University of Regina

CS 455/855
Mobile Computing

Swift

Dr. Orland Hoeber
orland.hoeber@uregina.ca
http://www.cs.uregina.ca/~hoeber/cs455/2018F

Readings

- Textbook, Chapters 1-5
  - the first chapter serves as a refresher on general object oriented programming details
  - the next four outline the basics of the Swift language

- Alternate: Swift Documentation (swift.org)

  - ** it is very important that you also read the Developer Library documentation for the API

iOS API

- In order to program for an iPhone/iPod Touch/iPad, we must program for the iOS operating system
  - use the iOS API
  - most of the code will be written in Swift
  - although the API is written in Objective-C (and some legacy C), there are only a few special cases where we need to be concerned with this

- So, in order to program for the iOS API, we need to learn to program in Swift (or Objective-C).

Swift

- Swift is a fully object-oriented language
  - everything is an object (even primitive data types)
  - everything you have learned about OO languages holds for Swift

- Syntax is similar to other C-like languages, but with a few minor differences
  - line breaks signify the end of a statement
  - semi-colons are only mandatory when multiple statements appear on the same line
  - single statements can span multiple lines, but the breaks should be put at sensible places

Swift Compiler

- As a compiled language, your code must satisfy the compiler before it can be executed.
- The compiler is very strict, which is a good thing.
- The compiler is also smart, and will often give you a suggestion for how to fix the mistake.
- There are things that the compiler will warn you about; you should heed these warnings and fix the possible mistakes.

Variables

- Variables can be declared in one of two ways:
  - let
  - var
- Variables declared with let cannot have their values changed.
- The type of a variable is established when it is created, either explicitly or implicitly.

```
let two = 2                   // implied to be an Int
var hello : String            // explicitly defined as a String
```

Objects

- Three fundamental object types:
  - class
  - struct
  - enum

- Defined in the way you expect:

```
class Manny {
   let name = ”manny”                      // property of Manny
   func sayName() {                        // method of Manny
       print (name)
   }
}
```

Function Basics

- The syntax for a function is a bit different than what you may have seen before:

```
func sum (_ x:Int, _ y:Int) -> Int {
   let result = x + y
   return result
}

let z = sum (4,5)
```

- The underscore indicates that the parameter names are not externalized in the function call.

Externalized Parameter Names

- Parameter names can be externalized, making the meaning of the function call more obvious:

```
func echoString (_ s:String, times:Int) -> String {
   var result = “”
   for i in 1...times {
       result += s
   }
   return result
}

let s = echoString(“hi”, times:3)
```

- Internal and external parameters may be different:

```
func echoString (string s:String, times n:Int) -> String
```

Other Parameter Details

- There are a number of other details about function parameters that you should be aware of:
  - default values can be assigned in the function declaration
  - parameters can be variadic, allowing many values to be provided in the function (which go into an array)
  - parameters can be modified by defining them using the inout keyword
    - the variable in the function call must be modifiable (var, not let)
    - the address of the variable must be passed (using &)
  - all parameters that are instances of classes are automatically modifiable

Function Overloading

- Function overloading is perfectly legal and should be used when appropriate.
- In addition to defining different functions based on the parameters, they can also be defined based on their return type.
- We will use this heavily in the specification of class initializers.

Anonymous Functions

- When programming with event-driven APIs (like iOS), it is common to provide a function as a parameter value
  - this function may be called at a later date, such as when the API method finished its work
  - these are called handlers or blocks

- A common approach is to specify the function within the function call that uses it
  - because the function doesn't have a name, it is anonymous

Anonymous Function Example

- There are a number of different formats for anonymous functions; this is the simplest:

```
UIView.animate (withDuration: 0.4, animations: {
   self.myButton.frame.origin.y += 20
   }, completion: {
       print (“finished: \($0)”)
   })
```

- If there are any parameters passed into the function, they can be used with the magic variable names $0, $1, $2, …
- This is a common pattern in using the iOS API, so you should learn to decode it and eventually use it.

