Session 4

Part B - Digging Deeper into 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:

- Explain control flow and looping, structures, and classes syntax
- Explain inheritance and its syntax
- Explain how to implement closures and its syntax
- Explain protocols and delegates
- Explain adaptable and reusable generic codes in Swift

INTRODUCTION

One of the most important features of a programming language is its capability to make decisions and perform repetitive tasks. Swift supports this aspect by providing various features such as control flow, looping, and structures.

In this session, you will enhance your knowledge about the Swift language by delving into its various features. You will learn about the various syntax declarations of Swift, apply the concept of inheritance, and implement closures in Swift. You will also learn about the various protocols, delegates, and adaptable and reusable codes used in Swift.

INTRODUCTION TO VARIOUS SYNTAX DECLARATIONS

To make decisions, Swift provides the usual flow control statements through the use of the `If-Else` statement. For making multiple comparisons, Swift provides the `Switch` statement, which in addition to being similar to its counterpart in C, is much more powerful and flexible. Swift also supports the C-style `For` and `While` loops and introduces the new `For-In` loop to iterate through arrays, dictionaries, and strings.

Swift primarily provides two types of statements for flow control:

- **`If` Statement:** The `If` statement enables you to make a decision based on a certain condition(s).
- **`Switch` Statement:** If you need to make a decision based on several conditions, you can use the more efficient `Switch` statement, which enables you to specify the conditions without using multiple `If` statements.

How to Make Decisions Using Various Statements

Swift supports the traditional C-style `If` statement construct for decision-making. The syntax for the `If` statement is as follows:

```
if condition {
   statement(s)
}
```

Here is an example of the `If` statement:

```
var raining = true  //---raining is of type Boolean---
if raining {
  println("Raining now")  }
```

The output of the above code is as follows:

```
Raining now
```

In Swift, there is no need to enclose a condition within a pair of parentheses (`()`).

You can also explicitly compare using a comparison operator:

```
if raining == true {
  println("Raining now")
}
```

In C/C++, non-zero values are considered `true` and are often used in the `If` statement. Consider the following code snippet in C:

```
//---in C/C++---
Int number = 1  //--value is true as its one--
```

```
if (number) {
  //---number is non-zero---
}
```

In Swift, for non-Boolean variables (or constants), you are not allowed to specify a condition without an explicit logical comparison:

```
var number = 1  //---number is of Int type---
if number {     //---this is not allowed in Swift---
  println("Number is non-zero")
}
if number == 1 {            //comparison operator used for comparison
  println("Number is non-zero")
}
```

In C/C++, the assignment operator is commonly performed within the following condition:

```
//---in C/C++---
if (number=5) { //---number is non-zero---

}
```

In the preceding statement, the value `5` is assigned to a `number`. Because the `number is non-zero`, the condition evaluates to `true` and therefore executes the block in the `If` statement.

In Swift, this is not allowed:

```
if number = 5 { //---not allowed in Swift---
  println("Number is non-zero")
}
```

This limitation is useful in preventing unintended actions on the developer's part.

An extension of the `If` statement is the `If-Else` statement. The `If-Else` statement has the following syntax:

```
if condition {
   statement(s)
} else {
   statement(s)
}
```

Any statements enclosed by the `Else` block are executed when the condition evaluates to `false`. Here is an example:

```
var temperatureInCelsius = 25
if (temperatureInCelsius>30) {
  println("This is hot!")
} else {
  println("This is cooling!")
}
```

The preceding example will output the following:

```
This is cooling!
```

Often, you will need to perform a number of `If-Else` statements. Consider the case where you have an integer representing a day of a week. For example, 1 represents Monday, 2 represents Tuesday, and so on. If you were to use the `If-Else` statement, the code would be too cumbersome and complex. For this purpose, you should use the `Switch` statement. The `Switch` statement has the following syntax:

```
switch variable/constant {
   case value_1:
      statement(s)
   case value_2:
      statement(s)
   ...
      ...
   case value_n:
      statement(s)
   default:
      statement(s)
}
```

The values of the variable/constant are compared with the various values using `Switch`. The `Switch` statement follows a specified keyword case for comparing the various values specified (`value_1`, `value2`, …, `value_n`). If no match is found, any statements specified after the `default` keyword are executed.

In Swift, there is no need to specify a `Break` statement after the last statement in each case. This is unlike C and Objective-C, which uses a break after each statement. This is called implicit fallthrough. In Swift, there is no implicit fallthrough. Once the statements in a case are executed, the `Switch` statement ends.

How Fallthroughs Work in Swift

If you are familiar with C/C++, you would be tempted to do a fallthrough, as shown below:

```
var grade: Character
grade = "B"
switch grade {
   case "A":
   case "B":
   case "C":
   case "D":
       println("Passed")
   case "F":
       println("Failed")
   default:
       println("Undefined")
}
```

Fallthroughs are not allowed in Swift. In Swift, each case must have at least one executable statement (note: comments are not executable statements).

If you want to implement the fallthrough behavior in Swift, you need to explicitly use the `fallthrough` keyword:

```
var grade: Character
grade = "A"
switch grade {
   case "A":
       fallthrough
   case "B":
       fallthrough
   case "C":
       fallthrough
   case "D":
       println("Passed")
   case "F":
       println("Failed")
   default:
       println("Undefined")
}
```

In this case, after matching the first case (`"A"`), the execution will fallthrough to the next case (`"B"`), which will then fallthrough to the next case (`"C"`), and so on. The `Switch` statement ends after outputting the line `"Passed."`

Structures in Swift do not use reference counting as memory management. Whenever a structure is passed around the code, it is copied for operation.

How to Perform Looping

A key useful feature in a programming language is its capability to repeatedly execute statements. Swift supports the following loop statements:

- `For-In`
- `While`
- `For`
- `Do-While`

The `For-In` Loop Statement

Swift supports a new loop statement known as the `For-In` loop. The `For-In` loop iterates over a collection of items (such as an array or a dictionary) as well as a range of numbers.

The following code snippet prints the numbers from `0` to `9` using the `For-In` loop:

```
for i in 0...9 {             //i is a constant with initial value as 0
   println(i)
}
```

The closed ranged operator (represented by `...`) defines a range of numbers from `0` to `9` (inclusive). After executing the statement(s) in the `For-In` loop (as defined by `{}`), the value of `i` is incremented to 1, and so on. Because `i` is a constant, you are not allowed to modify its value within the loop, as shown below:

```
for i in 0...9 {
   i++   //---this is not allowed as i is a constant---
```

```
    println(i)
}
```

You can also use the `For-In` loop to output characters in Unicode, as shown below:

```
for c in 65 ... 90 {
  println(Character(UnicodeScalar(c)))  //---prints out 'A' to 'Z'---
}
```

The UnicodeScaler is a structure that takes in an initializer containing the number representing a character in Unicode. You then convert it to a `character` type.

If you want to perform an action a fixed number of times and do not care about the number of each iteration, you can simply specify an underscore (_) in place of a constant, as shown below:

```
//---print * 5 times---
for _ in 1...5 {
  print("*")  //---prints out *****---
}
```

The preceding code snippet outputs the asterisk five times.

The `For-In` loop also works with arrays, as shown below:

```
var fruits = ["apple", "pineapple", "orange", "durian", "guava"]
for f in fruits {
   println(f)
}
```

The preceding example iterates through the five elements contained within the fruits array and outputs them:

```
apple
pineapple
orange
durian
guava
```

The `For` Loop Statement

Swift also supports the traditional `For` loop in C, using the following syntax:

```
for initialization; condition; increment/decrement {
   statement(s)
}
```

Unlike C, in Swift you do not need to enclose the initialization condition increment/decrement block using a pair of parentheses (`()`). The following code snippet outputs the numbers from `0` to `4`:

```
//---print from 0 to 4---
for var i = 0; i<5; i++
{
```

```
   println(i)
}
```

The initializer must be a variable, not a constant, as its value needs to change during the iteration of the loop.

```
//---print from 0 to 4---
for var i = 0; i<5; i++ {          //---i is a variable---
   println(i)
}               // ---loop exits
println(i)   //---i is not defined---
```

When you define `i` without initializing it with a value, you need to specify its type. You can also rewrite the preceding statement by initializing the value of `i` and then omitting the initialization in the `For` loop:

```
//---print from 0 to 4---
var i = 0
for ; i<5; i++ {   //---the initialization part can be omitted---
   println(i)
}
println(i)   //--5---
}
```

The `While` Loop Statement

Swift also provides the `While` loop. The `While` loop executes a block of statements repeatedly as long as the specified condition is true:

```
while condition {
   statement(s)
}
```

The following code snippet outputs the series of numbers `0` to `4`:

```
var index = 0
while index<5 {
  println(index++)
}
```

The `While` loop is applicable when you do not know the number of iterations in a condition.

