SWIFT FIRST CUT

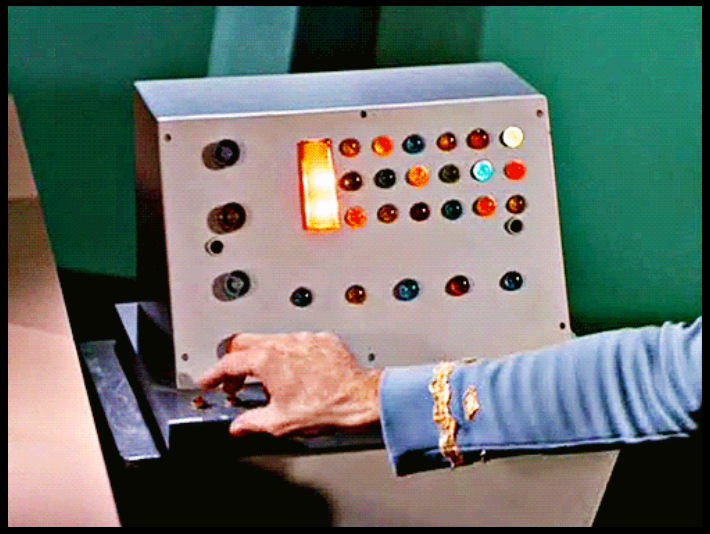
INTRODUCTION FOR OBJECTIVE-C DEVELOPERS

http://slides.com/tompiarulli/starting-swift/#/

Philly CocoaShops · January 2014

TRAINING CONTROLS

USE YOUR KEYBOARD OR MOUSE



← Go Back

Go Forward →

Access Bonus Content, Resources, Tips & Tricks ↓

Mouse Controls ↘

OH YEAH!

Down here is where you'll find extra content for if you're ahead of the game, or if you want to learn more later.

THANK YOU

WONDERFUL HOSTS

Tom Piarulli
Instructor
@tompark_io
tompark.io

Amit Rao
Teaching Assistant
@amitmrao

Kotaro Fujita
Teaching Assistant
@wild37
tomatoboy.co

Our Venue Sponsor
Indy Hall

Learn More about Indy Hall

INDY HALL

COWORKING SPACE

Indy Hall is a space where anyone can come to work and play. A community of folks who find that working together makes life more productive, interesting, and fun.

Memberships start at $20 per month.

Check it out →

EXPECTATIONS

For You:

- Latest Xcode Installed
- Project Files
- Slides Open
- Playground Open
- Basic Obj-C/iOS Knowledge

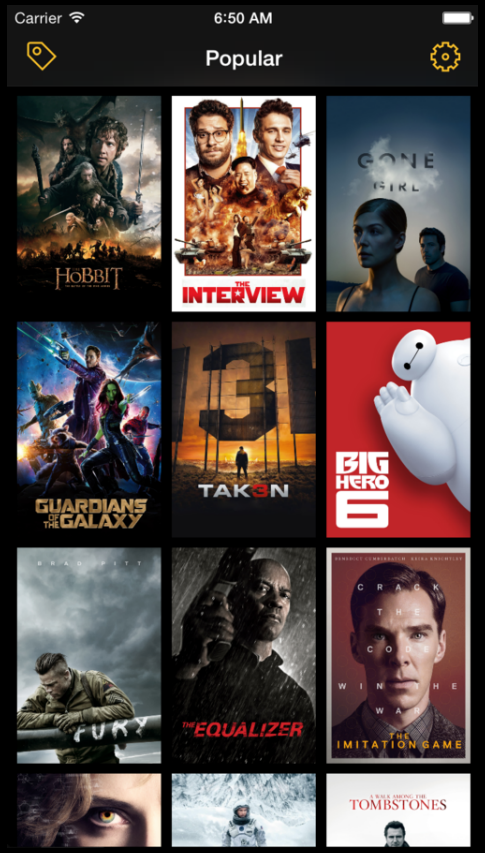
The Course:

