PART I
Building Applications with Swift

- **CHAPTER 1:** A Swift Primer
- **CHAPTER 2:** Writing a Swift Program
- **CHAPTER 3:** Classes, Structs, and Enums
- **CHAPTER 4:** Concurrency in Swift
- **CHAPTER 5:** Interfacing with Web Services
- **CHAPTER 6:** Storing Data with Core Data

1 A Swift Primer

WHAT'S IN THIS CHAPTER?

- Understanding Swift
- Declaring constants and variables and working with Swift's data types
- Transforming values with operators
- Controlling code execution with conditional statements and loops
- Defining and using Swift's enumerated data types
- Understanding, declaring, and using functions, anonymous functions, and closures

This chapter introduces the key concepts featured in the Swift programming language and covers the language's new syntax and data types. It is not intended as an introductory guide to Swift, but rather, as a way for programmers who have already worked with Swift to refresh their knowledge of the language. Prior knowledge of C and Objective-C programming on iOS and OS X is also assumed, although the information presented in this chapter should still make sense to those with no experience in C and Objective-C.

If you are already familiar with the foundations of the Swift programming language, you may want to dive right in with Chapter 2. Chapter 2 also shows you how to use Xcode's new playgrounds feature, which allows you to experiment with the effects of Swift code as you write it. The examples in this chapter can be entered directly into a playground so you can see the results of your code immediately, without having to create an Xcode project and compile the code.

WHAT IS SWIFT?

Swift is a brand-new language, developed by Apple, that is meant to offer an alternative to Objective-C for iOS and OS X development. Although it is designed to interoperate seamlessly with Objective-C, as well as C and C++, Swift is not an evolution of Objective-C, but rather an entirely new language with a much different pedigree. It drops a number of classic Objective-C language features, while introducing a host of new ones designed to make development of iOS and OS X programs safer and faster, as well as to make the development process itself more expedient.

Swift is the product of several years of development, as well as the research and insight gained from many languages that came before it. Far from being a simple improvement over Objective-C, its feature set is inspired by programming languages as diverse as Haskell, C#, Ruby, Python, and Rust. Swift also incorporates many features of the Cocoa and Cocoa Touch frameworks, such as key-value observing, into the language itself. The Swift compiler draws on much of the research and experience gained in creating LLVM and the clang compiler for Objective-C.

Unlike Objective-C, Swift is not a dynamically typed language. Instead, it uses static typing to help ensure the integrity and safety of your programs. Swift also prevents many of the problems inherent to the C language (and, by extension, Objective-C), especially with regards to memory integrity. While experienced Objective-C developers may lament this supposed loss of flexibility in their program design and construct, Swift's new features make writing programs simpler and easier, while still allowing programmers a great deal of freedom. Apple intends Swift to be suitable for application development as well as systems programming, and the design of the language exemplifies both of these areas of development.

Fortunately, Apple has also taken great care to integrate Swift into its existing ecosystem. Swift seamlessly operates with existing Objective-C code, libraries, and frameworks, as well as code written in C and C++. Programs can even mix these languages with little extra effort from you. Swift code is also completely supported by Xcode, allowing you to continue to use the development tools, such as compilers and debuggers, that you were familiar with when developing in Objective-C.

In fact, the introduction of Swift has added a new tool to the Xcode ecosystem's arsenal: playgrounds. Playgrounds are an interactive development environment for Swift that give you instant feedback on what blocks of code will do when executed. Swift also supports a read-eval-print loop, or REPL, that allows you to test snippets of Swift code right in the console. You can read more about playgrounds in Chapter 2.

Swift represents a massive leap forward in iOS and OS X development. While Objective-C programs will continue to be supported for quite some time, Swift is the way forward for iOS and OS X developers, and it is crucial to master the new skills introduced by Swift. Luckily, most of what you know about iOS and OS X is still important, even if you are writing your programs entirely in Swift. Most of all, Swift's new features make writing iOS and OS X applications more exciting and fun than they have ever been.

Before diving into the finer details of Swift development, you should familiarize yourself with Swift's basic concepts. If you already have a firm grasp of these concepts, however, feel free to skip to Chapter 2.

WHY LEARN SWIFT?

A bigger question is: Why should you learn Swift instead of writing your iOS and OS X apps in Objective-C? The truth is that Objective-C is not going to disappear any time soon. Most of the frameworks included in iOS and OS X are written in Objective-C, along with a large selection of third-party libraries and frameworks, and most iOS and OS X development tutorials are written for Objective-C. A thorough knowledge of Objective-C is important for developers of both of these platforms. However, Apple has already thrown a lot of weight into the development of Swift, and it is clearly the way forward for iOS and OS X development. Apple designed Swift to be both an applications and systems programming language, and it is certain that many future operating system components, frameworks, and libraries will be written in Swift. Swift will become increasingly important in the future.

Moreover, Swift expands and improves upon a lot of concepts from Objective-C, while adding a number of advanced features, such as closures, as well as making existing features, such as enumerated types, much more rich. Swift's syntax is cleaner and easier to learn and use than Objective-C, and it adds a lot of flexibility to APIs. It also allows iOS and OS X developers to explore other paradigms than the object-oriented development model inherent to Objective-C, such as functional programming. Finally, Swift is fun and adds a new level of exploration to the Apple development landscape.

WORKING WITH CONSTANTS AND VARIABLES

Swift has all the data types you are familiar with from languages such as C and Objective-C: numerical types such as `Int`, `Float`, and `Double`; Boolean types such as `Bool`; and character types such as `String` and `Char`. It also adds support for more complex types such as `Array` and `Dictionary` at the language level, providing a neat syntax for declaring and working with these container types.

Swift uses names to refer to values. These names are called *variables*. In Swift, a new variable is introduced using the keyword `var`. As with C and Objective-C, the actual value referred to by a variable can change. In the example that follows, the variable `x` first refers to 10, and then to 31:

```
var x = 10
x = 31
```

Swift also includes support for variables whose values cannot change. In this context, "variable" is a misnomer; these identifiers are more correctly referred to as *constants*. Constants are introduced with the `let` keyword:

```
let x = 10
```

You cannot change the value of `x` after it is declared—doing so will result in a compiler error. The following bit of code is not allowed:

```
let x = 10
x = 31    // This line will generate an error
```

Because constants' values cannot change, it is much easier to reason about the state of a constant than a variable. In Swift, it is preferable to use constants whenever possible; you should only use a variable when you absolutely need a value to vary over a program's execution. Good Swift programs will make extensive use of constants and minimize the number of variables that they use.

> **NOTE** *Technically speaking, constants are not variables because their values do not vary. However, in the context of programming languages, the term* variable *often does not adhere to the strict mathematical definition of the term; instead it indicates a name used to refer to a variable in a program's code. In this book, the term* variable *is often used when talking about both variables and constants because almost all concepts that apply to variables also apply to constants in Swift. The text explicitly denotes times when there are differences between the two.*

Swift also allows you to use any Unicode character as a variable name—not just characters from the English alphabet. The following code is allowed in Swift:

```
let π = 3.14159
let r = 10.0
let area = π * r * r
```

Almost any Unicode character can be used in a variable name. There are some restrictions, however: Variable names cannot include mathematic symbols, arrows, private or invalid Unicode code points, or line- and box-drawing characters. Variable names cannot start with a number, either, although they may contain numbers after the first character. Even with these restrictions, however, variable names are certainly more flexible in Swift than in C and Objective-C.