The `Do-While` Loop Statement

A variation of the `While` loop is the `Do-While` loop. The `Do-While` loop has the following syntax:

```
do {
   statement(s)
} while condition
```

The following code snippet outputs the series of numbers from `0` to `4`:

```
index = 0
do {
   println(index++)
} while (index<5)
```

The `Do-While` loop executes the block of statement(s) enclosed by a pair of braces (`{}`) first, before checking the condition to decide if the looping will continue. If the condition evaluates to `true`, the loop continues. If it evaluates to `false`, then the loop exits.

Understanding the Basic Structure Syntax

Object-oriented programming (OOP) is one of the most important features in Swift programming. Structures and classes play an important role in supporting OOP. In Swift, structures and classes share many similarities, and many concepts that apply to classes apply to structure as well.

The following code snippet defines a `structure` named `Go`:

```
struct Go {
   var row = 0      //---0...18---
   var column = 0   //---0...18---
}
```

For structure names, you should use `UpperCamelCase` (such as `CustomerAddress` and `EmployeeCredential`). To create an instance of the `Go` structure, use the structure's default initializer syntax:

```
var stone1 = Go()
```

The preceding statement creates an instance of the `Go` structure, and the instance name is `stone1`. Both the `row` and `column` properties are initialized to `0` by default:

```
println(stone1.row)      //---0---
println(stone1.column)   //---0---
```

You access the properties using the dot (`.`) syntax. Just as you can access the value of the property, you can also change its value:

```
stone1.row = 12        //--change the row to 12---
stone1.column = 16     //---change the column to 16---
```

A structure is a `value` type. When you assign a variable/constant of a `value` type to another variable/constant, its value is copied over. Consider the following example:

```
var stone1 = Go(row:12, column:16, color:StoneColor.Black)
var stone2 = stone1

println("---Stone1---")
println(stone1.row)
println(stone1.column)
println(stone1.color.toRaw())
```

```
println("---Stone2---")
println(stone2.row)
println(stone2.column)
println(stone2.color.toRaw())
```

In the preceding code snippet, `stone1` is assigned to `stone2`. Therefore, `stone2` will now have the same value as `stone1`. This is evident from the output values of the preceding code snippet:

```
---Stone1---
12
16
Black
---Stone2---
12
16
Black
```

To prove that the value of `stone2` is independent of the value of `stone1`, you can modify the value of `stone1` as follows:

```
stone1.row = 6
stone1.column = 7
stone1.color = StoneColor.White
```

Then, print out the values for both stones again:

```
println("===After modifications===")
println("---Stone1---")
println(stone1.row)
println(stone1.column)
println(stone1.color.toRaw())

println("---Stone2---")
println(stone2.row)
println(stone2.column)
println(stone2.color.toRaw())
```

The preceding statements print out the following, proving that the values of the two stones are independent of each other:

```
===After modifications===
---Stone1---
6
7
White
---Stone2---
12
16
Black
```

In Swift language `string`, `array`, and `dictionary` types are implemented using structures. As such, when they are assigned to another variable, their values are always copied. You can compare structures using the == operator. This is because the compiler does not understand what defines two structures as being equal. Therefore, you need to overload the default meaning of the == and != operators, as shown below:

```
func == (stone1: Go, stone2: Go) -> Bool {
   return (stone1.row == stone2.row) &&
          (stone1.column == stone2.column) &&
          (stone1.color == stone2.color)
}
func != (stone1: Go, stone2: Go) -> Bool {
   return !(stone1 == stone2)
}
```

Essentially, the preceding two functions are overloading the two operators: equal (==) and not equal (!=). Each function takes two `Go` instances and returns a `Bool` value.

You can now use the == operator to test whether `stone1` and `stone2` are of the same value:

```
var stone1 = Go(row:12, column:16, color:StoneColor.Black)
var stone2 = Go(row:12, column:16, color:StoneColor.Black)

if stone1 == stone2 {
   println("Same!")
} else {
   println("Different!")
}
```

The preceding code snippet will output the following:

```
Same!
```

Understanding the Basic Class Syntax

A class is similar to a structure in many ways. You can define a class by using the `class` keyword, as shown below:

```
class ClassName {
}
```

Here is an example:

```
class MyPointClass {
}
```

The preceding code snippet defines a class called `MyPointClass`. When naming classes, you should use `UpperCamelCase` (such as `MyPointClass`, `EmployeeInfo`, and `CustomerDetails`). An important difference between Objective-C and Swift is that in Swift there is no need to have one file to declare a class and another file to define the implementation of a class; one file handles all the declaration and implementation.

To create an instance of a class, you call the class name followed by a pair of parentheses (`()`) and then assign it to a variable or constant:

```
var ptA = MyPointClass()
```

ADDITIONAL KNOWHOW

Swift allows you to define a class or a structure in a single file. The external interface of that class or structure is automatically made available implicitly for other code use. This type is not used in other languages.

Swift is a modern programing language derived from Objective-C. Unlike Objective-C, Swift also uses code blocks as functions, classes, and structures. As you know about the Swift syntax design and memory management, it is easy to write software for high-ended games and applications without considering memory and code complexities. For IT Industries, now it is possible to write large software for iPhones and iPads in less time.

INHERITANCE

Class inheritance is one of the cornerstones of OOP. It basically means that a class can inherit the properties and methods from another class. Class inheritance enables a high degree of code reuse, allowing the same implementation to be adapted for another use. Swift fully supports the capability of class inheritance.

Explaining Inheritance and its Syntax

In Swift, there are simple base and inherit classes.

The simple base class does not inherit from another class. For example, the following `Shape` class does not inherit from any class:

```
class Shape {
    //---stored properties---
    var length:Double = 0
    var width:Double = 0

    func perimeter() -> Double {
        return 2 * (length + width)
    }
}
```

The above `Shape` class contains two stored properties, `length` and `width`, as well as the `perimeter()` method.

Inherit classes use initializers to create instances. An example of a default initializer is shown below:

```
var shape = Shape()
```

To create an instance of the `Shape` class, you can take examples of the following shapes:

- Rectangle
- Circle
- Square

Now, you can create classes that inherit the `Shape` base class, and extend from it, if necessary. Swift does not support abstract classes. You need to use protocols to implement the concept of abstract classes. You can improvise an abstract method by using the private identifier together with an initializer, as shown below:

```
class Shape {
   //---stored properties---
   var length:Double = 0
   var width:Double = 0

   //---improvision to make the class abstract---
   private init() {
       length = 0
       width = 0
   }

   func perimeter() -> Double {
       return 2 * (length + width)
   }
}
```

In the preceding code snippet, you added a private initializer, `init()`, which limits its accessibility to within its physical file. That is to say, any code that is outside the physical file in which the `Shape` class is defined (say, `Shape.swift`) will not be able to call the initializer method.

To inherit from a base class, you create another class and specify the base class name after the colon (`:`):

```
class Rectangle: Shape {

}
```

In the preceding code snippet, `Rectangle` is a subclass of `Shape`. This means that it will inherit all the properties and methods declared in the `Shape` class.

You also need to create an initializer for the `Rectangle` class to create an instance of the `Rectangle` class. You can provide the initializer, as shown below:

```
class Rectangle: Shape {
   //---override the init() initializer---
   override init() {
       super.init()
   }
}
```

As shown above, you need to prefix the `init()` initializer with the `override` keyword. This is because the `init()` initializer is already in the base class (`Shape`). In addition, because you are overriding the initializer, you need to call the immediate superclass's `init()` method before exiting this initializer:

```
override init() {
       super.init()
   }
```

You will now be able to create an instance of the `Rectangle` class:

```
var rectangle = Rectangle()
```

You can also access the `length` and `width` properties of the `Shape` base class:

```
rectangle.length = 5
rectangle.width = 6
```

The following example shows how you can also access the `perimeter()` method defined in the `Shape` class:

```
println(rectangle.perimeter()) //---22.0---
```

You can also add another initializer to the `Rectangle` class. This is called **overloading initializers**.

```
class Rectangle: Shape {
    //---override the init() initializer---
    override init() {
        super.init()
    }

    //---overload the init() initializer---
    init(length:Double, width:Double) {
        super.init()
        self.length = length
        self.width = width
    }
}
```

You are overloading the initializer with two parameters: `length` and `width`.

