Introduction to Swift 3

COSC346

Why Swift?

Swift

- It is hard to appreciate Object Oriented programming until you write very complex software.
- Cocoa is a complex OO framework for creating User Interfaces
- It will demonstrate OO in action as well as enable you to put your new-found knowledge about User Interfaces into practice.
- Cocoa is written in Objective-C, but Objective-C is getting a bit old.
- Swift is new and exciting and compatible with Objective-C… and is also **object-oriented**.

Why Swift?

Swift

- Modern
  - Result of research on programming languages
  - Multi-paradigm – takes ideas from many languages, incorporating their best features (in this course we will focus on the Object-Oriented aspect)
- Safe
  - Compiler forces you to do things right
  - Emphasis on detecting errors at compile time rather than run-time
- Concise
  - Easier and faster to develop software
  - Easier to create development tools
- Cocoa environment – good example of natural progression from OOP to User Interfaces

Overview

Swift

- Programming patterns for safety
  - Type checking
  - Clear distinction between variables and constants
  - Fussy compiler (but really developer's best friend)
- Modern programming features for expressiveness
  - Elegant way to do error checking with optionals
  - Computed class properties
  - Unicode-compliance inherent in Strings
  - Elegant literals for arrays and dictionaries
- Objective-C like syntax for readability
  - External names for function arguments
  - May seem odd at first, unless you're used to Objective-C
- Multi-paradigm
  - Lots of options: object-oriented, procedural and functional

“Hello, World!”

Swift

```
import Foundation

print("Hello, World!")
```

- No header files
- No main function
- No semicolons (unless you’ve got multiple statements in a single line)
- Almost like a scripting language

Variables and constants

Swift

```
var x: Int = 3 // Variable of type Int
let y: String = "cosc346” //Constant of type String

x = 4              //Value of x can change
y = "cosc360"                    ! Cannot assign to ‘let’ value ‘y’
```

- The value of a variable can vary
- The value of a constant remains constant
- Variables and constants must be of specific type…

Variables and constants

Swift

```
var x = 3 // x is an Int, because 3 is an Int literal
let y = "cosc346" // y is a String

x = 4             //Value of x can change
y = "cosc360"                    ! Cannot assign to 'let' value 'y'
```

- The value of a variable can vary
- The value of a constant remains constant
- Variables and constants must be of specific type…
- …but that type can be inferred by the compiler

Branching

Swift

if

```
let a = 7
let b = 13
if a > b {
    //a is larger than b
} else if a < b {
    //a is smaller than b
} else {
    //a is equal to b
}
```

switch

```
let cmd: Character = "q"
switch cmd {

case "l":
    print("l is for list")
case "q":
    print("q is for quit")
default:
    print("Don't understand '\(cmd)'")

}
```

Functions—Swift 2.2

External/Internal name of the 1st argument

External name of the 2nd argument

Internal name of the 2nd argument

```
func biggerNumberFrom(let x: Int, let and y: Int) -> Int {
    if x > y {
        return x
    } else {
        return y
    }
}

let a = 7
let b = 13
let n = biggerNumberFrom(a, and: b)
```

Function call

Functions—Swift 3


```
func biggerNumber(from x: Int, and y: Int) -> Int {
    if x > y {
        return x
    } else {
        return y
    }
}

let a = 7
let b = 13
let n = biggerNumber(from: a, and: b)
```

External name of the 2nd argument

Internal name of the 2nd argument

External name of the 1st argument

Internal name of the 1st argument

Function call

String interpolation


```
func biggerNumber(from x: Int, and y: Int) -> Int {
    if x > y {
        return x
    } else {
        return y
    }
}

let a = 7
let b = 13
let n = biggerNumber(from: a, and: b)
print("The bigger number of \(a) and \(b) is \(n).")
```

Gives the following output:

The bigger number of 7 and 13 is 13.