Understanding Swift Data Types

Swift has its own version of all the fundamental data types you would expect from C and Objective-C, including various numerical, Boolean, and character types. It also includes its own support for `Array` and `Dictionary` container types, including a simple, straightforward syntax for declaring and working with those types. Swift also introduces a new data type, a tuple, which groups multiple values—possibly of different types—into a single compound type.

Using Numerical Types

Swift features three basic numerical data types: `Ints`, `Floats`, and `Doubles`. As you would expect, these match up with the same data types in C and Objective-C. `Ints` represent whole numbers with no fractional component, such as 1, 304, and 10,000. `Floats` and `Doubles` each represent a number with a fractional component, such as 0.1, 3.14159, and 70.0. As in C, they differ only in their precision: `Floats` are 32-bit types, whereas `Doubles` are 64-bit types. Because of their increased size, `Doubles` offer more precision than `Floats`.

> **NOTE** *Generally, it is better to prefer* `Doubles` *over* `Floats`*, unless you know for sure that you don't need the precision of a* `Double` *and you can't sacrifice the increased size necessary to store a* `Double`*.*

Numerical types in Swift are written identically to those in C and Objective-C. To increase readability, however, they may include underscores, which are typically used to separate a number into chunks. For example, the value 1,000,000 could be written like this:

```
let n = 1_000_000
```

Underscores can be used in `Ints`, as well as both `Floats` and `Doubles`.

Swift numerical types have all the arithmetic operators you would expect, including +, –, /, *, and %, as well as the increment (++) and decrement (–) operators from C. These operators are covered in more detail later in this chapter.

There is no functional distinction between "primitive types" and "objects," as exists in Objective-C. The numerical data types have methods associated with them. The Swift standard library offers a bevy of methods that work with numerical types.

You may have noticed the lack of more specific integral data types such as short and long. Unlike C and Objective-C, Swift only offers a single basic integral data type, the Int. The size, or bit width, of the Int matches the underlying platform: 32 bits on 32-bit processors, and 64 bits on 64-bit processors.

Swift also has data types for integrals of specific widths, as well as unsigned versions of each integral data type. These are named similarly to their C counterparts: `Int8`, `Int16`, `Int32`, and `Int64`. The `UInt` is the unsigned version of the `Int`, and like an `Int`, matches the size of the underlying platform. `UInt8`, `UInt16`, `UInt32`, and `UInt64` are also available if you need unsigned integers of a specific width.

Preferably, however, you won't need to worry about the specific size of integral data types. Swift's type inferencing system, described later in this section, is designed to work with the `Int` data type. Generally speaking, integers of a specific bit width are not necessary when programming in Swift, but they are available for the rare times when the need for them arises.

Boolean Types

Swift offers a single Boolean data type, `Bool`, which can hold one of two values: `true` or `false`.

Swift's `Bool` type differs a bit from similar types in C and Objective-C. C does not have a Boolean type at all, instead treating any nonzero value as true and any zero value as false. In C, `ints`, `chars`, and even pointers can be treated as true or false, depending on their value. Objective-C offers a Boolean type, `BOOL`, as well as the values `YES` and `NO`, but in Objective-C, a `BOOL` is really just an `unsigned char`, and `YES` and `NO` are just the values 1 and 0, respectively. And Objective-C follows the same "truthiness" rules as C, so any nonzero value in Objective-C is true in a Boolean context (such as an `if` statement), and zero is false—even when dealing with pointers.

Swift dispenses with this nonsense. You cannot assign any values other than true and false to a Swift `Bool`, and only a `Bool` type may be used in Boolean contexts, such as if statements. Comparison operators, such as `==` and `!=`, also evaluate to a `Bool`. This change adds a level of type safety to Swift programs, ensuring that you only use a Boolean value—and nothing but a Boolean value—in appropriate contexts.

Using Character Types

Two character-based data types are available in Swift: `String`s and `Character`s. `Character`s represent a single Unicode character. `String`s are a collection of zero or more Unicode characters. Both are denoted by double quotes:

```
let c: Character = "a"
let s: String = "apples"
```

Although conceptually similar to string and character types in C and Objective-C, Swift's `String` and `Character` types are substantially different in practice. Both the `Character` and `String` types are treated as objects and have methods associated with them, just like Swift's other types. More importantly, they are designed to interact with text in a Unicode-compliant way. Whereas C strings are essentially nothing more than collections of byte values, Swift strings should be thought of as an abstraction over encoding-independent Unicode code points.

This difference has some significant practical implications. One major concern is how Swift handles characters outside of the Basic Multilingual Plane, or BMP. These include emoji characters, which, with the advent of mobile messaging, are becoming more and more widely used. Cocoa's `NSString` class assumes that all Unicode characters can be represented with a 16-bit integer (specifically, a `unichar`, which is a type alias for an unsigned short integer). This is true of any character in the BMP, which can be represented in a 16-bit integer using UTF-16 (one of several encodings for Unicode text; the other popular alternative is UTF-8). But not all Unicode code points can be represented in 16 bits—emoji characters being one of them.

As a result, `NSString` objects that include Unicode code points outside of the Basic Multilingual Plane will not return an intuitive value for their length and will not intuitively enumerate over their characters. In fact, the length method will return the number of `unichars` required to encode a character. Consider the simple Objective-C program that follows:

```
NSString *s = @"\U0001F30D";
NSLog(@"s is %@", s);
NSLog(@"[s length] is %lu", [s length]);
NSLog(@"[s characterAtIndex:0] is %c", [s characterAtIndex:0]);
NSLog(@"[s characterAtIndex:1] is %c", [s characterAtIndex:1]);
```

This program contains a string with a single character: the emoji globe character, or 🌍. However, the second line of the program will print "2" for the string's length, because this is the number of `unichars NSString` uses to encode 🌍. The third and fourth lines will also print odd characters to the screen, because they are printing one byte out of the two bytes necessary to encode the 🌍 character.

Swift does not have this problem. An equivalent Swift program will correctly report that it has a length of 1. It will also correctly print the character at index 0 and crash if you try to print a character at index 1 (which does not exist). The equivalent Swift program is shown below:

```
let s = "🌍"
println("s is \(s)")
println("s.length is \(countElements(s))")
println("s[0] is \(s[advance(s.startIndex, 0)])")
println("s[1] is \(s[advance(s.startIndex, 1)])")
```

It is becoming increasingly important to work with Unicode text in programs. The fact that Swift correctly handles Unicode out of the box represents a huge win over `NSString`.

