GENERAL ASSEMBLY

MOBILE DEVELOPMENT

ENUM, STATIC, SWITCH

Kishin Manglani

AGENDA

- Recap
- Enums
- Static & Class
- Switch Statements

RECAP

WHILE LOOPS

WHILE LOOPS

- A while loop starts by evaluating a single condition. If the condition is true, a set of statements is repeated until the condition becomes false.

```
while condition {
    statements
}
```

REPEAT WHILE

- The other variation of the while loop, known as the repeat-while loop, performs a single pass through the loop block first, before considering the loop's condition. It then continues to repeat the loop until the condition is false.

```
repeat {
    statements
} while condition
```

COMPUTED PROPERTIES

COMPUTED PROPERTIES

- We can use computed properties when both getting and setting a variable

```
class Square {
    var sideLength = 100

    var area: Int {
        get {
            return sideLength * 2
        }
        set(newArea) {
            sideLength = newArea / 2
        }
    }
}
```

MORE ON FUNCTIONS

```
func someFunction(firstParameter: Int, secondParameter: Int) {
    //body
}
someFunction(2, secondParameter: 3)
```

INTRO TO SWIFT

```
func exampleFunction(firstName firstParameter: Int, secondParameter: Int) {
    //body
}
exampleFunction(firstName: 4, secondParameter: 9)
```

INTRO TO SWIFT

```
func similarFunction(firstName firstParameter: Int, secondName secondParameter: Int) {
    //body
}
similarFunction(firstName: 9, secondName: 9)
```

```
func anotherFunction(firstParameter: Int, _ secondParameter: Int) {
    //body
}
anotherFunction(2, 3)
```

STRUCTS

STRUCTS

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

CGGEOMETRY

CGGEOMETRY

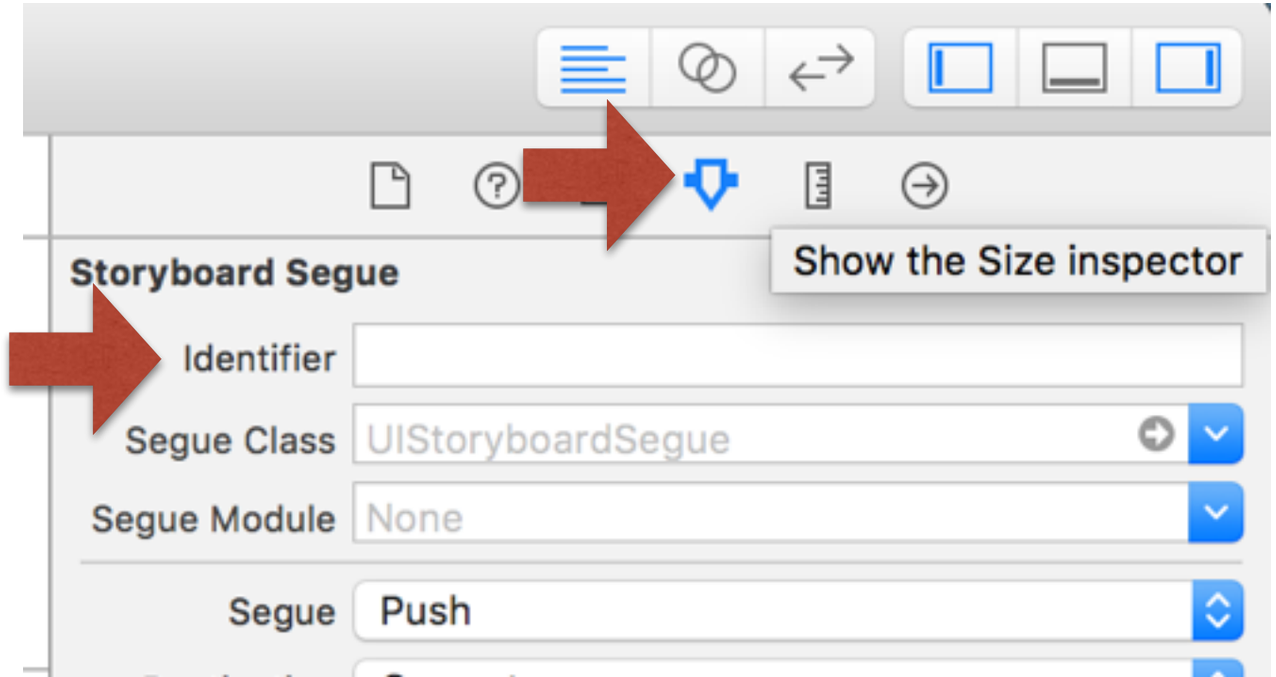
- CGPoint is a struct that represents a point in a two-dimensional coordinate system, typically an x coordinate and a y coordinate
- CGSize is a struct that represents the dimensions of width and height
- CGRect is a struct with both a CGPoint (origin) and a CGSize (size), representing a rectangle drawn from its origin point with the width and height of its size.
- To create a CGRect we pass in x, y, width and height values