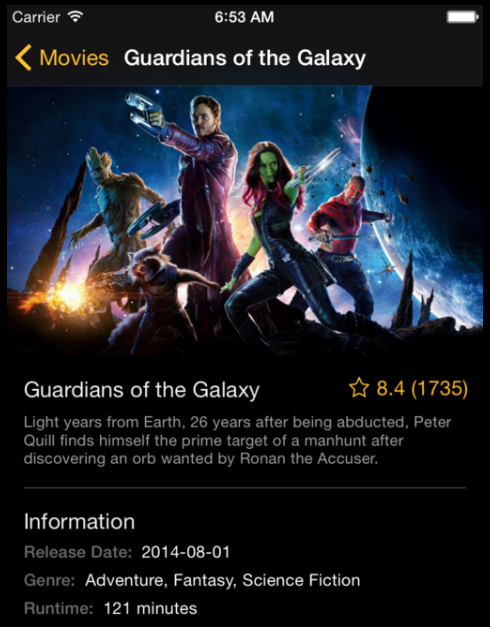
- Introductory
- There are no Swift experts
- Roundtable Encouraged (during breaks)

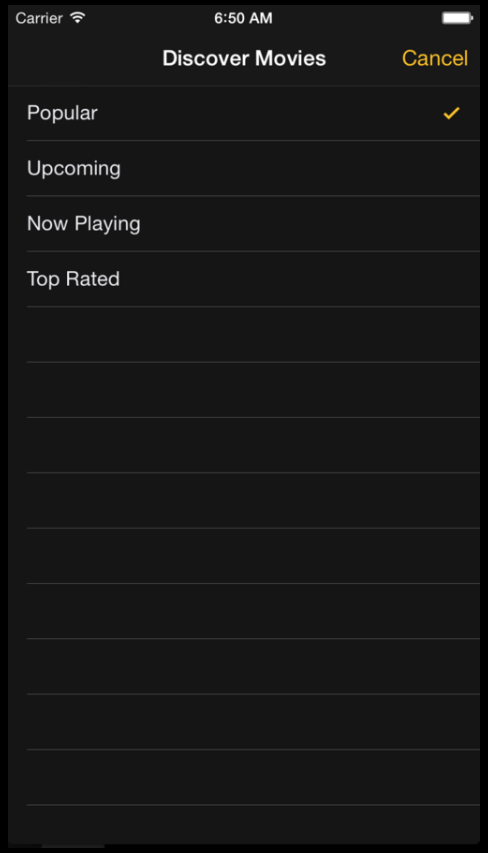
Download Project Files

WHAT WE'LL BE CREATING

SWIFT MOVIES - FUNCTIONAL, POLISHED, FUN







PRELUDE

YOUR TRAINING BEGINS

GETTING STARTED

VARIABLES, CONSTANTS, AND TYPES

```
// Playgrounds give us a chance to learn
// and experiment with Swift code

var myVariable = "If I can change and you can change..."

let myConstant = "You shall not pass!"

println(myConstant)

// No semicolons or @ prefix!
```

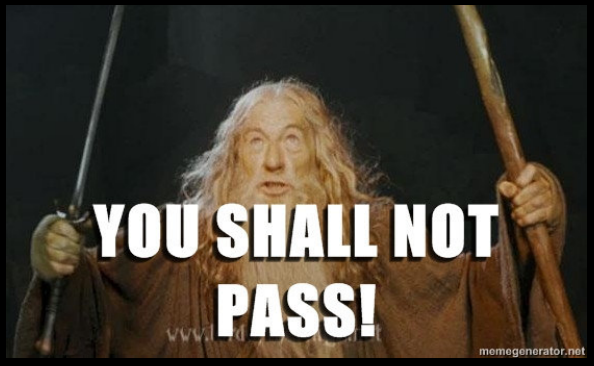
BONUS

Name the movie!

Bonus Answer

BONUS ANSWER

Rocky Balboa
Rocky IV (1985)



Gandalf
LOTR: Fellowship of the Ring (2001)

A WORD ON CONSTANTS

```
NSString *const APIUrl = @"api.mysite.com";
```

In Objective-C, constants were often used to avoid string literals or store macro app information (urls, versions, environments, etc.) whereas almost everywhere else we were using variables.

In Swift constants play a much larger role. In fact, you'll use them **more** than variables.

```
class Movie {
    let title: String
    let releaseYear: Int
    let rating: Double
    let voteCount: Int
    let cast: [Actor]
}
```

WHATS UP WITH THAT?

Swift is all about **code safety**.

The emphasis on safety can make Swift more frustrating than Obj-C at times, so its important to remember why.

Swift wants you to fail early and often so you can address bugs up front instead of in testing or *(*gasp*)* after they are discovered by a user.

By using constants wherever possible we mitigate one of the most common causes of bugs - values not being what we expect during execution.

TIP If you're unsure which to use, use a constant. You can always switch to a variable when its completely necessary. You gain the safety benefit and the compiler can better optimize the code.

