Swift: Beyond the Basics

Lecture 03

Jonathan R. Engelsma, Ph.D.

TOPICS

- Closures
- Tuples
- Optionals
- Objects
  - enum
  - struct
  - class
- Protocols

CLOSURES

- Self contained blocks of functionality that can be passed around and used in your code.
- Analogous to *blocks* in Objective-C and to *lambdas* in other programming languages.
- Can capture and store references to constants/variables from the scope in which they are defined. This is known as *closing*, hence the term *closure*.

CLOSURES

- Closures take one of 3 forms:
  - ***Global functions*** are closures that have a name and do not capture any values.
  - ***Nested functions*** are closures that have a name and capture values from their enclosing functions.
  - ***Closure expressions*** are unnamed closures written in lightweight syntax that capture values from their surrounding context.

CLOSURE EXPRESSIONS

- A way to write inline closures in brief focused syntax.
- Syntactical optimizations provided for writing closures without loss of clarity of intent.
  - parameter/retval type inferencing
  - implicit returns from single-expression closures
  - shorthand arg names
  - trailing closure syntax

DEFINING A CLOSURE

```
{ (params) -> returnType in
  statements
}
```

- enclosed in {}
- -> separates the params from the return type.
- code statements follow the keyword ***in***

SIMPLE CLOSURE EXAMPLE

```
// closure examples
let students = ["joe", "camila", "ryan", "roland"]
students.map({
    (student: String) -> String in
    "\(student) is getting swifter!"
})

// type inferencing - swift knows it is a string array, so we can write like this:
let studentMsgs = students.map({student in "\(student) is very swift!"})

// shorthand arguments
let studentStatus = students.map({$0 + " will definitely pass"})
```

TRAILING CLOSURES

- If a closure is passed as last argument of a function, it an appear outside of the ():

```
// trailing closures
let studentBadNews = students.map() {$0 + " will definitely fail"}
```

- If there are no params other than the closure itself we can get rid of the parens:

```
// trailing closures - if no params, other than closure, parens not needed!
let studentUtopia = students.map {$0 + " lands a paying job!"}
```

TUPLES IN SWIFT

- a ***tuple*** is a lightweight custom ordered collection of multiple values.
- Swift tuples do not have their analog construct in Objective-C!
- Example:

```
// declaring / initializing a tuple
var coordinate: (Int, Int) = (5,7)

var location: (String, Float, Float)
location = ("GVSU", 42.965175, -85.889405)
```

TUPLE USAGE

- What can we use tuples for?
  - Assign multiple values simultaneously:

```
// tuples can be used to assign multiple vals simultanously!
var someInt: Int
var someStr : String

(someInt, someStr) = (10,"hello")
println(someInt)
println(someStr)
```

TUPLE USAGE

- What can we use tuples for?
  - returning multiple return 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))
```

TUPLE USAGE

- What can we use tuples for?
  - swap values in two variables:

```
// A tuple can be used to simply swap values
var alpha = "first"
var omega = "last"

// the first shall become last, and the last first!
(omega, alpha) = (alpha, omega)
println(omega)
println(alpha)
```

ACCESSING TUPLE VALS

- How can we access values in a tuple?

```
// alternate #1 - use underscore notation
location = ("Grand Rapids", 42.960253, -85.657857)
var (description, _, _) = location
println(description)

// alternate #2 - use index literals
var descript2 = location.0
println(descript2)

// alternate #3 - give 'em names!
var favLocation = (name:"Newaygo", lat:43.419143, long:-85.800993)
println(favLocation.name)
```

SWIFT MAGIC…

- Observe that tuple syntax looks like function parameter lists!
- Not an accident - you can pass a tuple to an argument as its parameters!

