TATA CONSULTANCY SERVICES
Experience certainty.

Swift Programming

Basics

Introduction

Swift is an innovative new programming language for Cocoa and Cocoa Touch.

Development started in 2010, launched during WWDC 2014.

Swift took language ideas from Objective C, Rust, Haskell, Ruby, Python, C# etc.

Concise yet expressive syntax, apps run lightning-fast.

Works side-by-side with Objective C.

Intended to be more resilient against erroneous code than Objective C.

Built with the LLVM compiler included in Xcode 6.

Uses Objective C runtime, allowing C, Objective C, Objective C++ and Swift code to run within a single program.

Constants and Variables

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

Value of a constant cannot be changed once it is set

A variable can be set to a different value in the future

You can declare multiple constants or multiple variables in a single line, separated by commas

```
var x = 0.0, y = 0.0, z = 0.0
```

Constants and Variables

You can provide a type annotation when you declare a constant or a variable, to be clear about the kind of values the constant or variable can store

```swift
var welcomeMessage: String

welcomeMessage = "Hello"
```

```swift
var red, green, blue: Double
```

Swift does not require semi-colons after each statement. However, if you want to use multiple statements on a single line, semi-colons are required

```swift
let welcomeMessage = "Hello" ; println(welcomeMessage)
```

Constants and Variables

You can print the current value of a constant or a variable using the **println** function

```
println(welcomeMessage)

println("This is a string")

println("The current value of friendlyWelcome is \(welcomeMessage)")
```

Type Inference

Type inference enables a compiler to deduce the type of a particular expression automatically by examining the values you provide

```
let meaningOfLife = 42
// meaningOfLife is inferred to be of type Int

let pi = 3.14159
// pi is inferred to be of type Double

let welcomeMessage = "Hello"
// welcomeMessage is inferred to be of type String
```

If you combine integer and floating point literals in an expression, a type of **Double** will be inferred

```
let anotherPi = 3 + 0.14159
// anotherPi is also inferred to be of type Double
```

Integer and Floating Point Conversion

Conversions between integer and floating point numeric types must be made explicit

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi equals 3.14159, and is inferred to be of type Double
```

```
let integerPi = Int(pi)
// integerPi equals 3, and is inferred to be of type Int
```

Tuples

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and do not have to be of the same type as each other.

Particularly useful for returning multiple values from a function.

```
 let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals (404, "Not Found")
```

```
let (statusCode, statusMessage) = http404Error
println("The status code is \(statusCode)")
// prints "The status code is 404"
println("The status message is \(statusMessage)")
// prints "The status message is Not Found"
```

Tuples

Individual element values of a tuple can also be accessed using index numbers starting at zero

```
println("The status code is \(http404Error.0)")
// prints "The status code is 404"
println("The status message is \(http404Error.1)")
// prints "The status message is Not Found"
```

If you name the elements in a tuple, you can use the element names to access the values

```
let http200Status = (statusCode: 200, description: "OK")

println("The status code is \(http200Status.statusCode)")
// prints "The status code is 200"
println("The status message is \(http200Status.description)")
// prints "The status message is OK"
```

Optionals

Optionals are used in situations where a value may be absent.

An optional conveys that either there is a value, and it equals x or there isn't a value at all.

```
var surveyAnswer: String?
```

Example:

```
let possibleNumber = "123"
let convertedNumber = possibleNumber.toInt()
// convertedNumber is inferred to be of type "Int?", or "optional Int"

if convertedNumber != nil {
    println("convertedNumber has an integer value of \(convertedNumber!).")
}
// prints "convertedNumber has an integer value of 123."
```

Optional Binding

~~You use optional binding to find out~~ whether an optional contains a value, and if so, to make that value available as a temporary variable or a constant.