TYPE INFERENCE

*“ **Type inference** refers to the automatic deduction of the data **type** of an expression in a programming language. If some, but not all, **type** annotations are already present it is referred to as **type** reconstruction. - Wikipedia*

ENGLISH

Swift automatically sets the type based on the value assigned to it.

BONUS

Name the movie!

Bonus Answer

↓

BONUS ANSWER

Jules

Pulp Fiction (1994)

COMMON TYPES

(THAT YOU'LL RECOGNIZE FROM OBJ-C)

```
let myString: String = "a string"

let myInteger: Int = 100

let myDouble: Double = 1.234

let myBool: Bool = true

let myArray: Array = ["string 1", "string 2"]
let myIntArray: [Int] = [1, 2]

let myDictionary: Dictionary = ["key":"value"]
let myIntDictionary: [Int:Int] = [1:2, 3:4]
```

SHOULD I MANUALLY DECLARE MY TYPES?

Not unless it adds needed clarity to the code (or the compiler can't infer the type). Type Inference is there to create cleaner, more concise code - take advantage of it where it makes sense!

STRING MANIPULATION

```
let string1 = "Game Over"
let string2 = " Man"
let combinedString = string1 + string2

var quote = "Game Over ".stringByAppendingString("Man")
quote += ", Game Over!"
println(quote.capitalizedString)

let interpolatedString = "\(combinedString), Game Over!"
```

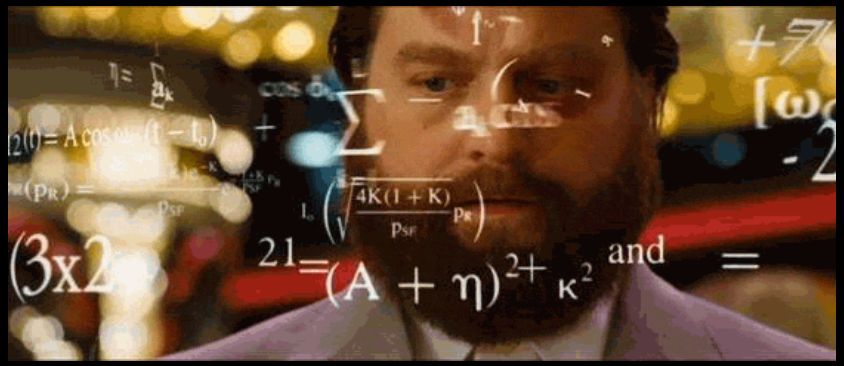
BONUS
Name the movie!

Bonus Answer

Private Hudson

Aliens (1986)

WORKING WITH NUMERICS




```
// Int & Double are default types
let myInt = 1
let myDouble = 1.234

// Underscores can indicate commas
let oneMillion = 1_000_000

// Conversions must be explicit
let result = myInt * myDouble // Error

let doubleResult = Double(myInt) * myDouble
let intResult = myInt * Int(myDouble)
```

```
+ // Addition
- // Subtraction
* // Multiplication
/ // Division
% // Remainder

// Add = for compound operation
myVariable += 10

// Unary Operators
++myVariable
--myVariable
myVariable++
myVariable--
```

BOOLEANS

```
let myBoolean = true

let myBoolean = YES // Error
let myBoolean = TRUE // Error

var myString = "testing"

myString == true // Error
if(myString) // Error
```

NOTE

Swift requires that you use the Bool type when you need boolean values. You can't integer values as was possible in Obj-C or perform if checks on variables/constants.

COLLECTIONS


```
let myArray = [1, 2, 3]

let myDictionary = ["key":"value", "key2":"value2"]
```

ARRAYS

```
var characters = ["Luke Skywalker", "Anakin Skywalker"]
characters[1] = "Darth Vader"

// Easy concatenate
let moreCharacters = ["C3PO", "R2D2"]
characters += moreCharacters

// Methods
characters.append("Han Solo")
characters.insert("Princess Leia", atIndex: 0)
characters.removeAtIndex(1)

characters.count
characters.isEmpty
characters.first
characters.last
```

NOTE

The Swift Array type can only contain a single type of object. NSArray is still available in Swift code, and can contain multiple object types.

DICTIONARIES

```
var professions = ["Boba Fett": "Bounty Hunter", "Han Solo": "Smuggler"]

// Access with key
professions["Han Solo"]

// Add with key
professions["Darth Vader"] = "Sith Lord"

// Methods
professions.removeValueForKey("Han Solo")
professions.updateValue("Boba Fett", forKey: "Sarlac Food")

professions.count
professions.isEmpty
```

NOTE

Similar to Array, Dictionary can't contain multiple different types of objects. However you could have one type as the key and another type as the value. NSDictionary is still available in Swift code.

