blogged at TUAW.com, O'Reilly's Mac Devcenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in computer science from Georgia Tech’s Graphics, Visualization and Usability Center. A geek, a programmer, and an author, she’s never met a gadget she didn’t love. When not writing, she and her geek husband have parented three geeks-in-training, who regard their parents with restrained bemusement, when they’re not busy rewiring the house or plotting global domination.
Editor's Note: We Want to Hear from You!

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

You can email or write me directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

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

Email: trina.macdonald@pearson.com

Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education
75 Arlington St., Ste. 300
Boston, MA 02116

Reader Services

Register your copy of The Swift Developer's Cookbook at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account*. Enter the product ISBN, 9780134395265, and click Submit. Once the process is complete, you will find any available bonus content under Registered Products.

*Be sure to check the box that you would like to hear from us in order to receive exclusive discounts on future editions of this product.
This page intentionally left blank
Optionals?! 

nil happens. When dictionary lookups fail, when instance properties are case dependent, when asynchronous operations have not completed, when failable initializers cannot create instances, and in dozens of other situations where values may or may not have been set, Swift may return nil instead of some other more concrete content. Swift provides nil as a powerful tool for expressing situations in which values are unavailable for use.

Swift differentiates these “no content” scenarios from error handling. In error handling, control flow changes to offer failure mitigation and reporting. With nil, a value-of-no-value represents an absence of data. It offers a testable placeholder to be used when no data is otherwise available.

Unlike in many other languages, in Swift, nil is not a pointer. It is a safe and expressive way to represent the potential for both a valid and invalid value within a single construct. In Swift, the Optional type encapsulates this concept and enables you to differentiate between successful value assignments and nil cases.

Learning how to recognize and use optionals is an essential step in mastering the Swift language. This chapter introduces optionals and surveys the supporting constructs you need to create, test, and successfully use optionals in your code.

Optionals 101

Question marks and exclamation points are the hallmark of Swift optionals. Any variable marked with ? may—or may not—contain a valid value. To get a sense of this language feature, consider the following example:

```swift
var soundDictionary = ["cow": "moo", "dog": "bark", "pig": "squeal"]
print(soundDictionary["cow"] == "moo") // prints true
print(soundDictionary["fox"]) // What does the fox say?
```

What does the fox say? In Swift, the answer is nil. In Swift, nil indicates “no value.” Unlike in Objective-C, in Swift, nil is not a pointer. Swift nil indicates a semantic missing
non-existence, a count of Monte Cristo, a cup of “not tea,” an honest politician. A nil item means “nothing to see here, move along, move along, move along.”

In this soundDictionary example, the variable stores a string dictionary. With strings for both keys and values, its type is Swift.Dictionary<Swift.String, Swift.String>. You can also represent this type as [String: String], using square brackets and a colon. Swift infers this type from the concrete data provided on the right-hand side of the soundDictionary assignment. Alternatively, you can use explicit type annotation in your code by adding a colon and a type in the variable declaration:

```swift
var soundDictionary: [String: String] = 
    {
        "cow": "moo", 
        "dog": "bark", 
        "pig": "squeal"
    }
```

Although the dictionary uses String keys and values, when you look up any item in this dictionary, the value returned is not a String. It’s typed String?. That question mark is critical to understanding dictionaries because it indicates an Optional type. Dictionary lookups may succeed or fail. Optional return types encapsulate both possibilities.

Contrast this behavior with arrays, where it’s the programmer’s job to check whether an index exists before accessing it. Both types could easily be implemented with the other convention, of course, but the Swift people chose the more likely use case for each type. Arrays are highly bounded with a small domain of legal lookup indexes. Dictionaries are often sparse compared against their possible key domain. Optionals enables dictionaries to better represent their “may or may not map to a value” results.

Confirm the return type with Quick Help. Enter the dictionary and the following assignment in Xcode. Then Option-click the sound symbol (as in Figure 3-1) or select the symbol and open View > Utilities > Show Quick Help Inspector. The declaration line in the Quick Help presentation confirms that the type assigned to sound is String?:

```swift
var sound = soundDictionary["cow"] // sound is typed String?
```

![Figure 3-1](https://via.placeholder.com/150)

**Figure 3-1** Xcode’s Quick Help reveals symbol typing.

While it might appear that the dictionary returns “moo” on success or nil for failed lookups, this is actually misleading. Print the output to the Xcode console, with print(sound). A successful result looks like Optional("moo"), with its value embedded in an optional wrapper. In this case, the Optional type uses a .Some enumeration, and its associated value is the "moo" string.
Unwrapping Optionals

You cannot use this optional directly as a string. The following example fails because you must unwrap Optional types before using their values:

```swift
let statement = "The cow says " + sound
```

The + operator in this case works on two string values, not on a string and an optional. Unwrapping gives you access to the string stored within the optional wrapper, enabling you to perform tasks like appending values.

### Unwrapping Optionals

Optional types always return either a wrapped value or nil. Wrapping means that any actual content is stored within a logical outer structure. You cannot get to that value (in this case, "moo") without unwrapping it. In the Swift world, it is always Christmas, and there are always presents—or at least variables—to unwrap.

Swift offers a variety of mechanisms for unwrapping optionals and recovering the underlying data type stored within them. There are more ways to get at a wrapped optional value than you might expect. The expressive possibilities expanded in Apple's Swift 2 update to provide greater developer flexibility.

#### Forced Unwrapping

The simplest way to extract an optional value is to append an exclamation point:

```swift
print("The sound is \(sound!)") // may or may not crash
```

This is also the most dangerous approach, and you want to generally avoid it in production code.

If the value is wrapped (that "optional" result you just read about), the exclamation point returns the value stored within the wrapped element. If the value is nil, you experience a fatal runtime error—which is not nearly as fun or gratifying for your workday.

The nil value is the Swift equivalent of your aunt's home-crocheted vest with a large friendly moose decoration on it. It's pretty important to check whether an item is nil before unwrapping it. You can more safely unwrap with an exclamation point within an if statement because this sound is nominally guaranteed to be non-nil in non-extreme (that is, nonthreaded) situations:

```swift
if sound != nil {
    print("The sound is \(sound!)") // doesn't crash
}
```

Safety is a relative thing. While this forced unwrap is pretty safe at compile time and runtime in its current use, it's dangerous while editing. If you accidentally move that print line out of the if block, the compiler cannot warn you about it, and you will ship broken code.
In this example, the `sound` variable is also not `weak`. It won’t deallocate between the `nil` test and unwrapping. Never use forced unwraps with `weak` variables because they may do just that.

As a rule, avoiding forced unwraps, even within `if` clauses, helps you avoid potential disasters.

**Conditional Binding**

Conditional binding provides a better and safer approach that avoids forced unwrapping. In conditional binding, you optionally bind (that is, assign) a value to a new constant or variable using an `if-let` or `if-var` statement. When the optional is not `nil`, Swift unwraps the value, assigns it, and executes a block of code. When `nil`, you either enter an `else` clause or continue to the next statement:

```
if let sound = soundDictionary[animal] {
    print("The \(animal) says \(sound)")
} else {
    print("Any sound the \(animal) says is unknown to modern science")
}
```

It’s conventional in Swift to use identical identifiers in conditional bindings for both wrapped and unwrapped versions. In the following snippet, a `mySound` constant is unwrapped within the `if-let` (or `if-var`) scope but wrapped outside it:

```
let mySound = soundDictionary[animal] // mySound is wrapped
if let mySound = mySound {
    print(mySound) // mySound is unwrapped
}
```

Using the same identifier makes it clear that you’re working with the same semantics in different states of undress. Conditional binding clauses tend to be short. This enables you to verify your overloaded symbol intent with a glance.

