```
var clients: [Person] = [
    (name: "Alice", age: 30),
    (name: "Bob", age: 25),
    (name: "Charlie", age: 22)]

for client in clients where client.
    print("\(client.name) is \(clie
}
```

```
Alice is 30 years old
```

Apple, Xcode
Playgrounds,
Strings, Col
Sequences, T
Operators, S
Flow Control
iOS, macOS,
watchOS, tvO

Swift

The Practical Guide

Kerem Koseoglu

Rheinwerk
Computing

Contents

Dedication 19

1 Introduction 21

1.1 About this Book 21

1.2 Welcome to Swift! 23

1.3 Why Learn Swift? 24

1.4 Swift Versus Other Programming Languages 26

1.4.1 Python 27
1.4.2 Java 28
1.4.3 C# 29
1.4.4 JavaScript 29
1.4.5 Dart 30
1.4.6 Kotlin 31
1.4.7 C++ 32
1.4.8 Rust 32
1.4.9 Picking the Ideal Language 34

1.5 Setting Up the Development Environment 35

1.5.1 Preparing Your Mac 35
1.5.2 Alternative Options for Swift Development 42

1.6 Summary 43

2 Variables 45

2.1 Variables 45

2.1.1 About Variables and Constants 46
2.1.2 Declaring Simple Variables and Constants 46
2.1.3 Type Annotations 48
2.1.4 Comments 50

2.2 Boolean 52

2.2.1 About Booleans 52
2.2.2 AND Operator 53
2.2.3 OR Operator 55
2.2.4 Nested Conditions 56

2.3 Numbers 58
2.3.1 Numeric Types 58
2.3.2 Arithmetic Operators 65
2.3.3 Comparison Operators 70
2.3.4 Numeric Literals 71

2.4 Text 74
2.4.1 About Strings 74
2.4.2 Basic String Literals 77
2.4.3 Multiline String Literals 77
2.4.4 Escape Sequences 77
2.4.5 Concatenation 79
2.4.6 String Interpolation 80
2.4.7 Checking String Contents 81
2.4.8 Substrings 84
2.4.9 Modifying Strings 87
2.4.10 Comparison Operators 88
2.4.11 String/Number Conversion 90
2.4.12 Regular Expressions 95

2.5 Dates and Times 98
2.5.1 Basic Data Types 98
2.5.2 Arithmetic Operations 100
2.5.3 Comparison 103
2.5.4 Measuring Durations 105
2.5.5 String/Date Conversion 107
2.5.6 Internationalization 111

2.6 Quantities 113
2.6.1 Units 113
2.6.2 Measurement 115
2.6.3 Unit Conversions 116
2.6.4 Arithmetic Operations 117

2.7 Tuples 120
2.7.1 Basic Tuples 120
2.7.2 Named Tuples 122
2.7.3 Tuple Type Annotations 122
2.7.4 Comparison 123

2.8 Type Aliases 124
2.8.1 Basic Types 124
2.8.2 Complex Types 125

2.9 Summary 125

3 Collections 127

3.1 Arrays 128
3.1.1 Creating Arrays 128
3.1.2 Arrays as Constants 129
3.1.3 Accessing Arrays 130
3.1.4 Array Derivation 134
3.1.5 Modifying Arrays 138
3.1.6 Iterating Through Arrays 141

3.2 Sets 148
3.2.1 Creating Sets 150
3.2.2 Sets as Constants 151
3.2.3 Accessing Sets 151
3.2.4 Set Derivation 152
3.2.5 Checking for Supersets 157
3.2.6 Modifying Sets 158
3.2.7 Iterating Through Sets 160

3.3 Dictionaries 162
3.3.1 Creating Dictionaries 163
3.3.2 Dictionaries as Constants 168
3.3.3 Accessing Dictionaries 168
3.3.4 Modifying Dictionaries 170
3.3.5 Iterating Through a Dictionary 172

3.4 Summary 173

4 Control Flow 175

4.1 Conditional Statements 176
4.1.1 If-Else 176
4.1.2 Logical Operators 184
4.1.3 Ternary Operator 187
4.1.4 Nil-Coalescing Operator 189
4.1.5 Switch 192

4.2 Loops 199
4.2.1 For-In Loops 199
4.2.2 While Loops 207
4.2.3 Repeat-While Loops 209

4.3 Control Transfer Statements 211
4.3.1 Break 211
4.3.2 Continue 213
4.3.3 Fallthrough 215
4.3.4 Guard 218

4.4 Summary 221

5 Functions 223

5.1 What Is a Function? 223

5.2 Defining and Calling Functions 226
5.2.1 Defining a Function 226
5.2.2 Calling a Function from the Program 227
5.2.3 Calling a Function from Another Function 229
5.2.4 Variable Scope in Functions 231

5.3 Input Parameters 234
5.3.1 Named Parameters 234
5.3.2 Omitting Argument Labels 237
5.3.3 Default Arguments 239
5.3.4 Guarding Functions 241

5.4 Returning Values 242
5.4.1 Returning Single Values 243
5.4.2 Returning Multiple Values 246

5.5 Variadic Parameters 250

5.6 Function Overloading 253

5.7 Inout Parameters 255

5.8 Nested Functions 258

5.9 Function Types 261
5.9.1 Basic 261
5.9.2 Function Types as Parameter Types 265
5.9.3 Function Types as Return Types 267

5.10 Closures 268
5.10.1 Closure Expressions 269
5.10.2 Trailing Closures 273
5.10.3 Escaping Closures 274
5.10.4 Autoclosures 279

5.11 Generic Functions 280
5.11.1 Generic Functions 280
5.11.2 Type Constraints 283
5.11.3 Generic Where Clauses 288
5.12 Summary 289

6 Optionals 291

6.1 What Are Optionals? 291
6.1.1 Tuple Optionals 291
6.1.2 Simple Variable Optionals 293
6.1.3 Collection Optionals 295
6.1.4 Function Optionals 295
6.2 Optional Binding 296
6.3 Implicitly Unwrapped Optionals 299
6.4 Optional Chaining 300
6.5 Summary 302

7 Enumerations 303

7.1 What Are Enumerations? 303
7.2 Declaring Enumerations 305
7.2.1 Case Values 305
7.2.2 Raw Values 308
7.2.3 Associated Values 310
7.3 Computed Properties 313
7.4 Functions in Enumerations 317
7.5 Iterating over Enumerations 319
7.5.1 Iterating over Case Values 319
7.5.2 Iterating over Raw Values 320
7.5.3 Iterating over Associated Values 321
7.6 Recursive Enumerations 322
7.7 Generic Enumerations 325
7.8 Summary 327

8 Structs 329

8.1 What Is a Struct? 329

8.2 Declaring Structs 331
8.2.1 Basic Struct Declaration 331
8.2.2 Mutable Properties 332
8.2.3 Complex Properties 333
8.2.4 Nested Structs 334
8.2.5 Default Property Values 337
8.2.6 Optional Properties 337

8.3 Static Properties 339

8.4 Computed Properties 341
8.4.1 Getters 341
8.4.2 Setters 343

8.5 Functions 345
8.5.1 Initialization 345
8.5.2 Nonmutating Functions 347
8.5.3 Mutating Functions 348
8.5.4 Static Functions 349

8.6 Encapsulation 351
8.6.1 Private Properties 352
8.6.2 Private Functions 354

8.7 Structs as Function Parameters 355

8.8 Generic Structs 357

8.9 Advanced Features 359
8.9.1 Lazy Properties 359
8.9.2 Property Observers 362
8.9.3 Property Wrappers 363
8.9.4 Subscripts 365
8.9.5 Failable Initialization 367

8.10 Summary 369

9 Classes 371

9.1 Declaring Classes 372
9.1.1 Basic Class Declaration 372

9.1.2 Reference Semantics ... 373
9.1.3 Composition ... 375

9.2 Functions ... 380
9.2.1 Mutability ... 380
9.2.2 Deinitialization ... 381
9.2.3 Convenience Initialization ... 384

9.3 Inheritance ... 385
9.3.1 Superclasses and Subclasses ... 385
9.3.2 Overriding Functions ... 392
9.3.3 Overriding Initializers ... 395
9.3.4 Overriding Computed Properties ... 397
9.3.5 Overriding Subscripts ... 399
9.3.6 Casting ... 401
9.3.7 AnyObject Type ... 407

9.4 Classes as Function Parameters ... 409

9.5 Memory Management ... 412
9.5.1 Strong References ... 412
9.5.2 Weak References ... 415
9.5.3 Unowned References ... 418

9.6 Summary ... 419

10 Protocols 421

10.1 Purpose of Protocols ... 421

10.2 Function Requirements ... 425
10.2.1 Nonmutating Function ... 425
10.2.2 Mutating Function ... 428
10.2.3 Static Function ... 429

10.3 Initializer Requirements ... 431

10.4 Property Requirements ... 433
10.4.1 Get Requirement ... 433
10.4.2 Get Set Requirement ... 435
10.4.3 Static Requirement ... 436

10.5 Implementing Multiple Protocols ... 437

10.6 Checking for Protocol Conformance ... 440

10.7 Protocols as Function Parameters ... 442
10.7.1 Protocol Type ... 442

10.7.2 Existential Type ... 443
10.7.3 Opaque Type ... 444
10.8 Associated Types ... 445
10.9 Using Standard Protocols ... 448
10.9.1 Equatable ... 448
10.9.2 Comparable ... 451
10.9.3 Identifiable ... 454
10.9.4 Hashable ... 455
10.9.5 CustomStringConvertible ... 457
10.10 Summary ... 459

11 Extensions 461

11.1 Extending Enumerations ... 462
11.2 Extending Structs ... 464
11.3 Extending Classes ... 466
11.4 Extending Protocols ... 468
11.5 Operator Extensions ... 470
11.5.1 Arithmetic and Boolean Operators ... 470
11.5.2 Equality and Comparison Operators ... 473
11.5.3 Prefix and Postfix Operators ... 475
11.5.4 Custom Operators ... 476
11.6 Extending Swift Data Types ... 478
11.6.1 Sample Usage ... 478
11.6.2 Avoiding Name Conflicts ... 479
11.7 Generic Extensions ... 480
11.8 Summary ... 482

12 Error Handling 483

12.1 Understanding Errors in Swift ... 484
12.2 Throwing Errors ... 486
12.2.1 Throwing Built-in Errors ... 486
12.2.2 Throwing Custom Errors ... 488
12.2.3 Propagating Errors ... 490

12.3 Catching Errors ... 492
12.3.1 Direct Pattern Matching ... 492
12.3.2 Error Pattern Matching ... 496
12.3.3 Localized Error Messages ... 499
12.3.4 Optional Try ... 502
12.4 Cleanup with Defer ... 504
12.5 Runtime Checks ... 508
12.5.1 Debug Runtime Checks ... 508
12.5.2 Release Runtime Checks ... 512
12.6 Summary ... 513

13 File Handling 515

13.1 Text Files ... 515
13.1.1 Reading Text Files ... 516
13.1.2 Writing Text Files ... 519
13.2 Binary Files ... 522
13.2.1 Reading Binary Files ... 523
13.2.2 Writing Binary Files ... 525
13.3 Working with Common Formats ... 529
13.3.1 JSON ... 529
13.3.2 Property List ... 535
13.4 File System Operations ... 542
13.4.1 Examining Folders ... 542
13.4.2 Manipulating Folders ... 545
13.4.3 Examining Files ... 548
13.4.4 Manipulating Files ... 550
13.5 Summary ... 554

