387: Swift Continued

Dan Goldsmith

May 2018

Introduction

Introduction

- Today we are going to continue looking at the swift language.
  - Recap (What we did yesterday)
  - Advanced Concepts
    - Classes and Inheritance
    - Error Handling
  - Practical Programming Task

Recap: Variables and Constants

- "Automatic" Variable types
- Type Safety
- Conversions

Variable Types:

- Core Variable types:
  - **String** Holds Text
  - **Int** Whole numbers
  - **Double** / **Float** Decimal Numbers
  - **Boolean** True / False

Defining Types:

```
//Automatic variable type of Integer
var radius = 5
//And of Double
var text = "Hello, World!"

//Manual Definition
var radius: Double
//And set value
radius = 5.0
```

Constants

- Used to hold items that value shouldn't change (like Pi)
- Once defined will case an error if you try to change them
- We use the **let** keyword.

```
let pi=3.1415
```

Variables

- Hold data values
- Can be changed at run time (remember type safety)
- use the **var** keyword

```
//Define the variable
var radius = 5.0

//And modify it
radius = 10.0
```

Recap: Collections of Objects

- Two main ways of holding Groups of objects
  - Lists (Groups of objects)
  - Dictionaries (Key, Value) pairs

Lists

- Enclose items in square brackets **[]**
- Remember indexing starts at 0
- Access using square bracket syntax **[<index>]**

Lists:

```
//Define a list of grades
var grades = [60, 55, 70]

//Print the 1st item in the list
print("Grade is \(grades[0])")

//Update the 2nd Item
grades[1] = 60
```

Lists: adding and removing objects

```
//Define a list
var grades = [60, 55, 70]

//How many items (will print 3)
print("Number of Grades \(grades.count)")

//Add an Item to the end of the list
grades.append(75)

//Insert an item at the start of the list
grades.insert(45, at: 0)

//Remove the first item from the list
grades.remove(at: 0)
```

Dictionaries:

- Hold "Key", "Value" pairs of data
- Useful for lookup tables etc.
- Similar syntax to lists **[<key>:<item>]**

```
//Define a dictionary of Airport Codes

var airports = ["DXB": "Dubai",
        "HKG": "Hong Kong"]
```

Dictionaries: Getting Data

```
//Define
var airports = ["DXB": "Dubai",
        "HKG": "Hong Kong"]

//Lookup HKG airport Code
print("Code HKG is \(airports["HKG"]))

//Add a new airport to the dict
airports["BHX"] = "Birmingham"
```

Recap: Comparisons

- == Equal To
- != Not Equal To
- > Greater Than
- < Less Than
- >= Greater or Equal to
- <= Less or Equal to

Recap: Logical Operators

- Allow us to **chain** comparisons together.
- **a && b** AND operator
- **a || b** OR operator
- **!a** NOT operator (inverts value)

NEW: Ranges

- Allow us to define Ranges of values we wish to work with
- Closed Ranges **a..b** runs from a to b (and includes both values)

```
//Values between x <= 0 && x >=10
//Ie 0,1,2,3,4,5,6,7,8,9,10
for x in 0..10
    ....
//What about 10 to 100
for x in 10..100
    ....
```

Half open ranges

- We can also use a **less than** for the second value
  - Think of this for lists based on the size

```
//Values between x <=0 and x <10
//Produces 0,1,2,3,4,5,6,7,8,9
for x in 0..<10
   ...

//Or for a list
for x in 0..<thelist.count
  ...
```

Recap: Selection

- Making choice based on a variables value
  - **If, then, else** Statements
  - **Switch** Statements

If Statements

- Remember Ordering of tests is important

```
if grade < 40 {
    // Fail
} else if grade >= 70{
    // First Calss
} else if grade >= 60 {
    // 2:1
} else {
    // A default condition
}
```

Switch Statements

- Alternative way of nesting large numbers of Ifs

```
switch x{
    case 0..<40:
        // Fail
    case 70..100:
        //First Class
    case 60..<70:
        //Secnd Class
    default:
        //Default condition
```

Recap Iteration

- Doing Things Many times
  - **For** loops
  - **While** Loops

For Loops

- Good when we know how many items we are dealing with
  - Iterate through Lists, Dictionaries, set ranges of values etc.
- **For-In** loops iterate through lists

```
//Define a list
thelist = ["foo","bar","baz"]

//Iterate
for item in thelist{
    print ("Item is \(item)")
}
```