Value types and reference types

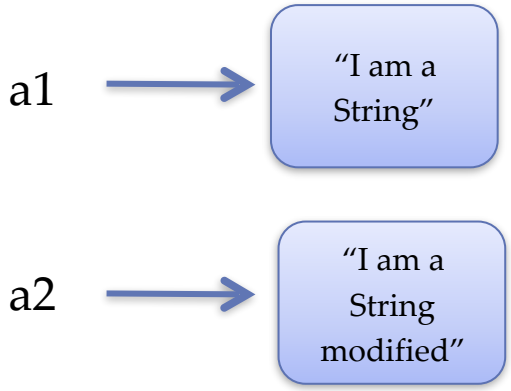
- Types have two flavours:
  - Value types – when copied or passed into a function, create a new value with same content; references to independent copies
  - Reference types – when copied or passed into a function, create a new reference to the original value; references to the same copy

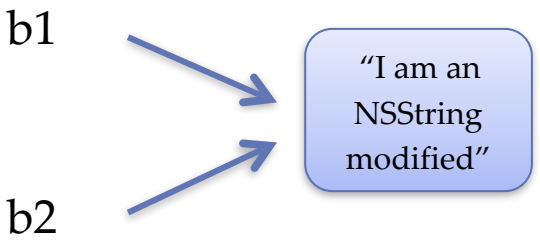
```
var a1: String = "I am a String" //New String
var a2 = a1                      //Copy of that String
a2 += " modified"

print("a1=\(a1)");      //Original string is intact
print("a2=\(a2)");      //Copy has been modified
```

```
// New NSString object
var b1 = NSMutableString(string: "I am an NSString")
var b2 = b1 //Copy of that object
b2.append(" modified")

print("b1=\(b1)");    //Original string is modified
print("b2=\(b2)");    //Copy has been modified
```





Value types and reference types

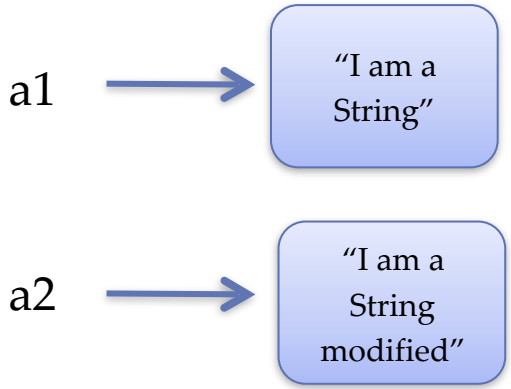
- Types have two flavours:
  - Value types – when copied or passed into a function, create a new value with same content; references to independent copies
  - Reference types – when copied or passed into a function, create a new reference to the original value; references to the same copy

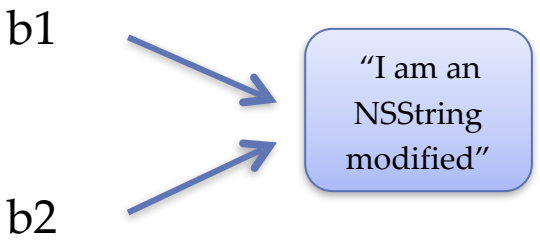
```
var a1: String = "I am a String" //New String
var a2 = a1                      //Copy of that String
a2 += " modified"

print("a1=\(a1)");      //Original string is intact
print("a2=\(a2)");      //Copy has been modified
```

```
// New NSString object
var b1 = NSMutableString(string: "I am an NSString")
var b2 = b1 //Copy of that object
b2.appendString(" modified")

print("b1=\(b1)");    //Original string is modified
print("b2=\(b2)");    //Copy has been modified
```





- All classes are reference types!

