Session 4

Part A - Introduction to Swift

MODULE OBJECTIVE

At the end of this module, you will be able to:

- Explain the syntax and design features of Swift

SESSION OBJECTIVES

At the end of this session, you will be able to:

- Use the basic syntax and development environment for testing the Swift code
- Explain the basic data types, new tuples, and optional data types in Swift
- Describe the various operators, strings, and characters used in Swift
- Describe Swift backward compatibility with the `NSString` in Objective-C
- Explain how a function is defined and called in Swift
- Explain arrays and dictionaries for collection of values

INTRODUCTION

The IT world is an extremely fast-changing one. Small changes occur almost everyday, and every now and then, something big happens that changes the entire industry, if not the world. For example, the iPhone, introduced in 2007, literally transformed the mobile industry overnight, spearheading the new era of the smartphones. The launch of the iPad three years later (2010) changed the way we use our computers, causing many to predict that we are all entering the end of the personal computer (PC) era.

For a long time after its inception in the 1980s, Objective-C was used by a company called NeXT for its NeXTSTEP operating system. Mac OS X and iOS both derived from NeXTSTEP, and therefore Objective-C was the natural choice of the language to use for Mac OS and iOS development. Developers starting on iOS development often complain that Objective-C does not look like a modern programming language (such as Java or C#), and that it is difficult to write and requires considerable time to learn.

For seven years, Apple has improved on the language and the iOS framework, making life easier for developers by introducing helpful features, such as Automatic Reference Counting (ARC), which takes the drudgery out of memory management, and Storyboard, which simplifies the flow of your application user interface. However, this did not stop all the complaints. Furthermore, Apple needed a new language that could take iOS and Mac OS development to the next level.

In 2014, at the Apple World Wide Developers Conference (WWDC), Apple took many developers by surprise by introducing a new programming language called Swift. After seven years, Apple finally released a new language that can replace Objective-C!

Swift is a modern programming language with an easy-to-read syntax. It provides a playground with expressive and innovative features that will allow you to experiment with Swift code and see the result in real-time. It is a neat language with which you can make more attractive iOS apps. In this session, which comprises of two parts, you will start with the basic of Swift from “Hello world” to an innovative app.

INTRODUCTION TO SWIFT

Apple designed Swift for **Cocoa (Mac OS X)** and **Cocoa Touch (iOS)** programming. The aim of Swift is to replace Objective-C with much more modern language syntax without worrying too much about the constraints of C compatibility. Apple has touted Swift as Objective-C without the C!

The syntax of Swift is similar to modern languages, such as Java and C#, and retains some of the core features of Objective-C, such as named parameters, protocols, and delegates. The language’s clear syntax makes your code simpler to read and maintain.

Consider the following example where you add two numbers in Objective-C:

```
// passing parameter to add two no.
int sum = [self addOneNum: 2 withAnotherNum:7] ;

// Below method add two no. and return sum
(int) addOneNum:(int)  num1 withAnotherNum:(int) num2
{
    return num1 + num2 ;
}
```

The following example displays the above parameters in Swift:

```
// calling function
    var sum = addTwoNumbers ( 2, 5 )

// function defined to add two no.
func addTwoNumber ( num1:Int, num2:Int ) -> Int
```

```
    {
        return num1 + num2
    }
```

From the two examples, it is obvious that Swift's syntax is simpler and easier to read than Objective-C.

Swift is also designed to be a type-safe language. Variables must be initialized before use. In most cases, you have to perform explicit type conversions when assigning values from one type to another. Also, variables that are not assigned a value cannot be used in a statement and will be flagged as errors.

In Swift, there is no implicit type conversion for safety reasons—you must explicitly convert an `Int` to a `Float` (or `Double`). For example, you cannot implicitly assign an `Int` variable to a `Float` variable:

```
    var f : Float
    var i : Int  =  5
    f = i  //---error---
```

Instead, you need to explicitly convert the value into a `Float` value:

```
    f  =  Float ( i )
```

BIG Picture

Apple claims that the Swift code works in tandem with the Objective-C code. They are inter-compatible. The frameworks like Cocoa and Cocoa Touch can be used with Swift just as they are used with Objective-C.

Significance of Swift

The importance of Swift can be attributed to the fact that it has minimum possibility to make mistakes. Apple did not create Swift for the sake of creating a new programming language. With the platform wars heating up, Apple desperately needed a language that would enable it to secure its long-term lead in the mobile platform market. Swift is strategic to Apple in a number of ways, as shown below:

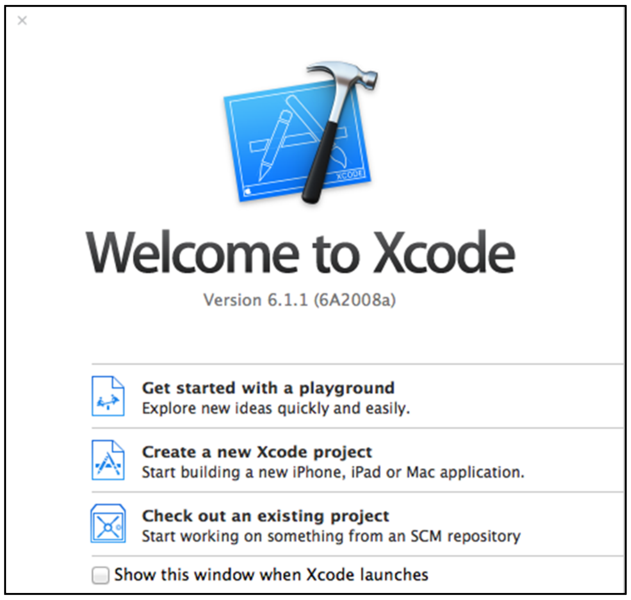
- It fixes many of the issues developers had with Objective-C—particularly, that Objective-C is hard to learn—replacing it with a language that is both fast to learn and easy to maintain.
- It delivers this easy-to-learn language while retaining the spirit of Objective-C, but without its verbose syntax.
- It is a much safer language than Objective-C, which contributes to a much more robust app platform.
- It is able to coexist with Objective-C, which gives developers ample time to port their code to Swift over time.

Real Life CONNECT

The Swift language will work along with Objective-C in your new apps. Swift is more concise, yet expressive and runs apps quickly. You will work on this language in your next projects for Cocoa and Cocoa Touch based iOS projects.

Setting up the Development Environment to Learn Swift

To test Swift code examples, you need a Swift compiler. The easiest way to obtain a Swift compiler is to download Xcode 6 from the Mac App Store and launch it. **Figure 1** shows the **Welcome** page of Xcode.



Welcome to Xcode

Version 6.1.1 (6A2008a)

Get started with a playground
Explore new ideas quickly and easily.

Create a new Xcode project
Start building a new iPhone, iPad or Mac application.

Check out an existing project
Start working on something from an SCM repository

Show this window when Xcode launches

Figure 1: Welcome Page of Xcode

There are two ways to test code in Swift:

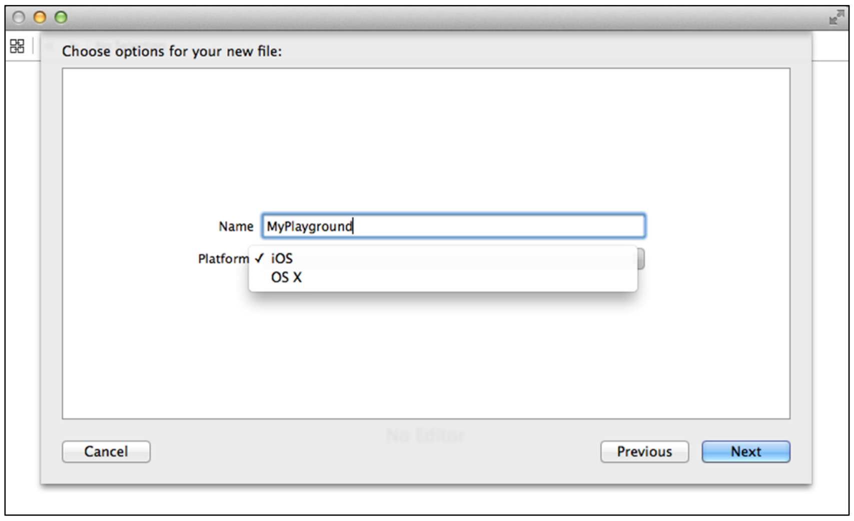
- Creating a playground project
- Creating an iOS project

Creating a Playground Project

Playground is a new feature of Xcode 6 that makes it easy and fun to learn Swift. As you enter each line of code, the playground will evaluate the line and display the results.

You can use the following steps to create a playground project:

1. Launch Xcode 6.
2. Select the **Get started with a playground** option on the **Welcome to Xcode** page (see **Figure 1**).
3. Name the playground project and select the platform you want to test it on (see **Figure 2**). Click **Next**.
4. A playground project will be created for the specified platform.



Choose options for your new file:

Name MyPlayground

Platform ✓ iOS
OS X

Figure 2: Select Options for Playground Project

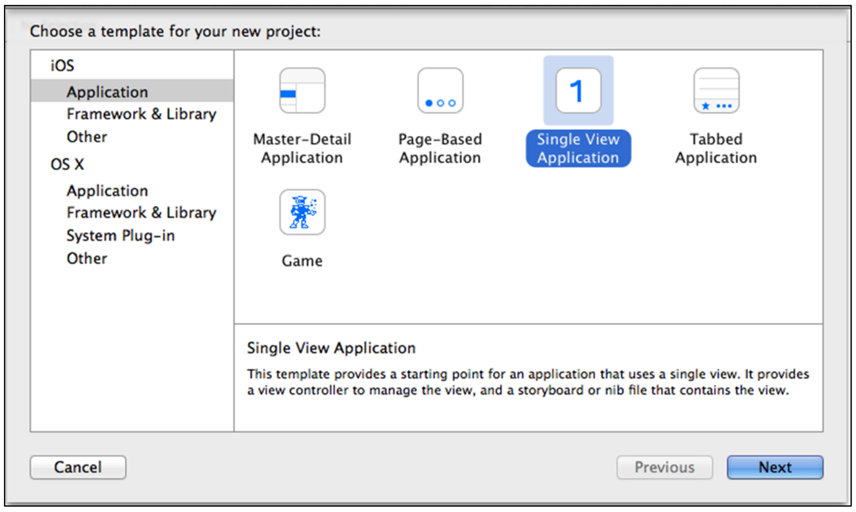
At the time of creating this session, Swift is available only for the iOS platform, while the version of Mac OS X is still in beta. Therefore, you will only be able to select the iOS platform.

