Mobile Application Development
Week 3: Swift Intro

When the iOS SDK (originally called the iPhone SDK) was first released in 2008, developers used Objective-C to develop their apps.
Objective-C was developed in the 1980s and used by NeXT, founded by Steve Jobs, for its NeXTSTEP operating system. Apple purchased NeXT in 1997 and Steve Jobs returned, bringing Objective-C to Apple where it resulted in iOS, macOS, watchOS, and tvOS. So whenever you see "NS" in any class names, that comes from NeXTSTEP.
Cocoa Touch is the part of the SDK that includes the frameworks developers use to develop iOS apps. These frameworks are mainly written in Objective-C.

Swift
Objective-C was not an easy or modern language to learn, so Apple developed Swift as an alternative OOP language for developing iOS, macOS, watchOS, and tvOS apps.
Swift was first announced in June 2014 at the WorldWide Developer Conference and released with Xcode 6 later that year. Version 2.2 was made open source in 2015 and it's usage has spread, including to server-side applications as well.
Swift works side-by-side with Objective-C so developers can use either, and you can even have files of both languages in one app. Apple is definitely promoting Swift as its preferred language moving forward.

Xcode 12.5/iOS 17.7 uses Swift 5.4.

Swift 5 changes
There were a lot of noticeable changes from Swift 2 to 3, including many of the API method names and headers changed in Swift 3, mainly to optimize readability. You might notice this if you look at any older code online. Since then there have been fewer major changes.
Swift 4.2 focused on making Swift easier, safer, and faster. Including developer productivity, improving performance, faster build times, and reducing the need for some boilerplate code.
The goal of Swift 5 was to increase stability. Swift 5.2 included some changes that reduced code size and memory usage. Swift 5.3 continued with those improvements and includes some new features. Swift 5.4 includes some compilation improvements, including better code completion in expressions with errors.

Xcode typically lets you compile in 2 or 3 versions of swift and includes a migrator tool to help move projects from the previous version to the current version of Swift.

Playground
(swift1 playground)
A Swift playground file allows you to test out Swift code, and see the results of each line in the sidebar. It's a great place to test out some code without having to set up a whole app to try it.
An alternative is using a REPL that runs in any browser such as https://repl.it/repls

Xcode
File | New | Playground
iOS | Blank template
Name: swift intro
Save

Delete what's there so you start with an empty file

Variables and constants

- Variables and constants associate a name with a value of a specific type
  - The value of a variable can change
  - The value of a constant cannot change
  - You can't change a constant into a variable or a variable into a constant
- Only use variables when the value will change, otherwise use constants.
- Variables use the keyword `var`, constants use `let`
- Can use almost any character in a variable or constant name
- Swift is a strongly typed language
  - Once you've declared a constant or variable of a certain type, you can't change its type.

```
var message : String = "Hello class"
let classMax : Int = 20
var age : Int
age = 20
age="old" (gives error, wrong type)
```

Data Types

Swift has all the standard data types (all listed in book and docs)

- **Int** represents integers
- **Double** represents 64-bit floating points
  - has a precision of at least 15 decimal digits
- **Float** represents 32-bit floating points
  - can have the precision be as little as 6 decimal digits
- **String** represents characters
  - Values must be in quotes
  - String is a value type
  - Use \ to escape a quote in a String
  - String is a value type so it's value is copied if it's assigned or passed
- **Bool** represents Boolean
  - Values true or false

Type Inference

- If you provide an initial value for a variable or constant Swift can infer the data type
- Because Swift is type safe, it performs type checks when compiling your code and flags any mismatched types as errors.
- Without an initial value for a variable or constant you must provide the type

```
var name = "Aileen"
firstName="Aileen"  (error, needs var or let)
var firstName="Aileen"
name="Pierce"
name=20 (gives error, wrong type)
```

print is a global function for output (there's no println in Swift)
To print out the value of a variable or constant as part of a string use a backslash and parentheses.

```
print("Who am I?")
```

```
print(firstName)
print("Hi " + firstName)
print("My name is \(firstName)")
print("\(firstName)"+"\(age)")
print("\(firstName)" + " \(name)")
```

Type Conversion

Swift does not provide implicit type conversion when you assign a variable of one numeric type to a variable of another numeric type, you must do the conversion explicitly.

```
let a = 42
let b = 0.123
let c = a + b (error)
```

You must convert the Int to a Double

```
let c = Double(a) + b
```

Comments

Comments are used to include notes, citations, or explanations in your code.