You can now create an instance of the `Rectangle` class like this:

```
    var rectangle = Rectangle(length: 5,width: 6)
```

In Xcode, code completion will automatically display two initializers that you can use for the `Rectangle` class.

Swift adopts the following rules for initializers:

- If a subclass does not have any initializers, then all the initializers of the base class are available to the subclass.
- If a subclass has at least one initializer, then it will hide all the initializers in the base class.

You can improvise abstract methods by using the `assert()` function.

An example is shown below:

```
class Shape {
    //---stored properties---
    var length:Double = 0
    var width:Double = 0

    //---improvision to make the class abstract---
```

```
    private init() {
        length = 0
        width = 0
    }

    func perimeter() -> Double {
        return 2 * (length + width)
    }

    //---improvision to make the method abstract---
    func area() -> Double {
        assert(false, "This method must be overridden")
    }
}
```

The `assert()` function takes two arguments: a condition and a message. When a condition evaluates to `false`, the program stops execution and a message is displayed.

The programming concept of classes and inheritance can be related to the real world. It is a good approach to understand this basic in the real world model and later in IT systems. Everything in the real world, whether it is an object, living being, or even an idea has so many aspects that it is very complex to represent it completely, but they all have a base or root class. Similarly, you have `NSObject`, which is the base class and contains a number of instance variables and instance methods. As the child inherits features from their parents, similarly derived classes inherit the features of a base class and then add some features of their own in it.

How to Create Different Class Types

You can create different class types to inherit the properties of base class. These classes are called subclasses.

Besides overloading initializers, you can also overload methods. Now using the `Shape` base class, the following code snippet creates another class called `Circle` that inherits from `Shape`:

```
class Circle: Shape { //---circle is a base class---

    //---initializer---
    init(radius:Double) {
        super.init()
        self.width = radius * 2
        self.length = self.width
    }
    //---override the perimeter() function---
    override func perimeter() -> Double {
        return 2 * M_PI * (self.width/2)
    }
    //---overload the perimeter() function---
    func perimeter(#radius:Double) -> Double {

        //---adjust the length and width accordingly---
        self.length = radius * 2
```

```
        self.width = self.length

        return 2 * M_PI * radius
    }
    //---override the area() function---
    override func area() -> Double {
        return M_PI * pow(self.length / 2, 2)
    }
}
```

You can now use the `Circle` class as follows:

```
var circle = Circle(radius: 6.8)
println(circle.perimeter())            // 42.7256600888212
println(circle.area())                 // 145.267244301992
//---need to specify the radius label---
println(circle.perimeter(radius:7.8))  // 49.0088453960008

//---call the perimeter() method above
// changes the radius---
println(circle.area())                 // 191.134497044403
```

Because the `perimeter()` method is overloaded, you can call it either with no argument or with one argument.

You can prevent a class from being inherited. Consider the following `Square` class:

```
final class Square: Rectangle {                //final keyword preventing inheritance
    //---overload the init() initializer---
    init(length:Double) {
        super.init()
        self.length = length
        self.width = self.length
    }
}
```

This `Square` class inherits from the `Rectangle` class and the class definition is prefixed with the `final` keyword. This indicates that no other class can inherit from it. For example, the following code is not allowed:

```
//---cannot inherit from Square as it is final---
class rhombus: Square {

}
```

In addition, if the `area()` method has been declared to be `final` in the `Rectangle` class, the `Square` class is not allowed to override it, as shown below:

```
final class Square: Rectangle {
    //---overload the init() initializer---
    init(length:Double) {
```

```
        super.init()
        self.length = length
        self.width = self.length
    }

    //---cannot override a final method---
    //---override the area() function---
    override func area() -> Double {
       ...
    }
}
```

ADDITIONAL KNOWHOW

While defining a class, it is important to specify a superclass; otherwise it automatically becomes a base class. Also, Swift classes do not inherit from universal base classes.

How to Create and Call Initializers in Subclasses

Initializers basically assign default values to variables in your class so that they all have "initial" values when the class is instantiated. In Swift, there are three types of initializers:

- Default initializer
- Designated initializers
- Convenience initializers

Default Initializer

The default initializer is the initializer that is created by the compiler when your class is instantiated.

For example, consider the following `Contact` class:

```
class Contact {
   var firstName:String = ""
   var lastName:String = ""
   var email:String = ""
   var group:Int = 0
}
```

When you create an instance of the `Contact` class, the compiler automatically generates a default initializer for the `Contact` class so that you can create the instance:

```
var c = Contact()
```

However, observe that all the stored properties in the `Contact` class are initialized to their default values. If they are not initialized to some values, such as the following, the compiler will complain that the class has no initializer:

```
class Contact {
   var firstName:String
   var lastName:String
   var email:String
```

```
    var group:Int
}
```

Designated Initializer

One way to fix the situation above is to initialize each stored property, or to explicitly create an initializer:

```
class Contact {
    var firstName:String
    var lastName:String
    var email:String
    var group:Int

    init() {
        firstName = ""
        lastName = ""
        email = ""
        group = 0
    }
}
```

In this case, you are creating your own initializer to initialize the values of the stored properties. This type of initializer is known as a designated initializer.

You can create another initializer with parameters, as shown in the following example:

```
class Contact {
    var firstName:String
    var lastName:String
    var email:String
    var group:Int

init() {
        firstName = ""
        lastName = ""
        email = ""
        group = 0
    }
    //---designated initializer---
    init(firstName: String, lastName:String, email:String, group: Int) {
        self.firstName = firstName
        self.lastName = lastName
        self.email = email
        self.group = group
    }
}
```

In the preceding example, the initializer is a designated initializer, as it initializes all the properties in the class. You can call the designated initializer as follows:

```
    var c2 = Contact(
            firstName:"Wei-Meng",
            lastName:"Lee",
            email:"weimenglee@learn2develop.net",
            group:0)
```

Convenience Initializer

The third type of initializer is known as a convenience initializer. To understand its use, consider the following example:

```
class Contact {
    var firstName:String
    var lastName:String
    var email:String
    var group:Int

    //---designated initializer---
    init() {
        firstName = ""
        lastName = ""
        email = ""
        group = 0
    }

    //---designated initializer---
    init(firstName: String, lastName:String, email:String, group: Int) {
        self.firstName = firstName
        self.lastName = lastName
        self.email = email
        self.group = group
    }

    //---designated initializer---
    init(firstName: String, lastName:String, email:String, group: Int,
    timeCreated:NSDate) {
        self.firstName = firstName
        self.lastName = lastName
        self.email = email
        self.group = group
        println(timeCreated)
    }

    //---convenience initializer; delegate to the designated one---
    convenience init(firstName: String, lastName:String, email:String) {
        self.init(firstName: firstName, lastName: lastName, email: email,
                  group: 0)
    }
```

```
    //---convenience initializer; delegate to another convenience
    // initializer---
    convenience init(firstName: String, lastName:String) {
        self.init(firstName:firstName, lastName:lastName, email:"")
    }
    //---convenience initializer; delegate to another convenience
    // initializer---
    convenience init(firstName: String) {
        self.init(firstName:firstName, lastName:"")
    }
}
```

Each convenience initializer calls another initializer. The convenience initializer with the fewest parameters calls the one with the next fewest number of parameters, and so on. This is **call initializer chaining**. Finally, the last convenience initializer calls the designated initializer. Initializer chaining enables you to ensure that all properties in a class are fully initialized before use.

Using Extensions in Swift

Extensions in Swift enable you to add additional functionalities (such as methods) to an existing class. Objective-C also supports extensions, except that it is called categories. Other languages that support extensions include C# and JavaScript.

