Mobile Development for iOS

Russell Mirabelli

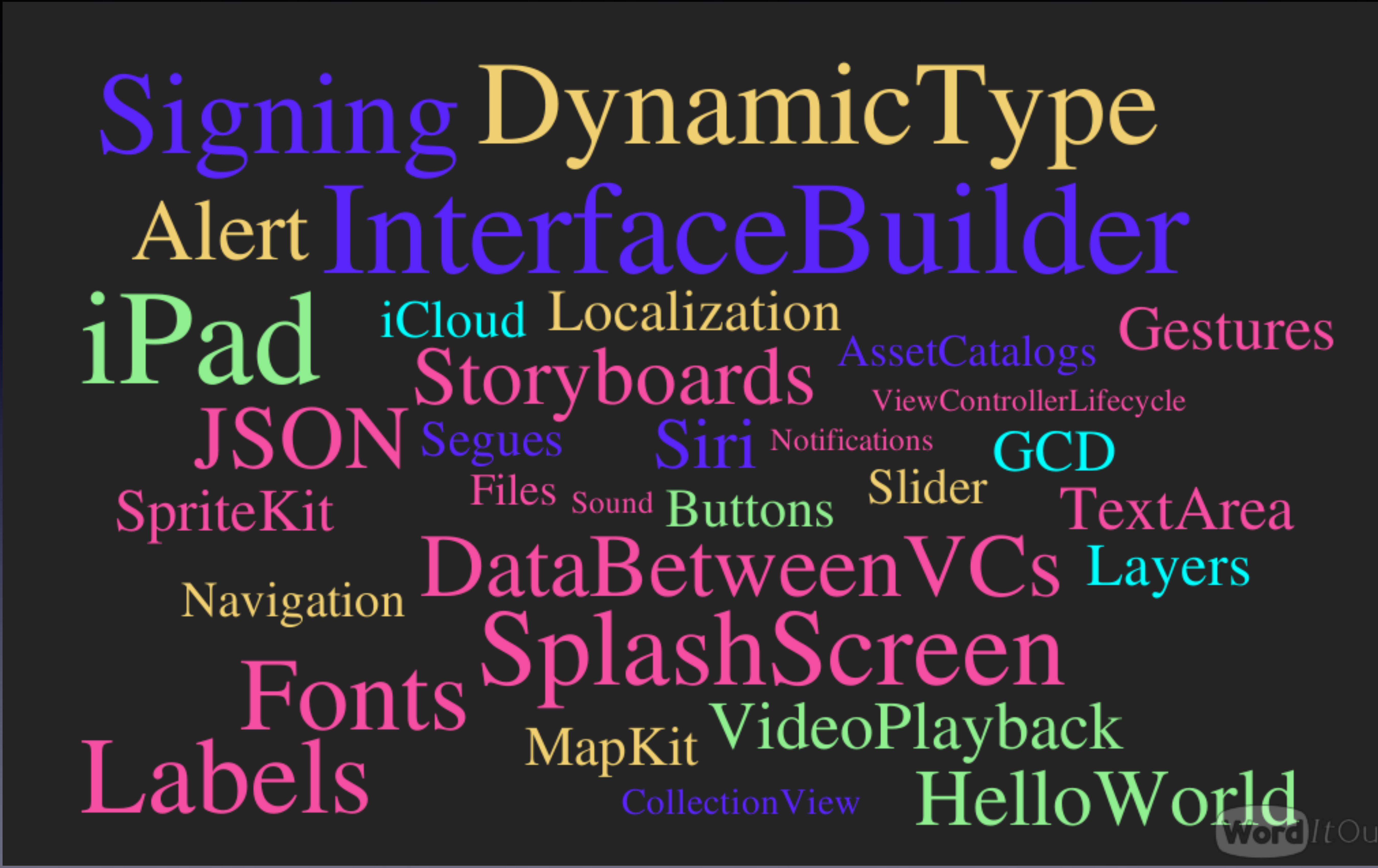
Today’s Schedule

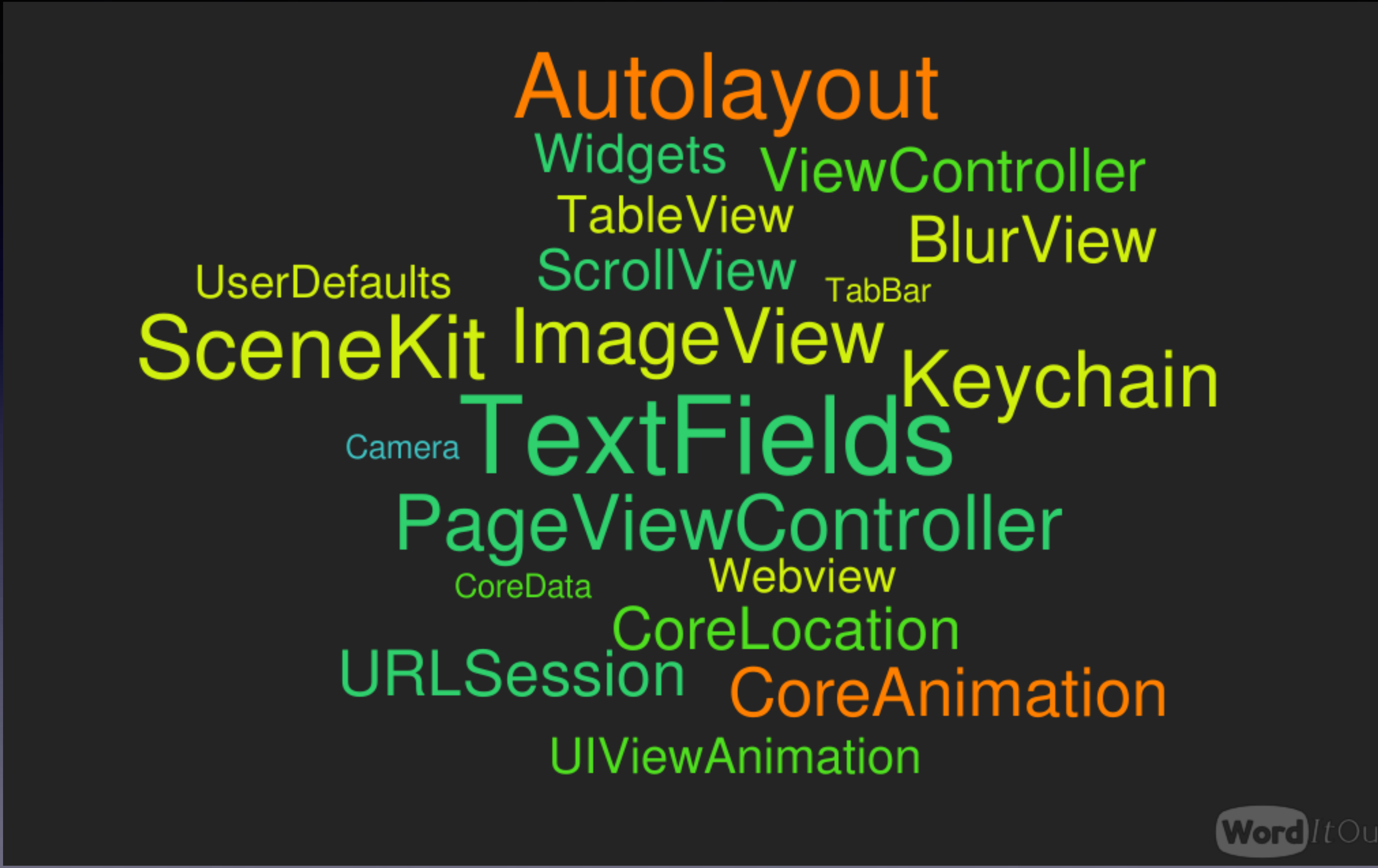
- Class Overview
- About me
- Syllabus
- About Xcode
- About Swift
- Hello, world!
- About the simulator and devices
- Ask Me Anything

Syllabus

- https://github.com/rmirabelli/UofD-Fall2017
- Quick summary:
  - No textbook
  - 4 projects
  - Office hours by appointment

About me


Signing
DynamicType
Alert
InterfaceBuilder
iPad
iCloud
Localization
Gestures
Storyboards
AssetCatalogs
ViewControllerLifecycle
JSON
Segues
Siri
Notifications
GCD
Files
Sound
Buttons
Slider
SpriteKit
TextArea
DataBetweenVCs
Layers
Navigation
SplashScreen
Fonts
VideoPlayback
MapKit
Labels
CollectionView
HelloWorld