The right-hand value in an `if-let` assignment must be an optional. You cannot use this construct with an already-unwrapped item. The following statement leads to an error, complaining that a conditional binding initializer must have `Optional` type:

```
if let sound = soundDictionary["cow"]! { // error!
    print("The cow says \(sound)")
}
```

**Conditional Binding and Type Casts**

Type casting enables you to re-interpret an instance’s type at runtime. Swift offers two distinct type-casting operators (`is` and `as`) with four variations: `is`, `as`, `as?`, and `as!`. These operators enable you to check value types and cast values to other types. The `is` operator checks whether you can convert an instance’s type to another, typically more specialized, type:
let value = 3
value is Any // true
value is Int // true
value is String // false

The `as` operator casts to other types and can test whether an item conforms to a protocol. These are its three forms:

- The `as` operator applies direct casts. The compiler must determine at compile time that the cast will succeed.
- The `as?` variant contains a question mark. It performs a conditional cast that always returns an optional value. If the cast succeeds, it's wrapped in an optional instance. If not, it returns `nil`. This is the cast I use most often in day-to-day coding. The compiler issues warnings when it detects that a cast will always succeed (for example, `String` to `NSString`) or always fail (`String` to `NSArray`).
- The forced variant `as!` includes an exclamation point. It either returns an unwrapped item or raises a runtime error. This is the most dangerous cast type. Use it when you know a cast will always succeed or when you want your application to prematurely crash.

Conditional binding and conditional casting work hand-in-hand. Figure 3-2 demonstrates why. In this screenshot, Swift code creates a string-indexed dictionary that stores arbitrary `Any` instance types. Even when a lookup returns a valid integer for the "Three" key, a forced unwrap still reports `Any` type. To use this value as a number, you must downcast from `Any` to `Int`.

```
let value = 3 // Int
let optionalValue = value as? Int // Int?
```

Conditional binding enables you to apply a cast, test the resulting optional, and then bind unwrapped values to a local variable. In the following example, `result` is bound as an `Int` constant within the if clause code block:

```
if let result = dict["Three"] as? Int {
    print("\(result + 3) = 6") // 6 = 6
}
```

Figure 3-2   Lookups from a `[String:Any]` dictionary return `Optional<Any>` values. The type remains `Any` even after forced unwrapping.
Chapter 3  Optionals?! 

This conditional pattern of `fetch`, `cast`, and `bind` commonly appears when working with structured data from web services and databases. Swift enables you to use type safety while navigating through key paths down to data leaves.

Recipe 3-1 searches a `UIView` hierarchy, using conditional casting to match subviews against a supplied type, returning the first subview that matches that type. The `as?` type operator returns an optional, which is conditionally bound to a local view variable. If the binding succeeds and a type match is found, the function returns. Otherwise, it recurses down the tree in a depth-first search.

**Recipe 3-1  Using Conditional Binding to Test Types**

```swift
public func fetchViewWithType<T>(
    type t: T.Type, contentView: UIView) -> T?
{
    for eachView in contentView.subviews {
        if let view = eachView as? T { return view }
        if let subview = fetchViewWithType(
            type: t.self, contentView:eachView) {
            return subview
        }
    }
    return nil
}
```

**Binding Cascades**

Although useful for accessing values, `if-let` bindings build into code structures colloquially known as *pyramids of doom*. Each successive test indents your code further and further to the right, creating a triangle of empty space to the left of the `if` statements and then the closing brackets:

```swift
if let cowSound = soundDictionary["cow"] {
    if let dogSound = soundDictionary["dog"] {
        if let pigSound = soundDictionary["pig"] {
            // use sounds here
        }
    }
}
```

It’s the sort of code you don’t want to bring home to your project manager—or your mummy.

Using multiple optional bindings within a single test enables you to avoid this structural mess. First introduced in Swift 1.2, cascading permits any number of conditional bindings tests within a unified conditional:
if let 
cowSound = soundDictionary["cow"],
dogSound = soundDictionary["dog"],
pigSound = soundDictionary["pig"] {
    // use sounds here
}

If any assignment fails, evaluation stops, and control moves past the if statement. This approach is particularly useful when working with web-sourced data, such as JSON, where each step may depend on the results of the previous conditional binding. The following example is from a script that checks prices on the iTunes App Store:

if let 
json = json as? NSDictionary,
resultsList = json["results"] as? NSArray,
results = resultsList.firstObject as? NSDictionary,
name = results["trackName"] as? String,
price = results["price"] as? NSNumber {
    // Process results
}

This code pulls data from the App Store and deserializes it with NSJSONSerialization. The results list is extracted from the JSON data, and the app results from that list, and so forth.

While this approach avoids pyramid indentation, it introduces a bad tendency toward code constipation. Increased blockiness makes if-let cascades hard to mentally process. Copious commenting and spacing can mitigate this. As shown here, the results are faster for the brain to parse, make it easier to spot errors, and are better for adding, removing, and reordering intermediate steps:

if let 
    // Access JSON as dictionary
    json = json as? NSDictionary,

    // Retrieve results array
    resultsList = json["results"] as? NSArray,

    // Extract first item
    results = resultsList.firstObject as? NSDictionary,

    // Extract name and price
    name = results["trackName"] as? String,
    price = results["price"] as? NSNumber {
        // Process results
    }
guard Statements

The guard statement was first introduced in Swift 2. It offers another way to unwrap and use optionals. Although guard is not limited to use with optionals—you can use it with general Boolean conditions as well—it provides a major development advantage that’s specific to optional handling. In particular, guard statements move code out of indented scopes by offering early-return error handling.

Early-return inverts the handling of success and failure paths. Errors are handled first, enabling success code to be written at the highest possible scope. Compare this to if-let, where you cannot put the failure path first and your success conditions must be handled in a child scope. Without guard statements, all conditionally bound values must be used with the if statement code block that binds them. For example, the following snippet conditionally binds a cowSound constant using if-let, and the cowSound constant is undefined outside that scope:

```swift
if let cowSound = soundDictionary["cow"] {
    // cowSound is unwrapped within this scope
}
// cowSound is undefined outside the scope
```

Like if-let, a guard statement conditionally unwraps and binds optional variables. When that binding fails, it executes a mandatory else clause, which must exit the current scope, using throw, break, continue, or return statements. Otherwise, the guard statement unwraps the optional value and binds it to a variable you can use throughout the remaining lifetime of the current scope:

```swift
guard let cowSound = soundDictionary["cow"] else {throw Error.MissingValue}
// cowSound now defined and unwrapped
```

Think of a guard statement as a soldier that will not permit the flow of execution to continue forward unless its condition or conditions are met. Unlike if-let, guard does not establish new scopes. Instead, it guides traffic through the existing scope.

Like if-let, a single guard statement can express multiple conditions, separated by comma delineators. For example, you might perform multiple constant assignments:

```swift
guard let
cowSound = soundDictionary["cow"],
dogSound = soundDictionary["dog"]
ext {throw Error.MissingValue}
```

Or you might include a more general test along with assignments:

```swift
guard
soundDictionary.count > 2,
let cowSound = soundDictionary["cow"],
let dogSound = soundDictionary["dog"]
ext {throw Error.MissingValue}
```
Unwrapping Optionals

Implicitly Unwrapped Optionals

Swift provides a feature called implicitly unwrapped optionals, which you declare by appending an exclamation point to the end of the type. Unlike normal optionals, these versions unwrap themselves, much like a jack-in-the-box or the hired entertainment at a bachelor party. Use implicitly unwrapped optionals cautiously. They’re a fairly dangerous feature as they introduce fatal runtime errors if you attempt to access values stored inside a nil case optional.