To understand how extensions work, consider the following example:

```
extension String {
    func getLatLng(splitter:String) -> (Double, Double) {
        var latlng = self.componentsSeparatedByString(splitter)
        return ((latlng[0] as NSString).doubleValue,
                (latlng[1] as NSString).doubleValue)
    }
}
```

The preceding code snippet extends the `String` class with a method named `getLatLng()`. Its main function is to take in a string containing a latitude and longitude with a separator in between (say, a comma), and return a tuple containing the latitude and longitude in the `Double` format. A sample string may look like this: "`1.23456,103.345678`". The `getLatLng()` method takes a `String` parameter (specifying the splitter between the latitude and longitude) and returns a tuple containing two `Doubles`.

To use the extension method, simply call it whenever you are dealing with a `String` variable or constant, as shown below:

```
var str = "1.23456,103.345678"
var latlng  = str.getLatLng(",")
println(latlng.0)
println(latlng.1)
```

Besides extending methods, extensions also work with properties, albeit with only computed properties. Extensions in Swift do not support stored properties. Consider the `Distance` class as shown below:

```
class Distance {
    var miles = 0.0
    var km: Double {
```

```
        get {
            return 1.60934 * miles
        }
        set (km) {
            miles = km / 1.60934
        }
    }
}
```

You can extend the `Distance` class by adding computed properties to it:

```
extension Distance {
    var feet: Double { return miles * 5280 }
    var yard: Double { return miles * 1760 }
}
```

In the preceding code snippet, you added two new computed properties to the `Distance` class.

You can use the newly added computed properties as shown below:

```
var d = Distance()
d.miles = 10
println(d.feet)    //---52800.0---
println(d.yard)    //---17600.0---
```

Using Access Controls in Swift

In Swift, the access control is modeled after the concept of modules and source files:

- **Module:** It is a single unit of distribution. The iPhone app that you developed and uploaded to App Store as a single unit is a module.
- **Source File:** It is a physical file within a module.

Swift's idea of access control is a little different from most other languages such as Java and C#. Most conventional OOP languages include three levels of scope:

- **Private Scope:** Member variables are accessible within the class to which they are declared.
- **Protected Scope:** Member variables are accessible within the class to which they are declared, as well as to subclasses.
- **Public Scope:** Member variables are accessible to all code, inside or outside of the class in which they are declared.

Swift provides three different levels of access for your code. These levels apply according to the location where an entity (constant, variable, class, or property) is defined.

- **Public Access:** The entity is accessible anywhere from within the file or module.
- **Private Access:** The entity is accessible only within the same physical file in which it is defined.
- **Internal Access:** By default, all entities defined in Swift have internal access, unless they are declared to be public or private.

Let's look at an example of how the various access control levels work. Assume that you have the following files:

- `ClassA.swift`
- `ClassB.swift`

`ClassA.swift` contains the following definition:

```
class ClassA {
   var a1 = 10
   //---same as---
   //internal var a1 = 10
}
```

`ClassB.swift` contains the following definition:

```
class ClassB {
   var b1 = 20
   //---same as---
   //internal var b1 = 20
}
```

By default, both `a1` and `b1` have internal access control. This means that as long as `ClassA.swift` and `ClassB.swift` are contained within the same module, `a1` is accessible by the code in `ClassB` and `b1` is accessible by the code in `ClassA`. For example, suppose that `ClassA.swift` and `ClassB.swift` are both part of an iPhone application project. In this case, both `a1` and `b1` are accessible anywhere within the iPhone project. Now if you add the `private` keyword to both the declarations of `a1` and `b1`:

```
class ClassA {
   private var a1 = 10
}
//--private--
class ClassB {
   private var b1 = 20
}
```

They will now be inaccessible outside the files. That is to say, `a1` is not accessible to the code in `ClassB`, and neither can the code in `ClassA` access `b1`.

If in `ClassA.swift` you now add another subclass that inherits from `ClassA`, then `a1` is still accessible:

```
//---these two classes within the same physical file---
class ClassA {
   private var a1 = 10
}

class SubclassA: ClassA {
   func doSomething() {
       self.a1 = 5
   }
}
```

In this case, `a1` is still accessible within the same file in which it is declared, even though it is declared as `private`.

If you add the `public` keyword to both the declarations of `a1` and `b1`:

```
class ClassA {         //---a1 (and b1) to be publicly accessible---
   public var a1 = 10
}
```

However, you will receive a compiler warning because `ClassA` has the default internal access, which essentially prevents `a1` from being accessed outside the module. To fix this, make the class `public` as well:

```
   public class ClassA {
   public var a1 = 10
}
```

`ClassA` and its property `a1` are now accessible outside the module.

Swift provides access control levels for individual types such as classes, structures, and enumerations. It also provides protocols to restrict certain context, such as global variables, constants, or functions. In addition, you do not need to specify access control explicitly while making single-target applications.

A good way to understand the `assert()` function is to think of its equivalent meaning in English—ensure. In other words, the `assert` statement means something like "ensure that the *condition* is true; otherwise, stop the program and display the *message*."

CLOSURES

An important feature in Swift is the closure. **Closures** are self-contained blocks of code that can be passed to functions to be executed as independent code units. Think of a closure as a function without a name. In fact, functions are actually special cases of closures.

Swift offers various ways to optimize closures. The various optimizations include the following:

- Inferring parameter types and `return` types
- Shorthand argument names
- Operator closure
- Implicit returns from single-statement closures
- Trailing closure syntax

Closure is Swift's answer to Objective-C's block syntax and C#'s Lambda expressions.

How to Implement Closures and its Syntax

The best way to understand closures is to use an example. Suppose you have the following array of integers:

```
let numbers = [5,2,8,7,9,4,3,1]
```

Assume you want to sort this array in ascending order. You can write your own function to perform the sorting, or you can use the `sorted()` function available in Swift.

```
var sortedNumbers = sorted(numbers, ascending)        //--sorted is a function---
```

The `sorted()` function takes two arguments:

- An array to be sorted
- A closure that takes two arguments of the same type as the array, and returns `true` if the first value should appear before the second value

Functions as Special Types of Closures

In Swift, functions are special types of closures.

Consider an example below:

```
func ascending(num1:Int, num2:Int) -> Bool {
//--ascending function take two arguments
   return num1<num2
}
```

The `ascending()` function takes two arguments of type `Int` and returns a `Bool` value. If `num1` is less than `num2`, it returns `true`. You can now pass this function to the `sorted()` function, as shown below:

```
var sortedNumbers = sorted(numbers, ascending)
```

The `sorted()` function will now return the array that is sorted in ascending order. You can verify this by outputting the values in the array:

```
println("===Unsorted===")
println(numbers)

println("===Sorted===")
println(sortedNumbers)
```

The preceding statements output the following:

```
===Unsorted===
[5, 2, 8, 7, 9, 4, 3, 1]
===Sorted===
[1, 2, 3, 4, 5, 7, 8, 9]
```

How to Declare and Use Closures in Functions

You can assign a closure to a variable. For example, the `ascending()` function discussed earlier can be written as a closure assigned to a variable:

```
var compareClosure : (Int, Int)->Bool =         //returns bool
{
   (num1:Int, num2:Int) -> Bool in
         return num1 < num2
}
```

To use the `compareClosure` closure with the `sorted()` function, pass in the `compareClosure` variable:

```
var sortedNumbers = sorted(numbers, compareClosure)
```

In general, a closure has the following syntax:

```
{
    ([parameters]) -> [return type] in
     [statements]
}
```

The above syntax shows how you pass a function into the `sorted()` function as a closure function.

A better way is to write the closure inline, which obviates the need to define a function explicitly or assign it to a variable. You can rewrite the earlier example to get the following results:

```
var sortedNumbers = sorted(numbers,
    {
        (num1:Int, num2:Int) -> Bool in
            return num1<num2
    }
)
```

As you can see, the `ascending()` function name is now gone. All you have supplied is the parameters list and the content of the function. If you want to sort the array in descending order, you can simply change the comparison operator:

```
var sortedNumbers = sorted(numbers,
    {
        (num1:Int, num2:Int) -> Bool in
        return num1>num2
    }
)
println("===Sorted===")
println(sortedNumbers)
```

The array will now be sorted in descending order:

```
===Sorted===
[9, 8, 7, 5, 4, 3, 2, 1]
If you want to sort a list of strings, you can write your closure as follows:
var fruits = ["orange", "apple", "durian", "rambutan", "pineapple"]

println(sorted(fruits,
    {
        (fruit1:String, fruit2:String) -> Bool in
            return fruit1<fruit2
    })
)
```

The output is as shown below:

```
[apple, durian, orange, pineapple, rambutan]
```

Various Operations Using Closures

If a closure is the final argument of a function, you can rewrite this closure as a trailing closure. A trailing closure is written outside of the parentheses of the function call.