14 Concurrency 555

14.1 What Is Concurrency? ... 555
14.2 Async Functions ... 557
14.2.1 Writing Async Functions ... 557
14.2.2 Calling Async Functions ... 559
14.2.3 Running Functions in Parallel ... 565

14.3 Tasks ... 569
14.3.1 Creating Tasks ... 570
14.3.2 Task Groups ... 574
14.3.3 Task Priorities ... 581
14.3.4 Task Cancellation ... 585

14.4 Async Streams ... 588
14.4.1 Starting an Async Stream ... 588
14.4.2 Canceling an Async Stream ... 590
14.4.3 Handling Async Stream Errors ... 591

14.5 Shared State Safety ... 593
14.5.1 Task-Local Values ... 594
14.5.2 Detached Tasks ... 597
14.5.3 Actors ... 599
14.5.4 Sendable Types ... 602

14.6 Summary ... 606

15 Modules in Swift 607

15.1 Introduction to Modules ... 607
15.1.1 What Is a Module? ... 607
15.1.2 Why Use Modules? ... 610

15.2 Working with Frameworks ... 613
15.2.1 Setting Up a New Framework ... 614
15.2.2 Adding Code to a Framework ... 615
15.2.3 Building a Framework ... 616
15.2.4 Importing a Framework ... 618

15.3 Working with Packages ... 621
15.3.1 Setting Up a New Package ... 622
15.3.2 Adding Code to a Package ... 624
15.3.3 Importing a Package ... 624

15.4 Access Control ... 629
15.4.1 Open ... 629
15.4.2 Public ... 629
15.4.3 Internal ... 630
15.4.4 File Private ... 630
15.4.5 Private ... 631

15.5 Summary ... 631

16 Conclusion 633

Appendices 635

A Unit Testing 635
- A.1 Introduction to Unit Testing 635
- A.2 Unit Testing with XCTest 638
- A.3 Unit Testing with Swift Testing 646
- A.4 Summary 653

B Debugging 655
- B.1 Debugging in Xcode 655
- B.2 Advanced Debugging Tools 662
- B.3 Support Commands 670
- B.4 Summary 671

The Author 673
Index 675

Chapter 3
Collections

Collections are sets of variables glued together as logical units. This chapter will teach you about collection types in Swift.

Now that you've learned about individual variables, you can advance your Swift journey with collections. In a nutshell, *collections* are data structures that group multiple variables into a single, organized container. Nearly all programming languages use collections—including Swift.

There are three main types of collections, which are the subject of this chapter:

- *Arrays* are ordered variable lists in which each variable has an index.
- *Sets* are unordered variable lists in which each value must be unique.
- *Dictionaries* are key-value pairs in which each key must be unique.

To paraphrase a possible question many of you may have: "But wait a minute—didn't we have groups of variables called *tuples* in the previous chapter? How are collections different than that?"

That's a good question! Collections (arrays, sets, dictionaries) and tuples both hold multiple variables, but they have key differences—which are listed in Table 3.1.

Feature	Tuples	Collections
Size	Fixed size; can't be changed after creation	Can grow or shrink dynamically; you can add/remove elements
Type uniformity	Can group a mix of different types	All elements should have the same type
Element access	Access using position or named properties	Access using indexes (arrays), iteration (sets), or keys (dictionaries)
Use case	Best for grouping related variables of different types	Best for storing a flexible number of variables of the same type

Table 3.1 Tuples Versus Collections

Due to such differences, a tuple is not considered a collection type; it is merely a flexible way of grouping similar variables together. Collections offer more advanced features, which will be highlighted in this chapter.

All clear? Great! Let's start with arrays, then continue with sets and dictionaries.

Screenshots

The previous chapter offered a coding debut to Swift and Xcode. To ensure that you could all get used to Xcode and could follow the examples correctly, screenshots of Xcode outputs were supplied for most of the examples.

Now that everyone is used to how and where Xcode displays outputs, we'll generally show the output of statements as inline comments from this point on. Separate results, such as screenshots or terminal outputs, will be provided only where necessary.

3.1 Arrays

An *array* in Swift is an ordered collection of elements of the same type, allowing you to store multiple values efficiently. For instance, if you are programming a queue system and want to store the customer names ordered by their time of arrival, you could store their names as strings in an array. In this section, you will learn how to create, access, and modify arrays, and learn about convenient features offered by Swift.

3.1.1 Creating Arrays

To begin this example, let's create an array of names for waiting customers, as shown in Listing 3.1.

```
var customers = ["Alice", "Bob", "Charlie"]
```

Listing 3.1 Basic String Array

Check Table 3.2 to see a visual representation of the `customers` array. Note that indexes begin with 0 as usual.

Index	Value
0	`"Alice"`
1	`"Bob"`
2	`"Charlie"`

Table 3.2 Visual Representation of Customers Array

And there you go: It's that easy! Now the `customers` array holds three distinct string values, reflecting the names of customers in the queue. As cashiers become available, Alice would be the first customer to be called, followed by Bob and then Charlie. You'll learn how to access those values shortly.

In Chapter 2, you learned about alternative ways of declaring variables using type inference and type interpolation. Likewise, there are alternative ways of declaring arrays using the same methods. Your knowledge of variables will be applicable in that sense. Listing 3.2 showcases the alternatives.

```
// Direct value assignment with type inference
var customers1 = ["Alice", "Bob", "Charlie"]

// Direct value assignment with type annotation
var customers2: [String] = ["Alice", "Bob", "Charlie"]

// Late value assignment with type annotation
var customers3: [String]
customers3 = ["Alice", "Bob", "Charlie"]

// Alternative syntax
var customers4: Array<String>
customers4 = ["Alice", "Bob", "Charlie"]
```

Listing 3.2 Different Methods for Array Creation

3.1.2 Arrays as Constants

In Chapter 2, you learned about the `var` and `let` keywords. The `var` keyword is used to declare a *variable*, which allows value changes later (mutable), whereas `let` is used to declare a *constant*, which won't allow its initial value to change (immutable).

The same feature applies to arrays too. An array declared with `var` would be mutable, while an array declared with `let` would be immutable. It is arguably more common to have mutable arrays, but both states are possible. Listing 3.3 demonstrates both syntaxes for number arrays.

```
var someNumbers = [2, 4, 10, 6, 1, 9]
let lostNumbers = [4, 8, 15, 16, 23, 42]
```

Listing 3.3 Declaration of Mutable and Immutable Arrays

Our examples so far have featured arrays built out of literals. Naturally, you can build mutable or immutable arrays out of variables too—as shown in Listing 3.4, which builds an immutable array out of numbers. The main prerequisite is to have variables of the same type; you can't mix numbers and strings in an array.

```
let n1 = 4
let n2 = 8
let n3 = 15
```

```
let n4 = 16
let n5 = 23
let n6 = 42

let lostNumbers = [n1, n2, n3, n4, n5, n6]
```

Listing 3.4 Building Array Out of Variables

You can be even more adventurous and build arrays out of complex types as well! Listing 3.5 showcases an example, which builds a mutable array out of tuples. This code snippet also highlights the comfort of using type aliases for tuples: Type uniformity for the array is ensured easily and in a human-readable way.

```
typealias Person = (name: String, age: Int, married: Bool)

let user1: Person = (name: "John", age: 30, married: true)
let user2: Person = ("Jane", 25, false)

var people: [Person] = [user1, user2]
```

Listing 3.5 Building Array Out of Tuples

To prevent any confusion, Table 3.3 features a visual representation of the `people` array.

Index	Value
0	`(name: "John", age: 30, married: true)`
1	`(name: "Jane", age: 25, married: false)`

Table 3.3 Visual Representation of People Array

Got it? OK, then! Now, let's go over how to access values in an array.

3.1.3 Accessing Arrays

In this section, you'll learn about accessing arrays in Swift. We'll explore a handful of options in that regard: basic array functions, index-based element access, and first/last elements.

Basic Array Functions

Let's start with array functions. Table 3.4 showcases some basic functions that are used frequently.

Function	Result
`isEmpty`	`true` if the array is empty; `false` otherwise
`count`	Number of elements in the array
`contains(element)`	`true` if the element is in the array; `false` otherwise

Table 3.4 Basic Array Functions

To see those useful functions in context, check Listing 3.6, which contains the output of each function as a comment. Now that you're familiar with Swift, this code snippet should be intuitive and self-explanatory.

```
let happyCustomers: [String] = ["Alice", "Bob", "Charlie"]
let sadCustomers: [String] = []

happyCustomers.isEmpty              // false
happyCustomers.count                // 3
happyCustomers.contains("Alice")    // true
happyCustomers.contains("Ann")      // false

sadCustomers.isEmpty                // true
sadCustomers.count                  // 0
```

Listing 3.6 Basic Array Properties

Index-Based Element Access

Now that you know about basic functions, let's move forward with element access. As you know, arrays are ordered lists in which each element has an index. Therefore, it's natural to expect the core functionality of being able to access elements via their indexes. As with tuples, indexes start with 0 and increment by one for each element.

Listing 3.7 demonstrates a code snippet for element access by index, in which you extract the first and second person in a bank queue. The intuitive `bankQueue[n]` expression returns the *n*th element in the array.

```
var bankQueue = ["Alice", "Bob", "Charlie"]

let firstInLine = bankQueue[0]      // Alice
let secondInLine = bankQueue[1]     // Bob
```

Listing 3.7 Array Element Access by Index

Of course, you can use a variable as an index too! In Listing 3.8, the `myIndex` variable is used as an array index. Instead of using literal values like `0` or `1`, you use the value of `myIndex`.

```
var bankQueue = ["Alice", "Bob", "Charlie"]

var myIndex = 0                       // 0
var myCustomer = bankQueue[myIndex] // Alice

myIndex += 1                          // 1
myCustomer = bankQueue[myIndex]     // Bob
```

Listing 3.8 Using Variable as Array Index

Check the Index First

If the index value exceeds the number of elements in the array, Swift will naturally generate an error. In Listing 3.8, `bankQueue[0]` (having the value `"Alice"`) or `bankQueue[1]` (having the value `"Bob"`) or `bankQueue[2]` (having the value `"Charlie"`) is fine. However, `bankQueue[3]` would generate an error because there is no such element.

To prevent such errors, you should always ensure that the index is less than the element count. In this example, the `if myIndex < bankQueue.count` expression can be placed as a condition before the element access.

Although you will learn much more about if statements in Chapter 4, this heads-up should be a useful detail to have in advance.

First and Last

Access via indexes is cool, but sometimes you simply want to access the first or last element of an array. Swift arrays feature two shortcut functions for that—namely, `first` and `last`. These are demonstrated in Listing 3.9.

```
var bankQueue = ["Alice", "Bob", "Charlie"]
let firstInQueue = bankQueue.first!     // Alice
let lastInQueue = bankQueue.last!       // Charlie
```

Listing 3.9 Accessing First and Last Elements of Array

Beyond this basic syntax, `first` and `last` also feature a search functionality. For example, if you have a string array, then you can invoke string functions against the elements and look for matches. In Listing 3.10, you run a search in `bankQueue` to find the first and last customers whose names contain the character `l`.

```
var bankQueue = ["Alice", "Bob", "Charlie"]

let firstWithL = bankQueue.first(where: { $0.contains("l") })   // Alice
```

```
let lastWithL = bankQueue.last(where: { $0.contains("l") })     // Charlie
let firstWithX = bankQueue.first(where: { $0.contains("x") })   // nil
```

Listing 3.10 Finding First/Last Search Results in String Array

As expected, `"Alice"` is returned as the first string with `l` and `"Charlie"` is returned as the last string with `l`. When we search for string containing the character `x`, we get `nil` as the result simply because there is none.