Creating an iOS Project

An alternative to creating a playground project is to create an iOS project. You can create an iOS project and test your application using the iPhone simulator included in Xcode 6. While the focus of this session is on the Swift programming language, testing your code in an iOS project enables you to test your code in its entirety.

You can use the following steps to create an iOS project:

1. In Xcode 6, select **File ➢ New ➢ Project...**. This will display a dialog box (see **Figure 3**).



Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

Figure 3: Choose a Template for a Project

2. Select **Application** under the **iOS** category in the left pane and then select the **Single View Application** template. Click **Next**.
3. The **Single View Application** template creates an iPhone project with a single view window.

QUICK TIP

The **Single View Application** template creates an iPhone project with a single view window. This is the best template to use for learning Swift without getting bogged down with how an iOS application works.

The Basic Syntax in Swift

Let's begin with creating constants and variables in Swift. They are created using the `let` keyword, as shown below:

```
// Constants
let radius = 3.45              // Radius is a Double
let numOfColumns = 5           // numOfColumns is an Int
                     let myName = "Wei-Meng Lee"  // myName is a String
```

In Swift, data types are not specified because they are inferred automatically.

```
// Variables
let radius = 3.4
var myAge = 25
var circumference = 2 * 3.14 * radius
```

Once a variable is created, you can change its value:

```
let diameter = 20.5
```

In Objective-C, the string literal is defined as follows:

```
NSString *myName = @”Wei-Meng Lee”    // “@” is needed before a string in Obj-C
```

However, it is not needed in Swift.

```
let myName = "Wei-Meng Lee"         // --Swift–
```

The `string` type in Swift is a primitive (value) type, whereas the `NSString` type in Objective-C is a reference type (object). Strings are discussed in detail later in this session.

One of the dreaded tasks in Objective-C is inserting values of variables in a string. You have to use the `NSString` class and its associated `stringWithFormat:` method to perform string concatenation, which makes your code really long. In Swift, this is an easy task by using the `\()` syntax, known as string interpolation. It has the following format:

```
 “Your string literal \(variable_name)”
```

The following example shows how:

```
let myName = "Wei-Meng Lee"
var strName = “My name is \(myName)”
```

You can use this method to include a `double` value in your string as shown below:

```
var strResult = "The circumference is \(circumference)"
```

In Swift, you can print the current values of variables or constants using the `println()` or `print()` function. The `print()` function prints a value, whereas the `println()` function prints a value as well as a line break. The `println()` and `print()` functions are similar to Cocoa's `NSLog` function of Objective-C:

```
  var strMyAge = "My age is " + String(myAge)
      println(strMyAge)             // prints strMyAge
```

In Swift, each statement does not end with a semicolon (;). If you want to include semicolons at the end of each statement, it is syntactically correct but not necessary. However, it is required when you combine multiple statements into a single line, as shown below:

```
let radius = 3.45; let numOfColumns = 5; let myName = “Wei-Meng Lee”;
```

In Objective-C, you also need to use '*' to indicate memory pointers whenever you are dealing with objects. In Swift, there is no need to use the '*', regardless of whether you are using objects or primitive types.

ADDITIONAL KNOWHOW

In languages such as C and Java, you are not allowed to nest comments. In Swift, comments can be nested. You can use // to comment a single line, or the /* and */ combination to comment a block of statements.

DATA TYPES IN SWIFT

Swift supports various basic data types that are available in most programming languages. These data types include:

- Integers
- Floating-point numbers
- Booleans

Swift also introduces new data types, which are not available in Objective-C. These new data types include:

- Tuples
- Optional types

In this section, you will first learn about the basic data types supported in Swift, followed by the new types. You will also learn about how to perform operations Swift as well as enumerations.

Integers

In Swift, integers are represented by using the `Int` type. The `Int` type represents both positive as well as negative values. If you only need to store positive values, you can use the unsigned integer `UInt` type.

```
println("Size of Int: \(sizeof(Int)) bytes")    // Int Store positive and negative both
println("Size of UInt: \(sizeof(UInt)) bytes")  // UInt Store only positive values
```

You can find the number of bytes stored by each data type using the `sizeof()` function:

```
// In iPhone 5s (which uses the 64-bit A7 chip)
Size of Int: 8 bytes
Size of UInt: 8 bytes

// In iPhone 5s (which uses the 32-bit A6 chip)
Size of Int: 4 bytes
Size of UInt: 4 bytes
```

The size of an `Int` type depends on the system on which your code is running. On 32-bit systems, `Int` and `UInt` use 32 bits for storage, whereas on 64-bit systems `Int` and `UInt` use 64 bits.

If you do not know the type of data a variable is storing, you can use the `sizeofValue()` function:

```
var num = 5
println("Size of num: \(sizeofValue(num)) bytes")
```

In Swift, various integer types are available for storing signed and unsigned numbers. However, you may want to explicitly control the size of the variable used in `Int` types, as shown below:

- `Int8` and `UInt8`
- `Int16` and `UInt16`
- `Int32` and `UInt32`
- `Int64` and `UInt64`

On 32-bit systems, `Int` is the same as `Int32`, while on 64-bit systems, `Int` is the same as `Int64`.

On 32-bit systems, `UInt` is the same as `UInt32`, while on 64-bit systems, `UInt` is the same as `UInt64`.

The following code snippet prints the range of numbers representable for each integer type:

```
//---UInt8  - Min: 0 Max: 255---
println("UInt8  - Min: \(UInt8.min) Max: \(UInt8.max)")

//---UInt16 - Min: 0 Max: 65535---
println("UInt16 - Min: \(UInt16.min) Max: \(UInt16.max)")

//---UInt32 - Min: 0 Max: 4294967295---
println("UInt32 - Min: \(UInt32.min) Max: \(UInt32.max)")

//---UInt64 - Min: 0 Max: 18446744073709551615---
println("UInt64 - Min: \(UInt64.min) Max: \(UInt64.max)")

//---Int8  - Min: -128 Max: 127---
println("Int8  - Min: \(Int8.min) Max: \(Int8.max)")

//---Int16 - Min: -32768 Max: 32767---
println("Int16 - Min: \(Int16.min) Max: \(Int16.max)")

//---Int32 - Min: -2147483648 Max: 2147483647---
println("Int32 - Min: \(Int32.min) Max: \(Int32.max)")

//---Int64 - Min: -9223372036854775808 Max: 9223372036854775807---
println("Int64 - Min: \(Int64.min) Max: \(Int64.max)")
```

For each integer type, the `min` property returns the minimum number representable and the `max` property returns the maximum number representable.

Floating-Point Numbers

Floating-point numbers are numbers with fractional parts. Examples of floating-point numbers are 0.0123, 2.45, and -4.521.

In Swift, there are two floating-point types:

- `Float`: It uses 32 bits for storage. `Float` has a precision of at least six decimal digits.
- `Double`: It uses 64 bits. `Double` has a precision of at least 15 decimal digits.

When assigning a floating-point number to a constant or variable, Swift will always infer the `double` type unless you explicitly specify otherwise:

```
var num1 = 3.14            //---num1 is Double---
var num2: Float = 3.14     //---num2 is Float---
```

Booleans

Swift supports the Boolean logic type—`Bool`. A `Bool` type can take either a `true` or `false` value. This is unlike Objective-C, in which a Boolean value can be `YES` or `NO`. The `Bool` values in Swift are similar to most programming languages like Java and C.

The following code snippet shows the `Bool` type in use:

```
var skyIsBlue = true
var seaIsGreen = false
var areYouKidding:Bool = true

skyIsBlue = !true    //---skyIsBlue is now false---
println(skyIsBlue)   //---false---
```

`Bool` variables are often used in conditional statements, such as the `If` statement:

```
if areYouKidding {
   println("Just joking, huh?")
} else {
   println("Are you serious?")
}
```

How to Perform Operations

`Int` operations are performed on the same integer types. If you try to add two numbers of different integer types, you will get an error. Consider the following example:

```
var i1: UInt8 = 255
var i2: UInt16 = 255
var i3 = i1 + i2       //---cannot add two variables of different types---
```

To fix this, you need to typecast one of the types to be the same as the other type:

```
var i3 = UInt16(i1) + i2  //---i3 is now UInt16---
```

When you add an integer constant to a `double`, the resultant type would also be a `Double` type. Likewise, when you add an integer constant to a `float`, the resultant type would also be a `Float` type, as the following examples illustrate:

```
var sum1 = 5 + num1   //---num1 and sum1 are both Double---
var sum2 = 5 + num2   //---num2 and sum2 are both Float---
```

However, if you try to add `Int` and `Double` variables, you will get an `error`:

```
var i4: Int = 123
var f1: Double = 3.14567
var r = i4 + f1          //---error---
```

To add two variables of different types, you need to cast the `Int` variable to a `Double`:

```
var r = Double(i4) + f1
```

When you add an integer to a floating-point number, the result would be a `Double` value. For example:

```
var someNumber = 5 + 3.14
```

In the above statement, `someNumber` would be inferred to be a `Double`.