More on Variables

- Declared with let (contant) or var (variable).
- The variable type is defined during instantiation, and cannot be changed.
- Variable types can be inferred:
  - var x = 1
- Or you can specify them explicitly:
  - var x : Int = 1
  - let separator : CGFloat = 2.0

Let and Objects

- Just because an object instance is declared with let doesn't mean that its internal parameters cannot be changed
  - let specifies that the variable cannot be changed (that it is a constant)
  - when the variable in question is a class, this means that the address to the corresponding object cannot be changed
  - the internal properties can still be manipulated, as the class specification allows
    - any properties that are declared using var can be changed

Computed Variables

- Rather than storing a value, a variable can be defined in terms of functions.
- Why?
  - to avoid duplicate storage of information
  - to provide a read-only variable
  - as a façade for a function
  - as a façade for other variables
- Specified explicitly with get and set options
  - set is optional; get is mandatory
  - if there is no set, the keyword get can be excluded

Computed Variables

```
var now : String {
   get {
      return Date().description
   }
   set {
      print(newValue)      // newValue is the set value
   }
}

var now : String {
   return Date().description       // this is get
}

print (now)
now = “Jan 10, 2017”
print (now)
```

Computed Variables

```
class HiddenValue {
  private var _p : String
  var p : String {
      get {
          return self._p
      }
      set {
          self._p = newValue
      }
  }
}
```

Setter Observers

- It is possible to add *observer* functions to a variable that are executed as the variable is set:
  - willSet happens *before* the variable is assigned a new value
  - didSet happens *after* the variable is assigned a new value

- The variable must be a var (not let).
- newValue is available in willSet.
- oldValue is available in didSet.

Setter Observers

```
var angle : CGFloat = 0 {
   didSet {
       // angle must be within the [0, 5] range
       if self.angle < 0 {
           self.angle = 0
       }
       if self.angle > 5 {
           self.angle = 5
       }
       // update location where this is used
       self.transform = CGAffineTransform (rotationAngle:
             self.angle)
   }
```

Built-In Simple Types

- The usual suspects for built-in simple types are present:
  - Bool
  - Int
  - Double
  - String
  - Character

- Other less common simple types
  - Range
  - Tuple
  - Optional

Range

- The Range object represents a pair of endpoints (start, end) and a range operator:
  - a…b is a closed range (from a to b, including b)
  - a..<b is a half-open range (from a to b, not including b)

- Range endpoints are typically Int, but could be other classes if they have an iterator defined.

- Commonly used in for-in loops.
- The range is in the positive direction; use .reversed() on the range to reverse it.

Range and For-In Loops

```
for i in 1...3 {
   print (i)
}

for i in (1...3).reversed() {
   print (i)
}

for x in 0..<5 {
   print (myArray[x])
}
```

Tuple

- A tuple is a lightweight custom ordered collection of multiple values
  - each value may be a different data type
  - there can be as many values as needed

```
var pair : (Int, String)
var pair : (Int, String) = (1, “Two”)
var pair = (1, “Two”)
```

- Why: this provides a simple solution to returning just one value from a function call, and has much less overhead than creating a class or struct.

Tuples

□ Internal variable names can also be used when defining tuples:

```
var (ix, s) = (1, “Two”)

let pair = (1, “Two”)
let (_, s) = pair           // s = “Two”

var s1 = “hello”
var s2 = “world”
(s1, s2) = (s2, s1)         // tuples perform safe swaps
```

Optional

- An optional is an object that wraps any other object.
- Its purpose is to support the fact that there is no nil data type in Swift.
- This is important, since nil is a common data type in Objective-C (and therefore the iOS API) to signify an uninitialized object.

- Optionals will look weird when you first see them, but will make sense to you once you use them.

Declaring an Optional Variable

- Since Optional is a special class, there is short-hand syntax for using this class

```
var stringMaybe = Optional (“hello”)
var stringMaybe : String?
var stringMaybe : String? = “hello”
```

- When assigning to an Optional, you can essentially ignore this and assign to the variable as normal (Swift will wrap the value in an Optional)

```
stringMaybe = “world”
```

Unwrapping an Optional

- In order to unwrap an Optional to get the embedded object, you have to tell Swift to do so

```
func giveMeAString (_ s:String) {}

let stringMaybe : String? = “hello”
giveMeAString(stringMaybe!)
```

- If you want to send a message to an object wrapped in an Optional, you need to unwrap it first

```
let upper = stringMaybe!.uppercased()
```

Implicitly Unwrapped Optional

- It is possible to define an Optional such that it implicitly unwraps itself as needed
  - this means that the value can be used wherever the embedded object is expected

```
func giveMeAString (_ s:String) {}
let stringMaybe : String! = “hello”
giveMeAString(stringMaybe)
```

  - this gives the functionality we expect for Optional providing a mechanism for handling nil values
    - we can test against nil, and assign nil to the Optional

Main Object Types

- There are three main object types:
  - enum
  - struct
  - class

- The declaration of an object may include the following:
  - initializers
  - properties
  - methods
  - other features that are used in special circumstances

Values vs. References

- Struct and Enum objects are values
- Class objects are references

- This implies something very important in terms of function side-effects. What is the implication?