Closures

`{ $0.contains("l") }` and similar expressions are *closures*, which are self-contained blocks of code passed as parameters. They are similar to lambda functions in other programming languages. You'll learn more about closures in Chapter 5. For now, you can accept them as common syntax elements and keep your focus on collections.

Naturally, the search functionality can be invoked for other data types too. Listing 3.11 demonstrates a code snippet in which a number search is executed using the `< 20` condition.

```
var bingoNumbers = [59, 19, 36, 55, 28]
let firstSmallNumber = bingoNumbers.first(where: {$0 < 20 })    // 19
```

Listing 3.11 Finding First Search Result in Number Array

You can get a little more adventurous and search through an array of complex types as well—such as tuples. In Listing 3.12, there is a `patientQueue` built out of tuples, in which the `name` and `age` of each patient is declared. To find the oldest and youngest patient, you can search through the array using the `age` property of the tuples.

```
typealias Person = (name: String, age: Int)

var patientQueue: [Person] = [
    (name: "John", age: 30),
    (name: "Jane", age: 15),
    (name: "Jim", age: 80),
    (name: "Jill", age: 20)]

let firstOldPatient = patientQueue.first(where: { $0.age > 65 })    // Jim
let firstYoungPatient = patientQueue.first { $0.age < 18 }          // Jane
```

Listing 3.12 Finding First Search Result in Tuple Array

On the last line of the code snippet, you can also see the shortcut version of running a search; as shown, the parentheses and `where:` prefix can be omitted if you like.

If you don't want to fetch the resulting element and only want to find the index, you can use `firstIndex` and `lastIndex` functions just like you would use *first* and *last*. Listing 3.13 demonstrates using those functions to find the indexes of the first/last elements for the given search conditions.

```
typealias Person = (name: String, age: Int)

var patientQueue: [Person] = [
    (name: "John", age: 30),
    (name: "Jane", age: 15),
    (name: "Jim", age: 80),
    (name: "Jill", age: 20)]

let firstOldIndex = patientQueue.firstIndex { $0.age > 65 }        // 2 (Jim)
let lastJIndex = patientQueue.lastIndex { $0.name.contains("J") }  // 3 (Jill)
```

Listing 3.13 Demonstration of firstIndex and lastIndex Functions

3.1.4 Array Derivation

In Chapter 2, we looked at alternative ways to extract substrings from strings, remember? Swift features similar functions to derive new subarrays from existing arrays. In this section, we'll go through some significant functions for that purpose.

Slicing

The most basic method of array derivation is to *slice* a subarray from an existing array. For example, if you have an array of seven elements, then we can extract the elements between 1 and 3 as a new array. Listing 3.14 demonstrates an example in which workdays are extracted from weekdays using indexes.

```
let weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
let workDays = Array(weekDays[1...5]) // Mon, Tue, Wed, Thu, Fri
```

Listing 3.14 Slicing Out Subarrays

In the end, `workDays` becomes an independent array, usable on its own like any other array—just like extracted substrings.

Filter

Another popular technique is *filtering*, in which you filter values of a bigger array to extract values as a smaller array. Listing 3.15 features a basic code snippet in which you filter a number array twice. In this code snippet, `$0` represents elements of the main array.

```
let numbers = [1, 2, 3, 4, 5]
let bigNumbers = numbers.filter { $0 > 3 }          // 4, 5
let evenNumbers = numbers.filter { $0 % 2 == 0 }    // 2, 4
```

Listing 3.15 Filtering Numbers

In the end, `bigNumber` contained numbers greater than 3 and `evenNumbers` contained even numbers; both are individual arrays of their own.

The same technique can be used with strings as well. In Listing 3.16, city names are filtered using their character counts, and short city names are extracted into a new array called `shortCities`. Once again, `$0` represents elements of the main array. We could have used any string function, but this example features `$0.count` to filter over string length.

```
let cities = ["New York", "San Francisco", "Los Angeles", "Chicago"]
let shortCities = cities.filter { $0.count < 10 } // New York, Chicago
```

Listing 3.16 Filtering Strings

Why not get a bit more adventurous and filter over tuples, too? The core logic of the filtering won't change—so Listing 3.17 should be pretty easy to follow!

```
typealias Person = (name: String, age: Int)

let people: [Person] = [
    ("Alice", 30),
    ("Bob", 25),
    ("Charlie", 16),
    ("David", 14)
]

let youngPeople = people.filter { $0.age < 18 } // Charlie, David
```

Listing 3.17 Filtering Tuples

Naturally, filters may contain multiple logical conditions too—as demonstrated in Listing 3.18.

```
let numbers = [231, 12, 334, 423, 25]
let smallOddNumbers = numbers.filter { $0 < 100 && $0 % 2 == 1 }    // 25
```

Listing 3.18 Filter with Multiple Conditions

Here we have two distinct conditions, bound together with the `&&` operator. Swift will process those conditions sequentially as demonstrated in Table 3.5 and produce the result shown: an array with a single element.

Condition	Meaning	Result
`$0 < 100`	Number must be less than 100	`[12, 25]`
`$0 % 2 == 1`	Number must be odd	`[25]`

Table 3.5 Multicondition Filter Process

Map

Mapping is a technique that returns a new array of the same size but with transformed elements. Listing 3.19 features a demonstration that uses the `map` function to create a new array containing `numbers` multiplied by 2.

```
let numbers = [1, 2, 3, 4, 5]
let doubleNumbers = numbers.map { $0 * 2 }  // 2, 4, 6, 8, 10
```

Listing 3.19 Mapping Numbers

As usual, `$0` represents each element in the array. The `map { $0 * 2 }` expression declares the intention to multiply each number by 2 and return the results as a new array.

As a second demonstration, Listing 3.20 features a map operation on strings. This time, `map { $0.uppercased() }` builds a new array out of `students`, with each name converted to uppercase.

```
let students = ["Alice", "Bob", "Charlie"]
let upperStudens = students.map { $0.uppercased() } // ALICE, BOB, CHARLIE
```

Listing 3.20 Mapping Strings

Reduce

Arrays feature a useful function called `reduce`, which is called when you want to run a certain calculation over array elements and return a single result.

Listing 3.21 features a basic example that calculates the sum of all elements in an array.

```
let numbers = [1, 2, 3, 4, 5]
let sum = numbers.reduce(0) { $0 + $1 } // 15
```

Listing 3.21 Summing Numbers in Array

Here, `(0)` is the initial value of the result, and `{ $0 + $1 }` means that you want to add up each number until the array is finished. Table 3.6 showcases the detailed iteration executed by the `reduce` function.

Iteration	Current Result	Operation	New Result
1	0	*+ 1*	1
2	1	*+ 2*	3
3	3	*+ 3*	6
4	6	*+ 4*	10
5	10	*+ 5*	15

Table 3.6 Number Iteration Executed by reduce Function

This useful function can be applied to any array summarization, such as concatenating strings in an array. Listing 3.22 demonstrates this approach through a clean example, followed by its detailed iteration in Table 3.7.

```
let words = ["Hello", "world", "!"]
let sentence = words.reduce("") { $0 + " " + $1 }   // Hello world !
```

Listing 3.22 Concatenating Strings in Array

Iteration	Current Result	Operation	New Result
1	`""`	*+ " " + "Hello"*	`" Hello"`
2	`" Hello"`	*+ " " + "world"*	`" Hello world"`
3	`" Hello world"`	*+ " " + "!"*	`" Hello world !"`

Table 3.7 String Iteration Executed by reduce Function

Naturally, you also can supply an initial value and use other functions within the `reduce` clause. In Listing 3.23, we debut the result with the initial value `"I say:"` and continue by applying the uppercased version of each string in `words`.

```
let words = ["Hello", "world", "!"]
let sn = words.reduce("I say:") { $0 + " " + $1.uppercased() } // I say: HELLO
WORLD !
```

Listing 3.23 Using String Functions While Reducing

Join

You also have the option of *joining* two existing arrays to build a new one. How cool is that? You can simply use the `+` operator to combine two arrays, as demonstrated in Listing 3.24. You can imagine this technique as akin to concatenating the arrays.

```
let guestsOnTime = ["Alice", "Bob"]
let guestsLate = ["David", "Eve"]
let allGuests = guestsOnTime + guestsLate // Alice, Bob, David, Eve
```

Listing 3.24 Joining Arrays Using +

The same technique can be applied to arrays with elements of other types too. Just for the fun of it, Listing 3.25 demonstrates the join operation of two arrays containing `Measurement` objects. In the end, `allWeights` will contain four `Measurement` values—which is the joint output of `someWeights` and `otherWeights`.

```
import Foundation

let weight1 = Measurement(value: 100, unit: UnitMass.kilograms)
let weight2 = Measurement(value: 1, unit: UnitMass.grams)
let someWeights = [weight1, weight2]

let weight3 = Measurement(value: 44, unit: UnitMass.kilograms)
let weight4 = Measurement(value: 12, unit: UnitMass.grams)
let otherWeights = [weight3, weight4]

let allWeights = someWeights + otherWeights // 4 elements
```

Listing 3.25 Joining Arrays with Measurement Values

This fun example concludes our content on array derivation. Now you know how to create, access, and derive arrays. The next natural step is to modify existing arrays, which will be covered in the next section.

3.1.5 Modifying Arrays

In this section, you will learn how to *modify* arrays. Operations like appending new elements or modifying or deleting existing elements will be covered.

An initial reminder, though: Modifiable arrays must have been declared using a `var` statement. As you know, `let` declarations create static/constant arrays that can't be changed later. Therefore, modification examples in this section will inevitably declare arrays using `var` statements.

Appending Elements

Swift features multiple methods to *append* elements to an array, which you'll discover next. The most basic and straightforward way is to simply invoke the `append` function of the array object. In Listing 3.26, after the initial declaration of `numbers` as `[1, 2, 3]`, we execute the `append(4)` function, extending the array as `[1, 2, 3, 4]`.

```
var numbers = [1, 2, 3]
numbers.append(4) // [1, 2, 3, 4]
```

Listing 3.26 Appending Array Element Using append Function

That's pretty easy, right? It's almost plain English! Now, what if you wanted to append multiple elements instead of just one? Listing 3.27 demonstrates how to do so. For this purpose, you can still use the `append` function—but instead of providing a single element as the parameter, you provide the `contentsOf: [4, 5]` parameter, indicating the elements you want to append.

```
var numbers = [1, 2, 3]
numbers.append(contentsOf: [4, 5]) // [1, 2, 3, 4, 5]
```

Listing 3.27 Appending Multiple Elements to Array Using append Function

As a shortcut, you can also make use of the `+=` operator, which you used with strings in Chapter 2. The same logic applies: You can concatenate a subarray to a main array, like concatenating strings. Check the demonstration in Listing 3.28.

```
var numbers = [1, 2, 3]
numbers += [4, 5] // [1, 2, 3, 4, 5]
```

Listing 3.28 Appending Multiple Elements to Array Using += Operator

The examples so far have focused on appending elements to the tail of an array. What if you want to insert elements at a specific index? For that purpose, you should use the `insert` function of the array object, as demonstrated in Listing 3.29. This function accepts two parameters: the element to insert (`"Ann"`) and the insertion index (`2`).

```
var people = ["John", "Mary", "Alice"]
people.insert("Ann", at: 2) // John, Mary, Ann, Alice
```

Listing 3.29 Inserting Element at Specific Index

You know by now that indexes begin with 0. That's why the statement `at: 2` guided Swift to insert `"Ann"` after `"Mary"`: Counting 0, 1, 2 makes `"Ann"` the third element of the array.