For safety reasons, there is no implicit type conversion in Swift—you must explicitly convert an `Int` to a `Float` (or `Double`):

```
var f:Float
var i:Int = 5
f = i  //---error---
f = Float(i)
```

When you cast a floating-point value to an integer, the value is always truncated—that is, you will lose its fractional part:

```
var floatNum = 3.5
var intNum = Int(floatNum)  //---intNum is now 3---
```

Technical Stuff

When you assign a `Double` to a `Float` type, the compiler gives an error because the number stored in a `Double` type is not able to fit into a `Float` type, thereby resulting in an overflow. Therefore, to assign two values, you need to explicitly cast a `Double` to a `Float`:

```
num2 = num1            //---num1 is Double and num2 is Float---
num2 = Float(num1)     //---cast Double to Float---
```

Tuples

A **tuple** is a group of related values that can be manipulated as a single data type. Tuples are very useful when you need to return multiple values from a function.

It is not necessary that the values inside a tuple are of the same type; they can be of any type. For example, you want to store the coordinates of a point in the following coordinate space:

```
var x = 7
var y = 8
```

The above example uses two variables to store the x and y coordinates of a point. Because these two values are related, it is much better to store them together as a tuple instead of two individual integer variables, as shown below:

```
var pt = (7,8)  // value stored as a tuple
var pt: (Int, Int)     //pt is tuple
pt = (7,8)             //value stored in tuple pt
```

In the above example, the `point` is a tuple of type `(Int, Int)`.

Some more examples of tuples are as follows:

```
var flight = (7031, "ATL", "ORD")        //tuple of type (Int, String, String)
var phone = ("Chloe", "732-757-2923")    //tuple of type (String, String)
```

If you are not interested in values within the tuple, then use the underscore (_) character in place of variables or constants, as shown below:

```
let (flightno, _, _) = flight
println(flightno)
```

Alternatively, you can also access individual values inside a tuple by using an index, starting from 0:

```
println(flight.0)   //---7031---
println(flight.1)   //---ATL---
println(flight.2)   //---ORD---
```

Using an index to access individual values inside a tuple is not intuitive. A better way is to name the individual elements inside the tuple:

```
var flight = (flightno:7031, orig:"ATL", dest:"ORD")
```

Once the individual elements are named, you can access them using those names:

```
println(flight.flightno)
println(flight.orig)
println(flight.dest)
```

Optional Types

Swift uses a new concept known as optional. An **optional type** specifies a variable that can contain no value. Optional types make your code safer.

To understand this concept, consider the following code snippet:

```
let str = "125"        //str is string
let num = str.toInt()  //toInt() converts string to integer
```

Conversion from `Int` to a string may not be always successful. This is because the string may contain characters that cannot be converted to a number and the result returned to `num` may be an `Int` value or `nil`. Therefore, by type inference, `num` is assigned a type of `Int?`. The ? character indicates that this variable can *optionally* contain a value—it might not contain a value at all if the conversion is not successful.

In the preceding code snippet, any attempt to use the `num` variable (such as multiplying it with another variable/constant) will result in a compiler error:

```
”value of optional type 'Int?' not unwrapped; did you mean to use”.
!' or '?'?”:
let multiply = num * 2  //error
```

To fix this, you should use the `If` statement to determine whether `num` contains a value. If it does, you need to use the ! character after the variable name to use its value, as shown below:

```
let str = "125"
let num = str.toInt()
if num != nil {
   let multiply = num! * 2
   println(multiply)    //---250---
}
```

The ! character indicates to the compiler that you know the variable contains a value and you indeed know what you are doing. The use of the ! character is known as **forced unwrapping of an optional’s value**.

Enumerations

An **enumeration** is a user-defined type consisting of a group of named constants. For example, you want to create a variable to store the color of a bag. You can store the color as a string, as shown below:

```
var colorOfBag = "Black"
colorOfBag = "Yellow"        //--color can be changed as yellow—
```

However, this approach is not safe, as there are two potential pitfalls:

- The color may be set to a color that is invalid. For example, a bag's color can only be Black or Green. If the color is set to Yellow, your code will not be able to detect it.
- The color specified might not be the same case as you expected. If your code expected "Black" and you assigned "black" to the variable, your code might break.

In either case, it is always better to be able to define your own type to represent all the different colors of a bag. In that case, you create an enumeration containing all the valid colors. The following code snippet defines an enumeration named `BagColor`:

```
enum BagColor {
   case Black
   case White
   case Red
   case Green
   case Yellow
}
```

The `BagColor` enumeration contains five cases (also known as members): Black, White, Red, Green, and Yellow. Each member is declared using the `case` keyword. You can also group these five separate cases into a single case, separated by using commas (,), as shown below:

```
enum BagColor {
   case Black, White, Red, Green, Yellow
}
var colorOfBag:BagColor             // variable declare
colorOfBag = BagColor.Yellow        // assign value to variable
```

In Swift, you need to specify the enumeration name followed by its member. This is different from Objective-C, for which you just need to specify the member name, such as `UITableViewCellAccessoryDetailDisclosureButton`. The approach in Swift makes the code more comprehensible.

Enumerations are often used in Switch statements. The following code snippet checks the value of `colorOfBag` and outputs the respective statement:

```
switch colorOfBag {
   case BagColor.Black:
       println("Black")
   case BagColor.White:
       println("White")
   case BagColor.Red:
       println("Red")
   case BagColor.Green:
```

```
            println("Green")
        case BagColor.Yellow:
            println("Yellow")
```

You can define a function within an enumeration. The following code snippet adds a function named info to the `DeviceType` enumeration:

```
enum DeviceType {
  case Phone (NetworkType, String)
  case Tablet(String)
  var info: String {
      switch (self) {
          case let .Phone (networkType, model):
              return "\(networkType.rawValue) - \(model)"
          case let .Tablet (model):
              return "\(model)"
      }   }        }
```

In the above code snippet, the `info()` function returns a string. It checks the member that is currently selected (using the `self` keyword) and returns either a string containing the network type and model (for phone) or simply the model (for tablet). To use the function, simply call it with the enumeration instance, as shown below:

```
println(device1.info)  //---LTE - iPhone 5S---
println(device2.info)  //---iPad Air---
```

Technical Stuff

You can make explicitly-typed enumeration values more readable by omitting the enumeration name by a dot and its member name:

```
var colorOfBag:BagColor      // variable declare
colorOfBag = BagColor.Yellow // assign value to variable
//---or ---
colorOfBag = .Yellow         //--Short form---
```

OPERATORS, STRINGS, AND CHARACTERS IN SWIFT

Common and New Operators in Swift

Operators in Swift work with various data types to enable you to make logical decisions, perform arithmetic calculations, and change values. Swift supports the following types of operators:

- Assignment
- Comparison
- Logical
- Arithmetic
- Range
- Nil coalescing

BIG Picture

Swift mostly supports standard operators from the C language and improves common coding errors. For example, using the assignment operator (=) in place of the equal to operator (==) does not return any value to prevent this code error. Swift also provides arithmetic operators (+, -, *, /, %) to detect and disallow overflow of values when working with numbers.

Assignment Operator

The **assignment operator (=)** sets a variable or a constant to a value.

In Swift, you can create a constant by assigning a value to a constant name, as shown below:

```
let companyName ="Developer Learning Solutions"
      let factor = 5
```

You can also create a variable using the assignment operator:

```
var customerName1 = "Richard"
```

In addition to assigning a value to a variable or a constant, you can also assign a variable or a constant to another:

```
var customerName2 = customerName1
```

You can also assign a tuple directly to a variable or a constant:

```
let pt1 = (3,4)
```

You can decompose the value of a tuple into multiple variables or constants by using the assignment operator:

```
let (x,y) = (5,6)
println(x)   //---5---
println(y)   //---6---
```

Unlike Objective-C, the assignment operator does not return a value in Swift. Therefore, you cannot do something as follows:

```
   if num = 5 {               //---error---
           ...
       }
```

This is a good feature, as it prevents programmers from accidentally doing an assignment instead of an equality comparison.

Arithmetic Operators

Swift supports the following arithmetic operators:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)
- Modulus (`%`)

Swift requires the operands in all arithmetic operations to be of the same type. This enforces type safety, as it requires you to explicitly perform typecasting.

Consider the following statements:

```
var a = 9     //---Int---
var b = 4.1   //---Double---
```

Using the type inference, `a` is `Int` and `b` is `Double`.

Because they are of different types, the following operations are not allowed:

```
println(a * b)  //---error---
println(a / b)  //---error---
println(a + b)  //---error---
println(a – b)  //---error---
```

You need to convert the variables to be of the same type before you can perform arithmetic operations on them.

Addition Operator (+)

The **addition operator (+)** adds two numbers together.

When adding numeric values, you should know some important subtleties, such as:

```
println(5 + 6)      //---integer addition (11)--
println(5.1 + 6)      //---double addition (11.1)--
println(5.1 + 6.2)    //---double addition (11.3)—
```

Subtraction Operator (−)

The **subtraction operator (−)** enables you to subtract one number from another.

As with the addition operator, you need to be aware of its behavior when subtracting two numbers of different types. Some important subtleties are shown below:

```
println(7 – 8)        //---integer subtraction (-1)---
println(9.1 – 5)      //---double subtraction (4.1)---
println(9 – 4.1)      //---double subtraction (4.9)---
```

Multiplication Operator (*)

The **multiplication operator (*)** multiplies two numbers.

Like the addition and subtraction operators, multiplying numbers of different types yields results different types, as shown below:

```
println(3 * 4)        //---integer multiplication (12)---
println(3.1 * 4)      //---double multiplication (12.4)---
println(3.1 * 4.0)    //---double multiplication (12.4)---
```

Division Operator (/)

The **division operator (/)** divides a number by another number.

