Swift

```
var people = ["Dave", "Brian", "Alex", "A
let name = "Alex"
if let index = find(people, name) {
    println("\(name) is person \(index +
    delegate?.didFindPersonWithName(name,
} else {
    println("Unable to find \(name) in th
}
```

Welcome to Swift

About Swift

Swift is a new programming language for iOS and OS X apps that builds on the best of C and Objective-C, without the constraints of C compatibility. Swift adopts safe programming patterns and adds modern features to make programming easier, more flexible, and more fun. Swift's clean slate, backed by the mature and much-loved Cocoa and Cocoa Touch frameworks, is an opportunity to reimagine how software development works.

Swift has been years in the making. Apple laid the foundation for Swift by advancing our existing compiler, debugger, and framework infrastructure. We simplified memory management with Automatic Reference Counting (ARC). Our framework stack, built on the solid base of Foundation and Cocoa, has been modernized and standardized throughout. Objective-C itself has evolved to support blocks, collection literals, and modules, enabling framework adoption of modern language technologies without disruption. Thanks to this groundwork, we can now introduce a new language for the future of Apple software development.

Swift feels familiar to Objective-C developers. It adopts the readability of Objective-C's named parameters and the power of Objective-C's dynamic object model. It provides seamless access to existing Cocoa frameworks and mix-and-match interoperability with Objective-C code. Building from this common ground, Swift introduces many new features and unifies the procedural and object-oriented portions of the language.

Swift is friendly to new programmers. It is the first industrial-quality systems programming language that is as expressive and enjoyable as a scripting language. It supports playgrounds, an innovative feature that allows programmers to experiment with Swift code and see the results immediately, without the overhead of building and running an app.

Swift combines the best in modern language thinking with wisdom from the wider Apple engineering culture. The compiler is optimized for performance, and the language is optimized for development, without compromising on either. It's designed to scale from "hello, world" to an entire operating system. All this makes Swift a sound future investment for developers and for Apple.

Swift is a fantastic way to write iOS and OS X apps, and will continue to evolve with new features and capabilities. Our goals for Swift are ambitious. We can't wait to see what you create with it.

A Swift Tour

Tradition suggests that the first program in a new language should print the words “Hello, world” on the screen. In Swift, this can be done in a single line:

```
1  println("Hello, world")
```

If you have written code in C or Objective-C, this syntax looks familiar to you—in Swift, this line of code is a complete program. You don’t need to import a separate library for functionality like input/output or string handling. Code written at global scope is used as the entry point for the program, so you don’t need a `main` function. You also don’t need to write semicolons at the end of every statement.

This tour gives you enough information to start writing code in Swift by showing you how to accomplish a variety of programming tasks. Don’t worry if you don’t understand something—everything introduced in this tour is explained in detail in the rest of this book.

> NOTE
>
> For the best experience, open this chapter as a playground in Xcode. Playgrounds allow you to edit the code listings and see the result immediately.

Simple Values

Use `let` to make a constant and `var` to make a variable. The value of a constant doesn’t need to be known at compile time, but you must assign it a value exactly once. This means you can use constants to name a value that you determine once but use in many places.

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

A constant or variable must have the same type as the value you want to assign to it. However, you don't always have to write the type explicitly. Providing a value when you create a constant or variable lets the compiler infer its type. In the example above, the compiler infers that `myVariable` is an integer because its initial value is a integer.

If the initial value doesn't provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon.

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

EXPERIMENT

Create a constant with an explicit type of `Float` and a value of `4`.

Values are never implicitly converted to another type. If you need to convert a value to a different type, explicitly make an instance of the desired type.

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

EXPERIMENT

Try removing the conversion to `String` from the last line. What error do you get?

There's an even simpler way to include values in strings: Write the value in parentheses, and write a backslash (`\`) before the parentheses. For example:

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces of fruit."
```

EXPERIMENT

Use `\()` to include a floating-point calculation in a string and to include someone's name in a greeting.

Create arrays and dictionaries using brackets (`[]`), and access their elements by writing the index or key in brackets.

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

To create an empty array or dictionary, use the initializer syntax.

```
let emptyArray = String[]()
let emptyDictionary = Dictionary<String, Float>()
```

If type information can be inferred, you can write an empty array as `[]` and an empty dictionary as `[:]`—for example, when you set a new value for a variable or pass an argument to a function.

```
shoppingList = []   // Went shopping and bought everything.
```

Control Flow

Use `if` and `switch` to make conditionals, and use `for`-`in`, `for`, `while`, and `do`-`while` to make loops. Parentheses around the condition or loop variable are optional. Braces around the body are required.

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
   10  teamScore
```

In an `if` statement, the conditional must be a Boolean expression—this means that code such as `if score { ... }` is an error, not an implicit comparison to zero.

You can use `if` and `let` together to work with values that might be missing. These values are represented as optionals. An optional value either contains a value or contains `nil` to indicate that the value is missing. Write a question mark (`?`) after the type of a value to mark the value as optional.

```
var optionalString: String? = "Hello"
```

```
optionalString == nil

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

EXPERIMENT

Change `optionalName` to `nil`. What greeting do you get? Add an `else` clause that sets a different greeting if `optionalName` is `nil`.

If the optional value is `nil`, the conditional is `false` and the code in braces is skipped. Otherwise, the optional value is unwrapped and assigned to the constant after `let`, which makes the unwrapped value available inside the block of code.

Switches support any kind of data and a wide variety of comparison operations—they aren't limited to integers and tests for equality.

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let x where x.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(x)?"
default:
    let vegetableComment = "Everything tastes good in soup."
}
```

EXPERIMENT

Try removing the default case. What error do you get?

After executing the code inside the switch case that matched, the program exits from the switch statement. Execution doesn't continue to the next case, so there is no need to explicitly break out of the switch at the end of each case's code.

You use `for-in` to iterate over items in a dictionary by providing a pair of names to use for each key-value pair.

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
largest
```

EXPERIMENT

Add another variable to keep track of which kind of number was the largest, as well as what that largest number was.

Use `while` to repeat a block of code until a condition changes. The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

```
var n = 2
while n < 100 {
    n = n * 2
}
n

var m = 2
do {
    m = m * 2
} while m < 100
m
```

You can keep an index in a loop—either by using `..` to make a range of indexes or by writing an explicit initialization, condition, and increment. These two loops do the same thing:

```
var firstForLoop = 0
for i in 0..3 {
    firstForLoop += i
}
firstForLoop

var secondForLoop = 0
for var i = 0; i < 3; ++i {
    secondForLoop += 1
}
secondForLoop
```

Use `..` to make a range that omits its upper value, and use `...` to make a range that includes both values.

Functions and Closures