Here’s an example of first an assignment to a normally typed constant and then to an implicitly unwrapped one. The difference is what happens when you access the wrapped value. That’s where the name comes from because during access, the value is implicitly unwrapped:

```swift
var animal = "cow" // 1
let wrappedSound = soundDictionary[animal] // Optional("moo")
let unwrappedSound: String! = soundDictionary[animal] // "moo"

// prints: The unwrapped sound is "moo"
let soundString = "\" + unwrappedSound + "\" // 2
print("The unwrapped sound is \(soundString")
```

In this example, when you use unwrappedSound in the string assignment, its value is not Optional("moo"). Its value is accessed as moo, and it behaves like a simple string. Implicit unwrapping extracts the value from an optional item and uses the data type you’d expect from a successful dictionary lookup. Once unwrapped, you can use the variable directly. You don’t have to add that “unwrap this variable” exclamation point or use conditional binding.

A real danger arises from unwrapping nil values. If you replace the assignment to animal in the line marked 1 with "fox" instead of "cow", this code raises a fatal error in the line marked 2. The runtime complains that it unexpectedly found nil while unwrapping an Optional value. With great unwrapping comes great responsibility. Take care that you properly guard these items to ensure that you don’t attempt to unwrap nil.

Limit implicit unwrapping to when you know in advance that a variable will always have a value after some logical point. For example, if you’re responding to button taps or menu clicks, you probably don’t have to wonder “does this button or menu item exist?” It does. If it didn’t, you would never have reached your callback.

Don’t use implicit unwrapping for more general cases like dictionary lookups. That’s just asking for trouble. You can, however, print unwrapped items and test them against nil:

```swift
print(unwrappedSound) // prints nil
if unwrappedSound != nil {
    print("Valid:", unwrappedSound)
}
```

An error won’t be raised until you perform an operation that attempts to access the value inside.
Chapter 3  Optionals?!  

As a rule, it is legal to assign a value, an optional value, or \texttt{nil} to an implicitly unwrapped variable. The following code is legal and will compile without error:

```swift
var myImplicitlyUnwrappedOptional: String!
myImplicitlyUnwrappedOptional = Optional("Value")
myImplicitlyUnwrappedOptional == "Value"  // true
myImplicitlyUnwrappedOptional = nil  // do not test except against nil
myImplicitlyUnwrappedOptional == nil  // true
myImplicitlyUnwrappedOptional = "Value"
myImplicitlyUnwrappedOptional == "Value"  // true
```

Assignments to both optional and non-optional values end with non-optional results, which you see by comparing against the non-optional string literal "Value". The danger lies in the \texttt{nil} assignment. A non-\texttt{nil} comparison tries to access the value and ends in a runtime error. The win is that you can perform an assignment to either optional or non-optional values with a single assignment.

An implicitly unwrapped scenario is common when working with Interface Builder. A view \textit{binding} is optional; it may or may not be properly instantiated in the interface file. The view \textit{use} is normally non-optional; presumably you set up the binding correctly. Implicit unwrapping simplifies code access to views at the cost of potential crashes during development and testing.

Swift 2’s one-line \texttt{guard} statement more or less offers a shorthand summary for the following initialization pattern:

```swift
let cowSound: String! = soundDictionary["cow"]
if cowSound == nil {throw Error.missingValue}  // Handle nil case
// cowSound’s value is now usable without explicit unwrapping
```

This assignment uses implicit unwrapping and then tests against \texttt{nil}. This is a safe check because the code makes no attempt to access an associated value. If the optional is \texttt{nil}, the if statement’s conditional block executes, transferring control away from this scope. Otherwise, execution continues, and the unwrapped \texttt{cowSound} constant is now available for use.

An implicitly unwrapped approach is both wordier and less safe than \texttt{guard}. It uses multiple statements compared with \texttt{guard}’s one, and it makes no conditions about leaving the current scope. If the \texttt{throw} request were replaced with a \texttt{print} statement, the \texttt{nil} case would continue forward. In doing so, it might encounter a use of \texttt{cowSound}’s nonexistent value and a nasty runtime crash.

Use implicit unwrapping cautiously and prefer \texttt{guard} where possible.

**Guarding Failable Initializers**

When an initializer returns an optional instance, it’s called a \textit{failable} initializer. The initializer may succeed in establishing a new instance, or it may fail and return \texttt{nil}. For example, the following structure can be initialized only with even numbers:
struct EvenInt {
    let number: Int
    init?(number: Int) {
        if number % 2 == 0 {self.number = number} else {return nil}
    }
}

Mark failable initializers by appending a question mark or an exclamation point to the `init` keyword (`init?` or `init!`). The punctuation choice depends on whether the initializer returns a normal optional instance or an implicitly unwrapped one. The implicitly unwrapped variation is almost never used in real life. According to my technical editor Kevin Ballard, you might theoretically encounter one when working with legacy code ported from Core Foundation or from any unaudited Objective-C API.

Use `guard` to test assignments from failable initialization. Here’s an example built around the `EvenInt` struct. When created with an odd number, an `EvenInt` initializer returns `nil`. The `guard` statement throws an error and exits the scope:

```swift
do {
    guard let even2 = EvenInt(2) else {throw Error.Odd}
    print(even2) // prints EvenInt(number: 2)
    guard let even3 = EvenInt(3) else {throw Error.Odd} // fails
    print(even3) // never gets here
} catch {print(error)}
```

Although you can use any approach for testing and unwrapping optionals, `guard` statements offer a natural synchronicity with failable initializers. Guarding initializers once again enables you to test for failed initialization at the point where you declare variables and constants and ensures that these bindings remain valid for the remaining scope.

**Optionals and Sentinel Values**

It is tempting to use optionals to signal when an operation has failed. The following snippet represents Swift prior to version 2 and is common to Apple’s traditional Cocoa patterns:

```swift
func doSomething() -> String? {
    let success = (arc4random_uniform(2) == 1) // flip a coin
    if success {return "success"} // succeed
    return nil // fail
}

if let result = doSomething() {
    // use result here
}
```

An unsuccessful operation returns `nil`, and a successful one returns a value.
Starting in Swift 2, reserve `nil` failures for initialization (although you can also use `throws` in initializers as well as in normal code) and prefer `guard` over `if-let`. Instead of using optionals as sentinels—that is, to signal success and fail conditions—use Swift’s new error-handling system. Error handling enables you to redirect the flow of control to mitigate errors and provide recovery support:

```swift
func betterDoSomething() throws -> String {
    let success = (arc4random_uniform(2) == 1) // flip a coin
    if success {return "success"} // succeed
    throw Error.failure // fail
}

do {
    let result = try betterDoSomething()
} catch {print(error)}
```

This refactoring skips optionals; the `nil` case is never of interest to the client code. Swift 2 error handling means you never have to unwrap.

When a thrown error is not of interest to its consumer, the `try?` operator can discard errors and convert results to optionals. This lets you integrate new-style errors with old-style optional handling. A `try?` expression returns a wrapped value for successful calls and `nil` for thrown errors:

```swift
guard let foo = try? somethingThatMayThrow else {
    // ...handle error condition and leave scope
}
if let foo = try? somethingThatMayThrow {}
```

The new error-handling system profoundly affects Cocoa-sourced APIs. Calls that used `NSError` pointers pre-Swift 2 change their return type from optional to non-optional, add the `throws` keyword, and eliminate error pointers from API calls. The new system sends `NSError` through `do-try-catch`. Compare the old approach with the new one:

```swift
// Old
func dataFromRange(range: NSRange, documentAttributes dict: [NSObject: AnyObject], error: NSErrorPointer) -> NSData?

// New
func dataFromRange(range: NSRange, documentAttributes dict: [String: AnyObject]) throws -> NSData
```

By introducing error handling, optionals can eliminate overloaded “failed call” semantics. It’s always better to use `throws` with well-defined errors than to use optional sentinel values. When you really have no information to pass back to the caller other than “I failed,” Swift 2’s updated error system simplifies creating an error enumeration to explain why that failure occurred. It is ridiculously easy to add informative errors that don’t require complicated `NSError` initializers:

```swift
enum Error: ErrorType { case BadData, MemoryGlitch, ItIsFriday }
```
Although many current APIs, especially asynchronous handlers and calls based on Core Foundation, have yet to transition to the new system, I encourage you to update your code to avoid using optionals as sentinel values. Return your optionals to the “contains a value or does not contain a value” semantics they were designed to handle.