Dividing an integer by another integer will return only the integer part of the result:

```
println(5 / 6)        //---integer division (0)---
println(6 / 5)        //---integer division (1)---
```

Dividing a double by an integer will return a double:

```
println(6.1 / 5)      //---double division (1.22)---
println(9.99 / 5)     //---double division (1.998)---
```

Dividing a double by a double will return a double:

```
println(6.1 / 5.5)   //---double division (1.10909090909091)---
```

Modulus Operator (`%`)

The **modulus operator (`%`)** returns the remainder of a division.

```
println(8 % 9)      //---modulo (8)---
println(9 % 8)      //---modulo (1)---
println(-5 % 3)     //---module (-2)---
```

A negative value for the second number is always ignored.

```
println(-5 % -3)    //---module (-2)---
```

The modulus operator also works with double values:

```
println(5 % 3.5)    //---module (1.5)---
println(5.9 % 3.5)  //---module (2.4)---
```

It is a common task in programming to increment or decrement the value of a variable by one. Swift provides the increment (`++`) and decrement (`--`) operators as shortcuts to these operations. For example, if you want to increment the value of a variable by one, you typically use the following code:

```
var i = 5
i = i + 1  //---i is now 6---
```

However, if you use the increment operator, you can rewrite the above code as follows:

```
var i = 5
++i        //---i is now 6---
```

Both the increment and decrement operators can be used as either a prefix or a postfix operator. Let's take a look at the increment operator:

```
i = 5
++i              //---i is now 6---
i++              //---i is now 7---
i = 5            //---i is now 5
var j = i++   //---j is now 5, i is now 6---
println(i)    //---6---
println(j)    //---5---
```

Now, you can use `++` as the prefix operator `?`.

Consider the following example:

```
i = 5
j = ++i       //---both i and j are now 6---
println(i)    //---6---
println(j)    //---6---
```

The same behavior of the postfix and prefix operators applies to the -- operator as well, as the following code snippet shows:

```
i = 5
j = i--         //---j is now 5, i is now 4---
println(i)      //---4---
println(j)      //---5---
i = 5
j = --i         //---both i and j are now 4---
println(i)      //---4---
println(j)      //---4---

var size = 2
size *= 3          //---size is now 6---

var width = 100
width /= 2         //---width is now 50---
```

ADDITIONAL KNOWHOW

The modulus operator is known as the remainder operator (%) in other languages, such as Java and C++. In Swift, its behavior for negative numbers means that it is, strictly speaking, a remainder rather than a modulus operation.

Comparison Operators

Swift supports standard comparison operators that are available in most programming languages. These comparison operators are:

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

Equal To (==) and Not Equal To (!=) Operators

You can use the equal to (==) operator to check for the equality of two variables. The "==" operator works with numbers as well as strings.

Consider the following example:

```
var n = 6
if n % 2 == 1 {
   println("Odd number")
} else {
   println("Even number")
}
```

The following example shows the == operator comparing string values:

```
var status = "ready"
if status == "ready" {
   println("Machine is ready")
} else {
   println("Machine is not ready")
}
```

In addition to the == operator, you can also use the not equal to (!=) operator.

The following code snippet shows the earlier example rewritten using the != operator:

```
   var n = 6
   if n % 2 != 1 {
      println("Even number")
   } else {
      println("Odd number")
   }
```

Likewise, you can also use the != operator for string comparisons:

```
   var status = "ready"
   if status != "ready" {
      println("Machine is not ready")
   } else {
      println("Machine is ready")
   }
```

The == and != operators also work with character types:

```
   let char1:Character = "A"
   let char2:Character = "B"
   let char3:Character = "B"
   println(char1 == char2)       //---false---
   println(char2 == char3)       //---true---
   println(char1 != char2)       //---true---
   println(char2 != char3)       //---false---
```

Greater Than (>) and Greater Than or Equal To (>=) Operators

The greater than (>) operator is used to determine whether a number is greater than another number. Consider the following examples:

```
   println(5 > 5)  //---false--
   println(6 > 5)  //---true---
```

You can use the greater than or equal to (>=) operator to determine whether a number is greater than or equal to another number. For example:

```
   println(7 >= 7) //---true---
   println(7 >= 8) //---false—
```

The > and >= operators do not work with the `String` type.

Less Than (<) and Less Than or Equal To (<=) Operators

You can use the less than (<) operator to determine whether a number is less than another number.

```
println(4 < 4)  //---false---
println(4 < 5)  //---true—
```

You can also use the less than or equal to (<=) operator to determine whether a number is less than or equal to another number:

```
println(8 <= 8) //---true---
println(9 <= 8) //---false—
```

The < operator can work with strings:

```
println("abc" < "ABC") //---false---
println("123a" < "123b") //---true---
```

However, the <= operator does not work with the `String` type.

QUICK TIP

Using the assignment operator (=) within a condition is a major source of bugs in C programs. Programmers often intend to use the comparison operator (==), but mistakenly use the assignment operator; and so long as they assign a non-zero value, the condition always evaluates to `true`.

Range Operators

Swift supports two types of range operators to specify a range of values:

- **Closed Range Operator (a...b):** It specifies a range of values starting from `a` right up to `b` (inclusive).
- **Half-Open Range Operator (a..<b):** It specifies a range of values starting from `a` right up to `b`, but not including `b`.

To demonstrate how these range operators work, consider the following example:

```
//---prints 5 to 9 inclusive---
for num in 5...9 {
   println(num)
}
```

The preceding code snippet uses the closed range operator to output all the numbers from 5 to 9:

```
5
6
7
8
9
```

To output only 5 to 8, you can use the half-open range operator:

```
//---prints 5 to 8---
for num in 5..<9 {
   println(num)
}
```

The preceding code snippet outputs 5 to 8:

```
5
6
7
8
```

The half-open range operator is particularly useful when you are dealing with zero-based lists, such as arrays. The following code snippet is a good example:

```
//---useful for 0-based lists such as arrays---
var fruits = ["apple","orange","pineapple","durian","rambutan"]
for n in 0..<fruits.count {
   println(fruits[n])
}
```

The preceding code snippet outputs the following:

```
apple
orange
pineapple
durian
rambutan
```

Logical Operators

Like most programming languages, Swift supports the following three logical operators:

- Logical NOT (!)
- Logical AND (&&)
- Logical OR (||)

Logical NOT (!) Operator

The **logical NOT (!) operator** inverts a `Bool` value so that `true` becomes `false` and `false` becomes `true`.

Table 1 shows how the NOT operator works on values:

Table 1: NOT Operator Values

a	**!a**
true	false
false	true

Consider the following statements:

```
var happy = true        //happy is of type BOOL and is set to true
happy = !happy          //used logical NOT operator to invert
```

Logical AND (&&) Operator

The **logical AND (&&) operator** creates a logical expression (`a && b`) where both `a` and `b` must be `true` in order to evaluate to `true`.

Table 2 shows how the AND operator works on two values. As you can see, both `a` and `b` must be `true` for the expression to be `true`.

Table 2: AND Operator Values

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Consider the following example:

```
   var happy = true
   var raining = false
   if happy && !raining {
      println("Let's go for a picnic!")
   }
   // Line "Let's go for a picnic!" will only be printed if you are happy and it is not
raining.
```

Swift does not need to wrap an expression using a pair of parentheses (which is required in Objective-C). However, you can always add it for readability, as shown below:

```
   if (happy && !raining) {
      println("Let's go for a picnic!")
   }
```

Swift supports **short-circuit evaluation** for evaluating the AND expression. If the first value is `false`, the second value will not be evaluated because the logical AND operator requires both values to be `true`.

Logical OR (||) Operator

The **logical OR (||) operator** creates a logical expression (`a || b`) where either `a` or `b` needs to be `true` in order to evaluate to `true`.

Table 3 shows how the OR operator works on two values. As you can see, as long as either `a` or `b` is `true`, the expression evaluates to `true`.

Table 3: OR Operator Values

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Consider the following example:

```
   var age = 131
   if age > 130 || age < 1 {
      println("Age is out of range")
   }
```

In the preceding example, the line "`Age is out of range`" will be printed if age is more than 130 or less than 1. In this case, the line is printed, as age is 131.

Often, you will use the `If-Else` statement to write simple statements such as:

```
var day = 5
var openingTime:Int
if day == 6 || day == 7 {
   openingTime = 12
 } else {
   openingTime = 9
}
```

In the preceding code snippet, you want to know the opening time (`openingTime`) of a store based on the day of a week (`day`). If it is Saturday (`6`) or Sunday (`7`), then the opening time is `12:00` noon; otherwise on the weekday it is `9:00` A.M. You can reduce such a statement by using the ternary conditional operator, as shown below:

```
openingTime = (day == 6 || day == 7) ? 12: 9
```

The syntax of the ternary conditional operator is as follows:

```
variable = condition ? value_if_true : value_if_false
```

The ternary conditional operator first evaluates the condition. If the condition evaluates to `true`, the `value_if_true` is assigned to a variable. Otherwise, the `value_if_false` is assigned to a variable.

Nil Coalescing Operator

Swift has introduced a new **nil coalescing operator**, which has the following syntax:

```
a ?? b
```

Consider the following optional variable:

```
var gender:String?
```

The `gender` variable is an optional variable that can take a `string` value or a `nil` value. Suppose you want to assign the value of `gender` to another variable. If it contains `nil`, you need to assign a default value to the variable. The code will be as follows:

```
var genderOfCustomer:String
       if gender == nil {
           genderOfCustomer = "male"
       } else {
           genderOfCustomer = gender!
       }
```

Here, you check whether the `gender` is `nil`. If it is `nil`, then you assign a default value of "`male`" to `genderOfCustomer`. If it is not `nil`, then its value is assigned to `genderOfCustomer`.

Now, if you use the Nil coalescing operator, the code will be as follows:

```
var gender:String?
var genderOfCustomer = gender ?? "male"    //---male---
```

The above code reads "unwrap the value of optional `a` and return its value if it is not `nil`; otherwise return `b`." Because the gender is `nil`, the `genderOfCustomer` is now assigned `male`.