Collection types

- Tuple – a list of mixed type data

```
var errMsg: (Int, String) = (404, "Not Found")
print("Error code \(errMsg.0): \(errMsg.1).")
```

- Array – indexed list of same type data

```
var shoppingList: [String] = ["Six Eggs", "Milk", "Flour", "Baking Powder", "Bananas"]
print("Third item is: \(shoppingList[2])")
```

- Sets – unordered list

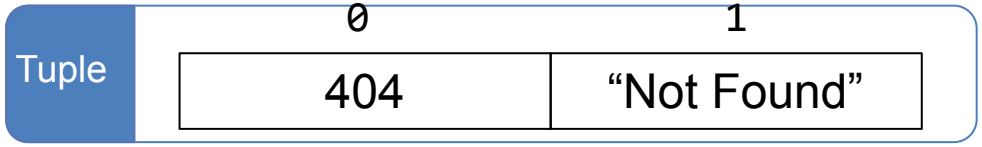
```
var favouriteGenres: Set<String> = ["Rock", "Classical", "Hip hop", "Jazz"]
if favouriteGenres.contains("Rock") {
    print("Rock is part of the set")
}
```

- Dictionary – hashed, keyword-addressable list

```
//Dictionary
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin", "LHR":
"Dublin Aiprort"]
let aname = airports["DUB"]
print("Airport DUB is \(aname!)")
```

Collection types

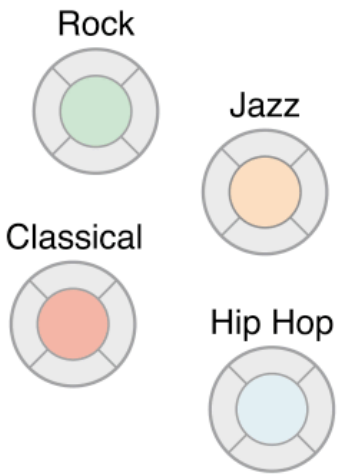
Swift



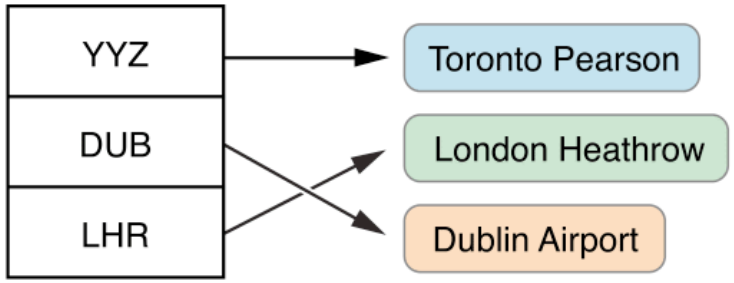
Array

Indexes	Values
0	Six Eggs
1	Milk
2	Flour
3	Baking Powder
4	Bananas

Set



Dictionary



Multi-value function return

Swift

Tuple declaration

```
func biggerAndSmallerNumber(from x: Int, and y: Int) -> (Int, Int) {
    if x > y {
        return (x, y)
    } else {
        return (y, x)
    }
}

let a = 7;
let b = 13;
let n = biggerAndSmallerNumber(from: a, and: b)
print("\(n.0) is bigger than \(n.1).")
```

Tuple creation

Tuple element access

Iteration

for

while

Swift

Range: 0, 1

```
for index in 0..<2 {
    print("index is \(index)")
}
```

Range: 0, 1, 2

```
for index in 0...2 {
    print("index is \(index)")
}
```

Range: 2, 1, 0

```
for index in (0...2).reversed() {
    print("index is \(index)")
}
```

Range: 1, 3

```
for index in stride(from:1,to:5,by:2) {
    print("index is \(index)")
}
```

Range: -1, 1, 3

```
for index in stride(from:-1,to:5,by:2) {
    print("index is \(index)")
}
```

```
var shoppingList: [String] = ["Six Eggs",
"Milk", "Flour", "Baking Powder", "Bananas"]
var index=0

while(index < shoppingList.count) {
    print("\(shoppingList[index])")
    index += 1
}
```