Consider the closure that you saw earlier:

```
println(sorted(fruits,
{
     (fruit1:String, fruit2:String) -> Bool in
          return fruit1<fruit2
    })
)
```

Observe that the closure is passed in as a second argument of the `sorted()` function. For long closures, this syntax might be a little messy. The preceding code snippet when rewritten using the trailing closure looks as shown below:

```
println(sorted(fruits)
            {
                (fruit1:String, fruit2:String) -> Bool in
                return fruit1<fruit2
            }
        )
```

Using the shorthand argument name, the closure can be shortened to the following:

```
println(sorted(fruits) { $0<$1 })
```

The array structure in Swift is a good example for examining how closure works. It has three built-in methods that accept closures as part of the argument list:

- `map()`
- `filter()`
- `reduce()`

The `map()` Function

The `map()` function enables you to transform elements from one array into another array. The `map()` function accepts a closure as its argument.

For the following examples, assume you have an array that contains the prices of some items:

```
let prices = [12.0,45.0,23.5,78.9,12.5]
```

Suppose you want to transform the `prices` array into another array with each element containing the dollar (`$`) sign, as shown below:

```
["$12.0", "$45.0", "$23.5", "$78.9", "$12.5"]
```

Instead of iterating through the original prices array and creating another one manually by copying each element, the `map()` function enables you to do it easily. Consider the following code snippet:

```
var pricesIn$ = prices.map(
            {
                (price:Double) -> String in
                   return "$\(price)"
            }
        )

println(pricesIn$)
```

The closure itself accepts a single argument representing each element of the original array. In the above example, the closure returns a string result. The closure is called once for every element in the array. You simply prefix each price with the `$` sign. The resultant array is assigned to `pricesIn$` and it now contains an array of `String` type:

```
 [$12.0, $45.0, $23.5, $78.9, $12.5]
```

The preceding code can be reduced to the following:

```
let prices = [12.0,45.0,23.5,78.9,12.5]

        var pricesIn$ = prices.map(
            {
                (price) -> String in
                    "$\(price)"
            }
        )

        println(pricesIn$)
```

Using the shorthand argument names, the code is as follows:

```
pricesIn$ = prices.map(
    {
          "$\($0)"
    }
)
```

The `filter()` Function

The `filter()` function enables you to filter the elements inside an array and return another array containing a subset of the original elements that satisfy the specified criteria.

Using the same `prices` array, the following code snippet shows how to apply a filter to the array to return all those elements greater than `20`:

```
let prices = [12.0,45.0,23.5,78.9,12.5]
        var pricesAbove20 = prices.filter(
            {
                (price:Double) -> Bool in
                    price>20
```

```
            }
        )
        println(pricesAbove20)
```

Like the `map()` function, the `filter()` function takes a closure. The closure itself accepts a single argument representing each element of the original array, and returns a `Bool` result. The closure is called once for every element in the array. The result will contain the element if the statement (`price>20`) evaluates to `true`. The preceding code snippet outputs the following:

```
 [45.0, 23.5, 78.9]
```

You can reduce code by using the type inference:

```
var pricesAbove20 = prices.filter(
   {
          (price) in
          price>20
   }
)
```

If you eliminate the named parameters, you will receive the following output:

```
var pricesAbove20 = prices.filter({ $0>20 })
```

The `reduce()` Function

The `reduce()` function returns a single value representing the result of applying a reduction closure to the elements in the array. This function enables you to return the elements inside an array as a single item.

Using the same `prices` array, the following code snippet shows how to sum all the prices in the array:

```
let prices = [12.0,45.0,23.5,78.9,12.5]
var totalPrice =  prices.reduce(
   0.0,
           {
               (subTotal: Double, price: Double) -> Double in
                   return subTotal + price
           }
       )
   println(totalPrice)
```

The `reduce()` function takes two arguments. The first argument takes the initial value (in this case, `0.0`), and the second argument takes the first element in the array. The closure is called recursively. The result is passed in to the same closure as the first argument and the next element in the array is passed in as the second argument. This happens until the last element in the array is processed. The closure recursively sums up all the prices in the array and outputs the following result:

```
171.9
```

If you apply type inference, it reduces the closure to the following:

```
var totalPrice =  prices.reduce(
           0.0,
```

```
            {
                (subTotal, price) in
                    return subTotal + price
            }
    )
```

If you remove the named parameters, it yields the following closure:

```
var totalPrice =  prices.reduce(0.0, { $0 + $1 })
println(totalPrice)
```

If you use an operator function, the closure can be further reduced as follows:

```
var totalPrice =  prices.reduce(0.0,  + )
```

The closure style of programming is very important while working on iOS and OS projects, as it allows you to make lightweight syntax applications. Basically, the style of coding like nested functions, shorthand parameters used, global functions defined are part of closures. Closures allow you to make fast-run and more expressive applications.

Instead of giving named arguments, you can use shorthand argument names in a closure, as Swift automatically provides shorthand names to the parameters. You can refer to the arguments within a closure as `$0`, `$1`, and so on.

SWIFT DESIGN PATTERNS

The use of protocols and delegates reflects one of the most important design patterns in Swift programming. Protocols are used to enforce the content of a class and delegates help to create events and event handlers.

Explanation of Protocols and Delegates

A **protocol** is a blueprint of methods and properties. It describes what a class should have, and it does not provide any implementation. A class that conforms to a protocol needs to provide the implementation as dictated by the protocol. A protocol can be implemented by a class, a structure, or an enumeration.

A **delegate** is an instance of a type (such as a class) that can handle the methods of a class or a structure. Think of a delegate as an event handler. A class or structure can fire events, and it needs something to handle these events. In this case, the class or structure can "delegate" this task to an instance of a type. This instance of a type is the delegate.

How to Create and Use a Protocol

To create a protocol, you use the `protocol` keyword followed by the name of the protocol:

```
protocol ProtocolName {
    func method1()
    func method2()
    ...
}
```

Here is an example of a protocol:

```
protocol CarProtocol {
   func accelerate()
   func decelerate()
}
```

The preceding code snippet declares a protocol named `CarProtocol` containing two methods: `accelerate()` and `decelerate()`. A class that wants to implement a car, which can accelerate or decelerate, can conform to this protocol. To conform to a protocol, specify the protocol name(s) after the class name, as shown below:

```
class ClassName: ProtocolName1, ProtocolName2  {
  ...
}
```

If you are conforming to more than one protocol, separate them by using a comma (`,`). If your class is also extending from another class, specify the protocol name(s) after the class it is extending:

```
class ClassName: BaseClass, ProtocolName1, ProtocolName2  {
  ...
}
```

The following code snippet shows an example of how to conform to a protocol:

```
class Car: CarProtocol  {
  ...
}
```

In the preceding code snippet, the `Car` class is said to "conform to the `CarProtocol`." To conform to the `CarProtocol` protocol, the `Car` class might look as shown below:

```
class Car: CarProtocol  {
   var speed = 0

   func accelerate() {
       speed += 10
       if speed > 50 {
           speed = 50
       }
       printSpeed()
   }

   func decelerate()  {
       speed -= 10
       if speed<=0 {
           speed = 0
       }
       printSpeed()
   }
```

```
    func stop() {
        while speed>0 {
            decelerate()
        }
    }

    func printSpeed() {
        println("Speed: \(speed)")
    }
}
```

The `Car` class is free to implement other methods as required. In this case, it also implements the `stop()` and `printSpeed()` methods. If any of the methods declared in `CarProtocol` are not implemented in the `Car` class, the compiler will flag an error. You can now create an instance of the `Car` class and call its various methods to accelerate the car, decelerate it, as well as to make it stop:

```
var c1 = Car()
c1.accelerate()  //---Speed: 10---
c1.accelerate()  //---Speed: 20---
c1.accelerate()  //---Speed: 30---
c1.accelerate()  //---Speed: 40---
c1.accelerate()  //---Speed: 50---
c1.decelerate()  //---Speed: 40---
c1.stop()         //---Speed: 30---
                 //---Speed: 20---
                 //---Speed: 10---
                 //---Speed: 0---
```

You can provide the option for a car to determine whether it will implement a particular method by using an **optional method** using the `optional` keyword. The following code snippet shows the `CarProtocol` has an optional method called `accelerateBy()`:

```
@objc protocol CarProtocol {
    func accelerate()
    func decelerate()
    optional func accelerateBy(amount:Int)
}
```

You can use the `@objc` tag before the `protocol` keyword to indicate to the compiler that your class is inter-operating with Objective-C. You need to prefix the protocol with this tag to declare optional methods in your protocol, even if you do not intend to use your class with the Objective-C code. All optional methods are prefixed with the `optional` keyword.