- // is used for single line comments
- /* comment */ is used for multi-line comments

Tuples

A tuple is a way to group values

- Any type
- As many values as you want
- Access values using dot notation starting with 0
- Can also assign to more useful names

Useful way to pass multiple values around easily

```
let violet = ("#EE82EE", 238, 130, 238)
print("Violet is \(violet.0)")
let (hex, red, green, blue) = violet
print("Violet is \(hex)")
```

Operators

- Swift supports the standard arithmetic and comparison operators (in books and docs)
- Equality is = =
- String concatenation: +
- The ++ and – shorthand is not supported in Swift 3 and later
    - Use += or -= instead
    - a=a+2 or a+=2
- Logical operators
    - Logical NOT: !
    - Logical AND: &&
    - Logical OR: ||
- Range operators are shortcuts for expressing a range of values
    - The closed range operator (a…b) defines a range that includes a and b.
    - The half-open range operator (a..<b) defines a range that includes a but NOT b
    - One-sided ranges let you omit the value on one side of the range (similar to Python)
        - (…b) starts at the beginning and goes up to b

- (a…) starts at a and goes to the end
- and the equivalent for the half-open range operator where the final value isn't part of the range

Conditionals

If/else

- if and if/else statements are the same as in other languages.
- The test condition does not need to be in parentheses (but can be)
- if and else bodies MUST be in curly braces, even if it's only 1 line

```
let young = "you're young"
let notyoung = "you're not young"

if age < 21 {
    print(young)
}
    else {
    print(notyoung)
}
```

(change the value you're testing age against and see how the playground changes)

Swift also has the ternary operator ?: which can be used to shorten an if/else statement. It evaluates the Boolean expression before the ? and executes either the statement before the : if the result is true, or the statement after the : if the result is false.

```
age < 21 ? young : notyoung
```

Some believe the ternary operator makes code hard to read, but you might see it.

switch

- A switch statement compares a value against possible matching cases
- Case statements can have multiple values separated by commas, or can be a range
- Cases do not automatically fall through so you don't need a break in each case.
- Switch statements must be exhaustive
  - A case for each possible value OR
  - Default case for any values that don't match a case

```
switch age{
     case 0...5: print("You're a wee bitty one")
     case 6...21: print("Enjoy school")
     case 22...55: print("Welcome to the real world")
     default: print("I don't know what you're doing")
}
```

before the switch statement add
`age=21`
and change the second case to be ..<21 and the third case to start at 21
this is using the half open range operator
(watch the playground change)

Loops

Swift supports for and while loops

for loop

- Swift 3 removed support for the C-style for loop
- The **for-in** loop is used to iterate over collections of items
- Loops don't need parentheses around the conditional
- No need for 'var' for the counter in a for loop

while loop

- While loops evaluate its condition at the start of each pass through the loop
- Repeat-while loops evaluate its condition at the end of each pass through the loop
    - Similar to do-while loops

```
for count in 0...age
{
    print("\(count)")
}
```

Functions

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

- Keyword func
- Function name
- Argument list (optional)
- Return type (optional)
- Body in { }

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

Call the function

```
sayHi()
```

Parameters are named and typed (can't be inferred like variables because there are no initial values)

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

When calling a function with parameters you need to include the parameter names. Use auto complete!

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

Functions can also have external parameter names to be used when the function is called

- Place the external parameter name before the local one
- If no external parameter is supplied it's taken to be the same as the internal parameter name

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

Call a function using the external parameter name.

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

An underscore in the parameter list means there's no external parameter name so you don't need to supply one when calling the function

- You will see this in the SDK methods a lot

```
func sayWhere(_ place:String){
    print(place)
}
```

When you call a function with an _ for the external parameter name you don't need a named parameter.

```
sayWhere("here")
```

Functions can also return a value

- List the return type with an arrow after the parameter list
- You can use a tuple to return multiple values
- Void means there's no return type (this is not needed)

```
func sayWho(firstName : String, lastName : String) -> String {
    return "Who " + firstName + " " + lastName + "?"
}

let msg2=sayWho(firstName: "Jim", lastName: "Adams")
print(msg2)
```

Optionals

One of the unique aspects of Swift is the concept of optionals (book page 595, Swift programming language book towards the end of the basics section)

Defining a variable as an optional allows it to not have a value

- 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

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

(ignore warnings)

Change score to nil (and then back). If score was not defined as an optional this would cause an error (remove the ? to see the error).

To access the value of an optional you need to unwrap it. You can force unwrap it by adding an '!'.

```
print("score is \(score!)")
```

If you force unwrap an optional that is nil your program will crash so you should always check first to find out if an optional has a value before unwrapping it.

```
if score != nil {
    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 block scope of the if statement.

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

Sometimes it's clear that an optional will always have a value and never be nil
We can unwrap these optionals without the need to check it each time
These are called implicitly unwrapped optionals
'!' after the type indicates it's an implicitly unwrapped optional

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

- No "!" is needed to access the optional because it's an implicitly unwrapped optional
- Implicitly unwrapped optionals should not be used when there is a possibility of a variable becoming nil at a later point.
- You will see that the variables created as outlet connections are implicitly unwrapped. That's because once the view is loaded we can be sure that object exists because it's part of the UI.

If a variable is not defined as an optional, it cannot have the value nil, it must have a value.

```
var finalScore : Int
finalScore = nil //error
```

Swift 5.5
Xcode 13 will support Swift 5.5 which continues with improvements and new features with concurrency handling being one main area.