Iteration

for while

Swift

```
for index in 0..<2 {
    print("index is \(inde
}


for index in 0...2 {
    print("index is \(index)")
}


for index in (0...2).reverse() {
    print("index is \(index)")
}


for index in stride(from:1,to:5,by:2) {
    print("index is \(index)")
}


for index in stride(from:-1,to:5,by:2) {
    print("index is \(index)")
}
```

```
hoppingList: [String] = ["Six Eggs",
 "Flour", "Baking Powder", "Bananas"]
ex=0

index < shoppingList.count) {
(shoppingList[index])")
```

index is 0
index is 1

index is 0
index is 1
index is 2

index is 2
index is 1
index is 0

index is 1
index is 3

index is -1
index is 1
index is 3

Iteration

Swift

for

```
var favouriteGenres: Set<String> =
["Rock", "Classical", "Hip hop", "Jazz"]
for genre in favouriteGenres {
    print("\(genre)")
}
```

Rock
Classical
Jazz
Hip hop

```
var airports: [String: String] =
    ["YYZ": "Toronto Pearson",
     "DUB": "Dublin Airport",
     "LHR": "Heathrow Airport"]
for (code, name) in airports {
    print("\(code): \(name)")
}
```

DUB: Dublin Airport
LHR: Heathrow Airport
YYZ: Toronto Pearson

Type Conversion


```
let h = 5.0
let i = 100
let j = h/i
```

Binary operator '/' cannot be applied of type 'Double' and 'Int'

Type Conversion


```
let h = 5.0 //h is a Double
let i = 100 //i is an Int
let j = h/i
```

Binary operator '/' cannot be applied of type 'Double' and 'Int'

- Type conversion can be used to change a variable types in an expression.

```
let h = 5.0 //h is a Double
let i = 100 //i is an Int
let j = h/Double(i)
```

Converting i to a Double

Optionals

- May or may not hold a value.

```
var status: Int?     //Optional – can hold an Integer value
                     //or nil
var Failure: Int     //Must hold an Integer

status = nil
status = 7

Failure = nil
```

Optional declared with a ? following the type

! Cannot assign a value of type 'nil' to a value of type 'Int'

Failure is not an Optional

Optionals

- May or may not hold a value.


```
//Dictionary
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin Airport"]

//Code string
var airportCode: String = "YOW"
//Optional variable for name
var airportName: String?

//Get the name from dictionary
airportName = airports[airportCode]

//If dictionary returned non-nil, then a name has been found
print("\(airportCode): ")
if airportName != nil { //Optionals must be unwrapped in order to access data
    print("\(airportName!)")
} else {
    print("not found")
}

//Optionals can be checked for nil and unwrapped at the same time using the let keyword
print("\(airportCode): ")
if let name = airportName {
    print("\(name)")
} else {
    print("not found")
}
```

Optional unwrapped with a !
following the variable reference

Revisiting value / reference types

- Common value types:
  - struct
  - enum
  - tuple
  - Array
  - Dictionary
  - String, Int, Bool, Int8, Int16, Int32, Int64, UInt, UInt8, UInt16, UInt32, UInt64, Float, Float80, Double, …
- Common reference types:
  - class
  - NSObject

Value/reference copy playground

- Here's the playground content I was using in lectures. (Intended for copy/paste, rather than readability here!)

```
import Foundation

var a:Int = 1
var b:Int = a
a = 2
print("\(a),\(b)")

var c:String = "blah"
var d:String = c
c = "blob"
print("\(c),\(d)")

class ClassCopyTest { var t:Int = 0 }
var e:ClassCopyTest = ClassCopyTest()
var f:ClassCopyTest = e
e.t = 1
print("\(e),\(f)")

struct StructCopyTest { var t:Int = 0 }
var g:StructCopyTest = StructCopyTest()
var h:StructCopyTest = g
g.t = 1
print("\(g),\(h)")
```