CONTROL FLOW


```
for i in 1..5 {
   println("O'doyle Rules!")
}
```

BONUS
Name the movie!

Bonus Answer

BONUS ANSWER

The O'Doyles

Billy Madison (1995)

LOOPS

```
for i in 1...3 {
   println("O'doyle Rules")
}

var range = Range(start: 1, end: 4)

var i = 0
while i < 5 {
   println("O'doyle Rules")
   i++
}

let oceans = ["George Clooney", "Brad Pitt", "Matt Damon"]
for actor in oceans {
   println(actor)
   i++
}
```

NOTE

When using a *for-in* loop Swift defines a new constant for each iteration. So *i* and *actor* above are actually constants and would be so even if the array contained variables.

IF STATEMENTS & CONDITIONS

```
if enemyType == "Wraith" {
   println("Use Fire")
} else if enemyType == "Werewolf" {
   println("Use Silver")
} else {
   println("Nuke it from orbit")
}
```

```
// Comparison Operators
==    // equal
!=    // not equal
>     // greater than
<     // less than
>=    // greater or equal
<=    // less or equal

// Logical Operators
!     // Not (invert)
&&    // And
||    // Or
```

NOTE

When using if statements in Swift you must always include braces, even if the statement after is one line (safety remember?)

THE ALMIGHTY SWITCH

In Objective-C the use cases for a switch statement were relatively limited, mostly because a switch could only evaluate primitive types.

```
switch hero {
   case "Thor":
       println("Asguard")
   case "Iron Man", "Captain America":
       println("New York, NY")
   case "Hawkeye":
       println("Waverly, Iowa")
   default:
       println("Unknown")
}
```

In Swift a switch can take any type, and are much more flexible. As a result switches are much more powerful and more common in Swift.

NOTE

A switch in Swift must be exhaustive, meaning there must be a case to cover every possibility (safety). Additionally, execution stops when a case is matched (hence the lack of break statements).

PRELUDE COMPLETE!

BREAK & REVIEW TIME

Ahead of the Game?

RESOURCES

SWIFT NO FRILLS INTRODUCTION

learn-swift.co

My favorite Swift cheatsheet.

INTRODUCING SWIFT

developer.apple.com/swift

Apple's official Swift site. Includes a blog (will be useful as Swift changes) and a list of resources such as WWDC videos and sample code.

LEARN SWIFT FROM OBJECTIVE-C

http://codewithchris.com/learn-swift-from-objective-c/

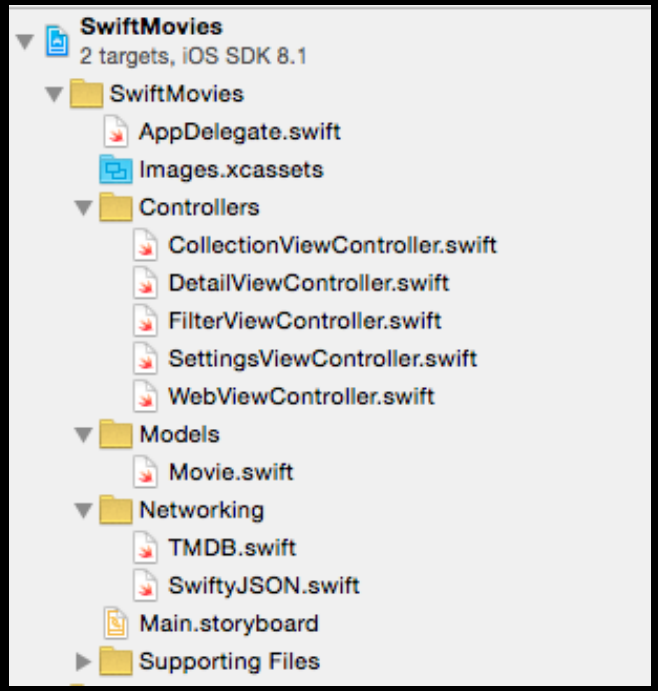
A series of brief tutorials with Objective-C and Swift side-by-side, so you can see the difference and take advantage of your existing knowledge.

ACT I

CLASSES, STRUCTS, EXTENSIONS, TUPLES

STARTING OUR PROJECT

STARTER PROJECT REVIEW



- Storyboard created with views
- Outlets connected to classes
- Images/Assets included

LOOK MA, NO PREFIXES

SWIFT NAMESPACING

In Objective-C we would prefix our classes with a few characters (TPViewController for example).