SEGUE

SEGUES

- We can set a specific identifier/name for a segue
- UIViewController has a method called:

```
self.performSegueWithIdentifier("identifier", sender: self)
```



ENUMS

ENUMS

- Let's say we're building a map app, and somewhere in this app we have a property called currentDirection
- To represent directions we could use Integers:
  North: 1
  South: 2
  East: 3
  West: 4
- That can quickly get confusing for users and even when you go back to change your code, you may wonder what 3 actually means
- Again, out of context it has no meaning, and we want to try to eliminate that

ENUMS

- Another alternative is to use Strings:
  North: “north”
  South: “south”…
- But this is prone to typos:
  currentDirection == “norht”

ENUMS

- Enums (enumerations) allow us to create a data type that has a specified set of possible values
- For example, we can create a direction enum with North, South, East and West as possible values
- We can use enums to specify suits of cards in a card game
- Or we can even use enums to specify sections of a UITableView

ENUMS

- Enums (enumerations) allow us to create a data type that has a specified set of possible values
- After defining an enum, Xcode will autocomplete it's values (no more typos!)

- For example, we can create a direction enum with North, South, East and West as possible values
- We can use enums to specify suits of cards in a card game
- Or we can even use enums to specify sections of a UITableView

ENUMS

```
enum Month {
    case January
    case February
    case March
    case April
    case May
    case June
    case July
    case August
    case September
    case October
    case November
    case December
}
```

```
enum Season {
    case Winter, Summer, Spring, Fall
}
```

ENUMS

```
func getSeason(month: Month) -> Season {
    //body
}
```

ENUMS

- If the type of an enum is known or can be inferred then we can use dot syntax to access members.
- UITableViewCellStyle is actually an enum

```
UITableViewCell(style: UITableViewCellStyle.Default, reuseIdentifier: nil)
UITableViewCell(style: .Default, reuseIdentifier: nil)
```

ENUMS

- In Swift, enums don’t actually have values that back them
- In other words, in Swift when I create a month enum, the case January doesn’t represent anything
- All it is, is a case in the month enum with the value January

ENUMS

- Of course, we can change that behavior
- We can make an enum take on a Int, String, and a whole lot more
- Then the enum will represent our case value, but will also have this other value that backs it
- The type that backs an enum is called a raw type and the value is called a raw value

ENUMS

```
enum Month: Int {
    case January
    case February
    case March
    case April
    case May
    case June
    case July
    case August
    case September
    case October
    case November
    case December
}
Month.January.rawValue          0
Month.February.rawValue         1
```

ENUMS

```
//Declaring an enum
enum Month: Int {
    case January = 1
    case February = 2
    case March = 3
    case April = 4
    case May = 5
    case June = 6
    case July = 7
    case August = 8
    case September = 9
    case October = 10
    case November = 11
    case December = 12
}
Month.January.rawValue                1
Month.February.rawValue               2
```

ENUMS

```
enum Month: Int {
    case January = 1
    case February
    case March
    case April
    case May
    case June
    case July
    case August
    case September
    case October
    case November
    case December
}
Month.January.rawValue          1
Month.February.rawValue         2
```

