Swift Objects (4.2)

Coming from a Java Perspective

Cody Henrichsen

2019

Class Structure

- The syntax of using a class in Swift is very similar to Java. Data members and methods are almost the same. Creation of objects is a bit different as there is no use of the new keyword when initializing an object
- The first and most obvious difference is that the constructor is not named the same as the class. Instead all classes and structs use init to create an instance
- Squiggles work the same – YAY
- Just like Java the order of components doesn't matter but patterns help
  - data members/properties
  - inits
  - methods

Visibility AKA Access Control

- The visibility of Swift types is a bit different than Java. By default your code can be seen across the project, so there is no need to import different Groups like you would with Java packages
  - The implicit (none) modifier is `internal`
    - Allows access across your project
  - Public & Private
    - These work just like Java
    - Great for explicitly defining visibility
  - Data members can be compound
    - Public get and private set

Init

- The init is how you instantiate an object in Swift. They are not methods and do NOT have a return type. Unlike Java, they do not have the same name as the class.
- There is a default init but…
- Inits are called like a constructor in Java but without the new key word
  - var someExample = FamousComputerScientist()
- ALL data members must be initialized or declared as lazy before the end of the init for compilation
  - There is no excuse for a NullPointerException
- If same signature as parent, must be marked as override
- **Must initialize sub class data before calling super.init()**

Init examples

```
init ()
{
    self.hasFlowers = true
    self.interests = "Running"
}

init (flowering : Bool)
{
    self.hasFlowers = flowering
    self.interests = "Computer Scientists Rock"
}
```

```
override init()
{
    topic = "CS"
    self.familiarity = 1
    super.init()
}

init(withTopic: String)
{
    topic = withTopic
    self.familiarity = 1
    super.init()
}
```

Data members / Properties

- Swift designers were sick of programmer caused crashes
- So ALL data must be either explicitly initialized, provided with a lazy initializer, or be Optional(?!)
  - Lazy modifier provides either a constructor or closure to initialize the variable
- All data members must have a Type assigned
  - Explicit
    - var graceHoppper : FamousComputerScientist
  - Implicit
    - Let phrase = “Katherine Johnson should be as well known as Alan Turing”
- Think about the required visibility

Properties

- Unlike Java, Swift provides direct access to all non-private data members with a property AKA dot level access instead of a method. This means you can assign into a data member with the assignment operator directly
- Stored
  - AKA public getter/setter but not separate from the data
- Computed
  - Indirect access/modification of internal data
  - Especially useful on complex components
- Data members can be made externally unchangeable but still accessible
  - Public get / Private set
- Allows for observers to be linked to the data

Property Example

```swift
public class Person
{
    private var hasFlowers : Bool
    var interests : String

    var phrase : String
    {
        get
        {
            return "\(interests) is so cool"
        }
        set
        {
            self.interests = "\(interests) \n \(newValue)"
        }
    }

    init ()
    {
        self.hasFlowers = true
        self.interests = "Running"
    }

    init (flowering : Bool)
    {
        self.hasFlowers = flowering
        self.interests = "Computer Scientists Rock"
    }
}
```

Private only data member

Stored property of a String

Computed property of a String

Implicit let for the set on a property

```swift
let me = Person()
print(me.interests)
me.interests = "Swift Programming"
me.phrase = "Allons-y"
print(me.phrase)


let ladyAdaLovelace = FamousComputerScientist(withTopic: "Algorithms")
print(ladyAdaLovelace.topic)
ladyAdaLovelace.interests = "Creating music with the Analytic Engine!"
print(ladyAdaLovelace.interests)
print(ladyAdaLovelace.isHowCoolAndWhy().reason)
```

Methods

- All methods in Swift have the keyword func at the front of their header, then the name, followed by the parameter(s), and finally the -> and its return type
- Swift continues the long method name approach that Objective-C used where the method name continues into the external variable names of the methods
- Methods can also return Tuples!!!
  - Multiple returns!!!
    - return (name: valuesSeparated, otherName: byCommas, named: inParens)
  - Signature options
    - Best: (name : Type, otherName : Type) // Much easier to read 😻
    - Valid: (type, type, type) //😺 You can do better

Method Example

```
private func demo() -> Double
{
    let numerator = arc4random()
    let denominator = arc4random() + 1
    return Double(numerator/denominator)
}
```

```
public override func viewDidLoad() -> Void
{
    super.viewDidLoad()
    showStructExamples()
    showExtensionExamples()
}
```

Tuple Example

```
public func isHowCoolAndWhy() -> (cool: Int, reason: String)
{
    let coolness = self.interests.count * 130
    let why = "because of their work in: \(topic) and #CompSci in general"

    return (cool: coolness, reason: why)
}
```

```
let ladyAdaLovelace = FamousComputerScientist(withTopic: "Algorithms")
print(ladyAdaLovelace.topic)
print(ladyAdaLovelace.interests)
print(ladyAdaLovelace.isHowCoolAndWhy().reason)
```

Static methods

- Swift also calls these Type methods
- These are methods that are called in a non-OOP fashion
- Utility methods!
- Add the keyword static before the func keyword
  - Obviously these are public (weird huh)

Inheritance

- No automatic inheritance hierarchy in Swift
  - There is no ur base class a la Object
- Swift uses the C++ and C# inheritance style via the : in the class header instead of using a keyword
- Subclasses must explicitly override methods or observers
- FYI: calls to the parent class' init are done at the end of the init!

