Advanced Mobile Application Development
Week 1: Advanced Swift

Swift 3
Swift was announced in 6/14 as a new programming language created by Apple and the first version was released in 9/14.
Swift became open source in 12/15 and now has an active open source community.

What's New in Swift https://developer.apple.com/videos/play/wwdc2015-106/
- language 15:05 -18:30 (skip 18:35-21:15 objective-C)
- core language 21:15-25:00
- type system 26:20-30:08 (optional)

What's New in Swift https://developer.apple.com/videos/play/wwdc2015-106/
- Pattern Matching 15:27 -18:20(guard)
- Error Handling 28:21-40:45

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

Optionals
One of the unique aspects of Swift is the concept of optionals(end of "The Basics" section)
- Defining a variable as an optional says it might have a value or it might not
- If it does not have a value it has the value nil
  - Nil is the absence of a value
- Optionals of any type can have the value nil
- A '?' after the type indicates it's an optional
- If you define an optional variable without providing a default value, the variable is automatically set to nil for you

```
var score : Int?
print("Score is \(score)")
score=80
print("score is \(score!)")
```

To access the value of an optional you must add an '!'. This is called **forced unwrapping**.
If you force unwrap an optional that is nil your program will crash so you should use an if statement to find out if an optional has a value before unwrapping it.
```
if score != nil {
    print("The score is \(score!)")
    print("The score is ", score!)
}
```

This is so common in Swift that there's a shorthand for it called **optional binding**. You can conditionally unwrap an optional and if it contains a value, assigns it to a variable or constant which has a local scope(only available where it's defined such as in this if statement)

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

Sometimes it's clear that an optional will always have a value, after the first value is set
We can unwrap these optionals without the need to check it each time
These are called **implicitly unwrapped** optionals

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

- Since newScore 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.
- '!' after the type indicates it's an implicitly unwrapped optional
- No "!" is needed to access the optional because it's an implicitly unwrapped optional
- You see this when you make outlet connections from Interface Builder. It's an optional but because you know it exists in the view it is implicitly unwrapped and we know it will never become nil
  - `@IBOutlet weak var name: UILabel!`

Implicitly unwrapped optionals should not be used when there is a possibility of a variable becoming nil at a later point.

Collection Types
[image]
Swift has three types of collections

- Arrays
  - ordered collections of values
- Sets
  - unordered collections of distinct values
- Dictionaries
  - unordered collections of key/value pairs
- The collection will be mutable if it's assigned to a variable, immutable if it's assigned to a constant
- Properties
  - .count returns the number of items in an array
  - .isEmpty is a boolean to see if count is 0

Arrays
Arrays store an ordered collection of values
Arrays start with an index of 0 just as in other languages

- remove(at:) and removeLast() return the removed item
- insert(_:at:) inserts an item into the array at a specified index

```
var shoppingList=["cereal", "milk"]
print(shoppingList[0])
```

```
shoppingList.append("bread")

if shoppingList.isEmpty{
    print("there's nothing you need")
} else {
    print("You need \(shoppingList.count)" + " items")
}

let removedItem = shoppingList.removeLast()

for item in shoppingList{
        print(item)
}
```

Dictionaries

Dictionaries store unordered key/value data pairs
.keys returns all the keys
.values returns all the values
updateValue() returns the old value for that key
removeValueForKey() returns the removed value or nil if no value existed

```
var newList=[String:String]()
```

creates an empty dictionary of whose keys are of type String and whose values are of type String.

```
var classes=["4120":"MAD", "3000":"Code"]
```

As with arrays, you don't have to write the type of the dictionary if you're initializing it with a dictionary literal whose keys and values have consistent types.

```
classes["3000"]
classes["2000"]="MIT"
classes.count

classes.updateValue("Mobile App Dev", forKey: "4120")
classes.removeValue(forKey:"3000")

for (num, name) in classes{
    print("\(num): \(name)")
}
```

Functions

Functions provide a way to group a set of instructions that perform a specific task.

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

sayHello()

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

```
}
```

Each function parameter has both an argument label and a parameter name. The parameter name is used as a local variable in the function. By default, parameter names are used as the argument label if a separate one isn't given. When you call a function you use the argument label followed by a : and then its value.

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

Or you can specify a different argument label before the parameter name.

```
func sayWhat (firstName first: String, lastName last: String){
    print("What \(first) \(last)?")
}

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

If you don't want an argument label for a parameter, use an underscore (_) instead of an explicit argument label for that parameter.

```
func sayWhere(_ first: String, last:String){
    print("Where's \(first) \(last)?")
}

sayWhere("Michelle", last: "Doe")
```

In documentation you would see this function referenced as sayWhere(_: last:) with the full declaration listed as the function header. You will see this in the SDK documentation a lot.

Return types are specified with an arrow.

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

let msg = sayWhy(first: "Jane", last: "Adams")
print(msg)
```

Closures

Closures are blocks of code that can be passed around and used in your code.