```
// function take tuples as args!
func max (int1: Int, int2: Int) -> Int
{
    if int1 > int2 {
        return int1
    } else {
        return int2
    }
}

let vals = (10, 20)
println(max(vals))
```

OPTIONALS IN SWIFT

- You can think of an ***optional*** in Swift as a box that potentially *wraps* an object of any type.
- An optional might wrap an object, or it might not!

CREATING AN OPTIONAL

- How we create an optional:

```
// Creating an optional
var mightBeAStr = Optional("Boo!")
println(mightBeAStr)

// The normal way we do it
var mightBeAStr2 : String? = "Boo"
```

- Note that these two variables are not Strings! They are Optionals.
- Assigning the wrapped type to an Optional variable causes it to automatically get wrapped!

PASSING OPTIONALS

- Optionals can be passed to functions where expected:

```
// passing an optional to a function
func gimmeAnOptional(s:String?) {}
let poof : String? = "kapoof!"
gimmeAnOptional(poof)

// if we assign the wrapped type, it gets wrapped automatically
gimmeAnOptional("cheese")
```

- If we pass the wrapped type (e.g. String) it gets automagically wrapped before passing.

PASSING OPTIONALS

- You CANNOT pass an optional type where the wrapped type is expected!

```
// we have a problem Houston!
var problemo : String? = "Make some trouble"
func gimmeAStringPls(s:String) {}
gimmeAStringPls(problemo)
                ⭘ Value of optional type 'String?' not unwrapped; did you mean to use '!' or '?'?
```

- You MUST *unwrap* an optional before using it…

UNWRAPPING OPTIONALS

- One way to unwrap an Optional is with the *forced unwrap operator,* a postfix '!':

```
// using the forced unwrap operator
var noProblemo: String? = "Behaving nicely"
func gimmeAStringPls(s:String) {}
gimmeAStringPls(noProblemo!)
```

- The ! operator simply unwraps the optional and passes to the function the string (in this case "Behaving nicely" to the function.

UNWRAPPING OPTIONALS

- We cannot send a message to a wrapped object before it is unwrapped!

```
var mightBeValid :String? = "howdy"
mightBeValid.uppercaseString
           ⛔ Value of optional type 'String?' not unwrapped; did you mean to use '!' or '?'?
```

- But this works fine:

```
var mightBeValid :String? = "howdy"
mightBeValid!.uppercaseString
```

IMPLICITLY UNWRAPPED OPTIONALS

- Swift provides another way of using an Optional where the wrapped type is expected: ImplicitlyUnwrappedOptional:

```
// implicitly unwrapped optionals
func gimmeANiceStr(s:String) {}
var mightBeNice : String! = "yeehah"
gimmeANiceStr(mightBeNice)
```

- In this situation, mightBeNice gets implicitly unwrapped by Swift when we pass it to the function.

DOES OPTIONAL CONTAIN A WRAPPED VALUE?

- To test if an optional contains a wrapped value, you can compare it to nil.
- To specify an Optional with no wrapped value, assign or pass nil.

```
// checking if an Optional wraps a value.
var mightBeNil : String? = "greetings"
if(mightBeNil == nil) {
    println("yup, it is empty") // does not print!
}
mightBeNil = nil
if(mightBeNil == nil) {
    println("yup, it is empty")
}
```

MORE OPTIONAL FACTOIDS

- Optionals get set to nil automatically.
- You cannot unwrap an Optional containing nothing, e.g. an Optional that equates to nil.
- Unwrapping an Optional that contains no value will crash your program!

CONDITIONAL UNWRAPPING

- Swift has syntax for conditionally unwrapping Optionals:
- This would be a problem:

```
// Conditional unwrapping
var mightNotBeValid :String? = nil
mightNotBeValid!.uppercaseString
```

- This would not be:

```
// Conditional unwrapping
var mightNotBeValid :String? = nil
mightNotBeValid?.uppercaseString
```

OPTIONAL COMPARISONS

- When Optionals are used in comparisons, the wrapped value is used. If there is no wrapped value, the comparison is false.

```
var aValue : String? = "amos"
if aValue == "amos" {
    println("amos is famous")
}

var aNumber : Int? = 1
if aNumber < 100 {
    println("is less")
}
```

WHY OPTIONALS?

- Provides the interchange of object values in the legacy Objective-C frameworks.
  - need a way to send / receive nil to / from Objective-C.
  - All Objective-C objects are handled as Optionals in Swift.
- Still some flux in the Swift wrappers of the legacy Objective-C frameworks!

OBJECT IN SWIFT

- Three Object Flavors in Swift
  - enum
  - struct
  - class

OBJECTS

- Declared with keyword enum, struct, or class respectively:

```
class Person { }
struct Guts { }
enum BloodType { }
```

- Can be declared anywhere in the file, within another object, within a function.
- Scope is determined by where the declaration is made.

OBJECTS

- Object declarations can have:
  - **Initializers** - also known as constructors in other languages.
  - **Properties** - variables declared at top level of an object. Can be associated with the object (instance) or class.
  - **Methods** - functions declared at top level of an object. Can be associated with the object (instance) or class.

ENUMS

- an object type whose instances represent distinct predefined alternative values. e.g. a set of constants that serve as alternatives.