In the `Car` class, you can choose to implement the optional `accelerateBy()` method, if desired, as shown below:

```
class Car: CarProtocol  {
    var speed = 0

    func accelerate() {
```

```
        speed += 10
        if speed > 50 {
            speed = 50
        }
        printSpeed()
    }

    func decelerate()  {
        speed -= 10
        if speed<=0 {
            speed = 0
        }
        printSpeed()
    }

    func stop() {
        while speed>0 {
            decelerate()
        }
    }

    func printSpeed() {
        println("Speed: \(speed)")
    }

    func accelerateBy(amount:Int) {
        speed += amount
        if speed > 50 {
            speed = 50
        }
        printSpeed()
    }
}
```

The following example shows how you can use the `accelerateBy()` method in your class:

```
var c1 = Car()
c1.accelerate()    //---Speed: 10---
c1.accelerate()    //---Speed: 20---
c1.accelerate()    //---Speed: 30---
c1.accelerate()    //---Speed: 40---
c1.accelerate()    //---Speed: 50---
c1.decelerate()    //---Speed: 40---
c1.stop()          //---Speed: 30---
                  //---Speed: 20---
                  //---Speed: 10---
                  //---Speed: 0---
c1.accelerateBy(5) //---Speed: 5---
```

```
   c1.accelerateBy(5) //---Speed: 10---

   @objc class Car: CarProtocol, CarDetailsProtocol {

     ...
   }
```

How to Create and Use a Delegate

If a car has reached its maximum speed, it is important that the class fires an event to inform about this to the user of the class. Similarly, if the car has completely stopped, it is also important to let the user know about this. It is also useful to notify the user whenever the car is accelerating or decelerating. All the preceding behaviors of the class can be implemented as a protocol, as shown below:

```
   @objc protocol CarDelegate {
      func reachedMaxSpeed(c: Car)          //reached max speed
      func completelyStopped(c: Car)        //car comes to complete stop

      optional func accelerating(c: Car)    //car is accelerating
      optional func decelerating(c: Car)    //car is decelerating
   }
```

To use the `CarDelegate`, you can add the following code to the `Car` class:

```
   @objc class Car: CarProtocol {
   // @objc tag use because the methods in the CarDelegate protocol contain references to the
Car class.
      var delegate: CarDelegate?
      var speed = 0
      func accelerate() {
          speed += 10
          if speed > 50 {
              speed = 50
              //---call the reachedMaxSpeed() declared
              // in the CarDelegate ---
              delegate?.reachedMaxSpeed(self)
          } else {
              //---call the accelerating() declared
              // in the CarDelegate ---
              delegate?.accelerating?(self)
          }
          printSpeed()
      }

      func decelerate()  {
          speed -= 10
          if speed<=0 {
              speed = 0
```

```
            //---call the completelyStopped() declared in
            // the CarDelegate---
            delegate?.completelyStopped(self)
        } else {
            //---call the decelerating() declared
            // in the CarDelegate ---
            delegate?.decelerating?(self)
        }
        printSpeed()
    }

    func stop() {
        while speed>0 {
            decelerate()
        }
    }

    func printSpeed() {
        println("Speed: \(speed)")
    }

    func accelerateBy(amount:Int) {
        speed += amount
        if speed > 50 {
            speed = 50
            //---call the reachedMaxSpeed() declared in
            // the CarDelegate---
            delegate?.reachedMaxSpeed(self)
        } else {
            //---call the accelerating() declared
            // in the CarDelegate ---
            delegate?.accelerating?(self)
//? sign indicate delegate variable is an optional variable (may be nil).
        }
        printSpeed()
    }
}
```

Because `delegate` is an optional type, using `?` will prevent the code from crashing if the `delegate` is `nil`. If the value is `nil`, there will be no effect by calling the `reachedMaxSpeed()` method. If you use `!` to wrap the optional value, the statement will crash if the `delegate` is `nil`.

Now that you have modified the `Car` class to have a variable of type `CarDelegate`, you can create a class that implements the `CarDelegate` protocol. The following `CarStatus` class conforms to the `CarDelegate` protocol:

```
class CarStatus: CarDelegate {
    func reachedMaxSpeed(c: Car) {
        println("Car has reached max speed! Speed is \(c.speed)mph")
```

```
    }

    func completelyStopped(c: Car) {
        println("Car has completely stopped! Speed is \(c.speed)mph")
    }

    //===optional methods===

    func accelerating(c: Car) {
        println("Car is accelerating...Speed is \(c.speed)mph")
    }

    func decelerating(c: Car) {
        println("Car is decelerating...Speed is \(c.speed)mph")
    }
}
```

The `CarStatus` class implements the four methods as declared in the `CarDelegate` protocol, two of which are optional. Think of the `CarStatus` class as the event handler for the `Car` class. You can now create an instance of the `CarStatus` class and assign it to the `delegate` property of the `Car` class, as shown below:

```
var c1 = Car(model: "F150")
c1.delegate = CarStatus()
c1.accelerate()  //---Car is accelerating...Speed is 10mph---
                 //---Speed: 10---
c1.accelerate()  //---Car is accelerating...Speed is 20mph---
                 //---Speed: 20---
c1.accelerate()  //---Car is accelerating...Speed is 30mph---
                 //---Speed: 30---
c1.accelerate()  //---Car is accelerating...Speed is 40mph---
                 //---Speed: 40---
c1.accelerate()  //---Car is accelerating...Speed is 50mph---
                 //---Speed: 50---
c1.accelerate()  //---Car has reached max speed! Speed is 50mph---
                 //---Speed: 50---
c1.stop()         //---Car is decelerating...Speed is 40mph---
                 //---Speed: 40---
                 //---Car is decelerating...Speed is 30mph---
                 //---Speed: 30---
                 //----Car is decelerating...Speed is 20mph--
                 //---Speed: 20---
                 //---Car is decelerating...Speed is 10mph---
                 //---Speed: 10---
                 //---Car has completely stopped! Speed is 0mph---
                 //---Speed: 0---
```

As you call the various methods of the `Car` instance, the various methods in the `CarDelegate` protocol will be fired and the result will be printed on the screen, as shown in the preceding code snippet.

How Protocols and Delegates are Used in Real-Life Apps

To see how protocols and delegates work in the iOS design pattern, consider a section that examines how you obtain location information. In iOS, the `CLLocationManager` class `(Location Manager)` helps you find the location of a device. To use the `CLLocationManager` class, create an instance of it, say, in your View Controller:

```
import CoreLocation

class ViewController: UIViewController {
   var lm: CLLocationManager!
```

You can then configure the instance of the `CLLocationManager` class. In particular, you set its `delegate` property, as shown below:

```
lm = CLLocationManager()
lm.delegate = self
lm.desiredAccuracy = 0
lm.distanceFilter = 0
```

When you set the `delegate` property to `self`, it means that the class containing the `lm` variable needs to conform to the protocol dictated by the `CLLocationManager` class, which in this case is `CLLocationManagerDelegate`. Therefore, you need to add the following to the `ViewController` class:

```
class ViewController: UIViewController, CLLocationManagerDelegate {
```

The `CLLocationManagerDelegate` protocol contains a number of methods that the conforming class can implement, including the following:

```
optional func locationManager(_ manager: CLLocationManager!, didUpdateLocations locations:
[AnyObject]!)
// Fired when new location data is available
optional func locationManager(_ manager: CLLocationManager!, didFailWithError error:
NSError!)
//Fired when the location manager is unable to retrieve the location value
```

