Introduction to Swift

Topics

- Swift Language Overview
- Variable and constants
- Strings
- Collection Types
- Control Flow
- Functions

The Swift Programming Language

- Introduced by Apple at WWDC 2014
- Expressive scripting language
- Successor to legacy Objective-C
  - Interoperable with Objective-C code
- Complete Reference to the Swift language
- EOS MacMinis:
  - 2024: Xcode 15.x + Swift 5.9 (or newer)

Why Replace Objective-C with Swift?

- Swift is 2.6x faster than Objective-C
- Easier for new programmers to get up to speed in iOS development
  - C-objective syntax is too complicated
- Swift includes modern features
  - Automatic memory management via Automatic Reference Counting (ARC)
    - Objects are automatically freed when there are no more references
  - Type Safe Programming Language:
    - Static typing: types are clearly identified and cannot change throughout runtime
  - Namespaced (similar to Java `package` declaration)
    - avoid identifier name conflicts when using other

Hello World

```
print (“Hello World”)
```

- No required imports / includes
- No main function
  - The first line in a `.swift` file that is not a method/function nor a class declaration is the first one to execute
- No semicolon
  - Unless you put multiple statements on one line

Data Types

Kotlin vs. Swift

	Kotlin	Swift
Constant (immutable)	`val`	`let`
Variable (mutable)	`var`	`var`

Constants

- Use the "`let`" keyword
- Data type can be inferred by the compiler or explicitly provided by the programmer

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

- Values do not have to be known at **compile time**, but can be assigned only ONCE

```
let secret: Int
// The actual random value will be
// known only at runtime
secret = Int.random(1...50)
```

- Attempting to assign the second time is a compile error

```
// Both these lines trigger a
// compiler error
explicitConst = "Hi"
secret = 9612
```

Variables

- Use the "var" keyword
- Values can be mutated
- Value assigned must be the same type
- Type conversion is explicit

```
var implicitVar = 25
var explicitVar: String = "hello"
```

```
implicitVar += 1
explicitVar = "howdy"
```

```
explicitConst = "Hi"
secret = 9612
```

String Interpolation and Conversion

```
// Swift
let secret = 9612
let message = "Secret code is " + String(secret)
let msg = "Secret code is \(secret)"
```

```
// Kotlin
val secret = 9612
val message = "Secret code is " + secret.toString()
val msg = "Secret code is ${secret}"
```

Strings

- String type in an **ordered collection** of characters
  - You can use a for-loop to iterate through all the characters
- Bridged to NSString in Objective-C
  - NSString is an object
  - Swift String is a struct
- String a **value** type, the struct itself is copied when passed to function
- NSString is a **reference** type, only the address of the string is copied when passed to a function

```
let planet = “Mars”

for ch in planet {
  print(ch)
}
```

Why NSxxxx?

Steve Job Short Historical Background

- 1976: founded Apple
- 1985: was fired from Apple
- 1985: found NeXT
  - NeXT computer: NeXTstation
  - NeXTStep operating system and software
- Some NextStep classes in iOS programming
  - NSObject, NSString, NSUser, and more

String Operations

```
// Swift
let msg1 = “hello”
let msg2 = “world”
let longMsg = msg1 + “ “ + msg2
let another = “I say \(msg1) \(msg2)”
```

```
if msg1 == “hello” {

}
```

```
if msg1.hasSuffix(“ly”) {

}
```

```
// Kotlin
val msg1 = “hello”
val msg2 = “world”
val longMsg = msg1 + “ “ + msg2
val another = “I say $msg1 $msg2”
```

```
if (msg1.equals(“hello”)) {

}
```

```
if (msg1.endsWith(“ly”)) {

}
```

Arrays

```
// Swift
var planets = [“Mars”, “Venus”, “Earth”]
var primes: [Int] = [2, 3, 5, 7, 11]
var weekTemps: Array<Float> = [24.5, 36.9]
for p in planets { print(p) }
```

```
print(“We live on \(planets[2])”)
print(“Number of planets \(planets.count)”)
```

```
planets.append(“Jupiter”)
primes.append(13)

for (pos, val) in planets.enumerated() {
  print (“Item \(pos+1): \(val)”)
}
```

```
// Kotlin
val primes = mutableListOf(2, 3, 5, 7, 11)

for (p in primes) { print(p) }
```

```
print(“Number of primes ${primes.size}”)
```

```
primes.append(13)

for ((pos, val) in primes.withIndex()) {
  print (“Item $pos: $val”)
}
```

