More Swift

Dan Goldsmith

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:

```
1  //Automatic variable type of Integer
2  var radius = 5
3  //And of Double
4  var text = "Hello, World!"
5
6  //Manual Definition
7  var radius: Double
8  //And set value
9  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.

```
1   let pi=3.1415
```

Variables

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

```
1  //Define the variable
2  var radius = 5.0
3
4  //And modify it
5  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:

```
1  //Define a list of grades
2  var grades = [60, 55, 70]
3
4  //Print the 1st item in the list
5  print("Grade is \(grades[0])")
6
7  //Update the 2nd Item
8  grades[1] = 60
```

Lists: adding and removing objects

```
1  //Define a list
2  var grades = [60, 55, 70]
3
4  //How many items (will print 3)
5  print("Number of Grades \(grades.count)")
6
7  //Add an Item to the end of the list
8  grades.append(75)
9
10 //Insert an item at the start of the list
11 grades.insert(45, at: 0)
12
13 //Remove the first item from the list
14 grades.remove(at: 0)
```

Dictionaries:

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

```
1  //Define a dictionary of Airport Codes
2
3  var airports = ["DXB": "Dubai",
4  "HKG": "Hong Kong"]
```

Dictionaries: Getting Data

```
1  //Define
2  var airports = ["DXB": "Dubai",
3  "HKG": "Hong Kong"]
4
5  //Lookup HKG airport Code
6  print("Code HKG is \(airports["HKG"]))
7
8  //Add a new airport to the dict
9  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

- Somehow I missed this yesterday
- 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)

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

Half open ranges

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

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

Recap: Selection

- Making choice based on a variables value
  - If Statements
  - Switch Statements

If Statements

- Remember Ordering of tests is important

```
1  if grade < 40 {
2      // Fail
3  } else if grade >= 70{
4      // First Calss
5  } else if grade >= 60 {
6      // 2:1
7  } else {
8      // A default condition
9  }
```

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

```
1  //Define a list
2  thelist = ["foo","bar","baz"]
3
4  //Iterate
5  for item in thelist{
6      print ("Item is \(item)")
7  }
```

For Loops, Ranges

```
1  for index in 0..10 {
2      print ("Value is \(index)")
3  }
4
5  //Use this to go through a list
6  thelist = ["foo","bar","baz"]
7
8  //Note the ..<
9  for index in 0..<thelist.count {
10     print ("Item at index \(index) is \(thelist[index])")
11 }
```

While Loops

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

```
1  thelist = ["foo","bar","baz"]
2  var index = 0
3
4  while index < thelist.count {
5      print ("Item at index \(index) is \(thelist[index])")
6      //IMPORTANT Increase index
7      index += 1
8  }
```

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

```
1  //   (Name)        (Parameters)          (Returns)
2  func demoFunction(parameter: String) -> String {
3      .. Do Something
4  }
```

Function Demo

```
1  func grade(mark: Double) -> String {
2      if mark < 40 {
3          return "Sorry, you failed"
4      } else if mark > 70 {
5          return "Congratulations you got a 1st")
6      } else if mark >= 60 {
7         return "Not bad, a 2:1"
8      } else if mark >= 50 {
9          return "OK, a 2:2"
10     }else if mark >= 40{
11         return "That sucks, a 3rd"
12     } else { //Catch things outside of expected range
13         return "Mark outside of boundries"
14 }
```

Calling Functions

- funtionName(<parameters>)

```
1  //Call the grade function
2  var text = grade(mark: 70)
3  // Print the Result (Contratulations ...)
4  print (text)
```

Recap: Classes and Objects

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

Creating Classes

- Use the class keyword
  - Define Class Variables / Attributes

```
1  class Circle {
2      var radius : Double
3  }
```

Constructors

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

```
1  class Circle {
2      var radius : Double
3
4      init(theradius: Double) {
5          //Update classes radius value with the one provided
6          radius = theradius
7      }
8  }
9
10 //Create a new Circule with radius 5
11 var theCircle = Circle(radius: 5)
```

Accessing / Modifing Attribures

- Use dotted syntax .

```
1  //Create a new Circule with radius 5
2  var theCircle = Circle(radius: 5)
3
4  //Print the Radius
5  print("Radius is \(theCircle.radius)")
6
7  //Change the Radius
8  theCircle.radius = 10
```

Class Methods

- Give a class functionality

```
1  class Circle {
2      var radius : Double
3
4      init(theradius: Double) {
5          //Update classes radius value with the one provided
6          radius = theradius
7      }
8
9      func getArea() -> Double {
10         //Calculate the area of the circle (NOTE: Bad Programming, Magic Numbe
11         return 3.14 * (radius * radius)
12     }
13 }
```

Using Methods

```
1  //Create a Circle
2  var theCircle = Circle(5)
3
4  //Call the Area Method
5  var theArea = theCircle.getArea()
6
7  //Print something
8  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 in Folder