For Loops, Ranges

```
for index in 0..10 {
    print ("Value is \(index)")
}

//Use this to go through a list
thelist = ["foo","bar","baz"]

//Note the ..<
for index in 0..<thelist.count {
    print ("Item at index \(index) is \(thelist[index])")
}
```

While Loops

- Keep going until we are told to stop
  - **REMEMBER** check your stop condition

```
thelist = ["foo","bar","baz"]
var index = 0

while index < thelist.count {
    print ("Item at index \(index) is \(thelist[index])")
    //IMPORTANT Increase index
    index += 1
}
```

Recap: Functions

- Modular Code is good.
  - Reuse functions
  - Reduce errors
  - Make our life easier (good Programmers are Lazy)

Defining Functions

- Use the **func** keyword
  - Parameters are **name:Type** pairs
  - Return types are also given

```
//   (Name)        (Parameters)          (Returns)
func demoFunction(parameter: String) -> String {
    .. Do Something
}
```

Function Demo

```
func grade(mark: Double) -> String {
    if mark < 40 {
        return "Sorry, you failed"
    } else if mark > 70 {
        return "Congratulations you got a 1st")
    } else if mark >= 60 {
       return "Not bad, a 2:1"
    } else if mark >= 50 {
        return "OK, a 2:2"
    }else if mark >= 40{
        return "That sucks, a 3rd"
    } else { //Catch things outside of expected range
        return "Mark outside of boundries"
}
```

Calling Functions

- funtionName(<parameters>)

```
//Call the grade function
var output = grade(mark: 70)
// Print the Result (Contratulations ...)
print (output)
```

Recap: Classes and Objects

- Create **Objects** representing items with similar traits
  - People
  - Shapes
  - Records

Creating Classes

- Use the **class** keyword
  - Define Class Variables / Attributes

```
class Circle {
    var radius : Double
}
```

Constructors

- Help us create new instances of objects
- use the special **init** method

```
class Circle {
    var radius : Double

    init(theradius: Double) {
        //Update classes radius value with the one provided
        radius = theradius
    }
}

//Create a new Circule with radius 5
var theCircle = Circle(radius: 5)
```

Accessing / Modifing Attribures

- Use dotted syntax .

```
//Create a new Circule with radius 5
var theCircle = Circle(radius: 5)

//Print the Radius
print("Radius is \(theCircle.radius)")

//Change the Radius
theCircle.radius = 10
```

Class Methods

- Give a class functionality

```
class Circle {
    var radius : Double

    init(theradius: Double) {
        //Update classes radius value with the one provided
        radius = theradius
    }

    func getArea() -> Double {
        //Calculate the area of the circle
        //(NOTE: Bad Programming, Magic Number)
        return 3.14 * (radius * radius)
    }
}
```

Using Methods

```
//Create a Circle
var theCircle = Circle(5)

//Call the Area Method
var theArea = theCircle.getArea()

//Print something
print ("Circle of Radius \(theCircle.radius) has area \(theArea)")
```

Recap: Warmup Task

- Create a new Playground
- Complete the Code provided.
  - Add a Get Circumference Function
  - Think about using Constants to remove the Magic Numbers
- Warmup.swift:

https://gist.github.com/djgoldsmith/5bfa8d31e002e8c2c51778a8c1ca0c99

Warmup Task: Code

```
class Circle {
    var radius : Double

    init(theradius: Double) {
        //Update classes radius value with the one provided
        radius = theradius
    }

    func getArea() -> Double {
        //Calculate the area of the circle
        return 3.14 * (radius * radius)
    }

   //Add a Function to Caluclate the Circumference (Pi * R * R)
   func getCircumference()
   {
      //Code Goes Here
   }
}
```

Warmup Task: Code (2)

```
//List of Circles
var theList = [2.0,4.0,6.0]

for item in theList {
    //New Circle
    var theCircle = Circle(theradius: item)

    //Print
    print ("Circle of Radius: \(item)")
    print ("\tArea: \(theCircle.getArea())")
    print ("\tCircum: \(theCircle.getCircumference())")
}
```