Strings can be concatenated using the + operator:

```
let s1 = "hello, "
let s2 = "world"
let s3 = s1 + s2
// s3 is equal to "hello, world"
```

You can also compare two strings to see if they are equal using the == operator:

```
let s1 = "a string"
let s2 = "a string"
let areEqual = s1 == s2
// areEqual is true
```

Naturally, you can see if two strings are unequal using the `!=` operator:

```
let s1 = "a string"
let s2 = "a string"
let areNotEqual = s1 != s2
// areNotEqual is false
```

NOTE *Operators like +, ==, and* `!=` *are discussed in greater detail later in this chapter.*

Swift also dispenses with Cocoa's delineation between immutable and mutable strings, as seen in the `NSString` and `NSMutableString` types. Swift has one single string type that uses the power of Swift constants and variables to declare an instance as mutable or immutable:

```
let s1 = "an immutable string"
var s2 = "a mutable string"
s2 += " can have a string added to it"
```

An immutable string, on the other hand, cannot be appended to. The Swift compiler will emit an error if you try to append a string to an immutable string.

You can also create a new string from other data types using *string interpolation*. When creating a string literal, variables referenced in a \() construct will be turned into a string:

```
let n = 100
let s = "n is equal to \(n)"
// s is "n is equal to 100"
```

You can interpolate expressions in a string, too:

```
let n = 5
let s = "n is equal to \(n * 2)"
// s is "n is equal to 10"
```

Using Arrays

Arrays are an integral part of the Swift language. They are akin to the Foundation framework's `NSArray` and `NSMutableArray` classes, but Swift has syntactic support for the easy creation and manipulation of array data types. Swift's `Array` type also differs from Objective-C's `NSArray`. Most importantly, the values contained in a Swift `Array` must all be of the same type, as opposed to `NSArray`, which can contain instances of any class. As with `Strings`, Swift also uses the same `Array` type for both mutable and immutable instances, depending on whether the variable is declared using `var` or `let`.

Swift also provides a new syntax for initializing arrays. An array can be initialized by writing its elements between brackets, as shown here:

```
let shoppingList = ["bananas", "bread", "milk"]
```

Elements of an array can also be accessed using bracket notation:

```
let firstItem = shoppingList[0]
// firstItem is "bananas"
```

If an array is declared using the `let` keyword, it is immutable: Elements cannot be added or deleted from it. On the other hand, if an array is declared using the `var` keyword, you are free to add or delete elements from it, or change the element stored at a specific index. For example, in the code that follows, the second element is changed from `"bread"` to `"cookies"`:

```
var shoppingList = ["bananas", "bread", "milk"]
shoppingList[1] = "cookies"
```

You can also change a range of elements:

```
var shoppingList = ["candy", "bananas", "bread", "milk", "cookies"]
shoppingList[1...3] = ["ice cream", "fudge", "pie"]
```

Afterwards, `shoppingList` will contain `["candy", "ice cream", "fudge", "pie", "cookies"]`. You can also append elements to an array:

NOTE *Range operators are discussed later in this chapter.*

```
var a = ["one", "two", "three"]
a += ["four"]
println(a)    // Will print ["one", "two", "three", "four"]
```

It is best to use immutable arrays, unless you absolutely need to alter an array's contents.

Using Dictionaries

Dictionaries are also a fundamental data type in Swift. They fill the same role as `NSDictionary` and `NSMutableDictionary` from the Foundation framework, albeit with a few implementation differences. As with arrays, Swift also has language syntax support for creating and manipulating dictionary instances.

Like Swift arrays, and unlike Cocoa's `NSDictionary`, dictionaries in Swift must use the same type for every key, and the same type for every value. Additionally, keys must be *hashable*. As with arrays, Swift also does not have separate types for immutable and mutable dictionaries. An immutable dictionary is declared using the `let` keyword, and a mutable dictionary is declared using `var`.

Dictionaries are initialized using a syntax similar to arrays. Syntactically, a dictionary looks like a list of key-value pairs enclosed in brackets; the key and value are separated with a colon, as shown here:

```
let colorCodes = ["red": "ff0000", "green": "00ff00", "blue": "0000ff"]
```

Individual keys of a dictionary can be accessed by putting the key in brackets after the variable name:

```
let colorCodes = ["red": "ff0000", "green": "00ff00", "blue": "0000ff"]
let redCode = colorCodes["red"]
```

If a dictionary is mutable, you can also change an element using the same bracket notation you use to access it:

```
var colorCodes = ["red": "ff0000", "green": "00ff00", "blue": "0000ff"]
colorCodes["blue"] = "000099"
```

Using Tuples

Swift has one new fundamental data type: the tuple. Tuples group multiple values into one single compound value. They are similar to arrays, but unlike arrays, the elements of a tuple may be of different types. For example, you can use a tuple to represent an HTTP status code:

```
let status = (404, "Not Found")
```

In the preceding example, `status` consists of two values, `404` and `"Not Found"`. Both values are of different types: `404` is an `Int`, and `"Not Found"` is a `String`.

You can *decompose* a tuple into its constituent elements:

```
let status = (404, "Not Found")
let (code, message) = status
// code equals 404
// message equals "Not Found"
```

You can ignore one or more elements using an underscore (_) in the decomposition statement:

```
let status = (404, "Not Found")
let (code, _) = status
```

NOTE *When decomposing tuples, you can use* `var` *instead of* `let` *if you plan to modify the decomposed elements.*

Tuples are most helpful when you want to return multiple values from a function or method. Functions or methods can only return a single value, but because a tuple *is* a single value (albeit one consisting of multiple values), you can use it to get around this restriction.

Working with Type Annotations

Swift is a *type safe* language, meaning that every variable has a type, and the compiler checks to ensure that you are assigning a value of the correct type to every variable and passing values or variables of the correct type to functions and methods. This ensures that your code is always working with an expected set of values and that it is not trying to perform an operation (such as calling a method) on a value that does not support that operation.

Type annotations are a method by which you, the programmer, communicate to the compiler the type of a variable. Annotations should be familiar to you if you have programmed in C or Objective-C. Consider the following code sample:

```
NSString *s = @"this is a string";
int x = 10;
```

In the preceding example, `s` and `x` are both variables. Each has a type annotation in front of the variable name: `s` is annotated with the type `NSString`, and `x` is annotated with the type `int`. These annotations are necessary to communicate to the compiler the types of values you expect `s` and `x` to hold. As a result, the compiler issues an error if you attempt to assign a value of some other type to the variable, as in the example that follows:

```
int x = 10;
x = @"this is a string";
```

Types help guarantee the correctness of code you write by ensuring that you are always working with a set of values that you expect. If a function expects an integer, you can't pass it a string by mistake.

Variables can be annotated in Swift, although the syntax differs from C and Objective-C. In Swift, you annotate a type by writing a colon and type name after the variable name. In the example that follows, `ch` is declared to be a `Character`:

```
let ch: Character = "!"
```

Essentially, the preceding declaration is the same as this declaration in C:

```
char ch = '!'
```

Type annotations can also be used in functions and methods to annotate the types of their arguments:

```
func multiply(x: Int, y: Int) -> Int {
    return x * y;
}
```

Here, both the parameters `x` and `y` have been declared to be of type `Int`.