To insert *multiple elements* at a specific index, you should modify the parameters of the `insert` function just as you did with the `append` function. Check the intuitive demonstration in Listing 3.30, which makes use of the `contentsOf` parameter once again.

```
var people = ["John", "Mary", "Alice"]
people.insert(contentsOf: ["Ann", "Bob"], at: 2) // John, Mary, Ann, Bob, Alice
```

Listing 3.30 Inserting Multiple Elements at Specific Index

Modifying Elements

Modifying an element of an array is as simple as inserting it. Check Listing 3.31, in which you change the second element of the array from `"Banana"` to `"Blueberry"`.

```
var fruits = ["Apple", "Banana", "Cherry"]
fruits[1] = "Blueberry" // Apple, Blueberry, Cherry
```

Listing 3.31 Modification of Array Element

To get a bit more adventurous, Listing 3.32 features another example of array element modification. Although it uses the exact same approach, the array contains tuples instead of a basic data type. But if you check the very last line, you will see that the modification syntax doesn't change at all!

```
typealias Instrument = (name: String, price: Double)

var instruments: [Instrument] = []

instruments.append((name: "Guitar", price: 100))    // Guitar
instruments.append((name: "Drums", price: 300))     // Guitar, Drums
instruments.append((name: "Piano", price: 200))     // Guitar, Drums, Piano

instruments[2] = (name: "Keyboard", price: 150)     // Guitar, Drums, Keyboard
```

Listing 3.32 Modification of Array Tuple

Deleting Elements

Deleting elements from an array is equally easy. Array objects feature ready-to-use functions with the `remove` prefix, which are listed in Table 3.8.

Function	Purpose
`removeFirst(n)`	Removes the first *n* elements of the array
`removeLast(n)`	Removes the last *n* elements of the array
`remove(at: n)`	Removes the *n*th element of the array
`removeAll()`	Clears the array completely

Table 3.8 Array Functions for Element Removal

A demonstration of those functions is provided in Listing 3.33. The syntax and results are self-explanatory.

```
var fibo = [1, 2, 3, 5, 8, 13, 21, 34, 55]
fibo.removeFirst(3)          // 5, 8, 13, 21, 34, 55
fibo.removeLast(2)           // 5, 8, 13, 21
```

```
fibo.remove(at: 2)          // 5, 8, 21
fibo.removeAll()            // Empty
```

Listing 3.33 Demonstration of Element Deletion from Array

One cool trick is to add a `where` condition to `removeAll`. Listing 3.34 demonstrates how to delete elements with a value greater than 10 from the array `fibo`. The syntax of the `where` condition is the same as in previous similar examples.

```
var fibo = [1, 2, 3, 5, 8, 13, 21, 34, 55]
fibo.removeAll(where: { $0 > 10 }) // 1, 2, 3, 5, 8
```

Listing 3.34 removeAll Function with where Condition

Sorting Arrays

Finally, we will show you how to sort arrays. And once again, it's very easy: You simply invoke the `sort` function of the array object. Listing 3.35 demonstrates how to sort an array in ascending and descending order.

```
var numbers = [5, 2, 8, 3, 1]
numbers.sort()      // 1, 2, 3, 5, 8
numbers.sort(by: >) // 8, 5, 3, 2, 1
```

Listing 3.35 Sorting Array in Swift

A similar function is `sorted`, which does almost the same job as `sort`. However, instead of mutating the original array, it returns a new sorted array. Check Listing 3.36 for a demonstration.

```
var numbers = [5, 2, 8, 3, 1]
var sortedNumbers = numbers.sorted() // 1, 2, 3, 5, 8
```

Listing 3.36 Creating New Sorted Array

The next step is *array iteration*, in which you loop through the elements of an array for bulk operations.

3.1.6 Iterating Through Arrays

When you have an array at hand, it is a natural expectation to visit each element sequentially. In many cases, this is the reason to build an array in the first place. Imagine an array of phone numbers, in which you have to call each customer in line sequentially. This would require iterating through the numbers in the array, right?

For such cases, Swift offers various methods to iterate through the elements of an array. In this section, we will discuss those iteration methods and their differences.

For Clause

The most fundamental approach to array iteration is to use a `for … in …` statement. Check the demonstration in Listing 3.37, which iterates through phone numbers.

```
var phones = ["123-4567", "890-1234", "543-2109", "234-5678"]

for phone in phones {
    print(phone)
}
```

Listing 3.37 Iteration Through Phone Numbers Using for Statement

The initial part of this code snippet is familiar: The `phones` array has been declared with some mock values. The next part is the interesting one! Using the `for phone in phones` statement, you tell Swift to iterate through all elements in `phones`, assigning a new value to the `phone` variable on every iteration. The code block between `{` and `}` will be executed for each `phone` sequentially.

That was a mouthful; let's break it down now! Table 3.9 showcases each iteration, including the value assigned to `phone` and how `print(phone)` looks.

Iteration	Value in Phone	Print Statement	Output
1	`"123-4567"`	`print("123-4567")`	**123-4567**
2	`"890-1234"`	`print("890-1234")`	**890-1234**
3	`"543-2109"`	`print("543-2109")`	**543-2109**
4	`"234-5678"`	`print("234-5678")`	**234-5678**

Table 3.9 Iteration Broken Down

It should be clear now! For each iteration, `phone` was assigned a new value, pulled from `phones` sequentially. The terminal output is shown in Figure 3.1 for even more clarity.

```
1 var phones = ["123-4567", "890-1234", "543-2109", "234-5678"]
2
3 for phone in phones {
4     print(phone)
5 }
```

```
123-4567
890-1234
543-2109
234-5678
```

Figure 3.1 Output of for Iteration

Naturally, the code between { and } can be as complex as needed, but the basic idea doesn't change: The code between { and } is executed with every `phone` value through the iteration. Listing 3.38 demonstrates the same iteration with slightly more complex code, making use of features you learned about before.

```
var phones = ["123-4567", "890-1234", "543-2109", "234-5678"]

for phone in phones {
    let cleanPhone = phone.replacing("-", with: "")
    let operatorText = "You can dial \(cleanPhone) now"
    print(operatorText)
}
```

Listing 3.38 Iteration with Slightly More Complex Code

As evident in the output in Figure 3.2, the iteration ran the same way as before—even if the code between { and } was a little different.

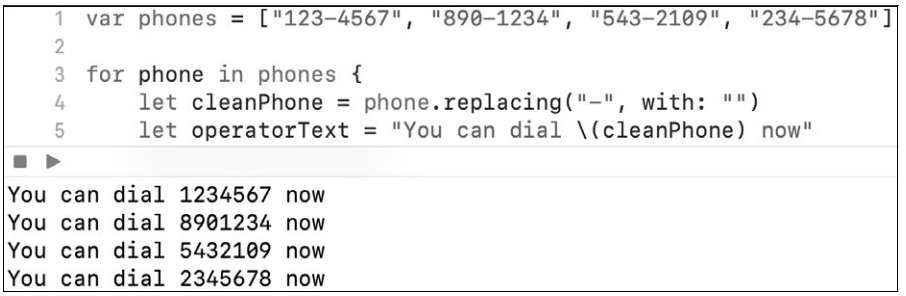


Figure 3.2 Output for Slightly More Complex Iteration

Using Enumerated

Arrays contain a cool function called `enumerated()`. When this function is used in a `for` iteration, you get access to the element and its index simultaneously. The example in Listing 3.39 invokes this functionality: Instead of executing the `for` iteration against the `bankQueue` array itself, you execute it against `bankQueue.enumerated()`. In return, you get access to the so-called `queueEntry` object, which contains the element index in `queueEntry.offset` (sequentially, `0`, `1`, `2`) and the element value in `queueEntry.element` (sequentially, `"James"`, `"John"`, `"Robert"`).

```
var bankQueue = ["James", "John", "Robert"]

for queueEntry in bankQueue.enumerated() {
    print(queueEntry.offset)
    print(queueEntry.element)
}
```

Listing 3.39 Accessing Index and Element Throughout Iteration

A breakdown of this code snippet is provided in Table 3.10, where the operation in each iteration is shown clearly.

Iteration	queueEntry.offset	queueEntry.element	Expected Output
1	0	"James"	**0** **James**
2	1	"John"	**1** **John**
3	2	"Robert"	**2** **Robert**

Table 3.10 Enumerated Iteration Broken Down

It's time to test the code and see if you get the expected result. Check the playground output in Figure 3.3: Things seem to be OK!

```
1 var bankQueue = ["James", "John", "Robert"]
2
3 for queueEntry in bankQueue.enumerated() {
4     print(queueEntry.offset)
5     print(queueEntry.element)
6 }
7
```

```
0
James
1
John
2
Robert
```

Figure 3.3 Output of Enumerated Iteration

Iteration Control Flow

It is possible to manipulate the iteration flow using keywords like `break` or `continue`, with which you might be familiar from other programming languages. This concept will be covered in Chapter 4, which is focused on control flow.

As stated before, the code between { and } can be as complex as needed. Listing 3.40 features the same iteration with slightly more complex code, in which you prepare a more intuitive cashier text for each customer in `bankQueue`.

```
var bankQueue = ["James", "John", "Robert"]

for queueEntry in bankQueue.enumerated() {
    let number = queueEntry.offset + 1
    let name = queueEntry.element
    let cashierText = "Call \(number). customer: \(name)"
    print(cashierText)
}
```

Listing 3.40 Slightly More Complex Enumerated Iteration

The output of this code snippet is shown in Figure 3.4. Once again, the core iteration didn't change at all; we merely changed the displayed output.

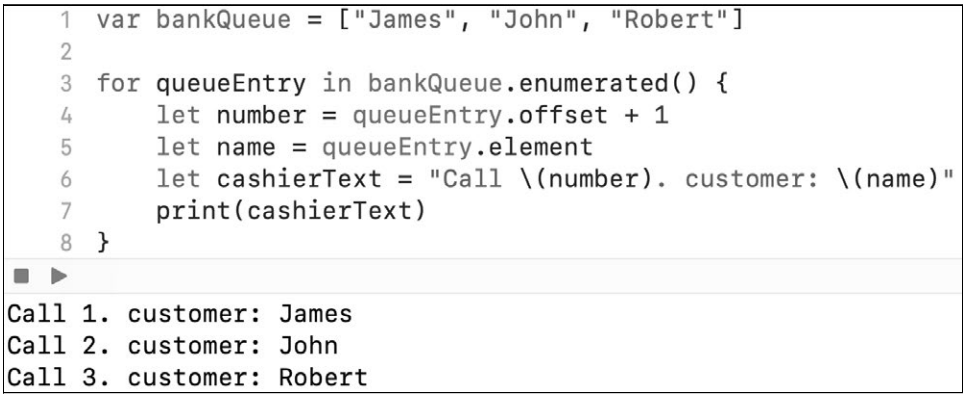


Figure 3.4 Output Containing Intuitive Cashier Text

Where Conditions

So far, we've gone through iterations in which we access all elements in an array sequentially. What if we want to access only some of them? There will be cases in which you only want to sequentially access elements matching a certain condition.

In such scenarios, you can add a `where` clause to the `for` iteration, containing the desired conditions. It works just like array filters, with a slightly different syntax.

Listing 3.41 demonstrates such an example. In this code snippet, `numbers` is a regular array. While iterating through `numbers`, you simply add the condition `where number > 100`. As a result, only values greater than `100` are processed through the iteration, and thus it only prints the values `435` and `522`.

```
let numbers = [1, 25, 38, 435, 522]

for number in numbers where number > 100 {
    print(number)    // 435, 522
}
```

Listing 3.41 Iterating with where Condition

All kinds of logical operators and parenthesis can be used in a `where` condition. Listing 3.42 demonstrates an example in which a `where` clause with two conditions is present.

```
let people = ["John", "Ann", "Alice", "Bob"]

for person in people
where person.count > 3 && person.hasPrefix("A") {
    print(person) // Alice
}
```

Listing 3.42 Complex where Condition with Logical Operators

A breakdown of those where conditions is provided in Table 3.11. In the end, Swift is only able to print the value `"Alice"`.