Advanced Class Methods

Advanced Class Methods

- Swift allows us to do some clever things with class methods
  - Use the same method to **get** or **set** values
  - Imagine a Square Class:
    - We could call **Square.Area()** to get the area of the square
    - We could call Square.Area = 10 to set the size of the square

Advanced Class Methods: Initial Code

```
class Square {
    //How long is each side
    var sidelength: Double

    init(length: Double){
        //Constructor:  Set the side length to specified v
        sidelength = length
    }
```

Advanced Class Methods: Getters and Setters

```
    func getArea() -> Double{
        //Calculate the area of the square
    return sidelength * sidelength
    }

    func setArea(area: Double){
        //Set the sidelength based on Area
    sidelength = area.squareRoot()
    }
}
```

Advanced Class Methods: Testing

```
var item = Square(length: 5)
//Get the Area
print ("Original Sides \(item.sidelength)")
print ("Original Area \(item.getArea())")
//Update the Area
item.setArea(area: 100)
print ("New Sides \(item.sidelength)")
```

Advanced Class Methods: Variables as functions

- Or we could to this:

```
//Define an Variable as a function
var area: Double {
      //Get this value
      get {
       return sidelength * sidelength
       }
 //Set the Value
 set {
      sidelength = newValue.squareRoot()
      }
 }

}
```

Advanced Class Methods: Variables as functions

```
var item = Square(length: 5)
//Get the Area
print ("Original Sides \(item.sidelength)")
print ("Original Area \(item.area)") //Note now a Variable
//Update the Area
item.area = 100
print ("New Sides \(item.sidelength)")
```

Inheritance

Inheritance

- Sometimes we can have lots of classes that share similar attributes
  - **Circles**, **Squares** and **Rectanges** are all types of **Shape**
  - **Students**, **Lecturers** are all types of **People**
- They share common attributes, or methods, but have individual differences

Super-class

- Defines the core functionality for groups of classes.
  - Common Variables
  - Common Functions

Shapes Superclass

- What do we need for our Shape superclass?
  - Print Details of the Shape
  - Calculate the Area
  - Calculate the Perimeter

Superclass:

```
class Shape {
    //Shapes will have a number of sides
    var numberOfSides = 0
    var name: String

    //Constructor
    init(name: String){
        self.name = name
    }

    //Place Holder for Area Functions
    func calcArea() -> Double {}

    //Place Holder for Perimeter function
    func calcPerimeter() -> Double {}

    //A simple Function to return the numberr of sides
    func description() -> String {
         return ("\(name): a Shape with \(numberOfSides) sides")
     }
}
```

Creating a Rectangle Class

- Lets subclass **shape** to create a rectangle
- Think about the rectangles attributes
  - Height, Width
- And Calculations
  - Area = H*W
  - Perimeter = 2*(H+W)

Setting up the Subclass

- We just add the name of the superclass to the definition

```
 class Shape {
    // <snip>
}

class Rectangle: Shape {
}

testRec = Rectangle(name:Rectange)
print ("Test the Rectangle \(testRec.description())")
```

Updating the class Variables

- We next give our **Rectangle** subclass relevant attributes
  - Note that the numberOfSides attribute is inherited from **shape**

```
class Rectange: Shape {
  //Variables specific to rectangles
  var width = 0.0
  var height = 0.0
}
```

Updating the Constructor

- And update the constructor to populate these variables
  - Here, we also update the superclass Variables (as we know what they are)
  - We need to initialise the superclass using an **super.init** call

Updated Constructor

```
class Rectange: Shape {
  //Variables specific to rectangles
  var width = 0.0
  var height = 0.0

  init(width: Double, height: Double){
      //First update Superclass using its init method
      super.init(name: "Rectangle")
      numberOfSides = 4 //And update the number of sides
      //Then Class Specific
      self.width = width
      self.height = height
      }
}
```

Testing the Updated Constructor

- Testing the code

```
//Create a new Rectange
var testRec=Rectange(width: 5, height: 10)
//Call the description function
print(testRec.description())
```

-Gives the expected output

```
$ swift testing.swift
Rectangle: a Shape with 4 sides
```

Completing the Class Methods

- Lets fill in the code for the class methods
  - Note we need to tell swift we are **overriding** superclass functions

```
class Rectange: Shape
   //.. <snip>
   override func calcArea() -> Double {
      return height*width
   }

   override func calcPerimeter() -> Double {
      return 2*(height+width)
   }
}
```