In this case, if you want to display the obtained location, the `ViewController` class should implement the first method, as shown below:

```
import CoreLocation

class ViewController: UIViewController, CLLocationManagerDelegate {

   var lm: CLLocationManager!

   required init(coder aDecoder: NSCoder)
   {
       super.init(coder: aDecoder)
   }

   override func viewDidLoad() {
```

```
        super.viewDidLoad()

        lm = CLLocationManager()
        lm.delegate = self
        lm.desiredAccuracy = 0
        lm.distanceFilter = 0

        if (UIDevice.currentDevice().systemVersion as
            NSString).floatValue>=8.0 {

            //---request for foreground location use---
            lm.requestWhenInUseAuthorization()
        }

        lm.startUpdatingLocation()
    }

    func locationManager(manager: CLLocationManager!,
        didUpdateLocations locations: [AnyObject]!) {

            var newLocation = locations.last as CLLocation
            println("\(newLocation.coordinate.latitude)")
            println("\(newLocation.coordinate.longitude)")
            println("\(newLocation.horizontalAccuracy)")
    }
```

In the above example, the `locationManager()` method will be called whenever the Location Manager is able to obtain new locations.

How to Write Adaptable and Reusable Codes - Generics

Generics are one of the most important features of Swift. They are so important that most of the types and classes in Swift are created using them. Generics enable you to write highly reusable functions that can work with a variety of data types. With generics, you specify a placeholder for the data type with which your generic code (functions, classes, structures, and protocols) is working. The actual data type to use is specified only at a later stage when the generic code is being used.

To understand generics, consider the following example of a function:

```
func swapNums(inout item1:Int, inout item2:Int) {     //declares two parameters
    let temp = item1
    item1 = item2
    item2 = temp
}
```

The `swapNums()` function is used to swap the values of two `Int` variables:

```
var num1 = 5
var num2 = 6
println("\(num1), \(num2)")   //---5,6---
```

```
swapNums(&num1, &num2)
println("\(num1), \(num2)")  //---6,5---
```

Note that the `swapNums()` function only allows you to swap two integer values.

If you want the function to swap two `String` variables, you need to create another function, as shown below:

```
func swapStrings(inout item1:String, inout item2:String) {
    let temp = item1
    item1 = item2
    item2 = temp
}
```

Both functions have the same implementation; only the type of variables you are dealing with is different. The same is true if you want to swap two `Double` values:

```
func swapDoubles(inout item1:Double, inout item2:Double) {
    let temp = item1
    item1 = item2
    item2 = temp
}
```

As you can see, creating separate functions for different data types creates a lot of duplication of code (and effort).

How to Implement Generics in Code

Using generics, you can rewrite the `swapNums()`, `swapStrings()`, and `swapDoubles()` functions using a single generic function:

```
func swapItems<T>(inout item1:T, inout item2:T) {
    let temp = item1
    item1 = item2
    item2 = temp
}
```

The generic version of the function looks almost the same as the other three functions, except that instead of specifying the type of arguments the function is expecting, you use `T` as the placeholder:

```
func swapItems<T>(inout item1:T, inout item2:T) {
```

Another common placeholder name is `ItemType`. If you use `ItemType` as the placeholder, then the function declaration would look like as shown below:

```
func swapItems<ItemType>(inout item1:ItemType, inout item2:ItemType) {
```

You can now call the `swapItems()` function just as you would when you call the `swapNums()` function:

```
var num1 = 5
var num2 = 6
swapItems(&num1, &num2)
println("\(num1), \(num2)")   //---6, 5---
```

The compiler will infer from the type of `num1` when you call the `swapItems()` function. In this case, `T` would be of type `Int`. Likewise, if you call the `swapItems()` function using arguments of type `String`, `T` would now be `string`:

```
var str1 = "blueberry"
var str2 = "apple"
println("\(str1), \(str2)")  //---blueberry, apple---
swapItems(&str1, &str2)
println("\(str1), \(str2)")  //---apple, blueberry---
```

The same behavior applies to `Double` types, as shown below:

```
var price1 = 23.5
var price2 = 16.8
println("\(price1), \(price2)")  //---23.5, 16.8---
swapItems(&price1, &price2)
println("\(price1), \(price2)")  //---16.8, 23.5---
```

You also have functions that accept arguments of multiple data types. For example, if you are writing a function that deals with `Dictionary` types, you have to deal with key-value pairs, as shown in the following function:

```
func addToDictionary(key:Int, value:String) {
    ...
}
```

In the preceding function stub, you have two parameters: one of type `Int` and the other of type `String`. The generic version of this function would look like as follows:

```
func addToDictionary<KeyType, ValueType>(key:KeyType, value:ValueType){
...
}
```

Here, `KeyType` and `ValueType` are placeholders for the actual data type that you will use.

How to Define Generic Classes and Structures

Generics are not limited to functions; you can also have `generic` types. `Generic` types can be any of the following:

- Classes
- Structures
- Protocols

Classes

For generics in classes, consider the following example:

```
class MyIntStack {
   var elements = [Int]()
   func push(item:Int) {
       elements.append(item)
   }
```

```
    func pop() -> Int! {
        if elements.count>0 {
            return elements.removeLast()
        } else {
            return nil
        }
    }
}
```

The above code snippet is a classic implementation of a stack data structure in Swift. A stack data structure enables you to `push` (insert) and `pop` (remove) items in the Last-In-First-Out (LIFO) fashion.

In the preceding implementation, `MyIntStack` is dealing only with the `Int` type. Observe that the `pop()` method returns a value of type `Int!` (implicit optional). This ensures that in the event that the stack is empty, a `pop` operation will simply return a `nil` value.

You can use `MyIntStack` as follows:

```
var myIntStack = MyIntStack()
myIntStack.push(5)
myIntStack.push(6)
myIntStack.push(7)
println(myIntStack.pop())  //---7---
println(myIntStack.pop())  //---6---
println(myIntStack.pop())  //---5---
println(myIntStack.pop())  //---nil---
```

You can rewrite the class as a generic class, as shown below:

```
class MyStack<T> {
    var elements = [T]()
    func push(item:T) {
        elements.append(item)
    }

    func pop() -> T! {
        if elements.count>0 {
            return elements.removeLast()
        } else {
            return nil
        }
    }
}
```

To use the `MyStack` class for `Int` values, simply specify the data type enclosed with angle brackets (<>) during instantiation of the class, as shown below:

```
var myIntStack = MyStack<Int>()
```

You can now use the class as usual:

```
myIntStack.push(5)
myIntStack.push(6)
myIntStack.push(7)
println(myIntStack.pop())     //---7---
println(myIntStack.pop())     //---6---
println(myIntStack.pop())     //---5---
println(myIntStack.pop())     //---nil---
```

You can also use the `MyStack` class with the `String` type:

```
var myStringStack = MyStack<String>()
myStringStack.push("Programming")
myStringStack.push("Swift")
println(myStringStack.pop())  //---Swift---
println(myStringStack.pop())  //---Programming---
println(myStringStack.pop())  //---nil---
```

Structures

Generics also apply to structures.

Consider the following implementation of a queue using a structure:

```
struct MyIntQueue {
   var elements = [Int]()
   var startIndex = 0

   mutating func queue(item: Int) {
       elements.append(item)
   }

   mutating func dequeue() -> Int! {
       if elements.isEmpty {
           return nil
       } else {
           return elements.removeAtIndex(0)
       }
   }
}
```

A queue is a data structure that allows you to `queue` (insert) and `dequeue` (retrieve) items.

In the preceding implementation, `MyIntQueue` deals only with the `Int` type. You can use it as follows:

```
var myIntQueue = MyIntQueue()
myIntQueue.queue(7)
myIntQueue.queue(8)
println(myIntQueue.dequeue())   //---7---
```

```
println(myIntQueue.dequeue())    //---8---
println(myIntQueue.dequeue())    //---nil---
```

If you rewrite the current implementation to use generics, you will get the following structure:

```
struct MyGenericQueue<T> {
   var elements = [T]()
   var startIndex = 0

   mutating func queue(item: T) {
       elements.append(item)
   }

   mutating func dequeue() -> T! {
       if elements.isEmpty {
           return nil
       } else {
           return elements.removeAtIndex(0)
       }
   }
}
```

You can now use the `MyGenericQueue` structure for any specified data type:

```
var myGenericQueue = MyGenericQueue<String>()
myGenericQueue.queue("Hello")
myGenericQueue.queue("Swift")
println(myGenericQueue.dequeue())  //---Hello---
println(myGenericQueue.dequeue())  //---Swift---
println(myGenericQueue.dequeue())  //---nil---
```

Protocols

Generics can also be applied to protocols.