**Coalescing**

Swift’s nil-coalescing operator `??` unwraps optionals and provides fallback values for the nil case. This next example checks uses nil-coalescing to assign a value to the `sound` constant:

```swift
let sound = soundDictionary["fox"] ?? "unknown"
```

If the lookup succeeds and the dictionary returns a wrapped value, the operator unwraps it and assigns it to `sound`. If not, it assigns the fallback value, which is "unknown". In either case, the assignment is not optional. `sound` uses a `String` type, not `String?`.

Use nil coalescing when you can supply fallback values that won’t interrupt a normal operational flow. When you cannot, choose `guard` instead and handle nil as an error case in the `else` clause.

**Note**

If the optional is non-nil, the right-hand side of the operator is never evaluated. The operation short-circuits, as with the Boolean operators `&&` and `||`.

**Optional Assignment**

In nil coalescing, you must supply a valid unwrapped fallback value. In cases where a fallback does not exist, consider optional assignment instead. This approach shortcuts any case where an optional value is unavailable. Normally, you embed assignments into an `if-let` scope. If the conditional binding succeeds, you assign the unwrapped value.

Alternatively, you might consider creating a custom operator that conditionally assigns values to a target, as in the following example:

```swift
// Thanks, Mike Ash
infix operator =? {}
public func =?(inout target: T, newValue: T?) {
    if let unwrapped = newValue {
        target = unwrapped
    }
}
```

This snippet builds an `=?` operator that supports simple assignment by wrapping and hiding its underlying `if-let` approach with a basic `infix` call.
The following assignments show this operator in action. The `s` string variable updates only for non-nil assignments:

```swift
var s: String = "initial value"
s =? "value 1" // value 1
s =? nil // value 1
```

Hiding if-let overhead makes the resulting conditional assignments cleaner and potentially more intuitively obvious to read through.

### Optional Patterns

A Swift `pattern` matches a value's structure rather than just the value itself. Patterns decompose and express a data structure, including component elements. They enable you to refer to an item's subconstructs when testing instead of dealing with instances as a monolithic whole. Using patterns is a powerful and nuanced way to represent an instance, and it's extremely handy for working with optionals.

The optional enumeration consists of two cases, `.None` and `.Some`. The `.Some` case contains an associated value of an arbitrary type:

```swift
enum Optional<T> {
    case None
    case Some(T)
}
```

With optionals, pattern matching enables you to limit statements to work only with the `.Some` case. You can reach into that case and bind the internal value using a single declaration. Follow the `case` keyword with a specific enumeration (the `.Some` case) and then bind the value nested within it using `let` or `var`:

```swift
switch soundDictionary[animal] {
    case .Some(let sound):
        print("Using case, the unwrapped sound is 
(sound)")
    default: break
}
```

The result is an unwrapped value, ready for access.

The `case .Some(let constant)` expression provides a one-to-one pattern match with the underlying optional. Optionals with the `None` case don't match this pattern, so the `case` code need not consider `nil` scenarios.

There's no denying that the `case .Some(let constant)` is awkward to process visually. It lacks grace. Responding to the complexity of this optional pattern-matching code, Swift 2 introduces shorthand using a postfixed question mark:
Unwrapping Optionals

```swift
switch soundDictionary[animal] {
    case let sound?:
        print("Using case, the unwrapped sound is \(sound) [2]")
    default: break
}
```

This postfixed question mark is nothing more than syntactic sugar for the `.Some case. The results are simpler and more readable because this case matches and unwraps its optionals.

The preceding example shortchanges Swift because there’s still no major advantage in using this one-case switch statement over, say, `if-let`. When you introduce a simple where clause, you begin to see where pattern-matching optionals adds power and concision.

The following snippet uses a `switch` statement to differentiate unwrapped optionals whose string values are longer than five characters from those that are shorter:

```swift
switch soundDictionary[animal] {
    case let sound? where sound.characters.count > 5:
        print("The \(sound) is long")
    case let sound?:
        print("The \(sound) sound is under 5 characters")
    default: break
}
```

Swift’s `switch` cases may experience significant logical overlap where small details select which case to execute.

Using pattern matching with `if` statements enables you to drop `switch` statement overhead and simplify one-case code even further. You lose the default statement and the surrounding `switch`, and you reduce the check to just a simple pattern comparison with where clause support. This `if` statement uses both pattern matching and a where clause for precise testing:

```swift
if case let sound? = soundDictionary[animal] where sound.hasPrefix("m") {
    print("Sound \(sound) starts with m")
}
```

The where clause isn’t limited to `if-case`. You can construct a similar statement using `if-let`:

```swift
if let sound = soundDictionary[animal] where sound.hasPrefix("m") {
    print("Sound \(sound) starts with m")
}
```

Pattern matching also enables you to iterate through an array and perform actions on unwrapped non-nil values. The `for-case-let` approach simplifies working with optional collections, as you see in the following snippet:

```swift
// Collect sound optionals into array
let soundOptionals = ["cow", "rhino", "dog", "goose", "hippo", "pig"].map({soundDictionary[$0]})
```
Optional Chaining

In Swift, you chain methods and properties by appending period-delimited selectors. Each function in the chain returns an intermediate value. This allows calls to be joined into a single statement without requiring variables that store intermediate results:

```
soundDictionary.description.characters.count
```

This approach creates a fluent interface, which is ideally a parsimonious and more readable expression of a set of operations you want to consider as a single unit. A danger, of course, lies in over-chaining. If you’re producing enormous lines of code that are difficult to debug and hard to read and that cannot be easily commented or differentiated on updates, you’re probably doing this wrong. Ask yourself, “Would I ever need to put a breakpoint in this statement or step through it?” If the answer is yes, you are over-chaining.

Swift introduces a powerful feature called optional chaining. Swift method calls may return optionals, and you must take this into account when forming chains. Swift provides a way that an entire chain can fail gracefully on encountering nil.

Optional chaining involves adding question marks after optional values. For example, you might look up an animal in the sound dictionary and use optional chaining to return a capitalized version of the sound:

```
soundDictionary[animal]?.capitalizedString // Moo or nil
```

Even though capitalizedString normally returns a non-optional, this chain returns String?. It may succeed or fail, depending on the lookup.

Add question marks to any chain participants that return optionals:

```
soundDictionary[animal]?.characters.first?.hashValue // returns Int?
```
You can add forced unwrapping to any chain item by replacing the question mark with an exclamation point. This usage comes with the same forced unwrapping dangers discussed earlier in this chapter:

```
soundDictionary[animal]!.capitalizedString // Moo or Runtime Error
```

Here’s a real-world example of where you might use optional chaining to simplify an if-let pattern. This code extends Array to return the index of a possible maximum element. Swift’s standard library `maxElement()` function returns an optional based on whether a sequence has values to compare (Apple writes in the standard library, “Returns the maximum element in `self` or `nil` if the sequence is empty.”):

```swift
extension Array where Element: Comparable {
    var maxIndex: Int? {
        if let e =
            self.enumerate().maxElement({$1.element > $0.element}) {
        return e.index
    }
    return nil
}
}
```

Introducing optional chaining greatly simplifies this code, enabling you to shortcut the index lookup and returning `nil` if the `maxElement` call fails. Recipe 3-2 returns the index of an array’s maximum value.