Initial Elements	Condition	Eliminated	Remaining
John Ann Alice Bob	person.count > 3	Ann Bob	John Alice
John Alice	person.hasPrefix("A")	John	Alice

Table 3.11 Breakdown of where Conditions

Such `where` conditions, and iterations in general, can be used with more complex data types too—like tuples. Listing 3.43 demonstrates an example in which an iteration through a tuple array is executed—including a `where` condition, too!

```
typealias Person = (name: String, age: Int)

var clients: [Person] = [
    (name: "Alice", age: 30),
    (name: "Bob", age: 25),
    (name: "Charlie", age: 22)]
```

```
for client in clients where client.age >= 25 {
    print("\(client.name) is \(client.age) years old")
}
```

Listing 3.43 Iteration Through Tuple Array

As you can see, the core syntax of the iteration didn't change at all; we simply threw in some tuples as a mental exercise. The output of this iteration is shown in Figure 3.5.

```
1  typealias Person = (name: String, age: Int)
2
3  var clients: [Person] = [
4      (name: "Alice", age: 30),
5      (name: "Bob", age: 25),
6      (name: "Charlie", age: 22)]
7
8  for client in clients where client.age >= 25 {
9      print("\(client.name) is \(client.age) years old")
10 }
■ ▶
Alice is 30 years old
Bob is 25 years old
```

Figure 3.5 Output of Tuple Iteration

Randomization

Swift empowers programmers with options for random access to array elements. Listing 3.44 features an example in which you access a random element in `fruits`. Every time you execute this code, `randomFruit` gets a random value, such as `"Apple"`, `"Cherry"`, or `"Orange"`.

```
let fruits = ["Apple", "Banana", "Cherry", "Mango", "Orange"]
let randomFruit = fruits.randomElement()
```

Listing 3.44 Picking Random Array Element

This code could be used, for example, in an app that suggests a random daily fruit to consume.

Another randomization feature lets you shuffle elements in an array. Just like shuffling cards before starting a card game, you can shuffle an array to randomly change the positions of its elements. Listing 3.45 features such an example, in which you use `shuffle()` to shuffle the names in `participants` and declare the first three as winners. It's a handy feature for a lottery app, right?

```
// List of participants
var participants = ["Alice", "Bob", "Charlie", "David", "Eve", "Frank", "Grace"]

// Shuffle the array
participants.shuffle()
```

```
// Get the first three as winners
let winners = participants.prefix(3)

// Print the winners
print("Winners:")
for (index, winner) in winners.enumerated() {
    print("\(index + 1). \(winner)")
}
```

Listing 3.45 Array Shuffle Example

Due to the nature of randomization, you will get a different output every time you execute this code. Nevertheless, Figure 3.6 demonstrates a possible output in which three random participants were declared as winners. Apparently, those names were in the first three positions in `participants` when `participants.shuffle()` was executed.

```
1  // List of participants
2  var participants = ["Alice", "Bob", "Charlie", "David", "Eve", "Frank", "Grace"]
3
4  // Shuffle the array
5  participants.shuffle()
6
7  // Get the first three as winners
8  let winners = participants.prefix(3)
9
10 // Print the winners
11 print("Winners:")
12 for (index, winner) in winners.enumerated() {
13     print("\(index + 1). \(winner)")
14 }
```

```
Winners:
1. Frank
2. Bob
3. Alice
```

Figure 3.6 One Possible Shuffle Output

That final example concluded our content on arrays, one of three major collection types, along with sets and dictionaries. We will continue the journey with sets, which are similar to arrays but bring the uniqueness constraint to the table. Take a break, get some fresh air if you need to, and see you there!

3.2 Sets

In Swift, a *set* is a collection, like an array; they share some common ground. The core difference is that a set is an unordered collection of unique values. Unlike arrays, sets don't allow duplicate elements and don't guarantee a specific order by index—but they promise a very high access speed, which is a significant benefit in large datasets.

Table 3.12 describes the differences between arrays and sets clearly.

Feature	Array	Set
Order	Ordered	Unordered
Duplicate values	Allowed	Not allowed
Access by index	Allowed	Not allowed
Element lookup speed	Slow	Fast

Table 3.12 Comparison Between Arrays and Sets

The selling points of a set are basically element uniqueness and access speed. Here are some general suggestions for working with a set:

- Use an array when you need ordered elements with possible duplicates.
- Use a set when you need fast lookup with unique elements.

To strengthen your understanding, Table 3.13 contrasts some use cases in which either an array or set would be preferred as an element container.

Case	Preference	Reason
Shopping cart	Array	Duplicate items should be allowed.
Queue	Array	Order matters and a queue is accessed in entry order.
Member list	Set	Members are unique and access by user name would be fast.
Product categories	Set	Categories are unique and order doesn't matter.

Table 3.13 Use Cases for Arrays and Sets

As you can see, arrays and sets can't fully replace each other. You could attempt to use an array instead of a set, but it won't ensure uniqueness and access speed would be unnecessarily slow. Likewise, you could attempt to use a set instead of an array, but your app will fail on duplicate elements and won't follow the given element order.

Why Are Sets Faster?

Sets are faster than arrays for lookups because they use hashing, while arrays use linear searches.

In more common terms, when checking if an element exists in an array, Swift must scan each item one by one until it finds a correspondence between elements. The runtime cost may be negligible in small datasets, but it would make a significant difference in large datasets.

Sets, meanwhile, use a hash table to help spot elements, which allows for instant lookup. The runtime cost is constant regardless of the set size. When an element is added to a set, Swift (secretly) computes a hash value, which is used as in index. When checking for an element, Swift directly jumps to the hashed index instead of scanning every item.

Now that you have a general notion of sets, we can move forward to hands-on examples to introduce the corresponding syntax and further features.

3.2.1 Creating Sets

Set declarations can be made using a familiar syntax, like that for arrays. No big surprise, right? Listing 3.46 showcases alternative methods for set declarations. Here, `set1`, `set2`, and `set3` will end up having the exact same elements.

```
// Explicit type declaration
var set1 = Set<Int>()
set1.insert(1)
set1.insert(2)
set1.insert(3)

// Explicit type declaration with values
var set2: Set<Int> = [1, 2, 3]

// Type inference
var set3 = Set([1, 2, 3])
```

Listing 3.46 Alternative Methods of Set Declaration

In the first part, you explicitly declare the type of `set1` as `Set<Int>()` and insert elements afterward. In the second part, you declare the type of `set2`—but insert the initial elements immediately. Finally, `set3` is declared using type inference, which means that you let Swift "guess" the types of elements on your behalf.

Insert Versus Append

You might have caught a syntax difference here. In arrays, new elements are added via the `array.append` function. In sets, new elements are added via the `set.insert` function. Although their purposes are similar, the difference is worth noting.

Note that `array.append` will add the new element to the tail of the array because an array is an ordered collection. However, `set.insert` will do the hash calculations and add the new element "somewhere"; no particular order is guaranteed.

You can create a set from an array as well! The catch is that you will lose duplicate entries in the process. Whether that's desirable or not depends on the use case. Listing 3.47 features such a conversion.

```
var allStudents = ["Alice", "Bob", "Charlie", "Bob", "Alice"]
var uniqueStudents = Set(allStudents) // Alice, Charlie, Bob
```

Listing 3.47 Array to Set Conversion

3.2.2 Sets as Constants

As with any other data type, sets can be declared as mutable via `var` or immutable (read-only) via `let`. To keep this section self-contained, Listing 3.48 demonstrates both declaration types.

```
var changeableSet = Set([1, 2, 3, 4, 5])
let readOnlySet = Set([1, 2, 3, 4, 5])
```

Listing 3.48 Set Declaration as Constant

In this example, you can modify the contents of `changeableSet` in due course—because it was declared to be mutable using `var`. On the other hand, `readOnlySet` can't be modified later—because it was declared to be immutable using `let`.

3.2.3 Accessing Sets

In this section, you will learn how to access set elements. We are going to cover basic set functions and how to access the first element of a set.

Basic Set Functions

First things first: Basic array functions, which were covered in Section 3.1.3, are available with sets too! To keep this section self-contained, Table 3.14 showcases those functions.

Function	Result
`isEmpty`	`true` if the set is empty; `false` otherwise
`count`	Number of elements in the set
`contains(element)`	`true` if the element is in the set; `false` otherwise

Table 3.14 Basic Set Functions

To see those functions in context, look at Listing 3.49; the functionality is quite intuitive.

```
var animals: Set<String> = ["dog", "cat", "bird", "elephant"]

animals.isEmpty             // false
animals.count               // 4
animals.contains( "bird" )  // true
animals.contains( "snake" ) // false
```

Listing 3.49 Demonstration of Basic Set Functions

A cool feature is to use a `where` clause within the `contains` function. Because you're already familiar with `where` clauses in Swift, the syntax of the demonstration in Listing 3.50 should be intuitive.

```
var animals: Set<String> = ["dog", "cat", "bird", "elephant"]
animals.contains(where: { $0.count > 5 })       // true due to "elephant"
animals.contains(where: { $0.contains("x") })   // false
```

Listing 3.50 Where Clause Within contains Function

First Function

Because sets are unordered collection types, they don't support index-based access. A supported complementary function is `first`, which returns the "first" element in the set. Be careful though: Because sets are not ordered like arrays, Swift doesn't guarantee that `first` returns the initial inserted element; instead, it simply returns an element based on the internal order of the set, which could be any of them.

Having that said, Listing 3.51 demonstrates the usage of the `first` function. Go ahead and try it: Every time you execute this code snippet, you should get a different element because the calculated hash values will change on each execution—and therefore, so does the "first" element behind the scenes.

```
var numbers: Set = [10, 20, 30, 40, 50]
numbers.first! // Output: Could be any of 10, 20, 30, 40, 50
```

Listing 3.51 Demonstration of first Function for Sets

3.2.4 Set Derivation

Just like arrays, sets feature functions through which you can derive new sets. They are mostly similar to those for array derivation—with the obvious deviation that index-based access would not make sense. In this section, we will walk through the available derivation functions for sets.

Filter

Set filters share the same logic and syntax as array filters. The `filter` function allows you to create a subset of a set based on a condition. Listing 3.52 showcases two examples, in which you extract even numbers and fruits starting with the letter *A*.

```
let numbers: Set = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let evenNumbers = numbers.filter { $0 % 2 == 0 }
print(evenNumbers) // 2, 4, 6, 8, 10; order varies

let fruits: Set = ["Apple", "Banana", "Cherry", "Avocado", "Blueberry"]
let aFruits = fruits.filter { $0.hasPrefix("A") }
print(aFruits) // Apple, Avocado; order varies
```

Listing 3.52 Set Filtering Demonstration

Although this code will work just fine, you can't ensure the element order in `evenNumbers` and `aFruits`; Swift will order them as it pleases, based on hash values.

