Seneca College of Applied Arts & Technology

PLAIN HTML VERSION: lecture4.161.txt.html
Xcode Project Archive: lecture4.161.zip

```
/*
file:    lecture4.161.swift
author:  Danny Abesdris
date:    February 3, 2016
purpose: MAP523AA/DPS923AA lecture #4

         Swift programming language topics:
         Classes, access modifiers and scope, instantiation, value vs reference types,
         properties, optional properties, property observers, failable initializers,
         Swift extensions, computed properties, protocols.

         TableView/Navigation GUI example.
*/

import Foundation
var response : String = ""

/*
Swift Classes:

In Swift, a class is a blueprint or template for an instance of that class. The term
object is often used to refer to an instance of a class. In Swift, however, classes and
structures are very similar and therefore easier and less confusing to use the term
instance for both classes and structures.

In Swift, both classes and structures can have properties (member variables) and method
(functions). Unlike C structures however, structures in Swift can be extended and conf
to protocols (a protocol is a list of methods that specify an interface) which will
be discussed below.

By convention, Class names are identifiers that use the camel-case naming scheme and
begin with an initial uppercase letter.
In an Xcode project, if you define a class in a separate .swift file, Xcode allows you
to use it in the projects other source-code files.
Although the access modifiers public, internal and private are different from the acces
modifiers used in other object-oriented programming languages like Java, C# and C++,
typically, if you want a class to be reusable in other apps, it should be declared as
public. However, if a class is used only in the files of the project in which its defi
the internal access modifier may be used, and if a class is used only in the file in
which its defined, it can be declared using the private modifier.

Access modifers:
private - accessible only from within the source file where its defined.
internal  accessible only from any file within the target (app) where its defined.
public  accessible from any file within the target (app) wheres its defined, and
         from within any other context that imports the current targets module.

The default access type in Swift is: internal

For example, the following defines a class named Person containing 3 stored properties
and 1 method:
```

```
*/

class Person {
    var firstName: String?          // properties
    var lastName: String?
    let gender = "female"

    func fullName( ) -> String {    // method
       var parts: String = ""

        if let first = self.firstName {
            parts += first
        }
        if let last = self.lastName {
            parts += last
        }
        return parts
    }
}

/*
Defining stored properties in a class definition is very similar to defining regular
variables and constants in that the var keyword is used to define a variable property a
let is used to define a constant property.
A variable property is read/write in that it allows you to get the propertys value fro
an object of the class and store a value in an object of the class using the dot (.)
operator (in much the same way members are accessed in C structures or clases).

Instantiation:
To create an instance of the Person class, you write:
var john = Person( )

Instantiating an instance of a class is very similar to invoking a function. To create
an instance, the name of the class is followed by a pair of parentheses, and the retur
value is assigned to a constant or variable.
*/

var p1 = Person( )
p1.firstName = "John"     // uses the Class's setter to set the name
p1.lastName  = "Selmys"
print(p1.firstName!)      // uses the Class's getter to get the name (unwrapped becaus
                          // was declared as Optional in the class
                          // the "getter" (get) and "setter" (set) methods are implici

print(p1.fullName( ))     // calling the method fullName( )

/*

Swift also provides computed properties that do not store data, but rather are used to
manipulate other properties. For example, a Circle class could have a stored property
radius and computed properties diameter, circumference and area that would perform
calculations using the stored property radius.

NOTE: In Swift, every stored property needs to have a value after initialization or be
      defined as an optional type. In the example above, the gender property has an
      initial value of "female" and that the property is of type String.
      Also, even though the gender property is defined as a constant, it is possible t
      change its value during the initialization of a Person instance.
      Once the instance has been initialized however, the gender property can no longe
      be modified since it was defined as a constant property with the let keyword.
```

```
Swift does not provide default values for a classs properties and must be initialized
before they can be used. To provide different values for a class's properties an initia
with (or without) parameters can be used to initialize new objects. In fact, Swift requ
an initializer call for every object thats created. For a class that does not explicitl
define any initializers, the compiler defines a default initializer (with no parameters
that initializes the classs properties to the default values specified in their definit
Initializers are like constructors in most other object-oriented programming languages

*/

class Person {
    var firstName: String?          // properties (all optional because they are not as
    var lastName: String?           // initial values)
    let gender: String?

    init(fn: String, ln: String, gender: String) {
       firstName = fn
       lastName = ln
       self.gender = gender         // local name "gender" masks class property "gender
                                    // so the self keyword is used.
    }
    init( ) {
       firstName = ""
       lastName = ""
       self.gender = "male"
    }
    func fullName( ) -> String {    // method
       var parts: String = ""

        if let first = self.firstName {
            parts += first
        }
        if let last = self.lastName {
            parts += last
        }
        return parts
    }
}
var airportCodes : [String:String] = ["yyz":"Toronto", "cdg":"Paris", "jfk":"New York"]
print(airportCodes["yyz"]!)