If you now assign a value to the `gender` and execute the preceding statements again, the `gender` would be `female`:

```
var gender:String? = "female"
var genderOfCustomer = gender ?? "male"   //---female---
```

Managing Strings and Characters in Swift

In Swift, a string literal is a sequence of characters enclosed by a pair of double quotes (""). The following code snippet shows a string literal assigned to a constant and another to a variable:

```
let str1 = "This is a string in Swift"        //---str1 is a constant---
var str2 = "This is another string in Swift"  //---str2 is a variable---
```

Because the compiler uses the type inference, there is no need to specify the type of constant and variable assigned to the string. However, if required, you can still specify the string type explicitly, as shown below:

```
var str3:String = "This is yet another string in Swift"
```

To assign an empty string to a variable, you can simply use a pair of empty double quotes, or call the initializer of the `string` type, as shown below:

```
var str4 = ""
var str5 = String()
```

The preceding statements initialize both `str4` and `str5` to contain an empty string. To check whether a variable contains an empty string, use the `isEmpty()` method of the `string` type, as shown below:

```
if str4.isEmpty {
println("Empty string")
}
```

You can concatenate strings in Swift using the addition (+) operator, as shown below:

```
var hello = "Hello"
var comma = ","
var world = "World"
var exclamation = "!"
var space = " "
var combinedStr = hello + comma + space + world + exclamation
println(combinedStr)  //---Hello, World!---
```

You can also use the addition assignment operator (+=) to append a string to another string:

```
var hello = "Hello"
hello += ", World!"
println(hello)   //---Hello, World!"
```

You can also compare two strings or characters using the equal to operator (==) or the not equal to operator (!=). Two strings are equal if they contain exactly the same Unicode scalars in the same order.

For example:

```
var string1 = "I am a string!"
var string2 = "I am a string!"
println(string1 == string2)   //--true--
println(string1 != string2)   //--false—
```

The following example compares two character variables, each containing a Unicode character:

```
var s1 = "é"        //---é---
var s2 = "\u{E9}"  //---é---
println(s1 == s2)  //---true---
```

The following example compares two string variables, each including a Unicode character:

```
var s3 = "café"       //---café---
var s4 = "caf\u{E9}" //---café---
println(s3 == s4)    //---true---
```

Defining a String and Characters

In Swift, a string is a value type and is made up of characters. Therefore, when you assign a string to another variable or pass a string into a function, a copy of the string is always created.

Consider the following code snippet:

```
var originalStr = "This is the original"
var copyStr = originalStr
```

In the preceding example, `originalStr` is initialized with a string literal and then assigned to `copyStr`. A copy of the string literal is copied and assigned to `copyStr`.

If you output the values of both variables, you can see that both variables output the same string literal:

```
println(originalStr)  //---This is the original---
println(copyStr)      //---This is the original---
```

Now, let’s change the `copyStr` variable by assigning another string literal to it:

```
copyStr = "This is the copy!"
```

In the above example, `copyStr` is now assigned another string. To prove this, you can output the values of both variables as follows:

```
println(originalStr)
println(copyStr)
```

The preceding code snippet will output the following:

```
This is the original
This is the copy!
```

The **mutability of a string** means whether it can be modified after it has been assigned to a variable. A string's mutability depends on whether it is assigned to a constant or a variable.

A string that is assigned to a variable is mutable, as the following example shows:

```
var myName = "Wei-Meng"
myName += " Lee"
println(myName)  //---Wei-Meng Lee---
```

A string that is assigned to a constant is immutable (that is not mutable—its value cannot be changed):

```
let yourName = "Joe"
yourName += "Sim"     //---error---
yourName = "James"    //---error---
```

A string is made up of characters. You can iterate through a string and extract each character using the `For-In` loop. The following code snippet shows an example:

```
var helloWorld = "Hello, World!"
for c in helloWorld {
   println(c)
}
```

The preceding statements output the following:

```
H
e
l
l
o
,
W
o
r
l
d
!
```

The `For-In` loop works with Unicode characters as well:

```
var hello = "您好"  //---hello contains two Chinese characters---
for c in hello {
   println(c)
}
```

The preceding code snippet outputs the following:

```
您
好
```

By default, the compiler will always use the string type for a character enclosed with double quotes. For example, `euro` is inferred to be of the `string` type in the following statement:

```
var euro = "€"
```

However, if you want `euro` to be the `character` type, you have to explicitly specify the `character` type:

```
var euro:Character = "€"
```

You can append a character to a string, but not vice-versa. This is because the `character` type can only hold a single character:

```
var euro:Character = "€"
var price = euro + "2500"  //---€2500---
euro += "2500"             //---error---
```

How to Use the Various Special String Characters

String literals can contain one or more characters that have a special meaning in Swift. These characters include:

- Double quote (`"`)
- Single quote (`'`)
- Backslash (`\`)

Double Quote (")

If you want to represent the double quote (`"`) within a string, prefix the double quote with a backslash (`\`):

```
var quotation = "Albert Einstein: \"A person who never made a mistake never tried
   anything new\""
println(quotation)
```

The preceding statement outputs the following:

```
Albert Einstein: "A person who never made a mistake never tried anything new"
```

Single Quote (')

If you want to represent the single quote (`'`) within a string, simply include it in the string:

```
var str = "'A' for Apple"
println(str)
```

The preceding statement outputs the following:

```
'A' for Apple
```

Backslash (\)

If you want to represent the backslash (\) within a string, prefix the backslash with another backslash (\):

```
var path = "C:\\WINDOWS\\system32"
println(path)
```

The preceding statement outputs the following:

```
C:\WINDOWS\system32
var headers = "Column 1\tColumn 2\tColumn3"    // \t represents a tab character:
println(headers)
```

The preceding statement outputs the following:

```
Column 1     Column 2     Column3
var column1 = "Row 1\nRow 2\nRow 3" // \n represents a newline character
println(column1)
```

The preceding statement outputs the following:

```
Row 1
Row 2
Row 3
```

How to Use Unicode Characters in Swift

In Swift, a character represents a single extended grapheme cluster. An **extended grapheme cluster** is a sequence of one or more Unicode scalars, which when combined produces a single human-readable character. Consider the following example:

```
let hand:Character = "\u{270B}"
let star = "\u{2b50}"
let bouquet = "\u{1F490}"
```

In the preceding code snippet, the three variables are of type `Character`, with the first one explicitly declared. Their values are assigned by using single Unicode scalars. Each Unicode scalar is a unique 21-bit number.

Here is another example:

```
let aGrave = "\u{E0}"   //---à---
```

In the preceding statement, `aGrave` represents the Latin small letter “a” with a grave, `à`. You can rewrite the same statement using a pair of scalars—the letter a followed by the Combining Grave Accent scalar:

```
let aGrave = "\u{61}\u{300}"
```

In either case, the `aGrave` variable contains **one** single character. For example, consider the following statement:

```
var voila = "voila"
```

In the preceding statement, `voila` contains five characters. If you append the Combining Grave Accent scalar to it as follows, the `voila` variable would still contain five characters:

```
voila = "voila" + "\u{300}"  //--- voilà---
```

This is because `a` has been changed to `à`.

Swift Backward Compatibility with the `NSString` Type in Objective-C

If you are familiar with the `NSString` class in Objective-C, you will be happy to know that the `string` type in Swift is bridged seamlessly (almost) with the `NSString` class in the `Foundation` framework in Objective-C. This means that you can continue to use the methods and properties related to `NSString` in Swift's `string` type. However, there are some caveats that you need to be aware of.

How Type Conversion Works for Strings

In Swift, there is no implicit conversion. You need to perform explicit conversion whenever you want to convert a variable from one type to another type.

Consider the following statement:

```
var s1 = "400" // s1 is string
```

To convert `s1` to an `Int` type, you need to use the `toInt()` method to explicitly convert it:

```
var amount1:Int? = s1.toInt()
```

You must specify the `?` character to indicate that it is an optional type; otherwise, the type conversion will fail. You can rewrite the preceding example using the type inference as follows:

```
var amount1 = s1.toInt()
```

Consider another example:

```
var s2 = "1.25"
```

If you call the `toInt()` method to explicitly convert it to the `Int` type, you will get the `nil` value:

```
var amount2 = s2.toInt()   //---nil as string cannot be converted to Int---
```

If you call the `toDouble()` method to explicitly convert it to the `Double` type, you will get an `error`:

```
var amount2:Double = s2.toDouble()  //---error---
```

This is because the `string` type does not have the `toDouble()` method. To resolve this, you can cast it to `NSString` and use the `doubleValue` property:

```
var amount2: Double = (s2 as NSString).doubleValue  //---1.25---
```

The following examples show how to convert from numeric values to `string` types:

```
var num1 = 200     //---num1 is Int---
var num2 = 1.25    //---num2 is Double---
```

To convert `num1` (which is of type `Int`), you can use the `String` initializer:

```
var s3 = String(num1)
```

Alternatively, you can use the string interpolation method.

How the `String` Type Inter-Operates with the `NSString` Class

Swift is almost bridged with the `NSString` class in the `Foundation` framework in Objective-C. This means that you can continue to use the methods and properties related to `NSString` in Swift's string type.

Consider the following statement:

```
var str1 = "This is a Swift string"
```

Based on the type inference, `str1` would be of the `string` type. However, you can continue to use the methods and properties already available in `NSString`, such as:

```
println(str1.uppercaseString)
println(str1.lowercaseString)
println(str1.capitalizedString)
```

`uppercaseString`, `lowercaseString`, and `capitalizedString` are all properties belonging to the `NSString` class, but you can use them in a string instance.

Swift will also automatically convert a result from `NSArray` of `NSStrings` to the `Array` class. The following example shows how:

```
var fruitsStr = "apple,orange,pineapple,durian"
var fruits = fruitsStr.componentsSeparatedByString(",")
for fruit in fruits {
   println(fruit)
}
```

The output of the preceding code is as follows:

```
apple
orange
pineapple
```

In Swift, you should consider some methods from the `NSString` class. These methods include:

- `containsString()` method
- `string countElements()` method
- `stringByReplacingCharactersInRange()` method
- `rangeOfString()` method

`containsString()` Method

The `containsString()` method is available in `NSString`, but if you call it directly in an instance of the `string` type, you will get an `error`:

```
var str1 = "This is a Swift string"
println(str1.containsString("Swift"))
//---error: 'String' does not have a member named 'containsString'---
```

In such a case, you first need to explicitly convert the `string` instance to the `NSString` instance using the `as` keyword, as shown below:

```
var str1 = "This is a Swift string"
println((str1 as NSString).containsString("Swift"))  //---true---
```

Once you have converted the string instance to the `NSString` instance, you can call the `containsString()` method.

