Advanced Mobile Application Development
Week 1: Advanced Swift

Swift 2

What's New in Swift https://developer.apple.com/videos/play/wwdc2015-106/ (32 mins)

- Swift 2 0-20:14
- Error Handling 28:21-40:45

Go into Xcode (swift3)
File | New | Playground
Name: playingwithswift
Platform: iOS
Save
Delete what's there so you start with an empty file

Optionals

```
var score : Int?
print("Score is \(score)")
score=80
if score != nil {
    print("The score is \(score!)")
}

if let currentScore = score {
    print("My current score is \(currentScore)")
}

let newScore : Int! = 95
print ("My new score is \(newScore)")
```

Since score is given an initial value, and it's a constant, we now it will always have that value and never be nil
Rather than placing an exclamation mark after the optional's name each time you use it, you place an exclamation mark after the optional's type when you declare it.

Functions

```
func sayHello () {
    print("Hello class")
}

sayHello()

func sayHello (first: String, last: String){
    print("Hi \(first) \(last)")
}

sayHello("Bill", last: "Adams")

func sayWhat (firstName first: String, lastName last: String){
```

```
    print("What \(first) \(last)?")
}

sayWhat(firstName: "Bill", lastName: "Adams")

func sayWhy (firstName first: String, lastName last: String)->String{
    return "Why " + first + " " + last + "?"
}

let msg = sayWhy(firstName: "Jane", lastName: "Adams")
print(msg)
```

Closures

Swift's standard library provides a method called `sort(_:)`, which sorts an array of values of a known type, based on the output of a sorting closure that you provide. Once it completes the sorting process, the `sort(_:)` method returns a new array of the same type and size as the old one, with its elements in the correct sorted order. The original array is not modified by the `sort(_:)` method.

```
let names=["Tom", "Jessie", "Megan", "Angie"]
```

The `sort(_:)` method accepts a closure that takes two arguments of the same type as the array's contents, and returns a `Bool` value to say whether the first value should appear before or after the second value once the values are sorted. The sorting closure needs to return `true` if the first value should appear *before* the second value, and `false` otherwise.

This example is sorting an array of `String` values, and so the sorting closure needs to be a function of type `(String, String) -> Bool`.

You could write a normal function and pass it to the sort(_:) method.

```
func backwards(s1: String, _ s2: String) -> Bool {
    return s1 > s2
}

var reversed = names.sort(backwards)
```

Or you can use a closure

```
//closures

reversed = names.sort({(s1:String, s2: String)->Bool in return s1 > s2})
print(reversed)
```

This is the same syntax as in the function but it's passed as a closure in {}

```
reversed = names.sort( { s1, s2 in return s1 > s2 } )
print(reversed)
```

Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns.

Because all of the types can be inferred, the return arrow (->) and the parentheses around the names of the parameters can also be omitted:

```
reversed = names.sort( { s1, s2 in s1 > s2 } )
print(reversed)
```

Because the closure's body contains a single expression (s1 > s2) that returns a Bool value, there is no ambiguity, and the return keyword can be omitted.

```
reversed = names.sort( { $0 > $1 } )
print(reversed)
```

Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names $0, $1, $2, and so on.
If you use these shorthand argument names within your closure expression, you can omit the closure's argument list from its definition, and the number and type of the shorthand argument names will be inferred from the expected function type. The in keyword can also be omitted, because the closure expression is made up entirely of its body:
$0 and $1 refer to the closure's first and second String arguments.

Enums

```
enum carType {
    case gas
    case electric
    case hybrid
}

var car = carType.electric
```

The type of car is inferred when it's initialized with a value of carType. You can then set it using the shortened dot notation.

```
car = .hybrid
```

Type Casting

Define a base class and 2 subclasses

```
class Pet {
    var name: String
    init(name: String){
        self.name = name
    }
}

class Dog : Pet {
    var breed: String
    init(name: String, breed: String) {
        self.breed=breed
        super.init(name: name)
    }
}
```

```
class Fish : Pet {
    var species: String
    init(name: String, species: String) {
        self.species=species
        super.init(name: name)
    }
}
```

Create an array with two Dog instances and 1 Fish instance

```
let myPets=[Dog(name: "Cole", breed: "Black Lab"), Dog(name: "Nikki", breed:
"German Shepherd"), Fish(name: "Nemo", species: "Clown Fish")]
```

The items stored in myPets are still Dog and Fish instances behind the scenes. However, if you iterate over the contents of this array, the items you receive back are typed as Pet, and not as Dog or Fish. In order to work with them as their native type, you need to check their type, and downcast them.

```
var dogCount = 0
var fishCount = 0

for pet in myPets {
    if pet is Dog {
        ++dogCount
    }
    else if pet is Fish {
        ++fishCount
    }
}

print("I have \(dogCount) dogs and \(fishCount) fish")
```

To print the appropriate breed/species of each pet we need to access each item as a Dog or Fish and not just as a Pet. We use the conditional form of the type cast operator (as?) to check the downcast each time through the loop.

```
for pet in myPets {
    if let dog = pet as? Dog {
        print("\(dog.name) is a \(dog.breed)")
    } else if let fish = pet as? Fish {
        print("\(fish.name) is a \(fish.species)")
    }
}
```