```
~~if let actualNumber = possibleNumber.toInt()~~
~~{~~
~~    println("\(possibleNumber) has an integer value of \(actualNumber)")~~
~~} else~~
~~{~~
~~    println("\(possibleNumber) could not be converted to an integer")~~
~~}~~
~~// prints "123 has an integer value of 123"~~
```

Basic Operators

Swift supports all the Unary, Binary and Ternary operators similar to C / Obj C.

Unlike the remainder operator (a % b) in C / Obj C, Swift's remainder operator can also operate on floating point numbers.

To determine the answer for a % b, the % operator calculates the following equation and returns *remainder* as its output:

a = (b × some multiplier) + remainder

```
9 % 4          // equals 1
-9 % 4         // equals -1
8 % 2.5        // equals 0.5
```

Range Operators

Closed Range Operator

The closed range operator (a...b) defines a range that runs from a to b, and includes the values a and b.

```
for index in 1...5
{
    println("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

Range Operators

Half-Open Range Operator

The half-open range operator (a..<b) defines a range that runs from a to b, but does not include b

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count
{
    println("Person \(i + 1) is called \(names[i])")
}
// Person 1 is called Anna
// Person 2 is called Alex
// Person 3 is called Brian
// Person 4 is called Jack
```

Arrays

An array stores multiple values of the same type in an ordered list.

Differ from NSArray and NSMutableArray classes, which can store any kind of object.

```
var shoppingList:[String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

Using type inference:

```
var shoppingList = ["Eggs", "Milk"]
```

Create immutable arrays using the ***let*** keyword, used for constants.

```
let shoppingList = ["Eggs", "Milk"]
//immutable array, contents cannot be changed
```

Dictionaries

Dictionary stores multiple values of the same type, each value associated with a unique key or identifier.

```
var airports:[String: String] = ["TYO": "Tokyo", "DUB": "Dublin"]
```

Using type inference:

```
var airports = ["TYO": "Tokyo", "DUB": "Dublin"]
```

Create immutable dictionaries using the ***let*** keyword.

```
let airports = ["TYO": "Tokyo", "DUB": "Dublin"]
```

Topics

- Control Flow
- Functions
- Enumerators
- Classes and Structures

Control Flow

Swift provides all the familiar control flow statements from C-like languages.

- ***for*** and ***while*** loops
- ***if*** and ***switch*** statements
- ***break*** and ***continue*** statements
- ***for-in*** loop

The cases of a switch statement do not “fall through” to the next case in Swift.

For Loops

for-in loop

```
for index in 1...5 {
    println("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

For Loops

for-in loop

If you don't need each value from the range, you can ignore the values by placing an underscore in place of a variable name.

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
println("\(base) to the power of \(power) is \(answer)")
// prints "3 to the power of 10 is 59049"
```

For Loops

for-in loop

If you don't need each value from the range, you can ignore the values by placing an underscore in place of a variable name.

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
      answer *= base
 }
println("\(base) to the power of \(power) is \(answer)")
// prints "3 to the power of 10 is 59049"
```

For Loops

for-in loop for iterating over arrays and dictionaries

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    println("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    println("\(animalName)s have \(legCount) legs")
}
// spiders have 8 legs
// cats have 4 legs
// ants have 6 legs
```

For Loops

for-in loop for iterating over a string

```
for character in "Hello" {
    println(character)
}
// H
// e
// l
// l
// o
```

For Loops

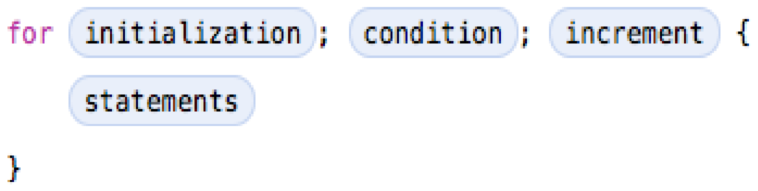
for-in loop for iterating over a string

```
for character in "Hello" {
    println(character)
 }
// H
// e
// l
// l
// o
```