```
// enum
enum BeeType {
    case Queen
    case Worker
    case Drone
}

// declaring/assigning a BeeType constant
let myBee = BeeType.Queen

// a handy shortcut, if type is known in advance.
let anotherBee: BeeType = .Worker

// Similarly when calling a function
func examineBee(type:BeeType) {}
examineBee(.Drone)
```

ENUM FACTOIDS

- Enums can be typed to Int, String, etc.
- Enums can have initializers (init), methods, and properties.
- By default, Enums are implemented as Ints with default values assigned (starting with 0).
- An enum is a value type, that is, if assigned to a variable or passed to a function, a copy is sent.

ENUMS WITH FIXED VALUES

- enums can be defined with *fixed values*:

```
// enums with fixed values
enum FancyBeeType : String {
    case Queen = "Queen"
    case Worker = "Worker"
    case Drone = "Drone"
}

// we can access the fixed value via the rawValue property
let fancyBee: FancyBeeType = .Queen
println(fancyBee.rawValue)
```

STRUCTS

- an object type in Swift, but could be thought of as a knocked down version of class. Sits in between enum and class in terms of its capabilities.

```
struct Bee {
    var type: FancyBeeType = .Worker
    var age: Int = 0
    init(type: FancyBeeType, age: Int) {
        self.type = type
        self.age = age
    }

    func description() -> String {
        return "This bee is a \(self.type.rawValue) of age \(self.age) days old."
    }
}

var busyBee: Bee = Bee(type: .Worker, age: 14)
println(busyBee.description())
```

This bee is a Worker of age 14 days old.

STRUCT FACTOIDS

- Nearly all of Swift's built-in object types are structs, Int, String, Array, etc.
- structs can have initializers, properties, and methods!
- An enum is a value type, that is, if assigned to a variable or passed to a function, a copy is sent.

CLASS

```
class Employee {
    var firstName: String
    var lastName: String
    var bio: String

    // no bio provided
    convenience init(firstName: String, lastName: String) {
        self.init(firstName: firstName, lastName: lastName, bio: "I ♡ Swift!")
    }

    // designated initializer.
    init(firstName: String, lastName: String, bio: String) {
        self.firstName = firstName
        self.lastName = lastName
        self.bio = bio
    }

    func computePay() -> Double {
        return 1000.0 * Double(count(bio))
    }
}

var steve = Employee(firstName: "Steve", lastName: "Jobs")

var tim = Employee(firstName: "Tim", lastName: "Cook", bio: "I ♡ Apple Watch!!!")
```

CLASS FACTOIDS

- A class is a *reference type*. That is, when assigned to a variable or passed to a function the reference is assigned or passed. A copy is not made!
- A class can *inherit* the properties/behavior of a parent class.
- A class instance is mutable in place. e.g., even if the reference is constant, values of the instance's properties can be changed via the reference.

PROTOCOLS IN SWIFT

- A *protocol* provides the the ability to define similarities between unrelated objects.

```
// protocols
protocol BrokenThing {
    func crash() -> String
}

class AirPlane : BrokenThing {
    func crash() -> String {
        return "Oops, this airplane just crashed"
    }
}

class Program : BrokenThing {
    func crash() -> String {
        return "I think this program has a bug."
    }
}
```

PROTOCOL FACTOIDS

- Swift protocols are equivalent to abstract methods in C++ or interfaces in Java.
- A class can subclass another class and implement one or more protocols, however, the parent class must always be listed first after the ':' in the class def, with the list of comma separated protocol names following.

READING ASSIGNMENT

- Chapter 4: Fundamentals (Neuburg)

WORKING WITH CLASSES

The RocketShip DEMO!!