NOTE *Functions are covered in greater detail in the section "Working with Functions," found later in this chapter.*

Of course, annotating each and every variable with a type quickly becomes unwieldy, and places the burden on the programmer to ensure that all variables are annotated properly. Languages such as C and Objective-C also allow you to *cast* values from one type to another, eroding the foundation of the meager type safety afforded in those languages. All in all, explicitly annotating types is cumbersome and error-prone. Luckily, Swift offers a solution to this madness: its type inference engine.

Minimizing Annotation with Type Inference

Many programmers prefer dynamically typed languages such as Python and JavaScript because such languages do not require that programmers annotate and manage types for every variable. In most dynamically typed languages, a variable can be of *any* type, and so type annotations are either optional or not allowed at all. Objective-C took a hybrid approach: It required type annotations, but any variable pointing to an instance of an Objective-C class (which includes anything derived from `NSObject`, but not primitive types like `int`s, `float`s, and so on) could simply be declared with type `id`, and therefore point to *any* Objective-C instance type. Even when stricter annotations were used, the Objective-C compiler did not make any strict guarantees about the type of an Objective-C variable.

While dynamically typed languages are often considered more pleasant to work with than statically typed languages, the lack of strict type safety means that the correctness of programs cannot be guaranteed, and they are often more difficult to reason about, particularly when working with third-party code or code you wrote years ago.

Statically typed languages are not guaranteed to be any safer than dynamically typed languages, however. C has static types and requires type annotations, but it is trivial to circumvent C's limited type safety through the use of type casting, void pointers, and similar language-supported chicanery.

Swift takes the best of both worlds: Its compiler makes strict guarantees about the types of variables, but it uses *type inferencing* to avoid making programmers annotate each and every variable manually.

Type inferencing allows a language's compiler to deduce the types of variables based on how they are used in code. In practice, this means that you can forgo adding type annotations to most variables and instead let the compiler do the work of figuring out the type. Don't worry—even without annotations, your programs still have all the type safety afforded by Swift's type system.

Consider the following snippet of code:

```
let s = "string"
let isEmpty = s.isEmpty
```

Neither of these constants has been annotated with a type, but Swift is able to infer that `s` is a `String` and `isEmpty` is a `Bool`. If you try to pass in either of these constants to a function that takes `Int`s, you will get an error:

```
func max(a: Int, b: Int) -> Int {
    return a > b ? a : b
}
max(s, isEmpty)
```

The preceding code will generate the error message `'String' is not convertible to 'Int'`, demonstrating that Swift knows that `s` and `isEmpty` are not `Int` types.

While Swift's type inferencing system greatly reduces the number of annotations you have to make manually in your code, there are still times when you must declare types. When writing functions, you must declare the types of all parameters, as well as the return type of the function.

You may also have to annotate a type when the type of a variable is ambiguous. For example, both `String`s and `Character`s are denoted by text surrounded by double quotes. The Swift compiler infers a single character in double quotes to be a `String`, not a `Character`. The code that follows will generate an error, because `c` is typed as a `String`:

```
func cId(ch: Character) -> Character { return ch; }
let c = "X"
cId(c)
```

If you want `c` to be a `Character`, you have to explicitly declare it to be a character. The following snippet of code works:

```
func cId(ch: Character) -> Character { return ch; }
let c: Character = "X"
cId(c)
```

However, such ambiguous cases are rare in Swift. Usually, the compiler has no trouble inferring the type you intend a variable to be, and you generally don't have to worry about explicit type annotations.

TYPE THEORY

Type inferencing is a huge topic in computer science and falls into the category of knowledge and research known as *type theory*. One important element of type theory is the Hindley-Milner type system, which forms the basis of many type inference engines. The type inference algorithm used in this system was intended as a way to describe types for the simply typed lambda calculus, a method of computation that is the foundation for many modern programming languages.

A full discussion of type theory is far beyond the scope of this book, but interested readers are encouraged to peruse the plethora of material on the subject available online.

Clarifying Code with Type Aliasing

In some cases, it may be helpful from the standpoint of code clarity to refer to a type in more specific terms. Let's say you write a function to calculate speed:

```
func speed(distance: Double, time: Double) -> Double {
    return distance / time
}
```

Certainly this function does the job, but what is `distance`? Is it measured in feet? Meters? And what is `time`? Is it seconds, or minutes, or hours? The function isn't clear what it expects. It would be much nicer to write the function like this:

```
func speed(distance: Feet, time: Seconds) -> FeetPerSecond {
    return distance / time
}
```

Of course, `Feet`, `Seconds`, and `FeetPerSecond` are not data types in Swift. However, Swift has a facility for providing more expressive type names: *type aliases*. You may be familiar with `typedefs` from C and Objective-C. Type aliases act the same way: They allow you to specify aliases for existing types.

You can modify the preceding code so it will actually compile using type aliases:

```
typealias Feet = Double
typealias Seconds = Double
typealias FeetPerSecond = Double

func speed(distance: Feet, time: Seconds) -> FeetPerSecond {
    return distance / time
}
```

Type aliases allow you to write your function so it is more expressive and more readable. It is a powerful feature that you should feel comfortable using in your own code.

WORKING WITH OPERATORS

Swift includes a number of built-in operators for working with data types. For the most part, these are arithmetic operators, intended to work with the numerical data types (`Ints`, `Floats`, and `Doubles`), but Swift also has an operator for string concatenation, as well as logical and comparison operators. Swift also introduces operators for representing and dealing with ranges of values. Almost all of them, with the exception of the range operators, are also a part of C and Objective-C, so you are undoubtedly familiar with most of them, and they behave nearly the same as their C and Objective-C counterparts.

Using Basic Operators

Swift supports four standard arithmetic operators:

- Addition (+)
- Subtraction (–)
- Multiplication (*)
- Division (/)

These operators behave exactly as you would expect, coming from C and Objective-C, with one minor difference: They do not allow overflow. That is, if a variable is the max value for its type (for example, the maximum value of an `Int`), and you try to add a value to it, an error will occur. If you want to overflow behavior you expect from C, you must use one of the overflow variants of these operators. Overflow operators are discussed in the next section.