For Loops

Traditional for loop

General form:



```
for initialization; condition; increment {
    statements
}
```

```
for var index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
```

while and do-while

while loop

General form:

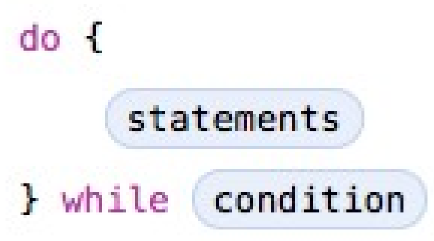
```
while condition {
    statements
}
```

```
var counter = 5
let target = 100
while counter < target
{
    counter *= 5
    println("counter is \(counter)")
}
// counter is 25
// counter is 125
```

while and do-while

do-while loop

General form:



```
do {
    statements
} while condition
```

```
var counter = 5
let target = 100
do
{
    counter *= 5
    println("counter is \(counter)")
}while counter < target
// counter is 25
// counter is 125
```

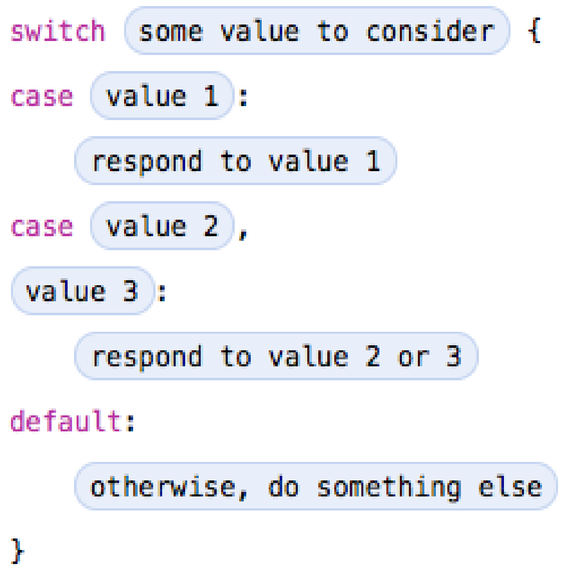
Conditional Statements

if statement

```
var temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    println("It's really warm. Don't forget to wear sunscreen.")
} else {
    println("It's not that cold. Wear a t-shirt.")
}
// prints "It's really warm. Don't forget to wear sunscreen."
```

Conditional Statements

switch statement



```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,
value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

Conditional Statements

switch statement

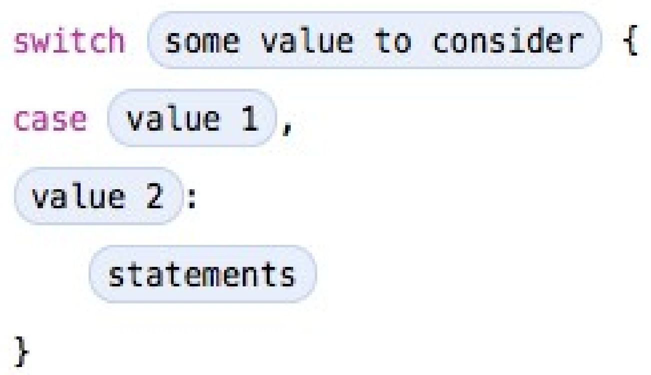
The body of each case must contain at least one executable statement.

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a":
case "A":
    println("The letter A")
default:
    println("Not the letter A")
}
// this will report a compile-time error
```

Conditional Statements

switch statement

Multiple matches for a single switch case can be separated by commas.



```
switch some value to consider {
case value 1,
value 2:
    statements
}
```

Functions

```
func sayHello(personName: String) -> String
{
    let greeting = "Hello, " + personName + "!"
    return greeting
}
```

```
println(sayHello("Anna"))
// prints "Hello, Anna!"
println(sayHello("Brian"))
// prints "Hello, Brian!"
```

Functions

Multiple input parameters:

```
func halfOpenRangeLength(start: Int, end: Int) -> Int {
    return end - start
}


println(halfOpenRangeLength(1, 10))
// prints "9"
```