**Recipe 3-2  Using Optional Chaining to Shortcut Evaluations**

```swift
extension Array where Element: Comparable {
    var maxIndex: Int? {
        return self.enumerate().maxElement(
            {$1.element > $0.element}?.index
        )
    }
}
```

Extend Recipe 3-2’s functionality to all collection types with the following snippet:

```swift
extension CollectionType where Generator.Element: Comparable {
    var maxIndex: Index? {
        return self.indices.maxElement({self[$1] > self[$0]})
    }
}
```

**Selector Testing and Optional Chaining**

Optional chaining isn’t just about transforming your code into bite-sized lines. It also acts as shorthand to test whether an item responds to a method or property selector. Optional
chaining offers a rough equivalent to Objective-C’s `respondsToSelector:` method, enabling you to determine whether it’s safe to execute calls on particular instances.

Commonly, you work with subclasses that are directly related to each other but that implement distinct method sets. For example, you might retrieve a collection of SpriteKit nodes from a scene and then adjust the line widths of the shape nodes. This snippet uses a failable cast followed by an optionally chained property assignment:

```swift
for node in nodes {(node as? SKShapeNode)?.lineWidth = 4.0}
```

This selector-testing approach also works in pure Swift, as in the following example:

```swift
// Root class put two subclasses
class Root {func rootFunc() {}}
class Sub1: Root {func sub1Func() {print("sub1")}}
class Sub2: Root {func sub2Func() {print("sub2")}}

// Create heterogeneous array of items
var items: [Root] = [Sub1(), Sub2()]

// Conditionally test and run selectors
(items[0] as? Sub1)?.sub1Func() // runs
(items[0] as? Sub2)?.sub2Func() // no-op, nil
```

This snippet constructs a heterogeneous array of `Root` subclasses. It then performs conditional casts and uses selector tests before calling class-specific methods.

Selector testing enables you to test whether a method exists before constructing a new instance. Adding the question mark ensures that the `NSString` call won’t fail with a runtime “unrecognized selector” exception:

```swift
let colorClass: AnyClass = UIColor.self
let noncolorClass: AnyClass = NSString.self

colorClass.blueColor?() // returns a blue color instance
noncolorClass.blueColor?() // returns nil
```

This is a special behavior of `AnyClass` and `AnyObject` that works only with Objective-C methods, for compatibility with `class` and `id`. These are special cases because these types return functions as implicitly unwrapped optionals. Other types don’t do that.

### Subscripts

Contrary to expectations, optional chaining with subscripts *doesn’t* introduce safe lookups. This is an important factor that you should internalize as soon as possible and recognize in your code. In the following example, if you try to access index 8 (aka the ninth element of this six-element array), your code dies with a fatal `Array index out of range` error:

```swift
let array: Array? = [0, 1, 2, 3, 4, 5]
array?[0] // 0
array?[8] // still fails
```
In this example, the question mark does not qualify the lookup for safety. It is required for subscripting after `array`, which is optional. With subscripts, you add chain annotations in-line after the optional value and before the subscript brackets.

Optional chaining is meant solely to set and retrieve values for subscripts used with optional values. It does not and cannot short-circuit failed subscripts unless you build a failable subscripting extension, as in the following example:

```swift
extension Array {
    subscript (safe index:.UInt) -> Element? {
        return Int(index) < count ? self[Int(index)] : nil
    }
}
```

Once you add a simple array `safe-index` extension, you can optionally chain the safe version of the subscript. In the following calls, the `Element?` results of the `safe:` subscript are now optional and can be chained:

```swift
print(array?[safe: 0].dynamicType) // nil
print(array?[safe: 8].dynamicType) // Optional(Swift.Int)
```

### Optional Mapping

Swift's `map` and `flatMap` functions enable you to conditionally apply functions to optional values. Their calls are similar, as you see in the following declarations, and both are incredibly useful tools:

```swift
/// If `self == nil`, returns `nil` . Otherwise, returns `f(self!)` .
func map<U>(f: @noescape (T) -> U) -> U?
/// Returns `f(self!)` iff `self` and `f(self)` are not nil.
func flatMap<U>(f: @noescape (T) -> U?) -> U?
```

The `map` closures return type `U`, which may or may not be an optional, while `flatMap` closures specifically return type `U?`, which is always an optional. This practical limitation simply means you can use `map` with closures that return non-optionals, but you cannot do the same with `flatMap`:

```swift
// flatMap must return optional
print(word.flatMap({string in string})) // compiles
// print(word.flatMap({string in string})) // errors
```

### Maps and Chaining

When working with optionals, `map` and `flatMap` both act like chaining, but you supply an arbitrary closure instead of a chained method name or property:
Chapter 3  Optionals?!

```swift
var word: String? = "hello"
var cap = word?.capitalizedString // Optional("Hello")
cap = word.map({$0.capitalizedString}) // Optional("Hello")
```

When you just want to unwrap-and-apply, use `map`. This mapping:

```swift
UIImage(named:"ImageName").map({print($0.size)})
```

is equivalent to this `if-let`:

```swift
if let image = UIImage(named:" ImageName ") {
    print(image.size)
}
```

Both the mapping and `if-let` include about the same level of code complexity for this particular example. Both unwrap the optional returned by `UIImage(named:)` and then print the size. You can argue which approach is better. Both bind the unwrapped result to a local constant whether that constant does or does not have an explicit name.

**Filtering nil Values with `flatMap`**

The `flatMap` function offers great utility both in the realm of optionals and outside it. With optionals, you can use `flatMap` to filter away `nil` values and easily convert an array of optionals to an array of unwrapped values:

```swift
let optionalNumbers: [Int?] = [1, 3, 5, nil, 7, nil, 9, nil]
let nonOptionals = optionalNumbers.flatMap({$0})
print(nonOptionals) // [1, 3, 5, 7, 9]
```

Recipe 3-3 uses a single `flatMap` call to eliminate `nil` instances and extract values from their optional wrappers.

**Recipe 3-3  Extracting Members from an Optionals Array**

```swift
func flatMembers<T>(array: [T?]) -> [T] {
    return array.flatMap({$0})
}
```

**Unmanaged Wrappers**

In rare cases (which are growing rarer by the day), a Core Foundation function may return a C pointer or an object reference embedded in an `Unmanaged` wrapper. You encounter these in the older, dustier, and stranger parts of Cocoa, where gruves still lurk in shadows. Keychain Services is a notorious offender in this regard. You must transfer `Unmanaged` references into the normal memory management system before working with them.
An Unmanaged wrapper, like an Optional wrapper, provides a layer of safety between your code and a potentially nasty crash. The Unmanaged<T> type stores a pointer whose memory is not controlled by the Swift runtime system. Before using this data, you take responsibility for how this memory should stay alive.

In Cocoa, this works very much like bridging in Objective-C. Unwrap any object with an existing +1 retain count using takeRetainedValue(). This applies to any item built with Create or Copy in its name. Use takeUnretainedValue() for +0 returns.

If you have an Objective-C framework or are developing one that you would like people to use in their Swift application—and if there are methods or functions in your Objective-C framework that return Core Foundation objects—decorate your methods or function names with CF_RETURNS_RETAINED or CF_RETURNS_NOT_RETAINED. If you don’t decorate your methods or functions, Core Foundation objects are returned as unmanaged.

In Swift 2, CF-IMPLICIT-BRIDGING-ENABLED and CF-IMPLICIT-BRIDGING-DISABLED automatically bridge based on Core Foundation naming conventions. So if you audit your APIs and ensure that they follow get/copy/create conventions, you can avoid specific method decoration.