As in C, applying the division operator (/) to `Int`s results in the maximum value that divides the second `Int` into the first:

```
let m = 11 / 5
// m is equal to 2
```

The addition operator (+) is also supported by strings. "Adding" two strings together concatenates them:

```
let s1 = "hello"
let s2 = ", world"
let s3 = s1 + s2
// s3 is now "hello, world"
```

Swift also has a remainder operator, `%`. The remainder operator returns the value left over after a division operation:

```
let rem = 11 % 5
// rem is equal to 1
```

The remainder operator is also supported for `Float`s and `Double`s, a departure from its behavior in C, which only allows the remainder operator to be used on `Int`s:

```
let rem = 5.0 / 2.3
// rem is equal to 0.4
```

NOTE *In C and Objective-C, the `%` operator is commonly known as the modulus operator. Applying the modulus operation to negative numbers results in values that, while mathematically correct, are counter-intuitive to many programmers. Swift uses the more intuitive behavior when applying the operation to negative numbers, and so refers to `%` as the remainder operator instead. When applied to positive operands, the modulus and remainder operations yield essentially the same result.*

Finally, Swift also supports the increment (++) and decrement (–) operators, which increase or decrease a value by 1.

```
var i = 10
i++
// i is now equal to 11
```

These operators come in both prefix and postfix varieties. The prefix form yields the value of the variable *after* the operation is applied, whereas the postfix form yields the value of the variable *before* the operation was applied. The following code gives an example of this.

```
var i = 10
var j = i++
// j is now equal to 10, and i is equal to 11
var k = ++i
// i and k are both equal to 12
```

Using Compound Assignment Operators

You can combine an arithmetic operation with an assignment in one fell swoop using the compound assignment operators:

- Addition assignment (`+=`)
- Subtraction assignment (`-=`)
- Multiplication assignment (`*=`)
- Division assignment (`/=`)
- Remainder assignment (`%=`)

These operators perform their associated operation *and* set a variable to a new value in a single expression:

```
var x = 20
x += 10
// x is now equal to 30
```

Using Overflow Operators

Swift's arithmetic operators do not allow values to overflow. In other languages such as C, if a variable is at its maximum value and you add another value to it, the variable *wraps around* to its minimum value. In Swift, a runtime error occurs instead. However, there may be occasions in which you want the overflow behavior. Swift provides this with the overflow variants of its arithmetic operators:

- Overflow addition (`&+`)
- Overflow subtraction (`&-`)
- Overflow multiplication (`&*`)
- Overflow division (`&/`)
- Overflow remainder (`&%`)

You use these operators in exactly the same way as the basic arithmetic operators. The only difference is that they allow overflow (or underflow). For example:

```
var num1: Int8 = 100;
var num2: Int8 = num1 &+ 100;
println(num2);  // Prints -56
```

Using Range Operators

You can easily represent ranges in Swift using its range operators. Swift provides two ways to represent ranges: a closed range operator (`...`) and an open range operator (`..<`). Both take two integer operands and yield a range of values between them. The closed range operator includes both operands' values in the range, whereas the open range operator includes the first operand's value, but not the second operand's. They are often used as counters in loops:

```
for i in 1...5 {
    // i will contain the values 1, 2, 3, 4, 5
    println("\(i)")
}
for i in 1..<5) {
    // i will contain the values 1, 2, 3, 4
    println("\(i)")
}
```

Using Logical Operators

Swift has three logical operators: logical NOT (`!`), AND (`&&`), and OR (||). `!` is a prefix operator and inverts a variable:

```
let b1 = true
let b2 = !b1
// b2 is false
```

`&&` and || both take two operands. `&&` returns `true` if *both* operands are true, and || returns `true` if at least one operand is true.

```
let b1 = true
let b2 = false
let b3 = b1 || b2
// b3 is true
let b4 = b1 && b2
// b4 is false
```

Swift also has a ternary condition operator, inherited from C. This operator takes three parameters:

- A condition
- A value to return if the condition is true
- A value to return if the condition is false

Essentially, it is an inline `if` statement. It looks like this:

```
let flag = true
let res = if flag ? 1 : 0
// res is equal to 1
```

Unlike C and Objective-C, these operators *only* work on `Bool` values. You cannot use them on non-Boolean values, such as `Ints`.

Using Comparison Operators

There are six comparison operators in Swift:

- Equal to (`==`)
- Not equal to (`!=`)
- Greater than (`>`)
- Less than (`<`)
- Greater than or equal to (`>=`)
- Less than or equal to (`<=`)

Like the logical operators, the comparison operators *only* return `Bools`, although they operate on most data types, including the numerical types and even classes (if classes define a custom operator). Each comparison operator takes two operands, returning the result of the operation. They behave identically to their counterparts in C and Objective-C.

As an example, consider the following:

```
    let a = 10
    let b = 20
    if (a > b) {
        println("a is greater than b")
    } else if (a == b) {
        println("a is equal to b")
    } else if (a < b) {
        println("a is less than b")
}
```

NOTE *Conditional statements are covered in greater detail in the section "Using Conditional Statements."*

Using Custom Operators

You can also define your own custom operators in Swift. Custom operators are discussed more thoroughly in Chapter 8.

MAKING DECISIONS WITH CONTROL FLOW

A programming language would not be good for much if it did not have a way to make decisions based on certain conditions. Swift offers all the familiar control flow statements to allow your program to make decisions during its execution.

Using Conditional Statements

Swift has two basic conditional statements: if statements and switch statements. An if statement executes its body only if the condition is true:

```
let flag = true
if flag {
    println("This statement is executed")
}
```

If statements can also take an else block, which is executed if the condition is false:

```
let flag = false
if flag {
    println("This statement is not executed")
} else {
    println("This statement is executed")
}
```

If statements can be chained together. The first block for which the condition is true is executed:

```
let x = 10
if x < 5 {
    println("This statement is not executed")
} else if x < 10 {
    println("This statement is not executed, either")
} else if x < 20 {
    println("This statement is executed")
} else {
    println("This statement is not executed")
}
```

As with the basic if statement, chained if statements do not require an else block.

Swift also has switch statements. Switch statements operate on a value and compare it against numerous cases. The first case that evaluates to true is executed. Switch statements are essentially like chained if statements, in a more compact, easier-to-read form.

Switch statements' most basic form is much like C, with a slightly different syntax:

```
let n = 20
switch n {
case 0:
    println("This statement is not executed")
case 10:
    println("Neither is this statement")
case 20:
    println("But this one is!")
default:
    println("And this one isn't")
}
```

Aside from syntax, Swift's switch statements differ from C's in several key ways. For one thing, the value used in a Swift switch statement does not need to be an integer, or even a numerical type; strings, tuples, enumerations (discussed later in this chapter), optional types (discussed in Chapter 8), and even custom classes may be considered as well.