Autolayout
Widgets
ViewController
TableView
BlurView
UserDefaults
ScrollView
TabBar
SceneKit
ImageView
Keychain
Camera
TextFields
PageViewController
CoreData
Webview
CoreLocation
URLSession
CoreAnimation
UIViewAnimation

Who has…

- An iPhone?
- Developed on Mac?
- Owns a Mac?
- Developed in *nix?
- Developed GUI

Swift 3

A modern language for iOS development

iOS Development Tools Overview

- Xcode
- Interface Builder
- Swift Playgrounds

Swift 1.0

- Swift was originally introduced at WWDC in 2014.
- First major new language in a very long time, first new language for Apple
- Not really ready for prime time

Swift 2.0

- Swift was significantly updated in 2015
- Compile time & run time improved
- Syntax updated

Swift 3.0

- Open source
- Language evolving
- Multiple elements cleaned up

Swift 4.0

- Coming in mid-September
- Very small changes

Why Swift Now?

- Swift brings features of multiple programming languages and paradigms
- Object oriented
- Functional Programming
- Increasingly important to Apple
- Most WWDC sessions are in Swift
- New OS features may be swift-only
- Swift is efficient to code
- Fun!

Let's Get Started

- Create a playground
- (File->New->Playground)

About Swift

- Modern language
- Based on Objective-C and the Objective-C runtime
- ARC
- Closures
- Collection literals
- Modules

Similarities with Objective C

- Cocoa / Foundation
- Named parameters
- Dynamic Object Model
- Swift can be mix-and-matched with Objective C

Differences from Objective C

- More friendly to new programmers
- Scripting-like language
- Not message-based
- Playgrounds

Hello, world!

```
print("Hello, world!")
```

An introduction to the parts of the playground

- Please notice: results to right

- Also, a console can be opened on the bottom to view print statements

- Other editors & functions available, including source control

More about that print statement

```
print("Hello, world!")
// Notice: no semicolon
// This should seem familiar
// Print statement INCLUDES newline
```

Constants and Variables

```
let myConstant = 42
var myVariable = 16
myVariable = 64
print("The meaning of life, the universe, and everything is \(myConstant)")
print("Will you still need me when I'm \(myVariable)")
```

Variable interpolation

- Notice that variables were placed in a string by enclosing in \ ()
- Format specifiers not required in Swift for most uses!

```
printf("Otherwise, I need to do this %d.", myVariable);
```

Variable & Constant Names

- Rule of thumb: camel case with initial lowercase letter
- Do not use snake case
- Do not use screaming case
- Do not use screaming snake case
- Use a long enough name for your code to be readable
- Variable names are for YOUR benefit, longer names don't cost extra

More on naming

- Emoji are possible-- be prepared to defend them
- Reserved keywords are possible-- be prepared to defend them to the death
- Generally treated as code smell

```
let 🍺 = "cheers"
let `class` = "class"
```

Constants

- Use 'let' to define a constant
- Prefer constants where possible for safety
- Start out with constants, switch to variable when needed
- You can't accidentally overwrite a constant!
- Compiler will warn you about variables that don't vary

Variables

- Use 'var' to define a variable
- You should know generally what a variable can do
- Variables will usually have a default value

Variable / Constant types

- The compiler will infer variable or constant type for you (implicit typing)

```
let myConstant = 42 // Int
let π = 3.14195 // Double
var response = "response" // String
```

Explicit typing

```
var myFloat: Float = 4.0
let explicitlyADouble: Double = 42
var wontWork: String = 13
```

Type Safety

- Objective C loves to live in the danger zone of "id" (void *)
- requires explicit casting, type checking at runtime, and risks instability
- Swift brings us back to the rational world of compile-time type checking
- Can be bypassed with "Any" type
- Experiment with this : make the "wontWork" example work in your playground

Base types

- Int : Int8,Int16, Int32, Int64, unsigned varieties
- Double
- Float
- String
- Boolean

Value Representations

```
Int.max, Int.min
0xff
0o32
0b1100110011
1_000_000
```

Quick Quiz

- In your playground, write an expression indicating why programmers can’t tell the difference between Christmas and Halloween

