Introduction to Swift 4

COSC346

Allowable Timetable Clash search

To check whether a timetable clash is allowable search for both paper codes in the clash in the box below. For the clash to be deemed Allowable, at least one of the papers must be listed and the clash must be consistent with the 'Details of clash to be allowed' provided. This means that if your timetable clash is approved, you can miss the listed paper to accommodate the clash, so long as you follow the 'Instructions for management of timetable clash' listed.

If your timetable clash meets these criteria, please write **allowable** in the comments box on the **Review and submit** page (see above).

If your timetable clash does not meet these criteria, it will be considered an Exceptional Timetable clash and will need to be assessed as such. You may also wish to consider revising your paper selection.

Please note that students are not normally permitted to have more than one allowable timetable clash per week per teaching period. Allowable clashes will be assessed, and may still be declined, as part of the Course Approval process.

Search below to find your paper

INFO301

Paper Code	Teaching Period	Details of clash to be allowed	Instructions for management of timetable clash
INFO301	S2	A one-hour clash per week in lectures is permitted.	You must review any lecture material and related media, and attend all other course related activities.

Search below to find your paper

COSC346

https://www.otago.ac.nz/study/OTAGO121418#10

Why 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

- 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 value: 'y' is a 'let' constant
```

- 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: 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 value: 'y' is a 'let' constant
```

- 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

No brackets around conditional

```swift
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

```swift
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

Swift

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 4

External name of the 2nd argument

Internal name of the 2nd argument

Swift

```
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 1st argument

Internal name of the 1st argument

Function call

String interpolation

Swift

```
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

Swift

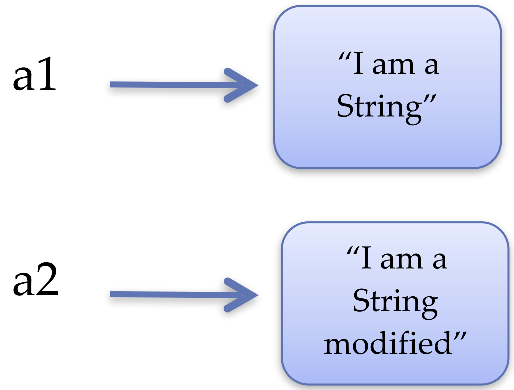
- Types have two flavours:
  - Value types – when copied or passed into a function, create a new value with same content; references to independent copies
  - Reference types – when copied or passed into a function, create a new reference to the original value; references to the same copy

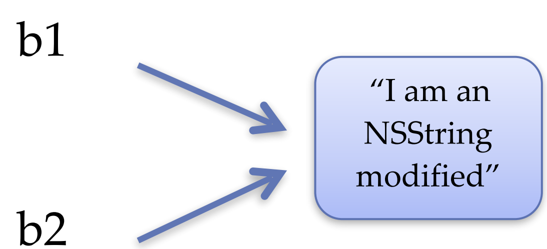
```
var a1: String = "I am a String" //New String
var a2 = a1                      //Copy of that String
a2 += " modified"

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

```
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

Swift

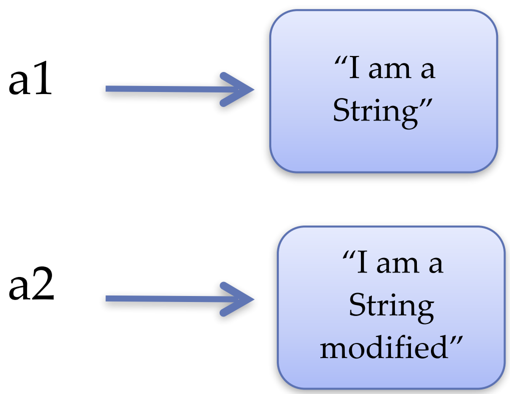
- Types have two flavours:
  - Value types – when copied or passed into a function, create a new value with same content; references to independent copies
  - Reference types – when copied or passed into a function, create a new reference to the original value; references to the same copy

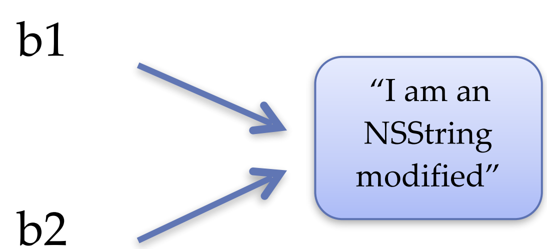
```
var a1: String = "I am a String" //New String
var a2 = a1                      //Copy of that String
a2 += " modified"

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