Warmup Task: Code

```
1  class Circle {
2       var radius : Double
3
4       init(theradius: Double) {
5           //Update classes radius value with the one provided
6           radius = theradius
7       }
8
9       func getArea() -> Double {
10          //Calculate the area of the circle (NOTE: Bad Programming, Magic Numbe
11          return 3.14 * (radius * radius)
12      }
13
14     //Add a Function to Caluclate the Circumference (Pi * R * R)
15     func getCircumference()
16     {
17        //Code Goes Here
18     }
19  }
```

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

```
1  class Square {
2      //How long is each side
3      var sidelength: Double
4
5      init(length: Double){
6          //Constructor:  Set the side length to specified value
7          sidelength = length
8  }
9
10     func getArea() -> Double{
11         //Calculate the area of the square
12 return sidelength * sidelength
13 }
14
15     func setArea(area: Double){
16         //Set the sidelength based on Area
17 sidelength = area.squareRoot()
18 }
19 }
20
21
```

Advanced Class Methods

- Or we could to this

```
1  class Square {
2      //How long is each side
3      var sidelength: Double
4
5      init(length: Double){
6          //Constructor:  Set the side length to specified value
7          sidelength = length
8  }
9
10     //Define an Variable as a function
11     var area: Double {
12          //Get this value
13       get {
14       return sidelength * sidelength
15       }
16   //Set the Value
17   set {
18        sidelength = newValue.squareRoot()
19       }
```

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:

```
1  class Shape {
2      //Shapes will have a number of sides
3      var numberOfSides = 0
4      var name: String
5
6      //Constructor
7      init(name: String){
8          self.name = name
9      }
10
11     //Place Holder for Area Functions
12     func calcArea() -> Double {}
13
14     //Place Holder for Perimeter function
15     func calcPerimeter() -> Double {}
16
17     //A simple Function to return the numberr of sides
18     func description() -> String {
19       return ("\(name): a Shape with \(numberOfSides) sides")
20   }
21 }
```

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

```
1  class Shape {
2      // <snip>
3  }
4
5  class Rectangle: Shape {
6  }
7
8  testRec = Rectangle(name:Rectange)
9  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

```
1  class Rectange: Shape {
2    //Variables specific to rectangles
3    var width = 0.0
4    var height = 0.0
5  }
```

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

```
1  class Rectange: Shape {
2    //Variables specific to rectangles
3    var width = 0.0
4    var height = 0.0
5
6    init(width: Double, height: Double){
7        //First update Superclass using its init method
8        super.init(name: "Rectangle")
9        numberOfSides = 4 //And update the number of sides
10       //Then Class Specific
11       self.width = width
12       self.height = height
13       }
14 }
```

Testing the Updated Constructor

- Testing the code

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

-Gives the expected output

```
1  $ swift testing.swift
2  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

```
1  class Rectange: Shape
2      //.. <snip>
3      override func calcArea() -> Double {
4          return height*width
5      }
6
7      override func calcPerimeter() -> Double {
8          return 2*(height+width)
9      }
10 }
```

Testing it

-Again we Test

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

- Looks like it works

```
1  $ swift testing.swift
2  Rectangle: a Shape with 4 sides
3  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)

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

```
1   class Square: Rectangle {
2         init(length: Double){
3               //Initialise the super class
4     super.init(width: length, height: length)
5     //Update the name
6     name = "Square"
7     }
8   }
```

Testing the Square

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

```
1  $ swift testing.swift
2  Square: a Shape with 4 sides
3  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

```
1  enum possibleError: Error {
2      case indexOutOfRange
3      case duplicateItem
4  }
```

Structs:

- Group variables together

```
1  struct Item {
2      var title:String
3      var description:String
4      var added:NSDate = NSDate()
5      var done:Bool = false
6  }
```

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

```
1  enum possibleError: Error {
2      case notEnoughMoney
3      case duplicateItem
4  }
```

Throwing Errors

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

```
1  var money = 10 //How much money do we have
2  var cost = 50  //How much does something cost
3
4  if cost > money {
5      throw possibleError.notEnoughMoney
6  } else {
7    //.. Do Stuff
8  }
```

Throwing errors in functions

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

```
1  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

```
1  //Start block of code
2  do {
3     //Block that could fail
4     let result = try afunction(input)
5     print (result) //Gets run if things are OK
6  }
7     catch {
8        print (error) /Gets run if things break
9  }
```

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

```
1  //Start block of code
2  do {
3     //Block that could fail
4     let result = try afunction(input)
5     print (result) //Gets run if things are OK
6  }
7     catch possibleError.notEnoughMoney{
8        print (error) /Gets run if things break
9  }  catch { //Default
10       print ("Unhandled Error \(error)")
```

Tasks

Time for some Work

- Either:
  - Keep playing with Subclasses / Inheritance
    - Can you make code for regular polygons?
    - What other shapes can you subclass
  - There is some example code and a worksheet available.