Introduction to Swift

Lecture 2

Jonathan R. Engelsma, Ph.D.

TOPICS

- Swift Language Overview
- Our first program!
- Variables and constants
- Strings
- Collection Types
- Control Flow
- Functions

THE SWIFT PROGRAMMING LANGUAGE

- A surprise introduced by Apple @ WWDC in June 2014.
- An industrial-quality systems programming language that is as expressive and enjoyable to use as a scripting language.
- Seamless access to existing Cocoa frameworks.
- Successor to legacy Objective-C language, but provides mix-and-match interoperability with Objective-C code.

WHY A NEW LANGUAGE?

- Easier for new programmers to get up to speed in iOS development.
- New language was necessary in order to ensure compatibility with existing frameworks.
- Easier to create a new language that feels modern, but still adapts to Objective-C / C conventions.

SWIFT CHARACTERISTICS

- Compiled: source —> bytecodes, not interpreted.
- Strong and static typing: types are clearly identified and cannot change (e.g. compiler generates faster safer code).
- Automatic ref counting: objects are automatically freed from memory when there are no more references.
- Name-spaced: makes our code easier to coexist with other people's code.

OUR OBLIGATORY FIRST PROGRAM!

- a complete Swift program!
- no imports / includes!
- no main program - code written at global scope becomes the entry point.
- **No semicolons!**

```
println("Hello World")
```

CONSTANTS

```
let implicitConst = 25
let explicitConst: String = "hello"

let explicitDouble: Double
explicitDouble = 3.14
```

- Use the "let" statement to define constants.
- values doesn't need to be known at compile time, but can be assigned only once.
- value assigned must be the same type as the variable name.
- Types can be implicit or explicit.

VARIABLES

```
var aNiceMessage = "bye"
var myCounter: Int = 1
++myCounter
aNiceMessage = "My counter is "
let msg = aNiceMessage + String(myCounter)
```

- Use the "var" statement to define variables - values can be mutated!
- value assigned must be the same type as the variable name.
- Types can be implicit or explicit.
- Type conversion is explicit.

STRINGS

```
var aString = "favorites: "
let g,r,b: String
g="green"; r="red"; b="blue"
aString += g + ", " + r + ", " + b
```

- String type is an ordered collection of Character: "hello, world"
- Bridged to NSString in Objective-C.
- String is a *value* type: copied when passed to function or method. (different than NSString in this regard!)
- Strings defined with var can be mutated, strings defined with let cannot be mutated!

STRINGS

- concatenation: can be accomplished with the + operator
- interpolation: can construct new strings from a mix of values/ expressions. using \().
- Compare using == operator.
- Use the hasPrefix / hasSuffix methods to check how a string starts/ends.

STRINGS

```
var language = "Swift"
var productivity = 10
var output = "My favorite language " + language + " makes me \(productivity)X more productive"

if language == "Swift" {
    println("hurrah")
} else {
    println("too bad!")
}

if output.hasPrefix("My favorite") {
    println("yup")
}

if(output.hasSuffix("10X more productive")) {
    println("you bet!")
}
```

ARRAYS IN SWIFT

- Arrays store *ordered* lists of values
- All values in a given array are of the *same type*
- Values are accessed via methods, properties and subscripting.

ARRAYS

```
var shoppingList: [String] = ["milk", "eggs", "bread"]
println("There are \(shoppingList.count) items in the list")
println("The last item is " + shoppingList[2])
shoppingList.append("chips")
shoppingList += ["dogfood","catfood"]
```

```
// iterating through an array in Swift!
for item in shoppingList {
    println(item)
}

for (index, value) in enumerate(shoppingList) {
    println("Item \(index + 1): \(value)")
}
```

DICTIONARIES IN SWIFT

- Dictionaries store *multiple values of the same type*
- Each value is associated with a *unique key*
- Do not have to be in a specified order
- Use just like a real world dictionary: e.g. when you want to look up the definition of a particular key value
- Keys must be *hashable*

DICTIONARIES

```
var languageLikes: Dictionary<String,Int> = ["Java":10,"Objective-C":2,"Swift":12,"VB": 0]
var javaLikes = languageLikes["Java"]
languageLikes["Ruby"] = 15
if(languageLikes["Swift"] > languageLikes["Ruby"]) {
    println("Swift wins!")
} else {
    println("Ruby wins!")
}
```

```
for(language, likes) in languageLikes {
    println("\(language): \(likes)")
}
```

```
Java: 10
VB: 0
Objective-C: 2
Ruby: 15
Swift: 12
```

CONTROL FLOW

- for-in statements:

```
// iterate over a range!
for index in 1...5 {
    println("\(index) times 5 is \(index * 5)")
}

// C-style for loops!
for var index = 1; index <= 5; ++index {
    println("\(index) times 5 is \(index * 5)")
}

// if you don't need the index values, use undercore instead of var name
let base = 3, power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
println("\(base) to the power of \(power) is \(answer)")
```

CONTROL FLOW

- while & do-while statements:

```
// while
var n = 5, cnt = 1
do {
    println("\(cnt) times 5 is \(cnt * 5)")
    cnt++;
} while(cnt <= 5)

// do-while
cnt = 1
do {
    println("\(cnt) times 5 is \(cnt * 5)")
    cnt++;
} while(cnt <= 5)
```

CONTROL FLOW

- if / if-else

```
var temp: Int = 50
if temp <= 32 {
    println("It is freezing out here")
} else if temp > 32 && temp <= 80 {
    println("How temperate it is here!")
} else if temp > 80 && temp <= 100 {
    println("It is a sweat shop here!")
} else {
    println("The thermometer is broken!")
}
```

SWITCH STATEMENT

- Similar to C syntax, but with some notable differences:
  - No need to use break statements!
  - No implicit fall through
  - Each case must contain at least one executable statement
  - Case condition can be scalar, range, or tuple!
  - Case condition can actually bind values.