ENUMS

```
enum Coin: Int {
    case Penny = 1, Nickel = 5, Dime = 10, Quarter = 25
}
```

ENUMS

```
enum TaskPriority: Int {
    case Low = 0
    case Medium = 1
    case High = 2
}
```

ENUMS

- A rawType gives an enum an initializer
- We can then initialize an enum with a rawValue
- These are “failable” initializers: if something goes wrong (like the rawValue is not compatible) the initializer can return nil

```
TaskPriority(rawValue: 0)          Low

TaskPriority(rawValue: 909)        nil
```

ENUMS

- Enums can also have properties and functions

```
enum TaskPriority: Int {
    case Low = 0
    case Medium = 1
    case High = 2

    static let sectionNames = ["Unimportant Tasks", "Medium Priority Tasks", "Urgent Tasks"]

    func sectionName() -> String {
        return TaskPriority.sectionNames[self.rawValue]
    }
}

TaskPriority.High.sectionName()
```

```
"Urgent Tasks"
```

```
"Urgent Tasks"
```

CLASS, STRUCT, ENUM

- Enums should represent a finite number of possibilities
- We can think of these as named labels

- Structs are a base type an cannot be subclassed,
- All functions and properties pertain to the instance
- Should not manipulate properties, etc. of other objects
- Is not the complete definition or model of your app

- Classes should be the primary actor/target of your app
- They represent the data model of your app
- Manipulate other objects properties

STATIC & CLASS

STATIC & CLASS

- Swift lets you create properties and methods that belong to a type, rather than to instances of a type
- Swift calls these shared properties "static properties", and you create one just by using the static keyword
- Because static methods belong to the class rather than to instances of a class, you can't use it to access any non-static properties from the class
- The class keyword does the same thing, but can only be used in classes
- When adding a property to an enum or a struct, we must use the static keyword
- Subclasses can override class methods, but cannot override static methods

STATIC & CLASS

- Why not make all properties and functions static?
- We cannot access properties belonging to an object in static functions
- Instance properties are what makes each object different
- When creating the building class, one thing we were considering was a numberOfBuildings property
- This would be a good candidate for a static variable

```
class Dog {
    static var numberOfDogs = 1

    init() {
        Dog.numberOfDogs += 1
    }
}
```

STATIC & CLASS

```swift
class Dog {
    static let dogNoise = "Bark bark"
    var defaultDogAge = 0

    func makeNoise() {
        print("\(self.defaultDogAge) \(Dog.dogNoise)")
    }

    static func makeNoiseStatic() {
        print("\(Dog.dogNoise)")
    }
}

Dog.makeNoiseStatic()
var puppy = Dog()
puppy.makeNoise()
```

```
"Bark bark\n"
"0 Bark bark\n"
```

Dog
Dog

SWITCH STATEMENTS

SWITCH STATEMENTS

- A switch statement considers a value and compares it against several possible matching patterns.
- It then executes an appropriate block of code, based on the first pattern that matches successfully.
- A switch statement provides an alternative to the if statement for responding to multiple potential states.

SWITCH STATEMENTS

```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,
value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

SWITCH STATEMENTS

```
var age = 30

switch age {
case 0:
    print("You are 0 years old")
case 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40:
    print("You are in your 30's")
case 40...49:
    print("you are in your 40's")
default:
    print("You are not 20 and not 30")
}
```

IF/ELSE, FOR LOOP, WHILE LOOP, SWITCH

- If else and switch are pretty similar
- If there are two cases, probably use an if
- If there are many cases, use a switch
- When iterating through an array use a for loop
- When iterating through a finite or known quantity use a for loop
- When iterating endlessly or an indefinite amount of time, use a while loop

UIIMAGE

UIIMAGE

- UITableViewCell also has an imageView property
- imageView has an image property which is of the type UIImage
- We can add images to the Xcode project through the asset catalog and create instances of them by calling `UIImage(named: "coffee")`

LAB