Functions without parameters:

```
func sayHelloWorld() -> String {
    return "hello, world"
}


println(sayHelloWorld())
// prints "hello, world"
```

Functions

Multiple return values:

```
func minMax(array:[Int]) -> (min:Int, max:Int)
{
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}


let bounds = minMax([8, -6, 2, 109, 3, 71])
println("min is \(bounds.min) and max is \(bounds.max)")
// prints "min is -6 and max is 109"
```

Functions

External parameter names:

```
func join(s1: String, s2: String, joiner: String) -> String {
    return s1 + joiner + s2
}


join("hello", "world", ", ")
// returns "hello, world"
```

Functions

External parameter names:

```
func someFunction(externalParameterName localParameterName: Int)
{
    // function body goes here, and can use localParameterName
    // to refer to the argument value for that parameter
}
```

```
func join(string s1: String, toString s2: String, withJoiner joiner: String)
    -> String
{
        return s1 + joiner + s2
}
```

```
join(string: "hello", toString: "world", withJoiner: ", ")
// returns "hello, world"
```

Functions

Default parameter values:

```
func join(string s1: String, toString s2: String,
    withJoiner joiner: String = " ") -> String
{
    return s1 + joiner + s2
}


join(string: "hello", toString: "world", withJoiner: "-")
// returns "hello-world"


join(string: "hello", toString: "world")
// returns "hello world"
```

Enumerations

An enumeration defines a common type for a group of related values.

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


var directionToHead = CompassPoint.West

directionToHead = .East
```

Enumerations

Associated values:

Enables you to store additional custom information along with the member value.

The associated value type can be different for each member of the enumeration.

8 85909 51226 3

```
enum Barcode
{
    case UPCA(Int, Int, Int, Int)
    case QRCode(String)
}

var productBarcode = Barcode.UPCA(8, 85909, 51226, 3)
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

Enumerations

Associated values:

Enables you to store additional custom information along with the member value.

The associated value type can be different for each member of the enumeration.

```
switch productBarcode {
case let .UPCA(numberSystem, manufacturer, product, check):
    println("UPC-A: \(numberSystem), \(manufacturer), \(product), \(check).")
case let .QRCode(productCode):
    println("QR code: \(productCode).")
}
// prints "QR code: ABCDEFGHIJKLMNOP."
```

Classes and Structures

Building blocks of your code

Swift does not require you to create separate interface and implementation files for custom classes and structures

External interface to a class or a structure is automatically made available to other code for use

Classes and Structures

Functionality	Classes	Structures
Properties	🟩	🟩
Methods	🟩	🟩
Subscripts	🟩	🟩
Initializers	🟩	🟩
De-initializers	🟩	🟥
Extensions	🟩	🟩
Protocols	🟩	🟩
Inheritance	🟩	🟥
Type Casting	🟩	🟥
Reference Counting	🟩	🟥

Definition Syntax

```
class SomeClass
{
    //Class definition goes here
}

struct SomeStructure
{
    //Structure definition goes here
}
```

Definition Syntax - Example

```
struct Resolution
{
    var width = 0
    var height = 0
}

class VideoMode
{
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

Creating Instances

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Simplest form of initializer syntax

Creates a new instance of the class or structure with any properties initialized to their default values

Accessing Properties

```swift
let someResolution = Resolution()
let someVideoMode = VideoMode()

println("The width of someResolution is \(someResolution.width)")
//prints "The width of someResolution is 0"

println("The width of someVideoMode is \(someVideoMode.resolution.width)")
//prints "The width of someVideoMode is 0"

someVideoMode.resolution.width = 1280
println("The width of someVideoMode now is \(someVideoMode.resolution.width)")
//prints "The width of someVideoMode now is 1280"
```

Swift enables you to set sub-properties of a structure property directly

Memberwise Initializers

All Structures have an automatically generated memberwise initializer

Class instances do not receive a default memberwise initializer