Dictionaries / Map

```
// Swift
var monthLen: [String:Int] = [
  “Jan”:31, “Feb”: 28, “Mar”:31, “Sep”:30]
print(monthLen.count)   // output 4
monthLen[“Feb”] = 29    // update value
print(monthLen.count)   // output 4
monthLen[“Dec”] = 31    // insert new
print(monthLen.count)   // output 5
```

```
print(monthLen[“Mar”])  // output 31
for (mon,len) in monthLen {
  print(“\(len) days in \(mon)”)
}

monthLen.removeValue(forKey: “Jan”)
monthLen.removeAll()
```

```
// Kotlin
var monthLen: Map<String,Int> = mutableMapOf(
  “Jan” to 31, “Feb” to  28, “Mar” to 31)
print(monthLen.size)   // output 3
monthLen[“Feb”] = 29    // update value
print(monthLen.size)   // output 3
monthLen[“Dec”] = 31    // insert new
print(monthLen.count)   // output 4
```

```
print(monthLen[“Mar”])  // output 31
for ((mon,len) in monthLen) {
  print(“$len days in $mon”)
}

monthLen.remove(“Jan”)
monthLen.clear()
```

Dictionaries / Map

```
// Swift
print(monthLen[“Mar”])  // output 31
for (mon,len) in monthLen {
  print(“\(len) days in \(mon)”)
}

for a in monthLen {
  print(“\(a.value) days in \(a.key)”)
}

for a in monthLen {
  print(“\(a.1) days in \(a.0)”)
}
```

```
// Kotlin
print(monthLen[“Mar”])  // output 31
for ((mon,len) in monthLen) {
  print(“$len days in $mon”)
}

for (a in monthLen) {
  print(“${a.value} days in ${a.key}”)
}
```

Alternate Syntax Notations

Terse Syntax	Verbose Syntax
`let numbers: [Int] = []`	`let numbers: Array<Int> = Array()`
`let daysInMonths: [String:Int] = [:]`	`let daysInMonths: Dictionary<String,Int> = Dictionary()`

Control Flow

For-in

```
// On a range (up)
for d in 3 ... 11 {
  print (d)
}
```

```
// On a range (down)
for d in stride(from: 10, to: 0, by: -2) {
  print (d)
}
```

```
// On an array
let arr = [23, 37, 71]
for n in arr {
  print(n)
}
```

```
// On a string
let msg = “Hello world”
for ch in msg {
  print (ch)
}
```

```
// On a dictionary
let httpCode: [Int:String] = [
  200: “Ok”,
  404: “Not found”]
for (code,msg) in httpCode {
  print (code, msg)
}
```

While / Repeat-While / If / If-else

```
var sum = 0
var term = 1
while sum < 10_000 {
  sum += term
  term += 1
}
```

```
var sum = 0
var term = 1
repeat {
  sum += term
  term += 1
} while sum < 10_000
```

```
var timeInSec = 200
if timeInSec < 60 {
  print (“Less than a minute”)
} else if timeInSec < 3600 {
  print (“Less than an hour”)
} else if timeInSec < 24 * 3600  {
  print (“Less than a day”)
} else {
  print (“Longer than a day”)
}
```

Switch

Similar to C syntax with some differences

- No implicit fall through, therefore no need to use `break` statements
- Each case must contain at least one executable statement
- Case condition must be **exhaustive**
- Case condition can be **scalar**, **range**, or **tuple**
- Case condition can bind value

```
var timeInSec = 200
switch timeInSec {
case -1: print(“Impossible negative second?”)
case 0 ... 59:          // closed range
  print (“Less than a minute”)
case 60 ... 3599:
  print (“Less than an hour”)
default:                // must be EXHAUSTIVE
  print (“Too long”)
}
```

```
var timeInSec = 200
switch timeInSec {
case 0 ..< 60:        // half-open range
  print (“Less than a minute”)
case 60 ..< 3600:
  print (“Less than an hour”)
default:
  print (“Too long”)
}
```

Switch: with bounding case condition

Similar to C syntax with some differences

- ....
- Case condition with a tuple
- Case condition can bind value

```
var testPoint = (20, 25)

switch testPoint {
case (0, 0):
  print("Origin")
case (let x, 0):
  print("On the X-axis: \(x)")
case (0, let y):
  print("On the Y-axis: \(y)")
default:
  print("elsewhere")
}
```

Switch: order of case conditions matters!

Only the **first** matching condition will execute at runtime

```
var testPoint = (0, 0)

switch testPoint {
case (0, 0):
  print("Origin")
case (let x, 0):
  print("On the X-axis: \(x)")
case (0, let y):
  print("On the Y-axis: \(y)")
default:
  print("elsewhere")
}
```

```
var testPoint = (0, 0)

switch testPoint {
case (let x, 0):
  print("On the X-axis: \(x)")
case (0, let y):
  print("On the Y-axis: \(y)")
case (0, 0):
  print("Origin")
default:
  print("elsewhere")
}
```

Functions

Functions: Parameters vs. Arguments

- Swift explicitly distinguishes **parameters** from **arguments**
  - Parameters are the **variable names** at the function header declaration
  - Arguments are the **data** supplied when the function is invoked

```
// Function declaration
func isHoliday(year: Int, month: Int, day: Int) -> Bool {
  // code that return true or false
}
```

Parameters: year, month, day

```
// Function invocation
let birthYear = 2001
if isHoliday(year: birthYear, month: 4, day: 23) {

}
```