```
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
```





- All classes are reference types!

Collection Types

Swift

- 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 – unique unordered list

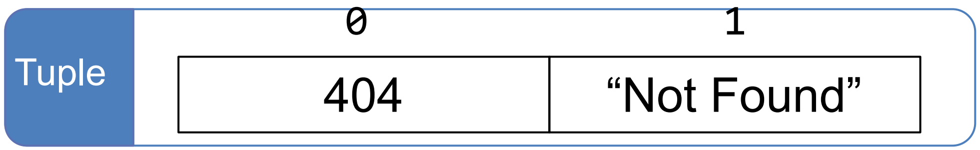
```
var favouriteGenres: Set<String> = ["Rock", "Classical", "Hip hop", "Jazz"]
if favouriteGenres.contains("Rock") {
    print("Rock is part of the set")
}
```

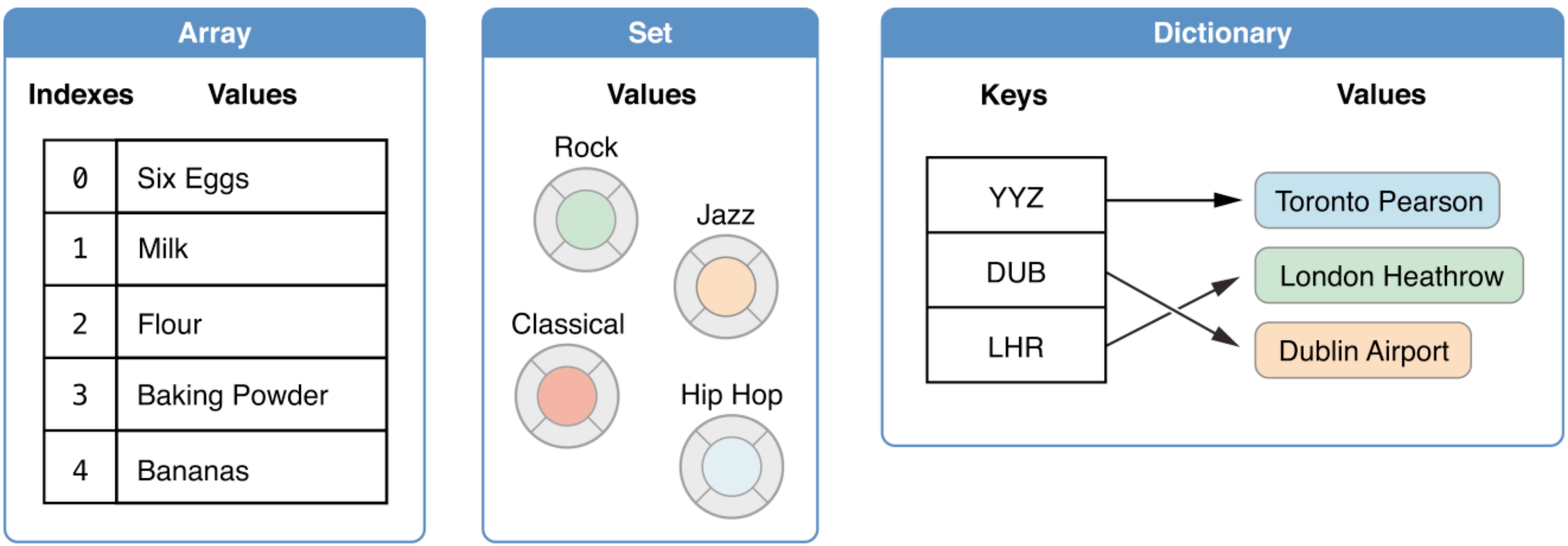
- Dictionary – hashed, keyword-addressable list

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

Collection types

Swift





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 m = biggerAndSmallerNumber(from: a, and: b)
print("\(m.0) is bigger than \(m.1).")
```

Tuple creation

Tuple element access

Iteration

Swift

for

while

Range: 0, 1

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

Range: 0, 1, 2

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

Range: 2, 1, 0

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

Range: 1, 3

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

Range: -1, 1, 3

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

Different syntax for different Swift versions

Iteration

for while

Swift

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

A index is 0
A index is 1

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

B index is 0
B index is 1
B index is 2

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

C index is 2
C index is 1
C index is 0

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

D index is 1
D index is 3

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

E index is -1
E index is 1
E index is 3

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

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

Iteration

for

Swift

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




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

Rock
Classical
Jazz
Hip hop

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

Type Conversion

Swift

```
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'
let k = i/h        ! Binary operator '/' cannot be applied of type 'Int' and 'Double'
```

- 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

Swift

- 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                    Nil cannot be assigned to type 'Int'

print(status)                    Expression implicitly coerced from 'Int?' to Any
```

Optional declared with a ? following the type

Failure is not an Optional

Optional(7)

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

Swift

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 let len = airportName?.count {
    print("The airport name is \(len) 'characters' long")
}else{
    print("🤷‍♀️")
}
```

Optional Chaining — if the airport name exists, call a method on it…

Swift

Error Handling

Adopt the Error protocol

```
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}
```

```
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

Use 'throws' in function definition and to generate error

```
do {
    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
```

catch the errors

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

Summary

- Value/Reference Types
- Optionals
- Constants and Variables
  - var and let
- Automatic Type Detection
  - But can specify the types
- Internal/External names for functions

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)")
```