Isaac Sheets
isaac.sheets@colorado.edu
ATLS 4320: Advanced Mobile Application Development

Week 1: Advanced Swift

Swift 5.0 and 5.1 Release Notes

Added a bunch of behind the scenes stuff that isn't that important for us. Their main thing as you saw in the WWDC video last class is ABI stability, which is very important for Swift as a whole but we won't really care that much about it in the context of this class. Basically, it just ensures that libraries and such written in Swift 5.0+ are guaranteed to be compatible with all apps and libraries written in other 5+ versions of Swift.

We did get some stuff that will be useful for us:

- Default values are respected by memberwise (implicit) initializers for structs
- Implicit returns for single expression functions — less code
- Raw strings
- Easily transforming all values in a dictionary — such as String to Int or vice versa

We'll go over some of these new features as well as some more advanced ones that have been around longer using a Swift playground.

New and Advanced Swift

Go into Xcode (swift5)
File | New | Playground
iOS Blank
Name: swift5
Save

String Literals and Raw Strings

Multi-line string literals can be useful. As of Swift 4.1 they have a simple syntax using 3 quotes.
The indentation of the closing delimiter determines how much whitespace is stripped from the start of each line.

```
let literal = """
    this string is pre-formatted
and it takes up
  multiple lines
"""
print(literal)
```

Raw strings allow the use of special characters within strings without the need to escape them

```
//raw string - no need to escape special characters!
let raw = #"I don't want to "go to work" I just want to "sleep""#
print(raw)

//interpolation is a little different for raw strings
```

```
let unreadMessages = 3
let rawInterpolation = #"You have \#(unreadMessages) unread messages"#
print(rawInterpolation)
```

This is especially useful for RegEx

```
let names = "Ariel, Danny, Aileen, David, Annie, Matt, Laura"
let aileenRegEx = "\\bAileen\\b"
let aileenBetterRegEx = #"\bAileen\b"#

let aileenRange = names.range(of: aileenBetterRegEx, options: .regularExpression)
print(names[aileenRange!])
```

It's also possible to combine raw and multiline strings

```
let rawLiteral = #"""
Literally such a raw string
Yeah haha, and super literal...
"""#
print(rawLiteral)
```

Boolean Toggle

Swift 4.2 added the toggle() function to Bool, making it really easy to toggle a Boolean.

```
var booleansAreEasyToToggle: Bool = false
print(booleansAreEasyToToggle)

booleansAreEasyToToggle.toggle()
print(booleansAreEasyToToggle)
```

Custom String Interpolation

New in Swift 5 is the ability to customize the way a variable of any type is printed when it's interpolated into a string. This can be useful for better logging during development, or if you'd like to make a class or struct print pretty if it'll be user facing.

```
//create struct
struct MenuItem {
    var name: String
    var price: Double
}

//instantiate
let coffee = MenuItem(name: "Coffee", price: 3.35)
let tea = MenuItem(name: "Tea", price: 2.25)
//show initial print statement format
print("""
Beverages:
\(coffee)
```

```
\(tea)
""")

//extend the String.StringInterpolation
extension String.StringInterpolation {
    //use the mutating keyword since we are modifying the properties of a value
type (String is a struct)
    mutating func appendInterpolation(_ value: MenuItem) {
        appendInterpolation("\(value.name)......$\(value.price)")
    }
}
```

Optionals

Let's review one of the unique aspects of Swift -- optionals (end of "The Basics" section). In Swift a variable/constant can only have the value of nil if it's defined as an optional.

- 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

```
//declare optional variable and no initial assignment, thus the initial value is
nil
var score: Int?
print("Score: \(score)")

//assign a value
score = 80
//force unwrap the optional -- not good since program will crash if the optional
is nil
print("Score: \(score!)")

//first check the value to ensure it is not nil -- still not the best way
if score != nil {
    //then force unwrap
    print("The score is \(score!)")
}

//optional binding! -- great!
if let currentScore = score {
    print("The score is \(currentScore)")
}
```

```
//unwrapping with a coalescing operator and a default value
score = nil
let myScore = score ?? 0
print("My score is \(myScore)")

//optional chaining: gracefully accessing members, properties, or methods of
optionals
struct Developer {
    var name: String
    var app: App?
}

struct App {
    var title: String
}

let nate = Developer(name: "Nate") //, app: App()) - uncomment to give the
optional property a value
print(nate.dietaryRestriction!.type) //runtime error! try to access property of
nil optional

//proper way to access optional property -- try it with an optional that has
if let app = nate.app?.title {
    print("\(nate.name) has released the following apps: \(app)")
} else {
    print("\(nate.name) has not released any apps")
}
```

Type Casting

Type casting is a way to check the type of an instance and treat that instance as a different type in its class hierarchy.