// adding a key value pair to a Dictionary
airportCodes["lax"] = "San Francisco"
// using the updateValue method to replace a value at a specific key
var oldVal = airportCodes.updateValue("Los Angeles", forKey: "lax")
print(airportCodes["lax"]!)

/*
NOTE: Like C++, when a constructor (initializer) is created, the default version
      is no longer provided and MUST be explicitly coded.

Also, Swifts types are either value types or reference types. In Swift, all class
variables are reference types whereas struct's are value types (see below):
*/

struct Point {      // replace with: class Point
   var xCoord: Int
   var yCoord: Int

   init(x: Int, y: Int) {
      xCoord = x
```

```
        yCoord = y
    }
}

var point1 = Point(x: 0, y: 0)
var point2 = point1

point1.xCoord = 10

print(point1.xCoord)    // Outputs 10
print(point2.xCoord)    // Outputs 0  when Point is declared as a struct and 10 when it



/*
All of a classs members are accessible. Class members declared as private in languages
such as Java, C# or C++ are accessible only within its defining class. In Swift, howeve
even a private class member can be accessed by the classs other members and in any othe
Swift code defined in the same source-code file. To prevent any code outside a class fi
accessing the classs private members (and to get the benefit of what private offers in
languages such as Java, C# and C++) the class must be defined in a file by itself.
Doing so enables the concept of encapsulation (i.e. to hide the classs implementation
details from the other source code in your module and any module into which your module
is imported).


More on Swift Classes:
Consider the following Time class below:
*/

public class Time {
       // an hour value in the range 0-23 (stored property)
    public var hour: Int = 0 {
        willSet {
            print("hour is \(hour); setting it to \(newValue)")
        }
        didSet {
            if hour < 0 || hour > 23 {
                print("hour invalid, resetting to \(oldValue)")
                hour = oldValue
            }
        }
    }
       // a minute value in the range 0-59 (stored property)
    public var minute: Int = 0 {
        willSet {
            print("minute is \(minute); setting it to \(newValue)")
        }
        didSet {
            if minute < 0 || minute > 59 {
                print("minute invalid, resetting to \(oldValue)")
                minute = oldValue
            }
        }
    }

       // a second value in the range 0-59 (stored property)
    public var second: Int = 0 {
        willSet (newSecondValue) {          // assigning a custom default name
            print("second is \(second); setting it to \(newSecondValue)")
        }
```

```
        didSet (originalSecondsValue) {     // assigning a custom default name
            if second < 0 || second > 59 {
                print("second invalid, resetting to \(originalSecondsValue)")
                second = originalSecondsValue
            }
        }
    }

    public init( ) { }                                      // default initializer
    public init(hour: Int, minute: Int=1, second: Int=1) {  // with default values
        self.hour = hour
        self.minute = minute
        self.second = second
    }

        // a failable initializer
    public init?(hour: Int, minute: Int, second: Int) {
        self.hour = hour
        self.minute = minute
        self.second = second
        if hour < 0 || hour > 23 || minute < 0 || minute > 59 || second < 0 || second > 5
            return nil // initialization failed
        }
    }

        // convert to String in universal-time format (HH:MM:SS)
        // read-only computed method
    public var universalDescription: String {
        return String(format: "%02d:%02d:%02d", hour, minute, second)
    }

        // convert to String in standard-time format (H:MM:SS AM or PM)
        // read-only computed method
    public var description: String {
        get {  // explicitly using get { } accessor to make the computed property read-o
            return String(format: "%d:%02d:%02d %@",
                   ((hour == 0 || hour == 12) ? 12 : hour % 12),
                     minute, second, (hour < 12 ? "AM" : "PM"))
        }
    }
}


// Testing class Time

// displays a Time object in 24-hour and 12-hour formats
func displayTime(header: String, time: Time) {
    print(String(format: "%@\nUniversal time: %@\nStandard time: %@\n",
                 header, time.universalDescription, time.description))
}

// create and initialize a Time object
let time = Time( ) // invokes Time default initializer
displayTime("AFTER TIME OBJECT IS CREATED", time: time)

// change time then display new time
print("SETTING A NEW TIME")
time.hour = 13
time.minute = 27
time.second = 6
displayTime("\nAFTER SETTING NEW HOUR, MINUTE, AND SECOND VALUES", time: time)
```

```
// attempt to set time with invalid values
print("ATTEMPTING TO SET INVALID PROPERTY VALUES")
time.hour = 99
time.minute = 99
time.second = 99
displayTime("\nAFTER ATTEMPTING TO SET INVALID VALUES", time: time)

var testObj = Time(hour: -1, minute: 0, second: 0)
displayTime("AFTER TIME OBJECT IS CREATED", time: testObj!)