Map

Like it is in arrays, the `map` function for sets in Swift is used to transform each element in the set into a new value. The output is a new set containing those new values. Listing 3.53 features two mapping examples.

```
let numbers: Set = [1, 2, 3, 4, 5]
let squaredNumbers = numbers.map { $0 * $0 }
print(squaredNumbers) // Output: [1, 4, 9, 16, 25] (order varies)

let fruits: Set = ["Apple", "Banana"]
let uppercasedFruits = fruits.map { $0.uppercased() }
print(uppercasedFruits) // Output: ["APPLE", "BANANA"] (order varies)
```

Listing 3.53 Set Mapping Demonstration

In the first example, you square the elements in `numbers` and collect them into a new set called `squaredNumbers`. In the second example, you convert the elements in `fruits` to uppercase and collect them into a new set called `uppercasedFruits`. As usual, element order is undeterminable in either sample.

Reduce

Like in arrays, the `reduce` function for sets in Swift allows you to combine all elements in a set into a single value. The detailed breakdown for this function was provided in Section 3.1.4 already, so you can jump directly into a couple of examples provided in Listing 3.54.

```
let numbers: Set = [10, 20, 30, 40, 50]
let sum = numbers.reduce(0) { $0 + $1 }
print(sum)  // Output: 150

let words: Set = ["Follow", "the", "white", "rabbit"]
let sentence = words.reduce("") { $0 + " " + $1 }
print(sentence)  // Output: " the Follow rabbit white" (order varies)
```

Listing 3.54 Set reduce Demonstration

In the first part, you have a set of numbers. By invoking `numbers.reduce(0) { $0 + $1 }`, you tell Swift to sum those numbers, the result of which is `150`. Because the element order doesn't affect the result of an addition operation, this part works just like it does for arrays.

In the second part, though, we have a different story. The `words` set contains a collection of unique strings. The `words.reduce("") { $0 + " " + $1 }` expression should return a concatenation of all strings in the set, right?

It kind of does—but with a twist: Because `words` didn't index the strings in the order they were provided, the `sentence` output will seemingly have shuffled the order of the strings—which is the expected behavior for sets! If you're aiming for control over the order of elements, you should use an array instead.

Union

You already know that it's possible to merge existing arrays; this topic was covered in Section 3.1.4. Likewise, it's possible to merge existing sets, combining all their elements—as shown semantically in Figure 3.7.

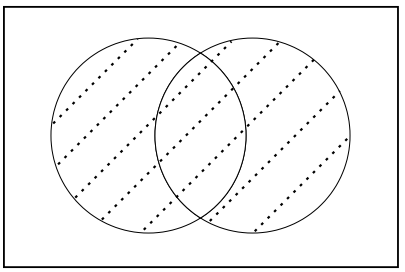


Figure 3.7 Scope of Set Union

You can either build a joint set out of two sets or insert rows of a set into another one to extend it. In this section, we will go over both options.

Listing 3.55 demonstrates the usage of the `union` function to merge two sets and create a new one. When `setA.union(setB)` is executed, Swift will merge all elements of `setA` and

setB into the target mergedSet variable. Meanwhile, the original setA and setB sets are not mutated.

```
let setA: Set = [1, 2, 3]
let setB: Set = [3, 4, 5]

let mergedSet = setA.union(setB)
print(mergedSet) // 1, 2, 3, 4, 5 (order varies)
```

Listing 3.55 Joining Sets Using union Function

Note that 3 is a common element in setA and setB. Due to the uniqueness requirement of sets, common elements like 3 are not duplicated in the target set.

What if you had three sets to merge instead of two? That's easy! You can simply chain the union function as shown in Listing 3.56. Because union returns a new set anyway, you can keep chain-executing this function for all sets you need to merge.

```
let setA: Set = [1, 2]
let setB: Set = [2, 3]
let setC: Set = [3, 4]

let mergedSet = setA.union(setB).union(setC)
print(mergedSet) // 1, 2, 3, 4 (order varies)
```

Listing 3.56 Chain-Executing Set Unions

The examples so far focused on producing a new set out of existing sets. But you can also merge a set into another set, extending the target set with new elements. The way to do so is to invoke the formUnion function of the target set. In the demonstration in Listing 3.57, setA was extended with new elements from setB. Naturally, setB is not affected or mutated by this operation.

```
var setA: Set = [10, 20, 30]
let setB: Set = [30, 40, 50]

setA.formUnion(setB)
print(setA) // 10, 20, 30, 40, 50 (order varies)
```

Listing 3.57 Merging Set into Another Set Using formUnion

Intersection

In this section, our purpose is to find common elements of two sets. The semantics of this operation is shown in Figure 3.8.

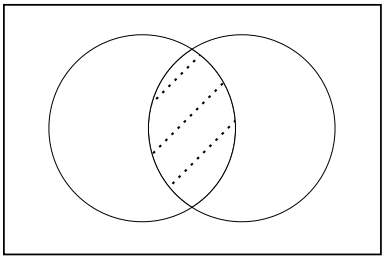


Figure 3.8 Scope of Set Intersection

For this purpose, you'll invoke the `intersection` function of either set. Listing 3.58 demonstrates how to do so; note the syntactic similarity to the former `union` function.

```
let setA: Set = [1, 2, 3, 4, 5]
let setB: Set = [3, 4, 5, 6, 7]

let commonElements = setA.intersection(setB)
print(commonElements) // 3, 4, 5 (order varies)
```

Listing 3.58 Finding Common Elements of Sets Using intersection Function

The `intersection` function doesn't mutate any of the sets; it simply creates a new set out of their common elements. An alternative is to invoke the `formIntersection` function, which will mutate the target set and reduce its elements to common elements of the two sets.

Listing 3.59 contains a demonstration of this function. After executing `setA.formIntersection(setB)`, `setA` will contain only elements in common with `setB`, replacing its former contents. `setB` is not mutated.

```
var setA: Set = [10, 20, 30, 40, 50]
let setB: Set = [30, 40, 50, 60, 70]

setA.formIntersection(setB)
print(setA) // 30, 40, 50 (order varies)
```

Listing 3.59 Demonstration of formIntersection Function

Subtracting

Now that you know about spotting the common elements of two sets, it's time to learn how to spot the differing elements of two sets! Figure 3.9 showcases the semantics of this operation.

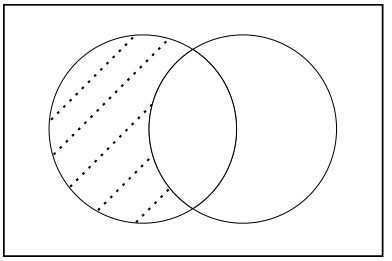


Figure 3.9 Scope of Set Subtraction

For this purpose, you can invoke the `subtracting` function (Listing 3.60), which will build a new set from the differing elements of the given sets.

```
let setA: Set = [1, 2, 3, 4, 5]
let setB: Set = [3, 4, 5, 6, 7]

let diffSet = setA.subtracting(setB)
print(diffSet) // Output: 1, 2 (order varies)
```

Listing 3.60 Finding Differing Elements of Sets Using subtracting Function

As with `union` and `intersection`, you can also mutate the original set with the differing values if you want. For that purpose, you can invoke the `subtract` function. Check Listing 3.61 for a demonstration: The execution of `setA.subtract(setB)` will remove elements from `setA`—but not the ones in common with `setB`. Meanwhile, `setB` is not mutated.

```
var setA: Set = [1, 2, 3, 4, 5]
let setB: Set = [3, 4, 5, 6, 7]

setA.subtract(setB)
print(setA) // Output: 1, 2 (order varies)
```

Listing 3.61 Demonstration of subtract Function

This example concludes our content on set derivation. We went through various functions that will help you produce new sets out of existing ones. Our next topic is a natural follow-up: How to check if sets are supersets or subsets of each other.

3.2.5 Checking for Supersets

For this topic, it might be a good idea to understand the terms *superset* and *subset* before jumping to the playground. Figure 3.10 illustrates a superset and subset.

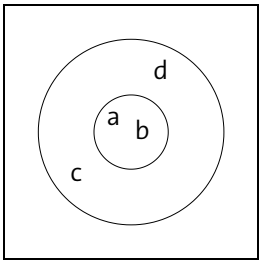


Figure 3.10 Superset and Subset Diagram

In this diagram, the inner circle is a subset and the outer circle is a superset—which contains the entire subset, plus more. The (inner) subset contains the elements *a* and *b*. The (outer) superset automatically contains the subset elements *a* and *b*, as well as the additional elements *c* and *d*. Table 3.15 showcases the element lists in pseudocode format.

Set	Elements
Subset	`["a", "b"]`
Superset	`["a", "b", "c", "d"]`

Table 3.15 Subset and Superset Elements

So far, so good! Moving forward to Swift, you can easily check whether a set is the superset of another set. Naturally, you can also check if a set is the subset of another one.

Listing 3.62 contains a demonstration of such checks. The `isSuperset` function is invoked to find if a set is a superset of another set, and `isSubset` is invoked to find out if a set is a subset of another set.

```
let set1: Set = ["a", "b"]
let set2: Set = ["a", "b", "c", "d"]
let set3: Set = ["e", "f"]

set1.isSubset(of: set2)     // true
set2.isSuperset(of: set1)   // true
set3.isSubset(of: set2)     // false
set2.isSuperset(of: set3)   // false
```

Listing 3.62 Checking for Superset and Subset Relations

3.2.6 Modifying Sets

Now that you know about creating, accessing, and deriving sets, it's time to learn about set mutation. In this section, you will learn about modifying sets and changing their contents.

Sets Aren't Indexed

Because sets don't have indexes, set-modification functions will deviate from their array counterparts. Although there is a reasonable overlap, index-based functions are naturally not available for sets.

Inserting Elements

In Section 3.2.1, we went through an example that created a set by using `insert` to insert its initial elements. However, you can invoke the `insert` function to insert elements later too! Listing 3.63 demonstrates this technique by adding new animals to the `animals` set.

```
var animals: Set = ["cat", "dog", "bird"]
animals.insert("fish")
animals.insert("snake")
print(animals) // cat, dog, bird, fish, snake (order varies)
```

Listing 3.63 Inserting Elements into Set

As you know by now, a feature of sets is their promise to contain unique values. Even if you attempt to insert duplicates into a set, as in Listing 3.64, Swift will ignore the duplicate insertion and preserve the uniqueness of the set's elements.

```
var animals: Set = ["cat", "dog", "bird"]
animals.insert("cat")
animals.insert("dog")
print(animals) // cat, dog, bird (order varies)
```

Listing 3.64 Inserting Duplicates into Set

Deleting Elements

Deleting elements from a set is a straightforward operation. Set objects feature ready-to-use functions via the `remove` prefix; these are listed in Table 3.16.

Function	Purpose
`remove(element)`	Removes the element from the set
`removeFirst()`	Removes the first element from the set (random)
`removeAll()`	Clears the set completely

Table 3.16 Set Functions for Element Removal

A demonstration of those functions is given in Listing 3.65. After the initial set definition, `numbers.remove(2)` is executed, which spots and removes the given element. In the

second part, `numbers.removeFirst()` is executed. Because you can't be sure of the element order, this statement removes a random element from the set. Finally, `numbers.removeAll()` clears the entire set.

```
var numbers: Set = [1, 2, 3, 4, 5]

numbers.remove(2)       // Removes specific element
print(numbers)          // 1, 3, 4, 5 (order varies)

numbers.removeFirst()   // Removes an arbitrary element
print(numbers)          // Remaining three elements, varies

numbers.removeAll()     // Clears the set
print(numbers)          // (empty)
```

Listing 3.65 Demonstration of Element-Removal Functions

Set Element Modification

Because set elements are uniquely hashed, you can't modify an element in a set like an array. As a workaround, you can emulate element modification by deleting the old value and inserting the new value.

3.2.7 Iterating Through Sets

Although sets are typically preferred for single-element access, it is possible to iterate through set elements too. In fact, the functions to iterate through sets are nearly identical to their array counterparts. In this section, you will discover those functions, as you did for arrays.