Testing it

-Again we Test

```
var testRec=Rectange(width: 5, height: 10)
print(testRec.description())
print("Area \(testRec.calcArea()) Perimeter \(testRec.calcPerimeter())"
```

- Looks like it works

```
$ swift testing.swift
Rectangle: a Shape with 4 sides
Area 50.0 Perimeter 30.0
```

Your Turn:

- Using the provided code (shapes.swift) create a Triangle Subclass
  - Think about the parameters
  - Area Calculation (W*H)/2
  - Perimeter Calculation (a bit harder)
-

<ghttps://gist.github.com/djgoldsmith/907424267ae067fe58321b8512

Thinking about Subclasses.

- Lets write a Square subclass
- Remember **Programmers are Lazy**
  - What do we know about Squares and Rectangles?
  - A Square is a Rectangle where the Width and Height are the Same

Subclassing a Subclass

- Lets Subclass Rectangle, and get it to take a Length attribute
  - Then set Height and Width to be this value
  - All the rest of the work is done for us

Square Sub-Subclass

```
class Square: Rectangle {
      init(length: Double){
            //Initialise the super class
       super.init(width: length, height: length)
       //Update the name
       name = "Square"
       }
}
```

Testing the Square

```
var testSq = Square(length: 5)
print(testSq.description())
print("Area \(testSq.calcArea()) Perimeter \(testSq.calcPerimeter())")
```

```
$ swift testing.swift
Square: a Shape with 4 sides
Area 25.0 Perimeter 20.0
```

Enumerations and Structs

Introduction

- I think of Enumerations and structs as "mini-classes"
  - Group several objects together
  - Can have functions associated with them

Enumerations

- Give a set of possible values that can be associated with the enumeration
- For example, if we know what sort of errors our code can return, we can group them together for convinience
- Lets say our code could return a **indexOutOfRange** or **duplicateItem** error

Enumerations Example

```
enum possibleError: Error {
    case indexOutOfRange
    case duplicateItem
}
```

Structs:

- Group variables together

```
struct Item {
    var title:String
    var description:String
    var added:NSDate = NSDate()
    var done:Bool = false
}
```

Error Handling

Why do Error Handling

- When dealing with user input mistakes happen.
  - We could try to get an item that doesnt exist
  - We could try to use input that doens't make sense
- If we do not deal with these errors, either the code crashes, or produces unexpected output

Defining Errors.

- We represent errors by using any type that supports the **error** protocol
- Can be defined in a struct to group things together
- By defining errors, we can start to classify things

```
enum possibleError: Error {
    case notEnoughMoney
    case duplicateItem
}
```

Throwing Errors

If we want the code to throw a specific error we can use the **throw** keyword

```
var money = 10 //How much money do we have
var cost = 50  //How much does something cost

if cost > money {
    throw possibleError.notEnoughMoney
} else {
  //.. Do Stuff
}
```

Throwing errors in functions

- We need to tell swift that we can throw an error in the function
  - use the **throws** keyword

```
func doSomething(param:String) throws -> String
```

Handling Errors

- When an error is thrown, the code will crash unless we deal with it
- A common way of dealing with code that could through an error is the **do-catch** block
  - We prefix the code that may fail with a **try** statement

Handling Errors: Syntax

```
//Start block of code
do {
   //Block that could fail
   let result = try afunction(input)
   print (result) //Gets run if things are OK
}
   catch {
      print (error) /Gets run if things break
}
```

Handling Errors: Dealing with specific errors

- As the function throws an error type, we can detect this and respond
- Also means we can deal with multiple errors in one block of code
    - Suffix the **catch** statement with the error type

Multiple Errors: Syntax

```
//Start block of code
do {
   //Block that could fail
   let result = try afunction(input)
   print (result) //Gets run if things are OK
}
   catch possibleError.notEnoughMoney{
      print (error) /Gets run if things break
}  catch { //Default
      print ("Unhandled Error \(error)")
```

Tasks

Time for some Work

- Keep playing with Subclasses / Inheritance
  - Can you make code for regular polygons?
  - What other shapes can you subclass
- Add Error handling.
  - Deal with negative side lengths
- There is Playground worksheet / code you can work through.