CONTROL FLOW

- switch statement

```
var t: Int = 32
switch t {
case 0...32:
    println("It is freezing out here")
case 33...80:
    println("How temperate it is here!")
case 81...100:
    println("It is a sweat shop here!")
default:
    println("The thermometer is broken!")
}
```

FUNCTIONS

- Self-contained chunks of code that perform a specific task.
- Functions are typed by the return type and type of the parameters.
- Swift supports first-class functions, e.g. functions can be passed as arguments and serve as return values.
- Functions in Swift can be nested.
- Swift functions can have *multiple return values*.

FUNCTIONS

```
// define a function
func stateYoMission(actionVerb: String) -> String {
    let mission = "I will " + actionVerb + " the dog!"
    return mission
}
// invoke the function
println(stateYoMission("walk"))

// multiple parameters
func stateYoMission(actionVerb: String, noun:String) -> String {
    let mission = "I will " + actionVerb + " the " + noun + "!"
    return mission
}
//invoke
println(stateYoMission("walk","cat"))
```

FUNCTIONS

```
// define a function
func stateYoMission(actionVerb: String) -> String {
    let mission = "I will " + actionVerb + " the dog!"
    return mission
}
// invoke the function
println(stateYoMission("walk"))

// multiple parameters
func stateYoMission(actionVerb: String, noun:String) -> String {
    let mission = "I will " + actionVerb + " the " + noun + "!"
    return mission
}
//invoke
println(stateYoMission("walk","cat"))
```

FUNCTIONS

Returning multiple values from a function:

```
// function with multiple ret vals
func findTheCenter(x1:Int, y1:Int, x2:Int, y2:Int) -> (x: Int, y:Int)
{
    return ((x2 - x1) / 2, (y2 - y1) / 2)
}
println(findTheCenter(0,0,10,10))
```

FUNCTIONS

- So far, all the functions we've seen defined what are known as *local parameter names*. E.g. they are only referred to in the function implementation.
- Sometimes its useful to name parameters when you call a function. These are called *external parameter names*.

FUNCTIONS

- external parameter names:

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


// now let's add external parameter names!
func join(string s1: String, toString s2: String, withJoiner joiner: String) ->String
{
    return s1 + joiner + s2
}
// now ain't this so very purty?
join(string: "hello", toString: "world", withJoiner: ", ")
```

FUNCTIONS

- shorthand for external parameter names:

```
// now checkout this even purtier shorthand!
func joinv2(#string: String, #toString: String, #withJoiner: String) ->String
{
    return string + withJoiner + toString
}
// now ain't this so very purty?
joinv2(string: "hello", toString: "world", withJoiner: ", ")
```

FUNCTION TYPES

- Variable that refer to functions can be defined:

```
func addTwoInts(a: Int, b: Int) -> Int
{
    return a + b
}

func multiplyTwoInts(a: Int, b: Int) -> Int
{
    return a * b
}

// call a function referred to by a variable of function type
var mathFunction: (Int, Int) -> Int = addTwoInts
println("Results: \(mathFunction(4,6))")

// call another!
mathFunction = multiplyTwoInts
println("Results: \(mathFunction(4,6))")
```

FUNCTION TYPES

- Functions can be passed as parameters:

```
// passing function as parameters
func printMathResults(mathFunction: (Int, Int) -> Int, a: Int, b: Int) {
    println("Result: \(mathFunction(a,b))")
}
printMathResults(addTwoInts, 3, 5)
```

FUNCTION TYPES

- Functions can be returned from functions:

```
// returning function from function
func stepForward(input: Int) -> Int {
    return input + 1
}
func stepBackward(input: Int) -> Int {
    return input - 1
}

func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}

var curVal = 3;
let moveNearToZero = chooseStepFunction(true)
curVal = moveNearToZero(curVal)
```

NESTED FUNCTIONS

- We can rewrite the previous examples with nested funcs:

```
// nested functions:
func chooseStepFunctionv2(backwards: Bool) -> (Int) -> Int {
    func stepForwardv2(input: Int) -> Int {
        return input + 1
    }
    func stepBackwardv2(input: Int) -> Int {
        return input - 1
    }
    return backwards ? stepBackwardv2 : stepForwardv2
}
curVal = 3;
let moveAwayFromZero = chooseStepFunction(false)
curVal = moveAwayFromZero(curVal)
```

READING ASSIGNMENT

- Chapter 1-3, 5: Fundamentals (Neuburg)