Each case may also take several matches, each one separated by a comma:

```
let ch: Character = "a"
switch ch {
case "a", "e", "i", "o", "u":
    println("\(ch) is a vowel")
case "y":
    println("\(ch) may be a vowel or a consonant")
default:
    println("\(ch) is a consonant")
}
```

Cases must have a body. However, instead of executable code, a case may have a `break` statement. The case is matched, but because there is no code to execute, execution jumps out of the `switch` statement:

```
let n = 10
switch n {
case 10:
    break
default:
    println("\(n) is not 10")
}
```

Ranges can also be matched in `switch` statements:

```
let n = 23
switch n {
case 0...10:
    println("\(n) is between 0 and 10")
case 11...100:
    println("\(n) is between 11 and 100")
case 101...1000:
    println("\(n) is between 101 and 1000")
default:
    println("\(n) is a big number")
}
```

Even tuples can be matched:

```
let color = (255, 0, 0)
switch color {
case (0, 0, 0):
    println("\(color) is black")
case (255, 255, 255):
    println("\(color) is white")
case (255, 0, 0):
    println("\(color) is red")
case (0, 255, 0):
    println("\(color) is green")
case (0, 0, 255):
    println("\(color) is blue")
default:
    println("\(color) is a mixture of primary colors")
}
```

In the case of tuples and many other composite data types, including enumerations and custom classes, it can be helpful to bind variables to specific elements of the data type. You can use Swift's *value bindings* to associate variables with variables and constants:

```
let color = (255, 0, 0)
switch color {
case (let red, 0, 0):
    println("\(color) contains \(red) red")
case (0, let green, 0):
    println("\(color) contains \(green) green")
case (0, 0, let blue):
    println("\(color) contains \(blue) blue")
default:
    println("\(color) is white")
}
```

The first case (`let red, 0, 0`) matches any tuple whose second and third elements are 0. The first element may match *any* value and is bound to the constant `red`.

You may also use variables in the binding, using the `var` keyword instead of `let`. If the binding is a variable, it may be modified in the case's body, just like any variable.

Case statements may also take a `where` clause to match on additional constraints outside of values:

```
let color = (255, 0, 0)
switch color {
case let (r, g, b) where r == g && g == b:
    println("\(color) has the same value for red, green, and blue")
default:
    println("\(color)'s RGB values vary")
}
```

In the first case, you can decompose a tuple to quickly bind the constants `r`, `g`, and `b` to the components of color.

Swift's switch statements do not implicitly fall through to the next statement in the switch block, as happens in C. Execution breaks out of a switch statement as soon as a case is matched and its body executed. You can explicitly fall through to another case with the `fallthrough` keyword:

```
let ch: Character = "y"
print("\(ch) is a ")
switch ch {
case "y":
    print("consonant, and also a ")
    fallthrough
case "a", "e", "i", "o", "u":
    println("vowel")
default:
    println("consonant")
}
```

Finally, the cases of a switch statement must be *exhaustive*; that is, the entire range of the condition's values must be accounted for. Otherwise, a compiler error will occur. A `default` case may be provided to cover the entire range of values, and the `break` statement may be used if no action should be taken on default values.

Using Loops

Swift offers two varieties of for loops: for-in loops, which enumerate over a set of values, and the more familiar for-condition loops, which loop over values until a certain condition is reached.

For-in loops may be unfamiliar to C programmers, and even Objective-C only got the feature with the release of Objective-C 2.0 in 2006. For-in loops, also referred to as "fast enumeration" in the Objective-C documentation, iterate over every element in a collection type, such as an array, dictionary, or even a `Range` object. At their most basic, for-in loops can utilize the range operators to loop over a set of monotonically increasing values:

```
for i in 1...5 {
    // iterates over the values 1, 2, 3, 4, 5
}
```

Instead of a range, you can also use a container type, such as an array or dictionary, to loop over a set of elements:

```
let fruits = ["lemon", "pear", "watermelon", "apple", "breadfruit"]
for fruit in fruits {
    // iterates over the values lemon, pear, watermelon, apple, breadfruit
}
```

You can also loop over a dictionary. Looping over a dictionary returns a 2-tuple containing each key and value. This 2-tuple can be decomposed into its constituent parts:

```
let nums = [0: "zero", 10: "ten", 100: "one hundred"]
for (num, word) in nums {
    // num will be 0, 10, 100
    // word will be zero, ten, one hundred
}
```

Swift's basic container types, such as arrays, dictionaries, and ranges, can all be iterated over in a for-in loop. Other objects that adopt the Sequence protocol can also be iterated over in a for-in loop. You will learn more about how to implement this protocol in your own classes in Chapter 8.

Swift also features for-condition loops, which are more familiar to those with C and Objective-C backgrounds. They are identical in spirit to C's for loop, although the syntax differs a bit:

```
for var i = 0; i < 5; ++i {
    // loops over 0, 1, 2, 3, 4
}
```

In for-conditional loops, variables are only valid within the scope of the loop; in the preceding example, `i` cannot be accessed outside of the loop. If you need to use `i` outside of the loop, you must declare it *before* the loop. The following code gives an example of this.

```
var i = 0;
for i = 0; i < 5; ++i {
    // loop body
}
```

If you do not wish to use the for loop's variable in a function, you can indicate this by using an underscore (_) as the variable name:

```
for _ in 1...10 {
    println("\n")
}
```

For-conditional loops are a bit superfluous because you can write the exact same construct using a for-in loop with a range, so you may not see them that often in practice. However, they are available if you decide you want to use them.

> **NOTE** *Some programming languages, such as Python, only have for-in loops. They dispense with for-condition loops entirely, instead relying on ranges to simulate for-condition loops. Some language designers consider for-in loops with ranges to be "safer" because they avoid issues with boundary conditions that are present in for-condition loops (commonly referred to as off-by-one errors). Swift still supports for-condition loops, but you may find it more convenient and less error-prone to use for-in loops in your code.*

For-in and for-conditional loops are considered to be *determinate* loops: They have a well-defined ending condition. Swift also offers two forms of *indeterminate* loops, or loops without a well-defined ending condition: while loops and their close cousins, do-while loops.

The bodies of while loops are simply executed until their condition becomes false:

```
var flag = true
var i = 1
while flag {
    println("i is \(i++)")
    if i > 10 {
        flag = false
    }
}
```

Do-while loops are similar, except that the body is always executed at least once, and the condition is checked at the *end* of the loop, rather than at the beginning:

```
var flag = false
var i = 1
do {
    println("i is \(i++)")
    flag = i <= 10
} while flag
```

Swift offers several varieties of loops, although all of them can be mimicked with the simple while loop. However, it may make your code clearer to use one of the other looping constructs. For example, when looping over a known range or set of items, a for-in loop communicates the intent much more clearly than a basic while loop.

Control Transfer Statements

Loops may contain statements that transfer control of code execution to different parts of code. These statements are identical to their behavior in C and Objective-C and may be used in all types of loops.

The `continue` statement causes execution to jump back to the top of the loop. The following code sample only prints odd numbers:

```
for i in 1...100 {
    if i % 2 == 0 {
        continue
    }
    println("\(i)")
}
```

The `break` statement causes a loop to immediately end. The following loop continues to execute until the user enters `q`:

```
while true {
    print("Enter 'q' to end: ")
    let input = getUserInput()
    if input == "q" {
        break
    }
}
```

Both `break` and `continue` statements break out of the innermost loop. However, you can label loops, which enables you to break out of an outer loop instead:

```
let data = [[3, 9, 44], [52, 78, 6], [22, 91, 35]]
let searchFor = 78
var foundVal = false
outer: for ints in data {
    inner: for val in ints {
        if val == searchFor {
            foundVal = true
            break outer
        }
    }
}
if foundVal {
    println("Found \(searchFor) in \(data)")
} else {
    println("Could not find \(searchFor) in \(data)")
}
```

GROUPING TYPES WITH ENUMERATIONS

Enumerations allow you to group related types together. In C and Objective-C, enumerations—designed with the `enum` keyword—are little more than constants grouped together. There is little compiler support to treat them as a "type" unto themselves. In Swift, on the other hand, enumerations are types unto themselves, giving them all the power of other types, such as structs and classes.