`countElements()` Method

The `countElements()` method is used in Swift to determine the length of a string. However, you can also use the `length` property available in the `NSString` class by typecasting it to `NSString`, as shown below:

```
var str1 = "This is a Swift string"
println((str1 as NSString).length)  //---22---
```

`stringByReplacingCharactersInRange()` Method

Some methods require arguments to be of a particular Swift type, even if the method is available in `NSString`. For example, the `stringByReplacingCharactersInRange()` method takes in two arguments: an instance of the type `Range<String.Index>` and a string instance. If you call this method and pass in an `NSRange` instance, an `error` will occur:

```
//---an instance of NSRange---
var nsRange = NSMakeRange(5, 2)

str1.stringByReplacingCharactersInRange(nsRange, withString: "was")
//---error: 'NSRange' is not convertible to 'Range<String.Index>'---
```

Instead, you need to create an instance of the type `Range<String.Index>` (a Swift type) and use it in the `stringByReplacingCharactersInRange()` method:

```
//---an instance of Range<String.Index>---
var swiftRange =
   advance(str1.startIndex, 5) ..< advance(str1.startIndex, 7)

str1 = str1.stringByReplacingCharactersInRange(
   swiftRange, withString: "was")
println(str1)    //---This was a Swift string---
```

In the above code example, the `..<` operator is called the half-open range operator. It has the following syntax: `a ..< b`, which specifies a range of values from `a` to `b`, but not including `b`.

An alternative way to dealing with strings is to declare a variable explicitly as `NSString` type:

```
var str2:NSString = "This is a NSString in Objective-C."
```

In the preceding statement, `str2` will now be an `NSString` instance. You can also rewrite the above statement as follows:

```
var str2 = "This is a NSString in Objective-C. " as NSString
```

You can call all the `NSString` methods directly via `str2`:

```
println(str2.length)                        //---35---
println(str2.containsString("NSString"))    //---true---
println(str2.hasPrefix("This"))             //---true---
println(str2.hasSuffix(". "))               //---true---
println(str2.uppercaseString)    //---THIS IS A NSSTRING IN OBJECTIVE-C.---
println(str2.lowercaseString)    //---this is a nsstring in objective-c.---
println(str2.capitalizedString)  //---This Is A Nsstring In Objective-C---

println(str2.stringByAppendingString("Yeah!"))
//---This is a NSString in Objective-C. Yeah!---

println(str2.stringByAppendingFormat("This is a number: %d", 123))
//---This is a NSString in Objective-C. This is a number: 123---
```

You can also create an `NSRange` instance and use it directly in the `stringByReplacingCharactersInRange()` method:

```
var range = str2.rangeOfString("Objective-C")
if range.location != NSNotFound {
   println("Index is \(range.location) length is \(range.length)")
   //---Index is 22 length is 11---
   str2 = str2.stringByReplacingCharactersInRange(
   range, withString: "Swift")
   println(str2)  //---This is a NSString in Swift.---
}
```

Here is another example of using the `rangeOfString()` method from `NSString` to find the index of the occurrence of a string within a string:

```
var path:NSString = "/Users/wei-menglee/Desktop"

//---find the index of the last /---
range = path.rangeOfString("/", options:NSStringCompareOptions.BackwardsSearch)

if range.location != NSNotFound {
   println("Index is \(range.location)")  //---18---
}
```

DEFINING FUNCTIONS IN SWIFT

A function is a group of statements that perform a specific set of tasks. In Swift, a function has a name. It may also accept parameters and optionally return a value (or a set of values). Functions in Swift work similarly to traditional C functions, and they also support features such as external parameter names, which enable them to mirror the verbosity of Objective-C methods.

How a Function is Defined and Called in Swift

In Swift, a function is defined using the `func` keyword, as shown below:

```
func doSomething() {
println("doSomething")
}
```

The preceding code snippet defines a function called `doSomething`. It does not accept any inputs (known as parameters) and does not return a value (technically it does return a void value).

To call the function, you can simply call its name followed by a pair of empty parentheses:

```
doSomething()
```

How to Define Various Parameters

An input parameter is used to define one or more named typed inputs in a function.

The following function takes in one single typed input parameter:

```
func doSomething(num: Int) {
println(num)     //num is parameter used internally within the function
}
doSomething(5)   //passing argument to function
//---or---
var num = 5
doSomething(num)
```

The following function takes in two input parameters, both of `Int` type:

```
func doSomething(num1: Int, num2: Int) {
   println(num1, num2)
}
doSomething(5, 6)           //passing two arguments to function
```

You can assign a default value to a parameter. The default value makes the parameter optional when you call it. Consider the following function in which you have three parameters:

```
func joinName(firstName:String,
              lastName:String,
              joiner:String = " ") -> String {         // default value of single space
   return "\(firstName)\(joiner)\(lastName)"
}
```

The third parameter has the default value of a single space. When calling this function with three arguments, you need to specify the default parameter name, as shown below:

```
var fullName = joinName("Wei-Meng", "Lee", joiner:",")
println(fullName)  //---Wei-Meng,Lee---
```

For default parameters, you need to specify the parameter name explicitly when calling a function. You can omit the default parameter when calling the function and it will use the default value of the single space for the third argument:

```
fullName = joinName("Wei-Meng","Lee")
println(fullName)   //---Wei-Meng Lee---
```

A function can also accept a variable number of arguments like in the case of calculating an average of a series. In this case, your function can be defined as follows:

```
  func average(nums: Int...) -> Float {
  var sum: Float = 0
  for num in nums {
     sum += Float(num)
  }
  return sum/Float(nums.count)
}
```

The `...` (three periods) indicates that the parameter accepts a varying number of arguments, which in this case are of type `Int`.

A parameter that accepts a variable number of values is known as a variadic parameter. You can call the function by passing it arguments of any length:

```
println(average(1,2,3))         //---2.0---
println(average(1,2,3,4))       //---2.5---
println(average(1,2,3,4,5,6))  //---3.4---
```

Some parameters are used to persist the changes made within the function. Sometimes you want a function to change the value of a variable after it has returned. This type of function is called an `inout` parameter. A variable passed into a function does not change its value after a function call has returned. This is because the function makes a copy of the variable and all changes are made to the copy. These variables include `Int`, `Double`, `Float`, `Struct`, and `String`. An example of an `inout` parameter is shown below:

```
func fullName(inout name:String, withTitle title:String)
//name parameter is prefixed with the inout keyword
{
   name = title + " " + name;
}
```

In the above example, the `name` parameter is prefixed with the `inout` keyword. This keyword specifies that changes made to the `name` parameter will persist after the function has returned. To see how this works, consider the following code snippet:

```
var myName = "Wei-Meng Lee"  // Original name is “Wei-Meng Lee”
fullName(&myName, withTitle:"Mr.")
println(myName)  //---prints out "Mr. Wei-Meng Lee"---
```

After the function has returned, its value has changed to "Mr. Wei-Meng Lee".

You need to know the following points when calling a function with the inout parameters:

- You need to pass a variable to an inout parameter; constants are not allowed.
- You need to prefix the & character before the variable that is passed into an inout parameter, to indicate that its value can be changed by the function.
- Inout parameters cannot have default values.
- Inout parameters cannot be marked with the var or let keyword.

BIG Picture

The function syntax used by Swift is expressive enough to understand anything from the simple C language to the Objective-C style. Passing default and variable parameters are easier and editable even after a function is executed with used inout parameters.

How to Return a Value from a Function

To return a value from a function, you can use the -> operator after the function declaration. The following function returns an integer value:

```
func doSomething(num1: Int, num2: Int, num3: Int) -> Int {
    return num1 + num2 + num3        //keyword return used to return the value
}
//---value returned from the function is assigned to a variable---
var sum = doSomething(5,6,7)
//---return value from a function is ignored---
doSomething(5,6,7)
```

In Swift, you can use a tuple type in a function to return multiple values. The following example shows a function that takes in a string containing numbers, examines each character in the string, and counts the number of odd and even numbers contained in it:

```
func countNumbers(string: String) -> (odd:Int, even:Int) {
  var odd = 0, even = 0
  for char in string {
     let digit = String(char).toInt()
     if (digit != nil) {
         (digit!) % 2 == 0 ? even++ : odd++
     }
  }
  return (odd, even)
}
```

The (odd:Int, even:Int) return type specifies the members of the tuple that would be returned by the function—odd (of type Int) and even (of type Int). To use this function, pass it a string and assign the result to a variable or constant, as shown below:

```
var result = countNumbers("123456789")
//Result is stored as a tuple containing odd and even integer.
println("Odd: \(result.odd)")          //---5---
println("Even: \(result.even)")        //---4---
}
```

ADDITIONAL KNOWHOW

When passing value types (such as `Int`, `Double`, `Float`, `Struct`, and `String`) into a function, the function makes a copy of the variables. However, when passing an instance of a reference type (such as classes), the function references the original instance of the type and does not make a copy.