/*
For each of the Time class's stored properties, a default value of 0 is provided.
Recall, that when all stored properties are defined with default values and the class
does not define an initializer, then the compiler supplies a default initializer
(with no parameters) that sets the stored properties to the default values specified
in their definitions.

Property Observers:
willSet { } and didSet { } are "property observers" that are invoked when a property is
about to be assigned a new value and after the new value has been assigned, respectivel
It is possible to add one or both of these property observers to any variable stored
property, including global or local variables, which are also stored properties.

Defining Property Observers:
To define a property observer, enclose it in braces { and } after the property has been
declared. When the willSet property observer is defined, it receives a constant with th
default name newValue that represents the value thats about to be assigned to the prope
Similarly, a didSet property observer receives a constant with the default name oldValu
that represents the propertys value before the assignment. In both cases, it is possibl
to specify a custom name for the constant by enclosing the name in parentheses between
willSet or didSet keyword and the opening left brace ({) of the property observers body

NOTE: For a stored property with property observers, the property's type must explicitl
      be specified, otherwise, a compilation error occurs.

Failable initializers:
To prevent an object from being created if invalid values are sent to an initializer
(constructor in C++), Swifts provides failable initializ
is invoked, it returns an optional of the classs type if the object is initialized pro
or returns nil otherwise.
Failable initializers are named init? with the ? indicating that the initializers resu
is a Time? (an optional Time).
This type of initializer requires that the object created first be unwrapped in order
to access it's data or call methods.

Swift Extensions:
Swift extensions allows programmers to add features to a customized classes, structures
enumeration types, as well as existing types (like Int, String, Array, etc.).
Class extensions can also be used to:

add methods
add protocol conformance
add computed properties and computed type properties.

General syntax:
extension NameOfTypeToExtend {
    // method, computed property, or initializers
}

Programmers often use extensions to organize their code into related groups of
```

```
functionality to make the code easier to maintain.
*/

extension Double {
    func celsiusToFahrenheit( ) -> Double {
        return self * 9 / 5 + 32
    }
    func fahrenheitToCelsius( ) -> Double {
        return (self - 32) * 5 / 9
    }
}

// usage:
let boilingPointCelsius : Double = 100.0
let boilingPointFarenheit : Double = boilingPointCelsius.celsiusToFahrenheit( )
print(boilingPointFarenheit) // 212.0


import UIKit

extension UIColor {

    class func fromRgbHex(fromHex: Int) -> UIColor {

        let red =   CGFloat((fromHex & 0xFF0000) >> 16) / 0xFF
        let green = CGFloat((fromHex & 0x00FF00) >> 8) / 0xFF
        let blue =  CGFloat(fromHex & 0x0000FF) / 0xFF
        let alpha = CGFloat(1.0)

        return UIColor(red: red, green: green, blue: blue, alpha: alpha)
    }

    class func customGreenColor() -> UIColor {
        let darkGreen = 0x008110
        return UIColor.fromRgbHex(darkGreen)
    }
}

// in ViewController
view.backgroundColor = UIColor.customGreenColor( )

Swift computed properties:
Computed properties do not provide their own storage. Rather, they perform tasks that
compute values, possibly using the types other members.
By setting a computed property, an object's method is recalculated accordingly and
permits the altering of other member variables (properties).

class Rectangle {
   var width = 100.0
   var height = 100.0

   // computing setter and getter
   var area: Double {
      get {
         return width * height
      }
      set(newArea) {
         let squareRootValue = sqrt(newArea)
         width = squareRootValue
         height = squareRootValue
      }
```

```
    }
}

var myRect = Rectangle( )
myRect.area = 16.0
print("width: \(myRect.width) height: \(myRect.height)")

/*
Swift Protocols:
In iOS development, a protocol is a set of methods and properties that encapsulates
a unit of functionality. The protocol doesnt actually contain any of the implementatio
it merely defines the required elements. Any class that declares
itself to conform to a protocol must implement the methods and properties declared in
the protocol.
*/

// Basic syntax:
protocol SampleProtocol {
    func someMethod( )
}

class MyClass: SampleProtocol {
    // Conforming to SampleProtocol
    func someMethod( ) {
    }
}

class AnotherClass: SomeSuperClass, SampleProtocol {
    // A subclass conforming to SampleProtocol
    func someMethod( ) {
       // must be coded here...
    }
}

protocol Animal {
   var lives:Int { get set }
   var limbs:Int { get }
   func makeNoise() -> String
}

class Cat: Animal {
   var lives = 9
   var limbs = 4
   func makeNoise() -> String {
      return "meow"
   }
}

let cat = Cat( )          // lives 9 limbs 4
print(cat.makeNoise( ))   // "meow"
print(cat.lives)          // 9
cat.lives = 8             // lives 8 limbs 4
print(cat.lives)          // 8
```