As a reminder: Keep in mind that sets don't guarantee an element order, so when you iterate through a set, the access order will be virtually random.

For Clause

The most fundamental approach to set iteration is to use a `for … in …` statement. Check the demonstration in Listing 3.66, which iterates through phone numbers.

```
var phones: Set = ["123-4567", "890-1234", "543-2109", "234-5678"]

for phone in phones {
    print(phone) // Prints each phone, order varies
}
```

Listing 3.66 Iterating Through Set Using for Clause

The logic of `for` clauses is identical in arrays and sets. To prevent content duplication, we won't dive into those details again here.

Using Enumerated

Likewise, the use of the `enumerated` function is also identical to its use in arrays. Check the demonstration in Listing 3.67, which iterates through each element of the set.

```
var customers: Set = ["James", "John", "Robert"]

for customer in customers.enumerated() {
    print(customer.offset)
    print(customer.element)
}
```

Listing 3.67 Iterating Through Set Using Enumerated

The output of this code snippet is shown in Figure 3.11, but there are a couple of things to keep in mind:

- Every time you execute this code, you will get a different order because values are hashed, not ordered.
- `customer.offset` should merely be seen as an index, starting from 0 and increasing with each element. It does not symbolize the (nonexistent) "element order" in the set.

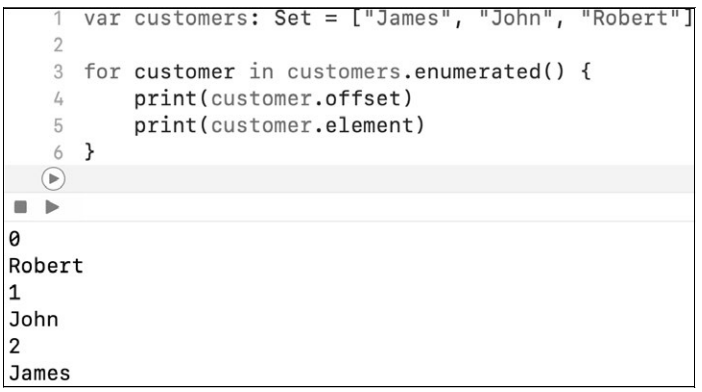


Figure 3.11 Set Enumeration Demonstration Output

Despite those limitations, the ability to browse through set elements is a neat feature to have!

Where Conditions

Just like arrays, sets can be partially iterated with the help of `where` conditions. Even the syntax is the same! Nevertheless, Listing 3.68 demonstrates an iteration example in which you iterate only through numbers bigger than `100`.

```
let numbers: Set = [1, 25, 38, 435, 522]

for number in numbers where number > 100 {
    print(number)    // 435, 522; order varies
}
```

Listing 3.68 Set Iteration Supported by where Condition

You can check Section 3.1.6 for additional `where` examples, ones used in our discussion of arrays.

Randomization

Finally, you will learn about fetching random elements out of a set—which is, once again, the same as for arrays. Listing 3.69 demonstrates the usage of `randomElement` in a set.

```
let fruits: Set = ["Apple", "Banana", "Cherry", "Mango", "Orange"]
let randomFruit = fruits.randomElement()
```

Listing 3.69 Picking Random Element from Set

> **No Shuffle for Sets**
>
> Because sets are unordered collections with a virtually random element order, it doesn't make sense to shuffle a set. That's why Swift sets don't have a `shuffle` function like arrays.

This section has concluded our content on sets. It was probably easy to follow through as sets share a lot of common ground with arrays. By reading about sets, your knowledge of arrays was also solidified. Now we'll move forward to a new collection type, one that's a little different than arrays and sets.

3.3 Dictionaries

The final collection type, dictionaries, is a little different than arrays and sets. Arrays and sets are lists of elements of the same type. Their selling point is their ability to contain multiple elements. A dictionary, meanwhile, is a flat structure containing key-value pairs.

The difference is highlighted in Figure 3.12. On the left side, `userNames` is an array holding a list of strings, reflecting the names of users in a system. On the right side, `currentUser` is a dictionary containing details about the current user.

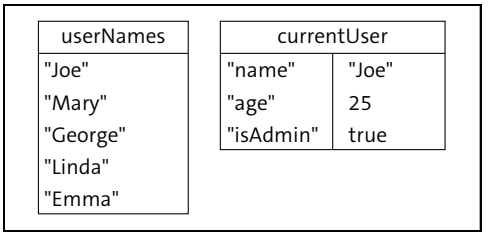


Figure 3.12 Array and Dictionary, Side by Side

In this example, you can imagine the array as a list of elements (user names) and the dictionary as zoomed in on an element (like a user) to show all its details. That's the usual selling point of a dictionary, anyway.

But wait: Didn't we use named tuples for that purpose in Chapter 2? How are dictionaries different? Good point! Their main differences are contrasted in Table 3.17.

Feature	Tuple	Dictionary
Structure	Fixed set of values	Key-value pairs
Access	By index or name	By key
Size	Fixed; set at declaration	Dynamic; can grow or shrink
Mutability	Can't add/remove elements	Can add/remove key-value pairs
Best use case	Grouping related values	Flexible key-value storage

Table 3.17 Tuples Versus Dictionaries

Despite those differences, tuples and dictionaries do have some overlapping functionality. If you are aiming for flexibility, though, dictionaries are the way to go because you can add new key-value pairs as needed.

As we go over some hands-on coding examples, you'll get a better idea of what makes dictionaries unique. Without further ado, let's begin!

3.3.1 Creating Dictionaries

In Chapter 2, you learned about alternative ways of declaring variables using type inference and type interpolation. Likewise, there are alternative ways of declaring dictionaries using the same methods, as we'll discuss in this section.

Dictionaries with a Single Type

Let's start with the basic example in Listing 3.70, featuring type inference, as a relaxed warm-up.

```
var myCar = [
    "make": "Nissan",
    "model": "Qashqai",
    "color": "Black",
    "bodyType": "SUV"
]

print(myCar["make"]!)    // Nissan
print(myCar["color"]!)   // Black
```

Listing 3.70 Dictionary Declaration Using Type Inference

In this example, `myCar` is a dictionary. It contains various properties as key-value pairs. If you had to list those properties in a table, it would look like Table 3.18.

Key	Value
`make`	`"Nissan"`
`model`	`"Qashqai"`
`color`	`"Black"`
`bodyType`	`"SUV"`

Table 3.18 Key-Value Pairs of myCar

That's clear, right? You can add as many properties as necessary. All those key-value pairs are logically properties of `myCar`.

The last part of Listing 3.70 demonstrates the basic way to access elements in a dictionary: `myCar["make"]` would return `"Nissan"`, while `myCar["color"]` would return `"Black"`.

Now that you have seen the basic approach to dictionary declaration, we can move forward to further methods. Listing 3.71 showcases examples of alternative syntax that serve the same purpose. You can pick any alternative that suits your needs.

```
// Type annotation
var myCar: [String: String] = [:]
myCar["make"] = "Nissan"
myCar["model"] = "Qashqai"

// Type annotation - alternative syntax
var herCar = Dictionary<String, String>()
herCar["make"] = "Hyundai"
herCar["model"] = "Accent"
```

```
// Type annotation with initial values
var hisCar: [String: String] = [
    "make": "Toyota",
    "model": "Corolla"
]
```

Listing 3.71 Alternative Dictionary Declaration Methods

Dictionaries with Multiple Types

So far, we have declared dictionaries with a single type, meaning that all values were strings. More often than not, though, a dictionary needs to contain values of various types. Revisiting the introduction to this section, Figure 3.13 features such an example.

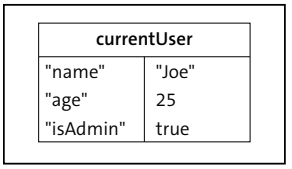


Figure 3.13 Dictionary with Multiple Types

If you look closely, you'll see that `currentUser` has keys with different data types, which are listed in Table 3.19.

Key	Data Type
`name`	String
`age`	Integer
`isAdmin`	Boolean

Table 3.19 Data Types of currentUser Keys

To declare such dictionaries with flexible/multiple data types, you need to use the `Any` keyword. An implementation of this is provided in Listing 3.72. Note that we have provided `Any` as the value data type here, indicating that Swift should behave in a flexible manner and accept any provided value type.

```
var currentUser: [String: Any] = [
    "name": "Joe",
    "age": 25,
    "isAdmin": true
]

print(currentUser["name"]!)     // Joe
```

```
print(currentUser["age"]!)      // 25
print(currentUser["isAdmin"]!)  // true
```

Listing 3.72 Declaration of Dictionary with Flexible/Multiple Data Types

Multidimensional Dictionaries

So far, we've covered *flat dictionaries*, in which each key corresponds to a single value. However, Swift supports *multidimensional dictionaries* too! You can declare a *nested dictionary*, in which an element is a collection instead of a simple variable.

Listing 3.73 demonstrates how to declare an array as a dictionary element. `bassGuitar` has an element called `availableColors`, which is a string array.

```
var bassGuitar: [String: Any] = [
    "brand": "Fender",
    "model": "Precision",
    "strings": 5,
    "availableColors": ["Black", "Sage Green", "Maple"]
]
```

Listing 3.73 Declaring a Dictionary Containing an Array

Likewise, a dictionary may contain a set as well—as demonstrated in Listing 3.74.

```
var bassGuitar: [String: Any] = [
    "brand": "Fender",
    "model": "Precision",
    "strings": 5,
    "availableColors": Set(["Black", "Sage Green", "Maple"])
]
```

Listing 3.74 Declaring a Dictionary Containing a Set

You can even include a dictionary inside another dictionary, making it a nested dictionary. In Listing 3.75, `specs` is a subdictionary of `bassGuitar`, containing key-value pairs of its own.

```
var bassGuitar: [String: Any] = [
    "brand": "Fender",
    "model": "Precision",
    "strings": 5,
    "specs": [
        "bodyWood": "Alder",
        "neckWood": "Maple",
        "fingerWood": "Rosewood",
        "quarterSawn": true,
```

```
        "scaleLength": 34,
        "frets": 21
    ]
]
```

Listing 3.75 Nested Dictionary Demonstration

As a mental exercise, Listing 3.76 demonstrates a complex dictionary, containing both a subset and a subdictionary as elements.

```
var bassGuitar: [String: Any] = [
    "brand": "Fender",
    "model": "Precision",
    "strings": 5,
    "availableColors": Set(["Black", "Sage Green", "Maple"]),
    "specs": [
        "bodyWood": "Alder",
        "neckWood": "Maple",
        "fingerWood": "Rosewood",
        "quarterSawn": true,
        "scaleLength": 34,
        "frets": 21
    ]
]
```

Listing 3.76 Complex Nested Dictionary Example

JSON Similarity

Readers with JSON experience might have noticed that complex dictionaries start to look like JSON files. That's correct—and you can use that similarity as a mental hook to understand dictionaries a little better.

Zipping Dictionaries

A cool trick in dictionary creation is to use the `zip` keyword. In Listing 3.77, the keys are in the `keys` array and the values in the `values` array.

```
let keys = ["brand", "model", "strings"]
let values = ["Fender", "Precision", "5"]

var bassGuitar = Dictionary(uniqueKeysWithValues: zip(keys, values))
print(bassGuitar) // ["brand": "Fender", "model": "Precision", "strings": "5"]
```

Listing 3.77 Dictionary Creation Using Zip

To build the bassGuitar dictionary, you "zip" keys with values: The first key (brand) gets the first value (Fender); the second key (model) gets the second value (Precision); and so on.