How to Define and Call `Function` Type Variables

Every function has a specific function type. To understand this, consider the following two functions:

```
func sum(num1: Int, num2: Int) -> Int {
  return num1 + num2
}

func diff(num1: Int, num2: Int) -> Int {
  return abs(num1 - num2)
}
```

Both functions accept two parameters and return a value of type `Int`. Therefore, the type of each function is:

```
(Int, Int) -> Int.
```

Usually, the type of each function is called the function signature in programming languages, such as Java and C#. As an example, the following function has the type `() -> ()`, which you can read as "a function that does not have any parameters and returns void":

```
func doSomething() {
   println("doSomething")
}
```

In Swift, you can define a variable or constant as a `Function` type:

```
var myFunction: (Int, Int) -> Int        // myFunction is variable
```

`myFunction` type is a function that takes in two `Int` parameters and returns an `Int` value.

```
myFunction = sum      //myFunction is of same type as the sum()
//---or---
var myFunction: (Int, Int) -> Int = sum       //Shortened the above example
```

Now, you can call the `sum()` function using the `function` type variable `myFunction`, as shown below:

```
println(myFunction(3,4))  //---prints out 7---
//myFunction assign another function(having (Int, Int) -> Int function type)
myFunction = diff
```

In the above example, to call `myFunction` again, you will call the `diff()` function:

```
println(myFunction(3,4))  //---prints out 1---
```

Table 4 shows the definition of some functions and their corresponding `function` types.

Table 4: Function Definitions and Types

Function Definition	Function Type (Description)

`func average(nums: Int...)`
`    -> Float {`
`}`
	`(Int...) -> Float`
	The parameter is a variadic parameter, and therefore you need to specify three periods (…).

`func joinName(firstName:String,`
`             lastName:String,`
`             joiner:String = " ")`
`    -> String {`
`}`
	`(String, String, String) -> String`
	You need to specify the type for the default parameter (third parameter).

`func doSomething(num1: Int,`
`                 num2: Int) {`
`}`
	`(Int, Int) -> ()`
	The function does not return a value, and therefore you need `()` in the `function` type.

`func doSomething() {`
`}`
	`() -> ()`
	The function does not have any parameter and does not return a value. Therefore, you need to specify `()` for both parameter and `return` type.

Technical Stuff

When you call two functions of the same name, you have to be careful while passing parameters in case of external and default parameters. Otherwise, this results into an error because the compiler is not able to resolve which function to call.

How to Return a `Function` Type from a Function

A `function` type can be used as the `return` type of a function.

Consider the following example:

```
func chooseFunction(choice:Int) -> (Int, Int)->Int {
//chooseFunction takes in an Int parameter
  if choice == 0 {
     return sum
  } else {
     return diff
  }
}
```

In the above example, if choice is `0`, then it returns the `sum()` function; otherwise it returns the `diff()` function.

To use the `chooseFunction()` function, call it and pass in a value and assign its return value to a variable or a constant as follows:

```
var functionToUse = chooseFunction(0)
```

The return value can now be called like a function:

```
println(functionToUse(2,6))        //---prints out 8---
functionToUse = chooseFunction(1)
println(functionToUse(2,6))        //---prints out 4---
```

You can also define functions within a function. This is known as **nested functions**. A nested function can only be called within the function in which it is defined.

You can rewrite the `chooseFunction()` function shown in the above example using nested functions:

```
func chooseFunction(function:Int) -> (Int, Int)->Int {
  func sum(num1: Int, num2: Int) -> Int {
     return num1 + num2
  }
  func diff(num1: Int, num2: Int) -> Int {
     return abs(num1 - num2)
  }
  if function == 0 {
     return sum
  } else {
     return diff
  }
}
```

QUICK TIP

For default parameters, you need to specify the parameter name explicitly when calling the function. In addition, there is no need to specify an external parameter name for a default parameter. You can use the # shorthand when defining the function as the default parameter implicitly. It indicates a named argument.

INTRODUCTION TO COLLECTION TYPES IN SWIFT

Swift provides two types of collections for storing data of the same type:

- **Arrays:** An **array** stores its items in an ordered fashion.
- **Dictionaries:** A **dictionary** stores its items in an unordered fashion and uses a unique key to identify each item.

In Swift, arrays and dictionaries store specific type of data. Unlike the `NSArray` and `NSDictionary` classes in Objective-C, arrays and dictionaries in Swift use either type inference or explicit type declaration to ensure that only specific types of data can be stored. This strict rule about data types enables developers to write type-safe code.

BIG Picture

`Array` and `Dictionary` types in Swift are implemented as generic collections. They use a key to store values and allow you to explicitly type the needed collections to ensure that you are aware of the value you retrieve from an array or a dictionary.

How to Create an Array

An array is an indexed collection of objects. The following statement shows an array containing three items:

```
var OSes = ["iOS", "Android", "Windows Phone"]
```

In Swift, you create an array using the `[ ]` syntax. The compiler automatically infers the type of items inside the array. In this case, it is an array of string elements.

An array declared with the `var` keyword is a **mutable array**—meaning the array's size is not fixed and during runtime you can add or remove elements from the array. If you declare an array using the `let` keyword, you are creating an **immutable array**. This means that once the array is created, its element(s) cannot be removed and new elements cannot be added.

```
//---immutable array---
let OSes = ["iOS", "Android", "Windows Phone"]
```

There are other shorthand syntaxes for arrays that specify the type using data types as shown below. If you mix the elements of an array with different types, the compiler will generate an `error`:

```
    var OSes = ["iOS", "Android", "Windows Phone", 25] //--Compilation error---
```

The fourth element is not compatible with the rest of elements, and therefore compilation fails.

```
    var OSes:Array<String> = ["iOS", "Android", "Windows Phone"] // same type elements
```

To specify the data type to be stored in an array, you can use the following syntax:

```
    var OSes:[String] = ["iOS", "Android", "Windows Phone"]      //store string
    var numbers:[Int] = [0,1,2,3,4,5,6,7,8,9]                    //store integer
```

Various Array Operations

To retrieve items inside an array, you can use the subscript syntax, as follows:

```
    var item1 = OSes[0]    // "iOS"
    var item2 = OSes[1]    // "Android"
    var item3 = OSes[2]    // "Windows Phone"
```

Subscripts enable you to access a specific item of an array directly by writing a value (commonly known as the index) in square brackets after the array name. Array indices start at 0, not at 1.

To insert an element into an array at a particular index, you can use the `insert()` function:

```
    //---inserts a new element into the array at index 2---
    OSes.insert("BlackBerry", atIndex: 2)
```

In the above code, the parameter name specified is `atIndex`. This is known as an external parameter name.

After inserting an element, the array now contains the following elements:

```
    [iOS, Android, BlackBerry, Windows Phone]
```

You can insert an element using an index up to the array's size. You can insert an element to the back of the array using the following code:

```
    OSes.insert("Tizen", atIndex: 4)
    //---index out of range---
    OSes.insert("Tizen", atIndex: 5)        //run-time failure
```

The above statement will result in a run-time failure, as the maximum accessible index is `4`.

To change the value of an existing item in the array, specify the index of the item and assign a new value to it, as shown below:

```
    OSes[3] = "WinPhone"
    [iOS, Android, BlackBerry, WinPhone]      //array now contains the updated element.
```

Note that you can only modify values of arrays that were declared using the `var` keyword. If an array is declared using the `let` keyword, its values are not modifiable.

To append an item to an array, use the `append()` function:

```
OSes.append("Tizen")
[iOS, Android, BlackBerry, WinPhone, Tizen]  //array now contains the appended element
```

Alternatively, you can also use the `+=` operator to append to an array:

```
OSes += ["Tizen"]
```

You can also append an array to an existing array:

```
OSes += ["Symbian", "Bada"]
```

The array now contains the following appended elements:

```
[iOS, Android, BlackBerry, WinPhone, Tizen, Symbian, Bada]
```

To get the length of an array, use the `count` property:

```
var lengthofArray = OSes.count  //---returns 7---
```

To check whether an array is empty, use the `isEmpty()` function:

```
var arrayIsEmpty = OSes.isEmpty
```

Array elements can be removed using the following functions:

```
var os1 = OSes.removeAtIndex(3)     // removes "WinPhone"
var os2 = OSes.removeLast()         // removes "Bada"
OSes.removeAll(keepCapacity: true) // removes all element
```

Both the `removeAtIndex()` and `removeLast()` functions return the item removed. The `removeAll()` function clears all elements in the array. If the `keepCapacity` parameter is set to `true`, then the array will maintain its size.

To iterate over an array, you can use the `For-In` loop, as shown below:

```
var OSes = ["iOS", "Android", "Windows Phone"]
for OS in OSes {
  println(OS)
}
```

You can also access specific elements in the array using its indices:

```
var OSes = ["iOS", "Android", "Windows Phone"]
for index in 0...2 {
   println(OSes[index])
}
```

If you need the index and value of each element in the array, you can use the global `enumerate` function to return a tuple for each element in the array:

```
var OSes = ["iOS", "Android", "Windows Phone"]
for (index, value) in enumerate(OSes) {
   println("element \(index) - \(value)")
}
```

The preceding code snippet outputs the following:

```
element 0 - iOS
element 1 - Android
element 2 - Windows Phone
```

In Swift, you can create an empty array of a specific data type using the initializer syntax, as follows:

```
var names = [String]()//empty array of type string
```

To populate the array, you can use the `append()` method:

```
names.append("Sienna Guillory")
names.append("William Fichtner")
names.append("Hugh Laurie")

for name in names {
  println(name)  //---print out all the names in the array---
}
```

To make names an empty array again, assign it to a pair of empty brackets:

```
names = []
```

The following example creates an empty array of type `Int`:

```
var nums = [Int]()
```

You can also create an array of a specific size and initialize each element in the array to a specific value:

```
var scores = [Float](count:5, repeatedValue:0.0)
```

The `count` parameter indicates the size of the array and the `repeatedValue` parameter specifies the initial value for each element in the array.