Define a base class and 2 subclasses

```
//declare a super class
class Commuter {
    var efficiency: Int
    init(efficiency: Int){
        self.efficiency = efficiency
    }
}

//two subclasses with unique properties and initializers
class Bike : Commuter {
    var type: String
```

```
    init(efficiency: Int, type: String) {
        //set the breed property
        self.type=type
        //pass name to init in super class
        super.init(efficiency: efficiency)
    }
}

class Skates : Commuter {
    var brand: String
    init(efficiency: Int, brand: String) {
        //set the brand property
        self.brand=brand
        //pass the efficiency to init in super class
        super.init(efficiency: efficiency)
    }
}

var myCommuters = [Bike(efficiency: 8, type: "road"), Bike(efficiency: 6, type:
"mountain"), Skates(efficiency: 4, brand: "Rollerblade"), Skates(efficiency: 3,
brand: "K2")]
```

(option click on myCommuters to see its type)

The items stored in myCommuters are still Bike and Skates instances behind the scenes. However, if you iterate over the contents of this array, the items you receive back are typed as Commuter, and not as Bike or Skate. To print the number of each type of vehicle we need to figure out if each item as a Bike or Skate and not just as a Commuter. The `"is"` type check operator lets you test whether an instance is of a certain class type

- Returns **true** if it's of that type
- Returns **false** if it's NOT of that type

```
var bikeCount = 0
var skateCount = 0
//the "is" operator tests if an object is a certain type and returns true or
false
for commuter in myCommuters {
    if commuter is Bike {
        bikeCount += 1
    }
    if commuter is Skates {
        skateCount += 1
    }
}
```

In order to work with them as their native type, you need to check their type, and then downcast them.

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
You can only case an instance as a different superclass or subclass from somewhere else in its own class hierarchy.
Casting does not actually modify the instance or change its value

```
//the "as?" operator attempts to downcast to a type returns an optional that is
nil if downcast was unsuccessful
//needed to work with objects that are subclass types
for commuter in myCommuters {
    if let bike = commuter as? Bike {
        print("Commuter is of type Bike")
    }
    if let skates = commuter as? Skates {
        print("Commuter is of type Skates")
    }
}
```

We use the conditional form of the type cast operator (as?) to check the downcast each time through the loop.

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.

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. *Useful for things like a completion handler for async functions.*

Closure syntax

```
{ (parameters) -> returnType in statements }
```

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.

```
//test array
let roster = ["Evan", "Eva", "Tommy", "Tom", "Nicole", "Sharon"]
```

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 backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}

var reversedRoster = roster.sorted(by: backward)
print(reversedRoster)
```

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.

```
var reversedRoster = roster.sorted(by: {(s1: String, s2: String) -> Bool in
return s1 > s2})
print(reversedRoster)
```

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:

```
var reversedRoster = roster.sorted(by: {s1, s2 in s1 > s2})
print(reversedRoster)
```

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.

```
var reversedRoster = roster.sorted(by: {$0 > $1})
print(reversedRoster)
```

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.

```
var reversedRoster = roster.sorted(by: >)
print(reversedRoster)
```

Enumerations

An enumeration defines a type for a group of related values that are fixed and known in advance.

```
//create enum
enum JobStatus {
    case notStarted
    case inProgress
    case done
}

//example struct that uses it
struct Job {
    var title: String
    var status: JobStatus
}

//example instance
var lab1 = Job(title: "Lab 1", status: JobStatus.notStarted)

//switch statement on the enum - the JobStatus type is implied...
switch lab1.status {
case .notStarted:
    print("Get to work!")
case .inProgress:
    print("I'm working on it")
case .done:
    print("Pay me")
default:
    print("unknown status")
}
```

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.

```
//create enum, conform to Error protocol
enum APIError: Error {
    case Forbidden
    case NotFound
    case RequestTimeout
    case UnknownResponse
}
```

Steps for throwing and handling errors with functions

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)

```
//dummy function to check response -- throws error if response is bad or returns
string if response is good or unknown
func checkResponse(status: Int) throws -> String {
    switch status {
    case 403: throw APIError.Forbidden
    case 404: throw APIError.NotFound
    case 408: throw APIError.RequestTimeout
    case 200: return "OK"
    default: throw APIError.UnknownResponse
    }
}
```

2. 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.

```
//call the function using a do catch statement and a try keyword
do {
    let status = try checkResponse(status: 400)
    print("Good to go, no worries")
} catch {
    print(error)
}

//better yet, catch specific types of errors and handle each individually
do {
    let status = try checkResponse(status: 404)
    print("Good to go, no worries")
} catch APIError.Forbidden {
    print("You don't have permission to access this resource")
} catch APIError.NotFound {
    print("Couldn't find the resource")
} catch APIError.RequestTimeout {
    print("Took too long to get resource")
} catch APIError.UnknownResponse {
    print("Something went wrong. Don't know what.")
}
```

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. (end of Control Flow section)
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.

Syntax

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

Example

```
//enum for an error
enum MathError: Error {
    case DivideByZero
}

//use a guard statement to exit early if the divisor is 0
func divide(dividend: Double, divisor: Double) throws -> Double {
    guard divisor != 0 else {
        throw MathError.DivideByZero
    }
    return dividend/divisor
}

do {
    print(try divide(dividend: 3, divisor: 0))
} catch MathError.DivideByZero {
    print("Cannot divide by 0!")
}
```