Quiz Answer

```
let 🎄 = 25
let 🎃 = 0o31

if (🎄 == 🎃) {
    print("It's all the same to me")
}
```

Playing around in the playground

- Now that we’ve covered the extreme basics, let’s do something vaguely fun in our playground

- Create a view at runtime

- See the view as your code progresses and changes its characteristics

Playground view

```swift
import PlaygroundSupport
let newView = UIView(frame: CGRect(x: 0, y: 0, width: 200, height: 200))
newView.backgroundColor = .red
newView.layer.cornerRadius = 20.0
newView.clipsToBounds = true
PlaygroundPage.current.liveView = newView
```

NOTICE

- import UIKit
- import PlaygroundSupport
- Even though view is constant (uses "let") it can be modified even though it can’t have a new value. This is an important distinction!!!
- See the view in the "assistant" view
- Turn on each individual line in the playground to see changes as they are applied

More with the view

```
let myLabel = UILabel(frame: CGRect(x: 0, y: 40, width: 100, height: 20))
myLabel.text = "howdy"
newView.addSubview(myLabel)
```

NOTICE

- Turn on per-line results in the playground
- This could be a great debugging tool for finding out exactly what in your code is ruining your view’s appearance

Short Break

- Let's take a short break.
- Ask any questions that you've been saving up.

Strings

- A string is a collection of characters
- Bridged with NSString
- All NSString operations are available
- Unicode at its core
- Strings are value types
- This means that a copy is passed as a parameter!
- This is different from Objective C
- Under the hood, the copy only ACTUALLY occurs when needed

Empty Strings

```
var emptyString  = ""
var emptyStringAgain = String()
if (emptyString == emptyStringAgain) {
    print("these are equal!")
}
```

String mutability

- Fundamentally, no such thing.
- If you want a constant, use let
- If you want a variable, use var

Walking through characters

- Strings are collections of characters

```
for character in "string".characters {
    print(character)
}
```

String concatenation

```
let firstString = "dog"
let secondString = "cow"
var concatenated = firstString + secondString
var spokenString = "moo"
spokenString += secondString
```

Other string operations

- What were those "NSString" operations?
- What does "NS" mean, anyhow?

```
let interestingString = "This is a chance to experiment."
print(interestingString.uppercased())
print(interestingString.replacingOccurrences(of: " ", with: "."))
```

Individual Characters

- are of type Type "Character"
- strings may have a character added to the end via “append”
- Please note that append modifies the string, and can only accept a single character

```
let period: Character = "."
var sentence = "Place it at the end of this"
sentence.append(period)
```

String Interpolation

- Replaces format strings

```
let insertedValue = 3
let message = "\(insertedValue) + itself = \(insertedValue +
insertedValue)"
```

Counting Characters

- You can’t directly get the length of the string
- A string is a collection of characters
- You can get the count of the collection
- This is changing in Swift 4

```
let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧,
Dromedary 🐪"
print("unusualMenagerie has \
(unusualMenagerie.characters.count) characters")
```

Why can't you directly count characters?

- Counting characters is fundamentally incorrect
- Usually this is done by counting bytes
- Swift is unicode, which means that characters may be (and often are) multibyte
- In Swift, you don't gain by knowing byte size of a string.

Comparing Strings

- Because strings are value types, comparing strings is based upon character equality, not address equality

```
let firstString = "hello"
let secondString = "hello"
if firstString == secondString {
    print("Yes, they are considered equal")
}
```

Comparing string instances

- If you REALLY need to know if two strings are the same object, use the '===' operator.

```
let firstString = "hello"
let secondString = "hello"
if firstString === secondString {
    print("Yes, they are exactly the same object")
} else {
    print("These are different objects")
}
```

Comparing Extended Grapheme Clusters

- Whoza what now?
- Strings are equivalent if their extended Grapheme Clusters are canonical equivalent: they must have the same linguistic meaning and appearance
- A Grapheme cluster is a sequence of unicode characters that are combined for a single value

Comparing Extended Grapheme Clusters

```
// "Voulez-vous un café?" using LATIN SMALL LETTER E WITH ACUTE
let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"

// "Voulez-vous un café?" using LATIN SMALL LETTER E and COMBINING ACUTE ACCENT
let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"

if eAcuteQuestion == combinedEAcuteQuestion {
    print("These two strings are considered equal")
}
```