For example, UTTypeCopyPreferredTagWithClass returns a +1 CFString instance. Assign this result with takeRetainedValue(), making sure to test for failed calls. Unwrapping nil causes nasty crashes that even blessed potions of restore life will not fix. Recipe 3-4 demonstrates how to use unmanaged wrappers by returning a preferred file extension for a given universal type indicator.

Recipe 3-4  Conditional Binding from Unmanaged Wrappers

```swift
import MobileCoreServices
import Foundation

enum Error: ErrorType { case NoMatchingExtension }

public func preferredFileExtensionForUTI(uti: String) throws -> String {
    if let result = UTTypeCopyPreferredTagWithClass(
        uti, kUTTagClassFilenameExtension) {
        return result.takeRetainedValue() as String
    }
    throw Error.NoMatchingExtension
}
```

This recipe uses conditional binding. The UTTypeCopyPreferredTagWithClass() function returns Unmanaged<CFString>?, which is an optional instance. If the call fails and returns nil, the function throws an error. Test the recipe by passing it a few common UTIs, such as public.jpeg and public.aiff-audio:
let shouldBeJPEG = try PreferredFileExtensionForUTI("public.jpeg")
let shouldBeAIFF = try PreferredFileExtensionForUTI("public.aiff-audio")

Use `takeUnretainedValue()` to unwrap any object built by a Core Foundation function with
Get in its name (for example, `CFAllocatorGetDefault()`) and constants passed as unmanaged
objects (for example, `kLSSharedFileListSessionLoginItems`). These items are not
automatically retained for you. Unlike with `takeRetainedValue()`, calling `takeUnretained-
Value()` won’t consume a retain upon unwrapping.

These functions follow the patterns established in Apple’s *Memory Management Programming
Guide for Core Foundation*, where you can read more about the “create rule,” the “get rule,” and
other details about memory ownership policies. Search the web for the latest update of this
document.

Wrap-up

Optionals are an invaluable component of Swift development. With their “possible value”
semantics, they enable you to store and represent the results of lookup operations whose data
may or may not exist. Optionals are a powerful workhorse construct that you regularly use in
day-to-day operations.

Swift’s new early-return `guard` statement is a gift that eliminates awkwardness from your code.
Now you can assign, unwrap, and use values with a clear path for missing values. Between `guard` and `nil` coalescing, Swift 2 can elegantly express both fail-on-nil and fallback-on-nil
scenarios with a minimum of overhead and indenting.

Swift 2’s revised error handling has also started to eliminate the role of optionals as sentinel
values. This usage continues to be common as it’s simple to use optionals to represent fail and
success states for method calls. Ideally, optional sentinels will shrivel and die as the language
matures and Apple’s APIs catch up with current language features. Until then, know that it’s
always better to use `throws` with well-defined errors than to use optional-style sentinels in the
code you control.

`nil` happens. Be prepared.
Index

Symbols
@objc keyword, 112
? (question mark), 44
optional chaining, 58-61
postfixed question marks, 57

A
abort() function, 128
access control, 159
access-level modifiers, 185-186
adopting protocols, 104-112
algebraic data types, 157-158
aliases, type aliases, 78
alternatives to try? operator, 149-151
Apple Developer Forums, 9
Apple Swift blog, 9
applications, 4
terminating
abort() function, 128
assertions, 126-127
fatalError() function, 126
preconditions, 127-128
arguments. See also parameters
format specifiers, 19-20
positional arguments, 75-76
separator, 13
shorthand argument names, 74-76
tuple arguments, 74
arrays
comparing with tuples, 73
multi-index array subscripting, 84, 194-195
safe indexing, 193-194
_slices, 195-196
typed arrays, 84
as_operator, 47
ASCII WWDC, 9
Ash, Mike, 33, 128
assertions, 126-127
associated types, 118
collating, 119-120
autoclosures, 87-91

B
Ballard, Kevin, 197
basic enumerations, 159-160
beautification, 117
best practices
for protocols, 124
Swift error handling, 130-137
  annotating with throws keyword, 134
  avoid nil sentinels, 131-132
  consuming errors, 135
  error types with visible access, 133
  throwing errors, 132-133
binding cascades, 48-49
blogs, Apple Swift blog, 9
bracket overload, 117
branching, 166-167
breaks, 167

building
convenience initializers, 181
convolution libraries, 92-93
custom mirrors, 31-34
enumerations, 175-176
errors, 137-139
functions, 65-72
  parameters, 69
generic types, 113-114
header documentation, 35-36
option sets, 176
protocols, 109
  with default implementations, 110-111
built-in Quick Look types, 27-28

C
callbacks, 94-95
capture lists, 85-86
capturing values
capture lists, 85-86
  unowned references, 86-87
chat rooms, 9
classes, 177-184
  initializers, 178-184
    convenience initializers, 179
deinitializers, 183-184
designated initializers, 179
failable initializers, 181-183
  optimization, 178
subclassing, 178
closures, 65
  autoclosures, 87-91
  capturing values
    capture lists, 85-86
    unowned references, 86-87
default values, adding, 90-91
inferred closure types, 76-77
initializers, rules, 179-181
no-escape closures, 88-90
passing as function parameters, 97-99
shorthand argument names, 74-76
trailing closures, 77
tuples
  arguments, 74
  return types, 82-83
types, 73-74
coalescing, 55
Cocoa coding guidelines, 68
collating associated types, 119-120
collections, protocol-based, 121
commenting
  Quick Help, 35-40
    adding images to header
documentation, 40
    building header documentation, 35-36
  keywords, 37-39
  markdown support, 36-37
  method/function annotations, 40
Comparable protocol, 160
comparing
  arrays and tuples, 73
  nil and error handling, 43
compiling, conditional compilation, 20-21
complex cases, 169
conditional binding, 46
type casts, 46-48
unmanaged wrappers, 62-64
conditional compilation, 20-21
conformance requirements, 115
conforming to the protocol, 102
constraints
  conformance requirements, 115
  same-type requirements, 115-117
constructors, 67
  referencing, 68
constructs, 158
consuming errors, 135
control flow, 189
  statement labels, 189-190
controlling line feeds, 14-15
convenience initializers, 179
  building, 181
converting
  Swift code to modern syntax, 2-4
    tuples to structs, 82
convolution libraries, building, 92-93
convolution operations, 91
copy-and-write-back, 156-157
cost of currying, 94
Curry, Haskell, 91
currying, 91-97
  callbacks, 94-95
    convolution libraries, building, 92-93
    cost of, 94
    partial application, 93
    reasons for using, 92
custom destinations, printing to, 15-18
custom mirrors, building, 31-34
CustomDebugStringConvertible protocol, 22
CustomStringConvertible protocol, 22, 177

D

De Franceschi, Davide, 160
dbgPrint() function, 21-23
  fallback cascade, 22
declarating
 operators, 191-192
 protocols, 105-106
 member declarations, 107-108
 Self requirements, 122
 default parameters, 70-71
 default values, adding to closures, 90-91
 defer statement, 151-153
 deinitializers, 183-184
 designated initializers, 179
 dictionaries, ? (question mark), 44
 didSet, 185
 documentation, Quick Help, 35-40
 adding images to header documentation, 40
 building header documentation, 35-36
 keywords, 37-39
 markdown support, 36-37
 method/function annotations, 40
 DoubleInitializable protocol, 109
 dump() function, 30

e
embedding values by type, 172-174
 enumerations, 139, 159-166
 associated values, 164-165
 basic enumerations, 159-160
 building, 175-176
 indirect values, 165-166
 raw value enumerations, 162-163
 Equatable protocol, 101-102
 error handling, 54, 125
 adding context to errors, 141-142
 building errors, 137-139
 calling throwing functions, 145-149
 error propagation, 147
 try operator, 145-146
 try! operator, 148
 try? operator, 148-149
 contextualizing strings, 142-143
 contextualizing throwing types, 143-144
 defer statement, 151-153
 enumerations, 139
 ErrorType protocol, 129-130
 good errors, 138
 guard statements, 151-153
 naming errors, 138-139
 versus nil, 43
 versus optionals, 130
 simplifying context, 144-145
 string descriptions, adding, 139-141
 extending strings, 140-141
 reasons for errors, supplying, 139
 simplifying output, 140
 type-specific errors, 141
 Swift rules, 130-137
 annotating with throws keyword, 134
 avoid nil sentinels, 131-132
 consuming errors, 135
 error types with visible access, 133
 terminate threaded error chains, 135-137
 throwing errors, 132-133
 ErrorType protocol, 129-130
 establishing
 generic types, 114
 protocols, 105
evaluating operator trade-offs, 192-193
explicit typing, 75
extending generic types, 120-121
extensions, 159, 186-187