Like C and Objective-C, Swift enumerations are introduced with the `enum` keyword:

```
enum Direction {
    case North
    case South
    case East
    case West
}
```

Each enum case may also be written on the same line, separated by commas:

```
enum Direction {
    case North, South, East, West
}
```

You can easily use enums in switch statements, where each case can handle one of the enumerated types:

```
let dir = Direction.North
switch dir {
case .North:
    println("Heading to the North Pole")
case .South:
    println("Heading to the South Pole")
case .East:
    println("Heading to the Far East")
case .West:
    println("Heading to Europe")
}
```

NOTE *Because the only cases associated with the* `Direction` *enum are North, South, East, and West, and the switch statement handles all of those cases, a default case is not necessary.*

Much like classes and structs, you can treat the individual cases in enumerated types as constructors and associate values with them. Associated values are similar to instance variables (or properties) in classes and structs (although even with this feature, enums are still not quite as powerful as classes). You can specify associated values when creating an enum case. For example, here is one way to describe JSON using enums:

```
enum JSValue {
    case JSNumber(Double)
    case JSString(String)
```

```
        case JSBool(Bool)
        case JSArray([JSValue])
        case JSDictionary([String: JSValue])
        case JSNull
    }
```

A `JSNumber` is not merely a data type: It also has a `Double` value associated it; a `JSString` has a `String` value associated; and so forth, for all types except for `JSNull` (which, by its nature, has no unique value associated with it).

You create an enum with an associated value similarly to how you would instantiate a class or struct:

```
let val = JSValue.JSNumber(2.0)
```

You can decompose enumerated values in switch statements to get the associated value, just as you can with a tuple:

```
let val = JSValue.JSNumber(2.0)
switch val {
case .JSNumber(let n):
    println("JSON number with value \(n)")
default:
    println("\(val) is not a JSON number")
}
```

Compared to C and Objective-C, enumerated types in Swift are very powerful. You can use them in many of the same ways you would use classes or structs. Enumerated types are covered in more detail in Chapter 3.

WORKING WITH FUNCTIONS

Swift functions are standalone blocks of code that you call to transform inputs into outputs, or simply to perform some action like writing to a file or printing output to the screen. Functionally they are the same as functions in C and Objective-C, although their syntax differs substantially. Functions can be top-level, or they can be nested within other functions (a feature not available in C or Objective-C). Functions can also be associated with classes, structs, and enums, in which case they are referred to as *methods*, although the syntax for declaring methods is more or less the same. (You learn more about classes and structs in Chapter 4.)

Declaring Functions

Every function has a name, which calls the function. Functions may also optionally take parameters. Functions are declared with the `func` keyword, followed by the name of the function. The name is followed by a set of parentheses; parameters may be declared within the set of parentheses. Finally, if a function returns a value, you specify the return value after a *return arrow* (`->`); you may omit this part of the function declaration if the function does not return a value. (Functions that do not return a value are called *void functions*.) The body of the function is specified in braces. Here is a simple definition for a function called `multiplyByTwo` that takes a single `Int` parameter, `x`, as input, and returns the parameter multiplied by 2:

```
func multiplyByTwo(x: Int) -> Int {
    return x * 2
}
```

Function calls look the same as in C and Objective-C:

```
let n = multiplyByTwo(2)
// n is equal to 4
```

If a function takes multiple parameters, the parameters are separated by commas:

```
func multiply(x: Int, y: Int) -> Int {
    return x * y
}
let n = multiply(2, 4)
// n is equal to 8
```

Functions with no return value can omit the return type:

```
func printInt(x: Int) {
    println("x is \(x)")
}
printInt(10)
// will print "x is 10" to the console
```

Functions can only return a single value. However, you can return multiple values by wrapping them in a tuple:

```
func makeColor(red: Int, green: Int, blue: Int) -> (Int, Int, Int) {
    return (red, green, blue)
}
let color = makeColor(255, 12, 63)
let (red, green, blue) = color
```

Specifying Parameter Names

Within a function, you use a variable by referencing the name specified in the parameter list. When calling the function, parameters are passed in the order they were declared in the function, and the caller does not need to specify the names of the parameters when calling the function.

Sometimes it may be useful to force the caller to specify parameter names. This can aid the readability of code that uses the function, for example. You specify the *external* parameter name before the *internal* parameter name:

```
func makeColor(red r: Int, green g: Int, blue b: Int) -> (Int, Int, Int) {
    return (r, g, b)
}
```

Within the makeColor function, the parameters are referred to as r, g, and b. However, *outside* of the function, they are specified as red, green, and blue. When calling a function with named parameters, you *must* specify the name:

```
let color = makeColor(red: 255, green: 14, blue: 78)
```

Often, you may want the internal parameter name and the external name to be the same. You can do this by prefixing the name with a hash symbol (#):

```
func makeColor(#red: Int, #green: Int, #blue: Int) -> (Int, Int, Int) {
    return (red, green, blue)
}
let color = makeColor(red: 255, green: 14, blue: 78)
```

Defining Default Parameters

You may also specify default values for function parameters. If you do not specify a value for such parameters when calling the function, the default value is used. You can still override the default value when calling the function, though:

```
func multiply(x: Int, by: Int = 2) -> Int {
    return x * by
}
let x = multiply(4)
// x is equal to 8
let y = multiply(4, by: 4)
// y is equal to 16
```

If you do not specify an external name for a default parameter, Swift uses the internal name as the external name as well. Regardless, you *must* use the external name when calling the function.

If you really do not want callers to use an external name when calling a default parameter, you can use an underscore (_) as its external name:

```
func multiply(x: Int, _ by: Int = 2) -> Int {
    return x * by
}
let x = multiply(4, 4)
// x is equal to 16
```

However, it makes your code more readable and is considered best practice to use an external name for default parameters.

Specifying Variadic Parameters

Functions may take a variable number of arguments. Variadic parameters are specified by placing three dots (. . .) after a parameter's type name:

```
func multiply(ns: Int...) -> Int {
    var product = 1
```

```
        for n in ns {
            product *= n
        }
        return product
    }
```

Variadic parameters are passed as an array; in the multiply function in the preceding code, the type of the `ns` parameter is [`Int`], or an array of `Int`s. You call the function with a comma-separated list of variables or values for the parameter:

```
    let n = multiply(4, 6, 10)
    // n is equal to 240
```

A function can take no more than one variadic parameter, and it must be specified last in the list of parameters.