3.3.2 Dictionaries as Constants

As with any other data type, dictionaries can be declared as mutable via var or immutable (read-only) via let. To keep this section self-contained, Listing 3.78 demonstrates a mutable and an immutable dictionary declaration.

```
// Mutable
var currentUser: [String: Any] = [
    "name": "Joe",
    "age": 25,
    "isAdmin": true
]

// Immutable
let previousUser: [String: Any] = [
    "name": "Mary",
    "age": 33,
    "isAdmin": false
]
```

Listing 3.78 Mutable and Immutable Dictionary Declarations

3.3.3 Accessing Dictionaries

Now that you know how to create dictionaries, it's time to access them. After all, why create a dictionary you will never read, right? We'll walk through various methods of accessing dictionaries in this section.

Direct Access Using a Key

Because dictionaries are basically key-value pairs, the first natural expectation would be to give the key and receive the corresponding value, right? Listing 3.79 demonstrates the basic syntax for this. The bassGuitar["brand"] expression returns the value of "brand" within the dictionary—and the same approach applies to "model".

```
var bassGuitar: [String: Any] = [
    "brand": "Fender",
    "model": "Precision",
    "strings": 5,
    "availableColors": ["Black", "Sage Green", "Maple"]
]
```

```
print(bassGuitar["brand"]!)     // Fender
print(bassGuitar["model"]!)     // Precision
```

Listing 3.79 Accessing Simple Dictionary Element

If you query a nonexisting key, Swift will simply return `nil`, as demonstrated in Listing 3.80.

```
var bassGuitar: [String: Any] = [
    "brand": "Fender",
    "model": "Precision",
    "strings": 5,
    "availableColors": ["Black", "Sage Green", "Maple"]
]

bassGuitar["price"] // nil
```

Listing 3.80 Accessing Nonexisting Dictionary Element

Optionals in Dictionary Access

Because it aims to be a type-safe language, Swift is strict about being absolutely sure about the type of any variable. When a dictionary is defined as `[String: Any]` as in Listing 3.80, this means that you can assign any type of value for a dictionary key. That's a potential area for safety issues because you could mistakenly unwrap an existing element to an incorrect variable type, like assigning the `bassGuitar["availableColors"]` array to an integer variable.

Or you could basically access a nonexisting element, as in `bassGuitar["price"]`, which may potentially lead to an app crash.

Due to such concerns, Swift always assumes that accessing a dictionary element "optionally" returns a value of the desired type. The dictionary may contain another type of data, or it may not contain such a key at all.

In Chapter 6, you will learn about optionals in Swift. In that chapter, once you understand the core logic of optionals, we will go over dictionary-specific examples of optional element access. But for now, we'll stick to the basics.

Checking If a Key Exists

Each dictionary comes with a property called `keys`. By invoking `keys.contains`, you can check if a dictionary contains a key or not. Listing 3.81 demonstrates the necessary syntax for that.

```
var bassGuitar: [String: Any] = [
    "brand": "Fender",
```

```
    "model": "Precision",
    "strings": 5,
    "availableColors": ["Black", "Sage Green", "Maple"]
]

bassGuitar.keys.contains("brand")   // true
bassGuitar.keys.contains("price")   // false
```

Listing 3.81 Checking if Key Exists in Dictionary

In this example, `bassGuitar.keys.contains("brand")` returns `true` because `"brand"` is an existing key in the dictionary. However, `bassGuitar.keys.contains("price")` returns `false` because `"price"` isn't a key within the dictionary. That's clear, right?

This feature will be useful when you learn about control flow in Chapter 4. You will be able to make your app behave differently depending on the existence of a key in a dictionary.

3.3.4 Modifying Dictionaries

In this section, you will learn how to make changes to dictionaries after their initial declaration. Adding, changing, and removing key-value pairs will be the focus.

Inserting Elements

Inserting a new key-value pair into a dictionary is as simple as merely providing the key and the value! Listing 3.82 features the required syntax.

```
var mobilePhone: [String: Any] = [
    "brand": "Apple",
    "model": "iPhone 15 Pro",
    "operatingSystem": "iOS 17",
    "screenSize": 6.1,
    "storageOptions": [128, 256, 512, 1024],
    "has5G": true,
    "colors": Set(["Black", "Blue", "White", "Titanium"])
]

mobilePhone["weight"] = 187
mobilePhone["waterResistant"] = true
```

Listing 3.82 Inserting New Key-Value Pairs into Dictionary

On the last two lines, the weight and water-resistance features of the mobile phone were added to the dictionary. It's simple as that!

Modifying Elements

Modifying the value for a key uses the exact same syntax as insertion. Listing 3.83 demonstrates both item modification and insertion to showcase them side-by-side.

```
var mobilePhone: [String: Any] = [
    "brand": "Apple",
    "model": "iPhone 15 Pro",
    "operatingSystem": "iOS 17",
    "screenSize": 6.1,
    "storageOptions": [128, 256, 512, 1024],
    "has5G": true,
    "colors": Set(["Black", "Blue", "White", "Titanium"]),
    "weight": 187
]

print(mobilePhone["weight"]!)           // Output: 187
mobilePhone["weight"] = 186             // Changes to 186
print(mobilePhone["weight"]!)           // Output: 186

mobilePhone["waterResistant"] = true    // Inserts new element
```

Listing 3.83 Dictionary Element Modification Demonstration

In this code snippet, `weight` was initially declared as `187`. The `mobilePhone["weight"] = 186` expression then changes this value to `186`. Note that this line has the exact same syntax as element insertion. The action here will depend on a couple of factors:

- If the dictionary contains an entry for `weight` already, Swift will update its value.
- Otherwise, Swift will insert the provided key-value pair, as happens for `waterResistant`.

Deleting Elements

In Swift, element removal from a dictionary is as straightforward as it gets. Listing 3.84 demonstrates two dictionary functions for this task.

```
var mobilePhone: [String: Any] = [
    "brand": "Apple",
    "model": "iPhone 15 Pro",
    "operatingSystem": "iOS 17",
    "screenSize": 6.1,
    "storageOptions": [128, 256, 512, 1024],
    "has5G": true,
    "colors": Set(["Black", "Blue", "White", "Titanium"]),
    "weight": 187
]
```

```
mobilePhone.removeValue(forKey: "has5G")    // Removes single element
mobilePhone.removeAll()                     // Removes all elements
```

Listing 3.84 Dictionary Element Removal Demonstration

The function `removeValue` will remove the key-value pair for the provided key, whereas `removeAll` will remove all key-value pairs from the dictionary, leaving an empty collection behind.

3.3.5 Iterating Through a Dictionary

There might be cases in which you have a dynamically created dictionary without knowing the exact key names. For example, a third-party library might have parsed a JSON file and returned a dictionary. In such a case, you might want to access all keys and/or values in the dictionary sequentially, as an array.

In such cases, you can use a `for` clause to iterate through key-value pairs, as demonstrated in Listing 3.85.

```
var mobilePhone: [String: Any] = [
    "brand": "Apple",
    "model": "iPhone 15 Pro",
    "operatingSystem": "iOS 17",
    "screenSize": 6.1,
    "storageOptions": [128, 256, 512, 1024],
    "has5G": true,
    "colors": Set(["Black", "Blue", "White", "Titanium"]),
    "weight": 187
]

for (key, value) in mobilePhone {
    print(key, value, separator: " : ")
}
```

Listing 3.85 Iterating Through Keys and Values of Dictionary

In this example, `for (key, value) in mobilePhone` is the iteration command—which works via the same logic as for arrays and sets. The code between { and } is executed for each key-value pair in the dictionary. On each iteration, Swift assigns the next key to the `key` variable and the next value to the `value` variable.

The output is shown in Figure 3.14. In this example, all we did between { and } was to print the keys and values. In upcoming chapters, we will do more interesting things as you learn more about Swift.

```
var mobilePhone: [String: Any] = [
    "brand": "Apple",
    "model": "iPhone 15 Pro",
    "operatingSystem": "iOS 17",
    "screenSize": 6.1,
    "storageOptions": [128, 256, 512, 1024],
    "has5G": true,
    "colors": Set(["Black", "Blue", "White", "Titanium"]),
    "weight": 187
]

for (key, value) in mobilePhone {
    print(key, value, separator: " : ")
}

has5G : true
colors : ["White", "Blue", "Black", "Titanium"]
weight : 187
model : iPhone 15 Pro
brand : Apple
operatingSystem : iOS 17
screenSize : 6.1
storageOptions : [128, 256, 512, 1024]
```

Figure 3.14 Output of Dictionary Iteration

And voilà! This final example concludes our content on dictionaries, as well as our entire chapter on collections.

3.4 Summary

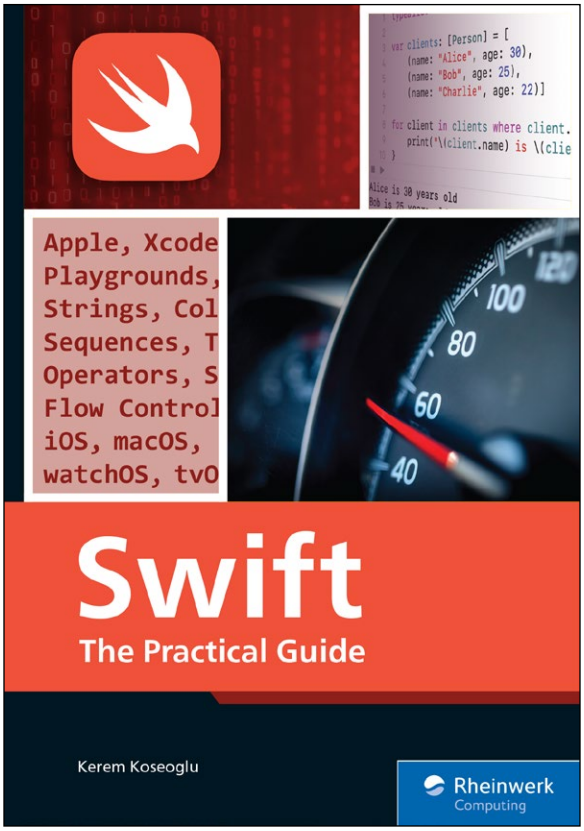
In this chapter, you learned about collections in Swift. Basically, collections are data structures that group multiple variables into a single, organized container. Swift offers three main collection types, which are summarized in Table 3.20.

Collection Type	Content	Duplicates	Use Case	Specialty
Array	Indexed/ordered collection of elements	Yes	Queue	Flexibility
Set	Hashed/unordered collection of elements	No	Member list	Fast element access
Dictionary	Hashed/unordered key-value pairs	No	Product properties	JSON-like

Table 3.20 Swift Collection Type Summary

Due to type safety concerns, Swift expects programmers to make use of optionals when accessing collections with flexible/uncertain data types, such as Any. You will learn more about optionals in Chapter 6 and will see corresponding examples there.

At this point, you know about variables and collections in Swift, which are the basic building blocks for any program. Now you can venture one step further and learn about control flow. In the next chapter, you will learn how to "flow" through different code snippets based on conditions.



Kerem Koseoglu

Swift

The Practical Guide

- Your comprehensive guide to the Swift programming language
- Work with variables, collections, enums, structs, and other language elements
- Practice as you learn with downloadable code snippets

rheinwerk-computing.com/6111

The Author

Dr. Kerem Koseoglu is a seasoned software engineer, author, and educator with experience in global software development projects. He works extensively with Swift, Python, and ABAP.

ISBN 978-1-4932-2718-1 • 680 pages • 12/2025

E-book: $54.99 • **Print book:** $59.95 • **Bundle:** $69.99

Rheinwerk Publishing