F
failable initializers, 52-53, 163, 181-183
fallthroughs, 167-168
fatalError() function, 126
flatMap() function, 62
FloatLiteralConvertible protocol, 109
forced unwrapping, 45-46
format specifiers, 19
  limitations, 19-20
Foundation interoperability with Swift, 203
frameworks, 5-6
functions
  abort(), 128
  annotations in Quick Help, 40
  assert(), 126-127
  building, 65-72
    parameters, 69
currying, 91-97
callbacks, 94-95
convolution libraries, building, 92-93
cost of, 94
partial application, 93, 95-97
reasons for using, 92
debugPrint(), 21-23
dump(), 30
fatalError(), 126
flatMap(), 62
map(), 61-62
naming conventions, 68-69
nested functions, 79-80
parameters
  default values, 70-71
  labels, 66
  let keyword, 71-72
  modifying, 72
  passing closures as, 97-99
  type inference, 77-78
  var keyword, 71-72
  variadic parameters, 83-84
print, 11-15
  controlling line feeds, 14-15
  fallback cascade, 22
  item separators, 13
  parameters, 12-13
  terminator parameter, 14
tuples
  arguments, 74
  return types, 82-83
types, 73-74

G
generics, 101-102. See also types
  associated types, collating, 119-120
  beautification, 117
  building, 113-114
  constraints
    conformance requirements, 115
    same-type requirements, 115-117
  extending, 120-121
  legal tokens, 117-118
  parameters, 113-114
  requirements, 114-120
getters, 185-186
good errors, 138

guard statements, 50, 151-153
failable initializers, 52-53

hash values, 160-162
Hashable protocol, 114
header documentation, building, 35-36
IEEE printf specification, 18
if statements, 14
images, adding to header
documentation, 40
implicitly unwrapped optionals, 51-52
indices, wrapping, 195
inferred closure types, 76-77
inheritance, multiple inheritance, 103
initializers, 158, 178-184
  convenience initializers, 179
    building, 181
deinitializers, 183-184
designated initializers, 179
failable initializers, 181-183
  rules, 179-181
initializing, strings, 19
inout parameters, 16
interpolation, 13-14
IRC (Internet Relay Chat), 9
item separators, 13

joinWithSeparator function, 203
kernels, 91
Keychain Services, 62
keywords
  @objc keyword, 112
    inout, 72
  let, 71-72, 169-170
override, 186-187
protocol, 106
Quick Help, 37-39
  label terms, 39
static, 107
throws, 134
var, 71-72, 169-170

label terms (Quick Help), 39
labels, 66
  conventions, 66-68
  statement labels, 189-190
  for tuples, 80-81
Lattner, Chris, 1
lazy evaluation
  lazy properties, 188
  lazy sequences, 187-188
learning Swift, 8-9
let keyword, 71-72, 169-170
libraries, 5-6
  convolution libraries, building, 92-93
  visualization libraries, 29
limitations, of format specifiers, 19-20
line feeds, controlling, 14-15
logging, mirroring, 30
  building custom mirrors, 31-34
  protocol extensions, 34
  recursive reflection, 32-33

Mandelbrot sets, 7
map() function, 61-62
markdown support, Quick Help, 36-37
member declarations, 107-108
raw value members, 163-164

members, 158

Memory Management Programming Guide
for Core Foundation, 64

methods
 annotations in Quick Help, 40
 naming conventions, 68-69
 tuple return types, 82-83

migrating code, 2-4

mirroring, 30
 building custom mirrors, 31-34
 protocol extensions, 34
 recursive reflection, 32-33

modifying parameters, 72

multi-index array subscripting, 84, 194-195

multiple inheritance, 103

mutating requirements, 108


N

naming
 functions, 68-69
 methods, 68-69

nested functions, 79-80

nil, 43-45
 coalescing, 55
 versus error handling, 43

no-escape closures, 88-90

NS_OPTIONS, 174-175

NSLog, 18


O

Objective-C

NS_OPTIONS, 174-175
 optional protocol requirements, 111-112

objects, protocol objects, 123

observers, 184-186

operators, 190-193
 as, 47
 conforming with, 192
 declaring, 191-192
 evaluating trade-offs, 192-193
 postfix printing operators, 23
 try, 145-146
 try!, 148
 try?, 149-151
 type casting, 46-48

option sets, 174-177
 building, 176
 building enumerations, 175-176
 viewing options, 176-177

optional chaining, 58-61
 selector testing, 59-60
 subscripts, 60-61

optional mapping, 61-62

optional patterns, 56-58

optional protocol requirements
 Objective-C, 111-112
 Swift, 112

optionals, 43-45
 ? (question mark), 44
 versus error handling, 130
 sentinel values, 53-55
 unwrapping, 45-58, 171-172
 conditional binding, 46
 failable initializers, 52-53
 forced unwrapping, 45-46
 guard statements, 50
 implicitly unwrapped optionals,
 51-52