Initializers

- □ An important part of object specification is to allow it to be initialized fully.
- □ The initializer function is called “init”, and is executed when the instance is instantiated

```
class Dog {
   var name = “”
   var license = 0
   init (name:String = “”, license:Int = 0) {
      self.name = name
      self.license = license
   }
}

let pepper = Dog (name: “Pepper”)
var fido = Dog ()
```

Properties & Methods

- Everything with properties and methods works as expected.
- Remember that properties are variables within the class, which means that everything we have discussed about variables holds within the context of an object.
- Properties and methods defined as static are accessed from the object itself, rather than an object instance

Enum and Named Cases

- The enum object type can be used to created a set of enumerated constants.

- This object type is often used as a mechanism for maintaining state within an application
  - it makes much more sense to name the states and hold them in an enum object than use Int values and have to remember what each state number means
  - within the enum object instance, you can hold extra information about the state

Struct - Lightweight Alternative to Class

- If your object does not need to support inheritance or pass-by-reference, you may find it easier to create a struct rather than a class
  - the specification and use is familiar

```
struct Digit {
   var number = 42
   init (number: Int) {
       self.number = number
   }
}
```

Classes and OO Functionality

- Classes work the way you'd expect
  - they are a reference type
  - they are mutable in-place (even if specified with let)
  - there can be multiple references to the same class instance
  - they support inheritance
  - a subclass can add new functionality to an existing superclass
  - when overriding existing functionality from the superclass, the override keyword must precede the new function specification
  - special care must be taken with class initialization to ensure that the superclass properties are initialized
  - polymorphism works as with other OO languages

Protocols & Protocol Conformance

- Protocols are ways of expressing commonalities between otherwise unrelated types.
- Protocols cut across class hierarchies, specifying what methods an object should define

```
protocol Flier {
    func fly ()
}

protocol FireBreather {
    func Burn()
}

struct Bird : Flier {
    func fly () {
    }
}

class Dragon : Flier, FireBreather {
   func fly () {
   }
}
```

Protocols & Cocoa

- The core iOS API (Cocoa) makes heavy use of protocols.
- The protocols tell another object what functionality will be expected of them.
- If your new class conforms to the protocol, then it can be used by Cocoa for pre-defined purposes (e.g., as a delegate object for a Cocoa object).

- You probably won't declare your own protocols very often, but you will make your classes conform to existing protocols (by creating the necessary functions that the protocol specifies).

Collection Types

- Array - numerically indexed values

- Dictionary – name/value pairs

- Sets – singleton sets that support set operations (join, union, intersection, etc.)

- Specific details on how to use these collections is provided in the textbook and Swift documentation

Usual Flow Control Constructs

- The usual constructs for flow control are present
  - if
  - if-else
  - switch-case
  - conditional evaluation (condition ? exp1 : exp2)
  - loops (for, while, repeat)

Conditional Binding

- Conditional binding is a convenient shorthand for safely unwrapping and passing an Optional into a block.
  - it combines an if statement with variable declarations
  - if any of the Optional unwrapping fails, the condition is false and the block is not executed
  - if there are multiple variable declarations, they are unwrapped left-to-right

```
if let ui = n.userInfo, prog = ui[“progress”] as? Number) {
   self.progress = prog.doubleValue
}
```

Memory Management & Retain Cycles

- Swift memory management is handled automatically (applause).
- You can ignore it, except for a special circumstance that you must be aware of: a retain cycle
  - when two class instances have references to one another, this retain cycle can result in a memory leak

```
class Dog {
   var cat : Cat?
}
class Cat {
   var dog : Dog?
}

let d = Dog()
let c = Cat ()
d.cat = c
c.dog = d
```

Protect Against Retain Cycles

- If your code is designed such that a retain cycle is possible, you have to guard against it manually
  - mark the retain cycle references as “weak”

```
class Dog {
   weak var cat : Cat?
}
class Cat {
   weak var dog : Dog?
}
```

  - now, if instances of these objects go out of scope from their regular variable pointers, the weak pointers will not force the objects to be retained
  - caution: only use weak references when you know you have a retain cycle; otherwise, let the memory manager do its job

Homework

- Make sure you have read Chapters 1 – 5
- Read Chapters 6 - 9
- Start reading and working through the demo in "Start Developing iOS Apps (Swift)"
- Next topic: Xcode and iOS Programming
- Project Proposal
  - due Sep19
- Assignment #1
  - due Oct 5