Specifying Constant, Variable, and In-Out Parameters

Just as you can use `let` or `var` to specify constants or variables, you can also use `let` or `var` in function parameter lists to specify if a parameter is a constant or variable. By default, all parameters are constants, so you cannot change their values in a function. The following is not allowed:

```
    func multiply(x: Int, y: Int) -> Int {
        x *= y
        return x
    }
```

However, if you declare the parameter using the `var` keyword, you can modify it within the body of the function:

```
    func multiply(var x: Int, y: Int) -> Int {
        x *= y
        return x
    }
```

Regardless of whether the parameter is a variable or a constant, changes to a parameter are seen only within the body of the function itself. The function does not change the value of variables outside of the function:

```
    let x = 10
    let res = multiply(x, 2)
    // res is equal to 20, but x is still equal to 10
```

Sometimes it may be useful for a function to affect variables outside of the function. In such cases, you can specify the parameter as an `inout` parameter:

```
    func multiply(inout x: Int, y: Int) {
        x *= y
    }
```

When calling a function with an `inout` parameter, you prefix the `inout` parameter with an ampersand (`&`). This syntax is familiar if you've programmed in C or Objective-C: An `inout` parameter is similar to a pointer parameter in C and Objective-C. In those two languages, `&` is also used to get the address of a variable in memory, which returns a pointer to that variable. Swift has largely dispensed with the concept of pointers but uses a similar syntax for its familiarity.

Although the preceding multiply function does not return a value, it will still change the value of variables passed into it:

```
var x = 10
multiply(&x, 2)
// x is now equal to 20
```

You may only pass *variables* as `inout` parameters; passing a constant or value will result in a compiler error.

Function Types

Functions are types, just like `Ints`, `Floats`, `Arrays`, and everything else in Swift. However, there is not one unifying Function type. Rather, each unique function signature (that is, the combination of a function's parameter types and return type) represents a different, unique type. Take, for example, the following function:

```
func multiply(x: Int, y: Int) -> Int {
    return x * y
}
```

The type is "a function that takes two `Int` parameters and returns an `Int`" and is denoted with type signature `(Int, Int) -> Int`.

If a function has no parameters, its parameter types are specified as an empty set of parentheses, `()`. Although referred to as *void*, it is essentially just a tuple with 0 elements. Likewise, a function that returns no value has a return type of `()`. Therefore, the type signature for the following function is `() -> ()`:

```
func printDash() {
    println("-")
}
```

As with other data types, you can declare a constant or variable that refers to a function:

```
func multiply(x: Int, y: Int) -> Int {
    return x * y
}
let m: (Int, Int) -> Int = multiply
let n = m(2, 4)
```

As the preceding code sample shows, you can call a variable or constant that refers to a function as though it were a function.

NOTE *As with other variables, you do not need to specify the type of a variable that points to a function unless you do not initialize it immediately, as Swift can infer the variable's type, just as it can with other data types such as* `Ints` *and* `Floats`.

Because functions are objects, just like any other data type in Swift, you can specify a function as a parameter to another function and return a function from a function. For example, the function map that follows takes a function and an array of `Ints`, and multiplies each by 2:

```
func multiplyByTwo(x: Int) -> Int {
    return x * 2
}

func map(fn: (Int) -> Int, ns: [Int]) -> [Int] {
    var res: [Int] = []
    for n in ns {
        let n1 = fn(n)
        res.append(n1)
    }
    return res
}

let nums = map(multiplyByTwo, [1, 2, 3])
// nums is now [2, 4, 6]
```

You can also return a function from a function:

```
func subtract(x: Int, y: Int) -> Int {
    return x - y
}

func add(x: Int, y: Int) -> Int {
    return x + y
}

func addOrSubtract(flag: Bool) -> (Int, Int) -> Int {
    if flag {
        return add
    } else {
        return subtract
    }
}

let fn = addOrSubtract(false)
let res = fn(4, 2)
// res is equal to 2
```

Using Closures

Functions' ability to take other functions as parameters and return functions is especially powerful when coupled with Swift's *closures*. Like functions, closures are blocks of code that can be passed around as objects. They are so named because they *close over* the scope in which they are declared, meaning that they can refer to variables and constants that are currently in their scope. Unlike functions, however, closures do not have to be named or declared; they can simply be defined when they are used.

In fact, Swift functions are just special cases of closures, meaning they are treated the same way by the language (and its supporting infrastructure, such as the Swift compiler).

Other than named functions, closures are most commonly used in a *closure expression*, a bit of code that defines an anonymous closure. Closure expressions are written in braces. Like functions, they optionally take a list of parameters specified in parentheses and optionally have a return value specified after a return arrow (`->`). The body of a closure begins after the keyword `in`.

The `addOrSubtract` function defined before could be rewritten to use closures instead of named functions:

```
func addOrSubtract(flag: Bool) -> (Int, Int) -> Int {
    if flag {
        return { (x: Int, y: Int) -> Int in return x + y }
    } else {
        return { (x: Int, y: Int) -> Int in return x - y }
    }
}

let fn = addOrSubtract(false)
let res = fn(4, 2)
// res is equal to 2
```

In many cases, the types of a closure's parameters can be inferred from context and can thus be omitted from the declaration. Furthermore, if a closure has only a single statement, the return keyword may be omitted as well. This leads to greater brevity when using closures, which is important because they are commonly used in Swift programming. The map function from before could be written and used as follows:

```
func map(fn: (Int) -> Int, ns: [Int]) -> [Int] {
    var res: [Int] = []
    for n in ns {
        let n1 = fn(n)
        res.append(n1)
    }
    return res
}

let nums = map({ x in x * 2 }, [1, 2, 3,])
// nums is now [2, 4, 6]
```

For even greater ease of writing, closures can also use shorthand parameter names in their bodies. The parameter names are prefixed with a dollar sign (`$`). Parameter names are numbered starting at 0. The first is referred to as `$0`, the second as `$1`, and so on. The `map` function declared previously could be called like this:

```
let nums = map({ $0 * 2 }, [1, 2, 3])
```

Swift offers an additional syntax for closures passed last to a function: In such cases, the closure can be specified *outside* of the parentheses in the function call. Because of this elegant syntax, functions that take a single function as a parameter often specify that closure as the final parameter. You can rewrite map so the function parameter is last:

```
func map(ns: [Int], fn: (Int) -> Int) -< [Int] {
    var res: [Int] = []
    for n in ns {
        let n1 = fn(n)
        res.append(n1)
    }
    return res
}
```

The call to map can then be written like this:

```
let nums = map([1, 2, 3]) { $0 * 2 }
```

Closures are an incredibly powerful part of Swift. They are similar to blocks in Objective-C but much more flexible—and, more importantly, they have a much nicer, easier-to-remember syntax than Objective-C's blocks. Utilizing the full power and potential of closures is a topic unto itself and will be discussed in more detail in Chapter 8.

SUMMARY

This chapter provided a crash course in Apple's new Swift programming language. While this was not intended to be a complete introduction to Swift, hopefully you were able to re-acquaint yourself with the rudiments of the language. If you are still unfamiliar with Swift, you may want to check out a more thorough guide on the basics of the language before moving on to the advanced concepts presented in the rest of this book.