You can rewrite the preceding statement without explicitly specifying the type:

```
var scores = Array(count:5, repeatedValue:0.0)
```

The type can be inferred from the argument passed to the `repeatedValue` parameter. If you print out the values for each element in the array, as shown below:

```
for score in scores {
  println(score)
}
```

You will be able to see initial values for each element:

```
0.0
0.0
0.0
0.0
0.0
```

Array equality can be done using the == operator. Two arrays are equal if they contain exactly the same elements and in exactly the same order. Consider the following example:

```
var array1 = [1,2,3,4,5]
var array2 = [1,2,3,4]//---two array are not equal as elements are not same---
println("Equal: \(array1 == array2)")  //---false---
```

Now append another element to `array2`:

```
array2.append(5)
```

These two arrays are now equal, as they have the same number of elements in the same exact order:

```
println("Equal: \(array1 == array2)")  //---true---
```

Suppose you have another array:

```
var array3 = [5,1,2,3,4]
```

It is not equal to `array1` because the order of the elements is not the same:

```
println("Equal: \(array1 == array3)")  //---false---
```

How to Create a Dictionary

A dictionary is a collection of objects of the same type that is identified using a key.

Consider the following example:

```
var platforms: Dictionary<String, String> = [
   "Apple": "iOS",
   "Google" : "Android",
   "Microsoft" : "Windows Phone"
]
```

Here, `platforms` is a dictionary containing three items. Each item is a key/value pair. For example, `"Apple"` is the key that contains the value `"iOS"`. The declaration specifies that the key and value must be of the `string` type. Due to type inference, you can shorten the declaration without specifying the `dictionary` keyword and type specifications:

```
var platforms = [
    "Apple": "iOS",
    "Google" : "Android",
    "Microsoft" : "Windows Phone"
]
```

Unlike arrays, the ordering of items in a dictionary is not important because items are identified by their keys and not their positions. You can rewrite the preceding statement as follows:

```
var platforms = [
    "Microsoft" : "Windows Phone",
    "Google" : "Android",
    "Apple": "iOS"
]
```

The key of an item in a dictionary is not limited to string—it can be any of the hashable types (i.e., it must be uniquely representable).

```
var ranking = [       // ---dictionary using an integer as its key--
    1: "Gold",
    2: "Silver",
    3: "Bronze"
]
```

The value of an item can itself be another array, as the following example shows:

```
var products = [
    "Apple" : ["iPhone", "iPad", "iPod touch"],
    "Google" : ["Nexus S", "Nexus 4", "Nexus 5"],
    "Microsoft" : ["Lumia 920", "Lumia 1320","Lumia 1520"]
]
```

To access a particular product in the preceding example, you will first specify the key of the item you want to retrieve, followed by the index of the array:

```
println(products["Apple"]![0])  //---iPhone---
println(products["Apple"]![1])  //---iPad---
println(products["Google"]![0]) //---Nexus S---
```

Note that you have to use `!` to enforce the unwrapping of the value of the dictionary. This is because the dictionary returns an optional value (it can potentially return a `nil` value if you specify a key that does not exist), as shown below:

```
var models = products["Samsung"]   //---models is nil---
```

The safest way to extract values from a dictionary is to test for `nil`, as shown below:

```
var models = products["Apple"]
if models != nil {
   println(models![0])   //---iPhone---
}
```

ADDITIONAL KNOWHOW

When creating a dictionary, its mutability depends on whether you use either the `let` or the `var` keyword. If you use the `let` keyword, the dictionary is immutable (its size cannot be changed after it has been created), as you are creating a constant. If you use the `var` keyword, the dictionary is mutable (its size can be changed after its creation), as you are creating a variable.

Various Dictionary Operations

To access an item in a dictionary using its subscript, specify its key, as shown below:

```
var platforms = [
    "Apple": "iOS",
    "Google" : "Android",
    "Microsoft" : "Windows Phone"
]

var ranking = [
    1: "Gold",
    2: "Silver",
    3: "Bronze"
]

println(platforms["Apple"])      //---Optional("iOS")---
println(ranking[2])              //---Optional("Silver")---
```

Because it is possible that the specified key might not exist in the dictionary, the returning result is an optional value of the dictionary's `value` type, which is `String?` in the first example and `Int?` in the second.

To get the number of items within a dictionary, use the `count` property (read-only):

```
var platforms = [
    "Apple": "iOS",
    "Google" : "Android",
    "Microsoft" : "Windows Phone"
]

println(platforms.count)   //---3---
```

To replace the value of an item inside a dictionary, specify its key and assign a new value to it:

```
var platforms = [
    "Apple": "iOS",
    "Google" : "Android",
    "Microsoft" : "Windows Phone"
]

platforms["Microsoft"] = "WinPhone"
```

If the specified key does not already exist in the dictionary, a new item is added. If it already exists, its corresponding value is updated.

Alternatively, you can also use the `updateValue(forKey:)` method and specify the new value for the item as well as its key:

```
platforms.updateValue("WinPhone", forKey: "Microsoft")
```

You can use the `updateValue(forKey:)` method to check whether the item has been updated or newly inserted:

```
if let oldValue = platforms.updateValue("WinPhone", forKey: "Microsoft111") {
  println("The old value for 'Microsoft' was \(oldValue).")
} else {
  println("New key inserted!")
}
```

To remove an item from a dictionary, you can simply set it to `nil`:

```
var platforms = [
    "Apple": "iOS",
    "Google" : "Android",
    "Microsoft" : "Windows Phone"
]

platforms["Microsoft"] = nil;
println(platforms.count)      //---2---
```

The number of items inside the dictionary would now be reduced by one.

Alternatively, you can use the `removeValueForKey()` method:

```
if let removedValue = platforms.removeValueForKey("Microsoft") {
  println("Platform removed: \(removedValue)")
} else {
  println("Key not found")
}
```

Like the `updateValue(forKey:)` method, the `removeValueForKey()` method returns the value of the key to be removed, and `nil` if the key does not exist.

There are a couple of ways to iterate through a dictionary. First, you can use the `For-In` loop, as shown below:

```
var platforms = [
    "Apple": "iOS",
    "Google" : "Android",
    "Microsoft" : "Windows Phone"
]

for platform in platforms {
  println(platform)
}
```

The preceding code will output the following:

```
(Microsoft, Windows Phone)
(Google, Android)
(Apple, iOS)
```

Observe in the above output that the values returned from a dictionary might not necessarily follow the order in which they are added.

You can also specify the key and value separately:

```
for (company, platform) in platforms {
  println("\(company) - \(platform)")
}
```

The preceding statement will output the following:

```
Microsoft - Windows Phone
Google - Android
Apple – iOS
```

The following example iterates through the values in a dictionary using the `values` property:

```
for platform in platforms.values {
  println("Platform - \(platform)")
}
```

The preceding code snippet will output the following:

```
Platform - Windows Phone
Platform - Android
Platform – iOS
```

You can also assign the keys or values of a dictionary directly to an array:

```
let companies = platforms.keys
let oses = platforms.values
```

In Swift, you can create an empty dictionary of a specific data type using the initializer syntax, as shown below:

```
var months = Dictionary<Int, String>()
```

The preceding example creates an empty dictionary of `Int` key type and `String` value type. To populate the dictionary, specify the key and its corresponding value:

```
months[1] = "January"
months[2] = "February"
months[3] = "March"
months[4] = "April"
```

To make `months` an empty dictionary again, assign it to a pair of brackets with a colon within it:

```
months = [:]
```

Equality of dictionaries can be done by using the == operator. Two dictionaries are equal if they contain exactly the same keys and values, as the following example illustrates:

```
var dic1 = [
  "1": "a",
  "2": "b",
  "3": "c",
]
var dic2 = [
  "3": "c",
  "1": "a",      //---two dictionaries are not equal as elements are not same---
]
println("Equal: \(dic1 == dic2)")  //---false---
```

If you add a new item to `dic2`, then it will evaluate to `true`:

```
dic2["2"] = "b"
println("Equal: \(dic1 == dic2)")  //---true---
```

ADDITIONAL KNOWHOW

Creation of immutable collections is a good practice, where the collection does not need to change. Creation of immutable collections enables the Swift compiler to optimize the performance of the collections you create.

Cheat Sheet

- The aim of Swift is to replace Objective-C with much more modern language syntax without worrying too much about the constraints of C compatibility. The syntax of Swift is similar to modern languages, such as Java and C#, and retains some of the core features of Objective-C.
- To test Swift code examples, you need a Swift compiler. The easiest way to obtain a Swift compiler is to download Xcode 6 from the Mac App Store and launch it. There are two ways to test code in Swift:
  - Creating a playground project
  - Creating an iOS project
- Swift supports the following data types:
  - Integer: Represented by the `Int` type
  - Floating-point number: Represented by `Float` and `Double`
  - Boolean: Represented by the `Bool` type
  - Tuple: Is a group of related values that can be manipulated as a single data type
  - Optional types: Specifies a variable that can contain no value
- An enumeration is a user-defined type consisting of a group of named constants. It can also contain a function within its definition.
- In Swift, a semicolon is only needed if you are combining multiple statements into a single line; otherwise not needed.
- Swift supports the following operators:
  - Assignment
  - Arithmetic
  - Comparison
  - Range
  - Logical
  - Nil coalescing
- Use the ? character for the optional variables if you are not sure whether variable is `nil`.
- In Swift, you can represent a string using the `String` or `NSString` type. You can use the `as` keyword to cast a `String` to `NSString`.
- Strings created using the `let` keyword are immutable. Strings created using the `var` keyword are mutable.
- To find the length of a string, you can either cast it to an `NSString` and then use the `length` property, or you can use the `countElements()` function in Swift.

- Use the `toInt()` method to convert a string to an integer. To convert a string into a double, cast the string to `NSString` and then use the `doubleValue` property.
- You can use the closed range operator as `(a...b)` and the half-open range operator as `(a..<b)`.
- By default, all the parameters in a function are constants. To modify the values of parameters, prefix them with the `var` keyword.
- A parameter that persists the changes made within the function is known as an `inout` parameter.
- You can directly add an array to another array by using the `append()` function or the `+=` operator.
- You can use the `count` property to check the size of an array.
- You can use the `removeAtIndex()`, `removeLast()`, or `removeAll()` function to remove an element from an array.
- To remove an element from a dictionary, you can specify the key of the item and set it to `nil`.
- When an array or dictionary is copied, a copy of it is made and assigned to the variable.