- Closures in Swift are similar to blocks in C and Objective-C and anonymous functions in JavaScript

Functions are really just closures with a name

Swift's standard library provides a method called `sorted(by:)`, 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 `sorted(by:)` 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 `sorted(by:)` method.

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

The `sorted(by:)` 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 sorted(by:) method.

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

var reversed = names.sorted(by:backwards)
```

Or you can use a closure. This is the same syntax as in the function but it's passed as a closure in {}. The start of the closure's body is introduced by the in keyword. This keyword indicates that the definition of the closure's parameters and return type has finished, and the body of the closure is about to begin.

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

```
reversed = names.sorted(by: { $0 > $1 } )
print(reversed)
```

Enumerations

An enumeration defines a type for a group of related values.

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

var car = carType.electric
print(car)
```

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
print(car)
```

Type Casting

Type casting is a way to check the type of an instance

- Use the "is" type check operator to test whether an instance is of a certain class type
  - Returns **true** if it is of that type
  - Returns **false** if is is not of that type

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 += 1
    }
    else if pet is Fish {
        fishCount += 1
    }
}

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.
Type casting lets you treat an instance as if its is a different class in its class hierarchy
When you believe an instance refers to the subclass type use the "as" type cast operator to try to downcast to the subclass type

- Use the conditional form "as?" when you're not sure if the downcast will succeed
    - Returns an optional
    - Returns nil if the downcast wasn't possible
- Use the forced form "as!" when you are sure the downcast will always succeed
    - Attempts the downcast and force-unwraps the result
    - You will get a runtime error if you try to downcast to an incorrect class type

Casting treats the instance being cast as an instance of the type to which it has been cast
Casting does not actually modify the instance or change its value

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)")
    }
}
```

AnyObject can represent an instance of any class type

- Objective-C does not have typed arrays so the SDK APIs often return an array of [AnyObject]
- If you know the type of objects in the array you can use the force form to downcast to that class type

Any can represent an instance of any type at all, including function types and non-class types

Error Handling

Error handling is the process of responding to and recovering from error conditions in your program. Enums are often used to represent error conditions.

```
enum WebError: Error{
    case Forbidden
    case NotFound
    case RequestTimeout
}
```

There are four ways to handle errors in Swift.

1. Propagate the error from a function to the code that calls that function

A throw statement returns an error and immediately transfers program control back to where the function was called. To indicate that a function can throw an error, you write the throws keyword in the function's declaration after its parameters. A function marked with throws is called a throwing function. Only throwing functions can propagate errors. (added in Swift 2)

```
func webPage(status: Int) throws -> String{
    switch status{
        case 403: throw WebError.Forbidden
        case 404: throw WebError.NotFound
        case 408: throw WebError.RequestTimeout
        default: return "OK"
    }
}
```

2. Handle the error as an optional value

Use the try keyword when calling a function that throws an error

You use try? to handle an error by converting it to an optional value. If an error is thrown while evaluating the try? expression, the value of the expression is nil.

```
var status = try? webPage(status: 400)
status = try? webPage(status: 404)
```

3. Handle the error using a do-catch statement

Use a do-catch statement to handle errors. If an error is thrown in the do clause, it is sent to the catch clause.

```
do {
    try webPage(status: 404)
} catch WebError.Forbidden {
    print("Forbidden")
} catch WebError.NotFound {
    print("File not found")
} catch WebError.RequestTimeout {
    print("Request time-out")
}
```

4. Assert that the error will not occur.

Sometimes you know a throwing function or method won't throw an error at runtime. In that case you can write try! before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, you'll get a runtime error.

Early Exit

In Swift 2 a new guard statement was introduced to avoid nested if statements (pyramids of doom)
A guard statement, like an if statement, executes statements depending on the Boolean value of an expression.
In a guard statement the condition must be true in order for the code after the guard statement to be executed. If it's false the code inside the else clause is executed.

- Lets you handle false conditions early, keeping the code that handles a violated condition next to the test condition
- Always has an else clause that MUST transfer control out of the code block. You can transfer control with an early exit
  - Continue: used in loops to skip that iteration and go to the next iteration of the loop
  - Break: used in loops or switch statements to exit completely out of the loop or switch and go on to the rest of the function
  - Return: exits out of the current scope. In functions this will return control to where the function was called
  - Throw: used to throw (return) an error
- The code that is typically run is kept in the main flow and not wrapped in an else block
- Code is more readable and easier to maintain
- Any variables or unwrapped optional in the guard remain in scope after the guard finishes, so you can use them.

guard boolean
    else {
      false
      transfer control
    }
true
main body continues

```
enum MathError:Error{
    case DivideByZero
}

func divide(number1: Double, number2:Double) throws -> Double{
    guard number2>0 else{
        throw MathError.DivideByZero
    }
    return number1/number2
}

var answer = try? divide(number1: 10, number2: 5)

do {
    try divide(number1: 10, number2: 0)
} catch MathError.DivideByZero {
    print("You can't divide by zero")
```