out of date code, upgrading, 4

OutputStreamType protocol, 15
 printing to strings, 16-17
<table>
<thead>
<tr>
<th>over-chaining, 58</th>
</tr>
</thead>
<tbody>
<tr>
<td>override keyword, 186-187</td>
</tr>
<tr>
<td>parameter keyword, 37</td>
</tr>
<tr>
<td>parameter-less subscripting, 197-198</td>
</tr>
<tr>
<td>parameters</td>
</tr>
<tr>
<td>default values, 70-71</td>
</tr>
<tr>
<td>labels, 66</td>
</tr>
<tr>
<td>conventions, 66-68</td>
</tr>
<tr>
<td>let keyword, 71-72</td>
</tr>
<tr>
<td>modifying, 72</td>
</tr>
<tr>
<td>NSLog, 19</td>
</tr>
<tr>
<td>passing closures as, 97-99</td>
</tr>
<tr>
<td>print function, 12-13</td>
</tr>
<tr>
<td>terminator parameter, 14</td>
</tr>
<tr>
<td>Swift conventions, 69</td>
</tr>
<tr>
<td>type inference, 77-78</td>
</tr>
<tr>
<td>type parameters, 113-114, 118</td>
</tr>
<tr>
<td>var keyword, 71-72</td>
</tr>
<tr>
<td>variadic parameters, 83-84</td>
</tr>
<tr>
<td>partial application, 93, 95-97</td>
</tr>
<tr>
<td>passing closures as function parameters, 97-99</td>
</tr>
<tr>
<td>pattern matching, optional patterns, 56-58</td>
</tr>
<tr>
<td>Permutation Generator, 203-205</td>
</tr>
<tr>
<td>playgrounds, 7</td>
</tr>
<tr>
<td>Quick Looks, 26-27</td>
</tr>
<tr>
<td>Point struct, 22</td>
</tr>
<tr>
<td>positional arguments, 75-76</td>
</tr>
<tr>
<td>postfix printing, 23</td>
</tr>
<tr>
<td>postfixed question marks, 57</td>
</tr>
<tr>
<td>preconditions, 127-128</td>
</tr>
<tr>
<td>print function, 11-15</td>
</tr>
<tr>
<td>controlling line feeds, 14-15</td>
</tr>
<tr>
<td>fallback cascade, 22-21</td>
</tr>
<tr>
<td>item separators, 13</td>
</tr>
<tr>
<td>parameters, 12-13</td>
</tr>
<tr>
<td>terminator parameter, 14</td>
</tr>
<tr>
<td>printing, 11-15</td>
</tr>
<tr>
<td>to custom destinations, 15-18</td>
</tr>
<tr>
<td>to custom output streams, 17-18</td>
</tr>
<tr>
<td>debugPrint() function, 21-23</td>
</tr>
<tr>
<td>dump() function, 30</td>
</tr>
<tr>
<td>option sets, 177</td>
</tr>
<tr>
<td>postfix printing, 23</td>
</tr>
<tr>
<td>string interpolation, 13-14</td>
</tr>
<tr>
<td>to strings, 16-17</td>
</tr>
<tr>
<td>product types, 158</td>
</tr>
<tr>
<td>property observers, 184-186</td>
</tr>
<tr>
<td>getters/setters, 179-181</td>
</tr>
<tr>
<td>protocol extensions, 34</td>
</tr>
<tr>
<td>protocol-based collections, 121</td>
</tr>
<tr>
<td>protocols, 101-103, 118.</td>
</tr>
<tr>
<td>See also types</td>
</tr>
<tr>
<td>adopting, 104-112</td>
</tr>
<tr>
<td>alias defaults, 119</td>
</tr>
<tr>
<td>best practices, 124</td>
</tr>
<tr>
<td>building, 109</td>
</tr>
<tr>
<td>with default implementations, 110-111</td>
</tr>
<tr>
<td>Comparable, 160</td>
</tr>
<tr>
<td>CustomDebugStringConvertible, 22</td>
</tr>
<tr>
<td>CustomStringConvertible, 22, 177</td>
</tr>
<tr>
<td>declaring, 105-106</td>
</tr>
<tr>
<td>Self requirements, 122</td>
</tr>
<tr>
<td>DoubleInitializable, 109</td>
</tr>
<tr>
<td>Equatable, 101-102</td>
</tr>
<tr>
<td>ErrorType, 129-130</td>
</tr>
<tr>
<td>FloatLiteralConvertible, 109</td>
</tr>
<tr>
<td>Hashable, 114</td>
</tr>
<tr>
<td>implementation, 103</td>
</tr>
<tr>
<td>member declarations, 107-108</td>
</tr>
</tbody>
</table>
multiple inheritance, 103
mutating requirements, 108
objects, 123
optional requirements
  Objective-C, 111-112
  Swift, 112
Self requirements, 108
static keyword, 107
as types, 121-123
pyramids of doom, 48

Quick Help, 35-40
  adding images to header documentation, 40
  building header documentation, 35-36
  keywords, 37-39
    label terms, 39
  markdown support, 36-37
  method/function annotations, 40
Quick Looks, 24-29
  built-in Quick Look types, 27-28
  Custom Types, 24-25
  playground Quick Looks, 26-27
  third-party rendering, 28-29
radix initializer, 198-199
ranges, 199-201
raw value enumerations, 162-163
raw value members, 163-164
recipes
  adding context to errors, 141-142
  adding index safety to arrays, 193-194
  conditional binding from unmanaged wrappers, 63
  currying and partial application, 95
  establishing generic types, 114
  mimicking try? with printing, 151
  multi-index array subscripting, 84
  multiple-array subscripting, 194-195
  multiple-instance collection, 205
  nested functions, 79-80
  optional chaining, 59
  postfix printing, 23
  printable option sets, 177
  printing and string formats, 18-21
    conditional compilation, 20-21
    format specifiers, 19
  printing to custom destinations, 15-18
    custom output streams, 17-18
    strings, 16-17
  same-type requirements, 115-117
  scrambled collections, 204
  striding indices with a Permutation Generator, 204-205
recognizers, 185
recursive reflection, 32-33
redirecting print output, 15-18
reference semantics, 178
reference types, 155-156
referencing constructs, 68
repeat initializers, 198
REPL (read eval print loop), 6-7
  playgrounds, 7
returns keyword, 37
safe indexing, 193-194
same-type requirements, 115-117
Schönfinkel, Moses, 91
scripting, 6
selector testing, 59-60
Self requirements, 108, 122
sentinel values, 53-55
separator argument, 13
setters, 185-186
shebangs, 6
shorthand argument names, 74-76
Sinclair, Jared, 186
slices, 195-196
splitting strings, 201
statement labels, 189-190
static keyword, 107
strings
  initializing, 19
  interpolation, 13-14
  printing to, 16-17
  radix initializer, 198-199
  ranges, 199-201
  repeat initializers, 198
  splitting, 201
  subscripts, 201-202
structs, 82
  Point, 22
subscripts, 60-61, 196-198
  multi-index array subscripting, 84, 194-195
  parameter-less, 197-198
  wrapping, 195
sum types, 158
The Swift Programming Language, 8
Swift Standard Library Reference, 9
Swift version 2, 1-2
Swiftsub, 8
switches, 166-172
  branching, 166-167
  breaks, 167
  complex cases, 169
  fallthroughs, 167-168
  pattern matching, 169-170
  tuples, 169
  unwrapping optional enumerations, 171-172
  where clauses, 170-171
terminating applications
  abort() function, 128
  assertions, 126-127
  fatalError() function, 126
  preconditions, 127-128
third-party libraries, 28-29
throwing errors, 132-133
throwing functions
  error propagation, 147
  try operator, 145-146
  try! operator, 148
  try? operator, 148-149
throws keyword, 37, 134
tokens, 103-104, 117-118
trailing closures, 77
try operator, 145-146
try! operator, 148
try? operator, 148-149
  alternatives to, 149-151
tuples, 80-83, 169
  comparing with arrays, 73
  converting to structs, 82
  currying, 91-97
  callbacks, 94-95
  convolution libraries, building, 92-93
visualization libraries

cost of, 94
partial application, 93, 95-97
reasons for using, 92
labels, 80-81
reflection, 81
return types, 82-83

type aliases, 78

type casts, 46-48
type constraints, 104
typed arrays, 84
types, 155
algebraic data types, 157-158
associated types, 118
collating, 119-120
classes, 177-184
initializers, 178-184
optimization, 178
subclassing, 178
copy-and-write-back, 156-157
enumerations
associated types, 164-165
basic enumerations, 159-160
indirect values, 165-166
raw value enumerations, 162-163
extending, 186-187
generic types
building, 113-114
extending, 120-121
overrides, 186-187
parameters, 113-114, 118
reference types, 155-156
strings
radix initializer, 198-199
ranges, 199-201
repeat initializers, 198
splitting, 201
subscripts, 201-202
value types, 155-156

U

uniq function, 117
unmanaged wrappers, 62-64
unowned references, 86-87
untyped closure assignment, 76-77
unwrapping optionals, 45-58, 171-172
conditional binding, 46
type casts, 46-48
failable initializers, 52-53
forced unwrapping, 45-46
guard statements, 50
implicitly unwrapped optionals, 51-52

updating Swift code to modern syntax, 2-4
upgrading out-of-date code, 4

Using Swift with Cocoa and Objective-C, 8

V

value types, 155-156
values
capturing
capture lists, 85-86
unowned references, 86-87
embedding by type, 172-174
var keyword, 71-72, 169-170
variadic parameters, 83-84
visualization libraries, 29
websites, Apple Developer Forums, 9
where clauses, 170-171
willSet, 184-185
wrapping indices, 195

Xcode
migrating code, 2-4
Quick Look for Custom Types, 24-25