```
let vga = Resolution(width: 640, height: 480)
```

Value Types

Structures and Enumerations are value types

Value types are copied when it is assigned to a variable or a constant, or when it is passed to a function

All of the basic types in Swift – integers, floating point numbers, booleans, strings, arrays and dictionaries – are value types

Value Types - Structures

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd

cinema.width = 2048

println("Cinema is now \(cinema.width) pixels wide")
//prints "Cinema is now 2048 pixels wide"

println("hd is still \(hd.width) pixels wide")
//prints "hd is still 1920 pixels wide"
```

Value Types - Enumerators

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

var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection

currentDirection = .East
if rememberedDirection == .West
{
    println("The remembered direction is still West")
}

//prints "The remembered direction is still West"
```

Reference Types

Classes are reference types

Reference types are not copied when it is assigned to a variable or a constant, or when it is passed to a function

A reference to the same existing instance is used instead

Reference Types – Sample Code

```
let tenEighty = VideoMode()

tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25

let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30

println("The frameRate property of tenEighty is now \
            (tenEighty.frameRate)")

//prints "The frameRate property of tenEighty is now 30.0"
```

Identity Operators

It is possible for multiple constants and variables to refer to the same single instance of a class

Identity operators can be used to find out if two variables or constants refer to exactly the same instance of a class

Swift provides two identity operators

- Identical to (===)
- Not Identical to (!===)

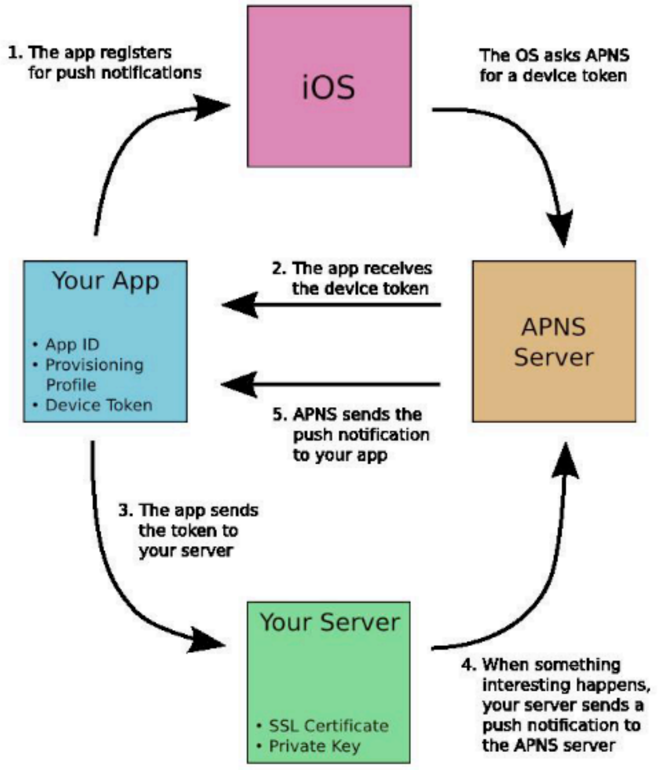
Identical to (===) does not mean the same thing as equal to (==)

Identity Operators – Sample Code

```
if tenEighty === alsoTenEighty
{
    println("tenEighty and alsoTenEighty refer to the same
              VideoMode instance")
}

//prints "tenEighty and alsoTenEighty refer to the same
   VideoMode instance"
```

Apple Push Notification Service


1. The app registers for push notifications
iOS
The OS asks APNS for a device token
Your App
• App ID
• Provisioning Profile
• Device Token
2. The app receives the device token
APNS Server
5. APNS sends the push notification to your app
3. The app sends the token to your server
Your Server
• SSL Certificate
• Private Key
4. When something interesting happens, your server sends a push notification to the APNS server

TATA CONSULTANCY SERVICES

Experience certainty.

TATA

THANK YOU