Arguments: birthYear, 4, 23

Functions: Parameters vs. Label Arguments

- Each parameter may be associated with an argument label
- When an argument label is present, it must be used during invocation

```
// Function declaration
func isHoliday (inYear y: Int, inMonth m: Int, day: Int) -> Bool {
  // Must use y, m, and day inside this function body.
  // Can't use inYear, inMonth

}
```

(internal) Parameters: y, m, day
(external) Label args: `inYear`, `inMonth`

```
// Use inYear inMonth when invoking the function
if isHoliday(inYear: 2025, inMonth: 4, day: 23) {

}
```

Function Overloading

- Overloading is allowed when the combination of ***function name***, ***parameters name-type***, and ***return type*** is unique

```
// These three functions are valid overloading of isHoliday
func isHoliday (year y: Int, inMonth m: Int, dayOfMonth: Int) -> Bool {
  // code that return true or false
}

func isHoliday (year y: Int, inWeek m: Int, dayOfWeek: Int) -> Bool {
  // code that return true or false
}

func isHoliday (year y: Int, inWeek m: Int, dayOfWeek: String) -> Bool {
  // code that return true or false
}
```

Functions: Omitting Label Arguments

- Use _ to omit argument label in function invocation

```
// Function declaration
func isLeap (year: Int) -> Bool {
  // code that return true or false
}
```

```
// Function invocation
if isLeap(year: 1996) {
}
```

```
// Function declaration
func isLeap (_ year: Int) -> Bool {
  // code that return true or false
}
```

```
// Function invocation
if isLeap(1996) {
}
```

Use Fn Label Arguments for Readability

```
// Ambigue: is 6 the desired length or the character position?
substring(“Allendale”, 2, 6)
```

```
// Avoid confusion
substring(“Allendale”, startAtPos: 2, length: 4)
substring(“Allendale”, startAtPos: 2, untilPos: 6)

// CAN’T swap argument order
substring(“Allendale”, untilPos: 6, startAtPos: 2) // ERROR
```

Functions: Returning Multiple Values

- Use tuple to return multiple results from a function

```
func findCheapestFlight (fromAirport: String, toAirport: String) -> (String, Float) {
  var airline: String
  var cost: Float
  // do the work here, and then return airline name and ticket price as a tuple
  return (airline, cost)
}
```

```
let (carrier, ticketCost) = findCheapestFlight(fromAirport: “GRR”, toAirport: “ORD”)

let result = findCheapestFlight(fromAirport: “GRR”, toAirport: “ORD”)
print(“Cheapest flight by \(result.0) with cost \(result.1)”
```

Function Signature (in documentations)

Use the parameter names to distinguish a specific variant of an overloaded function to use

```
isHoliday(year:month:day:)

isHoliday (year:inMonth:dayOfMonth:)

isHoliday (year:inWeek:dayOfWeek:)

distance(from:to:)

// Swift String class
init(contentsOf:encoding:)
init(contentsOfFile:encoding:)
```

Function Types

- Data can be **passed into** (as arguments) or **returned from** a function
- Same idea can be applied to code/function, i.e.
  - Function can be **passed into** (as arguments) or **returned from** a function
- For proper type compatibility checking between arguments and parameters, the compiler uses **function types**
- Consequence: Swift allows you to define variables whose type is a function

Function Types (Lambda Types)

Function types are derived from its parameter types and return type

Function signature	Function Type
`func sayHello(toName: String)`	`(String) -> Void`
`func genPassword(len: Int) -> String`	`(Int) -> String`
`func isHoliday(year:Int, month:Int, day:Int) -> Bool`	`(Int,Int,Int) -> Bool`
`func hasCommonFactor(n1:Int, n2:Int, n3:Int) -> Bool`	`(Int,Int,Int) -> Bool`
`func cheapestFlight(arr: [Flight]) -> (String, Float)`	`([Flight]) -> (String, Float)`

Passing Functions as Arguments

```
func doLogin(_ u: User, password: String, authenticateFn: (User,String) -> Bool) {
  if authenticateFn(u,password) {
    // Accept login
  } else {
    // Reject login
  }
}
```

```
func authenticateWithCaptcha (u: User, p: String) -> Bool {
  // more code here
}

// Function invocation
doLogin(____, password: “TopS3kret”, authenticateWithCaptha)

// Or if Two-Factor Authentication is desired
doLogin(____, password: “TopS3kret”, authenticateWith2FA)
```

Using typealias

```
func doLogin(_ u: User, password: String, authenticateFn: (User,String) -> Bool) {
  if authenticateFn(u,password) {
    // Accept login
  } else {
    // Reject login
  }
}
```

```
typealias AuthFunc = (User,String) -> Bool

func doLogin(_ u: User, password: String, authenticateFn: AuthFunc) {
  if authenticateFn(u,password) {
    // Accept login
  } else {
    // Reject login
  }
}
```

Reading Assignment

Appendix A1 - A6 Engelsma/Dulimarta textbook