Consider the following `MyStackProtocol` protocol:

```
protocol MyStackProtocol {
   typealias T
   func push(item:T)
   func pop() -> T!
   func peek(position:Int) -> T!
}
```

In this example, the `MyStackProtocol` protocol specifies that any class, which wants to implement a stack data structure, needs to implement the following three methods:

- **`push()`** —Accepts an argument of type `T`
- **`pop()`** —Returns an item of type `T`
- **`peek()`** —Accepts an integer argument and returns an item of type `T`

The protocol does not dictate how elements in the stack are to be stored. For example, one implementation can use an array, while another can use a double-linked list. Because the protocol does not dictate the data type that the stack needs to deal with, it declares an associated type using the `typealias` keyword, as shown below:

```
typealias T
```

`T` is the placeholder for the actual data type that would be used by the implementer of this protocol. When implementing the protocol, you need to implement the required methods declared in the protocol in your implementing class. The following code snippet shows an example:

```
class MyOwnStack: MyStackProtocol {

    typealias T = String

    var elements = [String]()

    func push(item:String) {
        elements.append(item)
    }

    func pop() -> String! {
        if elements.count>0 {
            return elements.removeLast()
        } else {
            return nil
        }
    }

    func peek(position:Int) -> String! {
        if position<0 || position>elements.count-1 {
            return nil
        } else {
            return elements[position]
        }
    }
}
```

Here, the `MyOwnStack` class conforms to the `MyStackProtocol` protocol. Because you are now implementing a stack to manipulate `String` types, you assign `T` to `String`, as shown below:

```
    typealias T = String
```

In fact, there is no need to explicitly declare the preceding statement; the type of `T` can be inferred from the implementation:

```
    func push(item:String) {    //---type of item is String---
        elements.append(item)
    }
```

You can rewrite the `MyOwnStack` class as shown below:

```
class MyOwnStack: MyStackProtocol {

   var elements = [String]()

   func push(item:String) {
       elements.append(item)
   }

   func pop() -> String! {
       if elements.count>0 {
           return elements.removeLast()
       } else {
           return nil
       }
   }

   func peek(position:Int) -> String! {
       if position<0 || position>elements.count-1 {
           return nil
       } else {
           return elements[position]
       }
   }
}
```

You can use the `MyOwnStack` class as follows:

```
var myOwnStack = MyOwnStack()
       myOwnStack.push("Swift")
       myOwnStack.push("Hello")
       println(myOwnStack.pop())    //---Hello---
       println(myOwnStack.pop())    //---Swift---
```

However, because we are talking about generics in this session, the `MyOwnStack` class should ideally be a generic class as well. Here is the generic implementation of the `MyOwnStackProtocol` protocol:

```
class MyOwnGenericStack<T>: MyStackProtocol {
   var elements = [T]()

   func push(item:T) {
       elements.append(item)
   }

   func pop() -> T! {
        if elements.count>0 {
            return elements.removeLast()
        } else {
```

```
            return nil
        }
    }

    func peek(position:Int) -> T! {
        if position<0 || position>elements.count-1 {
            return nil
        } else {
            return elements[position]
        }
    }
}
```

You can now use the `MyOwnGenericStack` class as follows:

```
var myOwnGenericStack = MyOwnGenericStack<String>()
      myOwnGenericStack.push("Swift")
      myOwnGenericStack.push("Hello")
      println(myOwnGenericStack.pop())    //---Hello---
      println(myOwnGenericStack.pop())    //---Swift---
```

Specifying Requirements for Generics in Associated Types

Suppose you have a function that compares two stacks to determine whether they are equal (i.e., have the same elements and count). Your function might look like this:

```
    func compareMyStacks
        <ItemType1:MyStackProtocol, ItemType2:MyStackProtocol>
        (stack1: ItemType1, stack2:ItemType2) -> Bool {

        ...
        return true
    }
```

In the `compareMyStacks()` function, you specified it as a generic function that takes two stacks as arguments, first of `ItemType1` and second of `ItemType2`. These two types must conform to the `MyStackProtocol` protocol. A use of the `compareMyStacks()` function might look as shown below:

```
var myOwnGenericStack1 = MyOwnGenericStack<String>()
var myOwnGenericStack2 = MyOwnGenericStack<String>()
var same = compareMyStacks(myOwnGenericStack1, stack2:myOwnGenericStack2)
```

In this case, because both stacks (`myOwnGenericStack1` and `myOwnGenericStack2`) use the `String` type, the stacks can be compared. However, what if you want to compare stacks of different types? In that case, it is not possible to perform the comparison and you need to enforce this restriction based on the type acceptable to the `compareMyStacks()` function. You can do so by specifying a `where` condition:

```
func compareMyStacks
    <ItemType1:MyStackProtocol, ItemType2:MyStackProtocol
    where ItemType1.T == ItemType2.T>
    (stack1: ItemType1, stack2:ItemType2) -> Bool {
    ...
    return true
}
```

In the preceding statement, the `where` condition dictates that the type used by the two arguments (which conforms to the `MyStackProtocol` protocol) must be the same. If you now try to compare two stacks of different types, the compiler will generate an `error`:

```
var myOwnGenericStack2 = MyOwnGenericStack<String>()
    var myOwnGenericStack3 = MyOwnGenericStack<Double>()

    //---error---
    compareMyStacks(myOwnGenericStack2, stack2: myOwnGenericStack3)
```

In the preceding code snippet, `myOwnGenericStack2` uses the `String` type and `myOwnGenericStack3` uses the `Double` type. Therefore, passing them as arguments to the `compareMyStacks()` function violates the `where` clause.

In addition to ensuring that the type for the two arguments is the same, you may need to enforce the constraint that the arguments are of a specific type, such as those that conform to the `Comparable` protocol:

```
func compareMyStacks<
    ItemType1:MyStackProtocol, ItemType2:MyStackProtocol
    where ItemType1.T == ItemType2.T, ItemType1.T:Comparable>
    (stack1: ItemType1, stack2:ItemType2) -> Bool {
    ...
    return true
}
```

Once you have specified this constraint, you will not be able to compare stacks that use the `Bool` type (the `Bool` type does not conform to the `Comparable` protocol):

```
var myOwnGenericStack4 = MyOwnGenericStack<Bool>()
var myOwnGenericStack5 = MyOwnGenericStack<Bool>()
//---error---
```

ADDITIONAL KNOWHOW

The Swift library is mostly built with generic code. In fact, you will use generics throughout the Language Guide. Swift's `Array` and `Dictionary` are also generic collection types. In-built functions like `swap()` are part of the Swift library.

During the lab hour of this session, you will create a simple four function calculator form using the Swift language.

Cheat Sheet

- By default, Swift does not support fallthroughs; however, you can explicitly initiate a fallthrough by using the `fallthough` keyword.
- Labeled statements allow you to specifically indicate with which loop you want to break out/continue.
- Stored property in a class directly stores the value of a property within the class. Computed properties in a class do not directly store the value of a property within the class; they store it using another property.
- For comparing class instances, use the identical to (===) and not identical (!==) operators.
- For comparing class equivalence, you need to overload the == and != operators.
- Swift does not officially support the concept of abstract classes to implement them, you can improvise using a private initializer.
- When applied to a method, the `final` keyword indicates that subclasses of the current class are not allowed to override the particular method. When applied to a class, this means that the current class cannot be subclassed by another class.
- Functions are special types of closures. A closure is a function without a name.
- An array's three closure functions are `map()`, `filter()`, and `reduce()`.
- You can use the `@objc` tag to indicate to the compiler that your class is inter-operating with Objective-C.
- You can use the `required` keyword to ensure that all subclasses of a class also provide an implementation for the initializer.
- Generics is a way of coding in which functions are written in terms of placeholder types that are later replaced with specific types provided as parameters.