Inheritance Example

```
public class Person
{
    private var hasFlowers : Bool
    var interests : String

    var phrase : String
    {
        get
        {
            return "\(interests) is so cool"
        }
        set
        {
            self.interests = "\(interests) \n \(newValue)"
        }
    }

    init ()
    {
        self.hasFlowers = true
        self.interests = "Running"
    }

    init (flowering : Bool)
    {
        self.hasFlowers = flowering
        self.interests = "Computer Scientists Rock"
    }
}
```

```
public class FamousComputerScientist : Person
{
    var topic : String

    override init()
    {
        topic = "CS"
        super.init()
    }

    init(withTopic: String)
    {
        topic = withTopic
        super.init()
    }
}
```

Overrides must be made explicit!

Subclass on the right of the colon

The call to the super init is AFTER your data is initialized

Protocol

- The protocol Type is even more powerful in Swift than the corresponding Java interface. A protocol allows for a more complex object relationship.
- As with Java a protocol consists of method headers that must be implemented by a class or struct that adopts the protocol.
- A protocol can also specify an init for a class or struct
- Objective-C provides for optional methods in a protocol that are often used in the UIKit framework
- If a protocol is designed to be used by a struct, any methods that change internal state must be marked as mutating

Protocol cont

- Delegate
  - Protocols designed to link to another class especially to listen for changes or handles specific responsibilities
  - Weird that it matches the English definition
- Datasource
  - Protocols that are designed to link to data :D

Structs

- In Swift a **`struct`** is a user defined Type that is similar to the C# datatype of the same name. Structs allow for conformation to protocols and extensions and therefore make sense for lightweight objects.
- Swift suggests the use of a **`struct`** for your default Types, especially if you are connecting the instance to data sources beyond your control.
  - All the defaults Types in Swift are Structs
    - `Int`, `Double`, `String`, …
  - Handle inheritance needs via protocols
- Structs are VALUE types not reference
  - Copy when passed as param or returned from method
- Limitations
  - No inheritance
  - Modifying internal state requires `mutating` keyword on the associated `func`

Structs Cont.

- A default init is provided by Swift that can be used to initialize each data member in order of declaration
- Custom inits are allowed provided all data members are initialized
  - As with Java, if you create a custom init you lose the default init
- Private data is allowed but must be defined in the init!
- Methods in a struct look and work just like those in a class

Struct examples

```
import Foundation

struct CodeArt
{
    let width : Int
    let height : Int
    var content : String
    private var badDesign : String

    init()
    {
        self.width = 8
        self.height = 10
        self.content = "default art"
        self.badDesign = "asda"
    }

    init(height:Int,content:String,width:Int)
    {
        self.badDesign = "asda"
        self.height = height
        self.content = content
        self.width = width
    }
}
```

```
import Foundation
struct ZombieHead
{
    var degradationPercent : Double
    var isFleshy : Bool


    private func demo() -> Double
    {
        let numerator = arc4random()
        let denominator = arc4random() + 1
        return Double(numerator/denominator)
    }

    public mutating func update() -> Void
    {
        self.degradationPercent = demo()
    }
}
```

Struct use example

```
private func showStructExamples() -> Void
{
    var firstZombie = ZombieHead(degradationPercent: 0.2, isFleshy: true)
    var myArt = CodeArt()
    let coolArt = CodeArt(height: 123, content: "amazing work", width: 1232134)
    firstZombie.update()
    myArt.content = "sakldjlkaj"
    print(myArt.content)
    print(coolArt.height)
}
```

Struct errors

```
private func showExamples() -> Void
{
    var firstZombie = ZombieHead(isFleshy: true, degradationPercent: 0.2)
    var myArt = CodeArt()
    let coolArt = CodeArt(widt
```

Argument 'degradationPercent' must precede argument 'isFleshy'

```
var myArt = CodeArt()
let coolArt = CodeArt(width: 234, height: 123, content: "amazing work")
```

Argument passed to call that takes no arguments

Extensions

- Swift recognizes that functionality of existing Types may not meet all needs of current and future developers. As such, the extension is provided to add functionality and/or computed (not stored) data to existing Types.
- Extensions can be declared for ANY Type in Swift
- They can be added to any file in your project
- I prefer keeping extensions together in the Resources group using the boring name Extensions.swift
- Remember to maintain MVC separations when designing an extension

Extension custom examples

```
extension ZombieHead
{
    func spooky() -> Void
    {
        print("boooooooo")
        print("boooooooo")
        print("boooooooo")
        print("boooooooo")
        print("boooooooo")
        print("boooooooo")
        print("boooooooo")
    }
}
```

```
extension CodeArt
{
    var area : Int
    {
        return width * height
    }

    init(area : Double)
    {
        let square :Int
        square = Int (Double.squareRoot(area)())
        self.init(height: square, content: "squares", width: square)
    }
}
```

Extension to built in type

```
extension String
{
    func utterlyUselessMethod() -> Void
    {
        //Comments only!!!
    }

    func overloaded() -> Int
    {
        return self.count
    }
}
```

Extension use

```
private func showExtensionExamples() -> Void
{
    let words : String = "sample for extensions"
    print(words.overloaded())
    words.utterlyUselessMethod()

    var structSample : CodeArt = CodeArt(area: 3.1415)
    structSample.content = "sadafas"
    print(structSample.area)

    let zombie = ZombieHead(degradationPercent: 0.8345, isFleshy: false)
    zombie.spooky();
}
```