This was to prevent naming collisions. If we brought in third-party code that had the same class name as our own code for example we would have a problem.

Swift has automatic Namespacing based on target. This means if you bring in a third party framework you access its classes using FrameworkName.Class - this prevents them from conflicting with your own classes.

NO HEADERS EITHER!

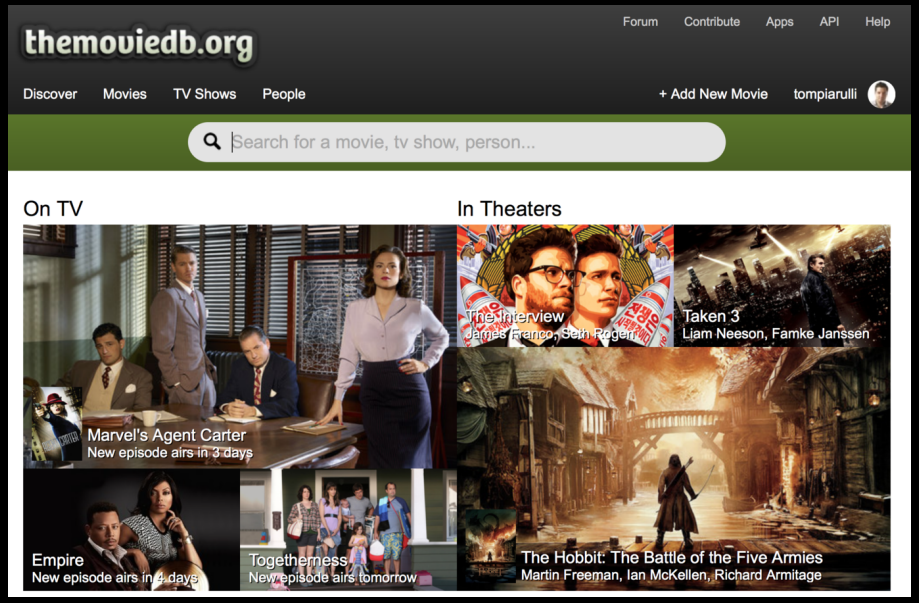
AUTOMATIC IMPORTS

In Objective-C we would have a .h file in which we would declare all of the public facing properties and methods of a class. We would then import those headers to gain access to those properties/methods.

In Swift the compiler handles the imports automatically, and we no longer have header files. Everything that you do not specify otherwise will be available in any file.

THE MOVIE DB (TMDB)

MOVIES API



View API Documentation

DATA MODELING

CREATING OUR MOVIE CLASS

```
final class Movie {
    let title: String
    let posterPath: String
    let averageRating: Double
    let voteCount: Int

    init(title: String, posterPath: String, averageRating: Double, voteCount: Int) {
        self.title = title
        self.posterPath = posterPath
        self.averageRating = averageRating
        self.voteCount = voteCount
    }
}
```

THE FINAL MODIFIER

If we place the word final before any class, method, or property in Swift we are telling the compiler that it cannot be overridden by a subclass.

NOTE

This isn't just to prevent overriding - marking a class as final will allow the compiler to perform additional optimizations on it - speedier code!

FUNCTIONS

init is a reserved word for a specific kind of function which creates an object. Other than the first word it has the same structure as any function in Swift:

```
func sum(number1: Int, number2: Int) -> Int {
   return number1 + number2
}

let operation:(Int, Int) -> Int = sum;
operation(10, 20)

let mySum = sum(10, 10)

func logSomething() {
   println("something")
}
```

STRUCTS

Our average rating and vote count are logically tied to one another. In a case like this it might make sense to use a struct.

```
struct Rating {
   let averageRating: Double
   let voteCount: Int
}
```

Structs are typically used for objects that are just meant to hold data.

Structs like classes:

- Can have properties
- Can have methods

Structs unlike classes:

- Cannot inherit from others
- Are pass by **value**
- Behave differently as constants

STRUCT VS CLASS

```
var aliensRating = Rating()
aliensRating.averageRating = 5.0

var referenceToAliensRating = aliensRating
aliensRating.averageRating = 9.5

aliensRating.averageRating              // 9.5
referenceToAliensRating.averageRating   // 5.0
// Values are copied

var variableRating = Rating()
let constantRating = Rating()

variableRating.averageRating = 9.0
constantRating.averageRating = 9.0 // -> Error: can't assign
constantRating = variableRating // -> Error
```

NOTE

The Swift Array and Dictionary types are actually structs, and therefore share the attributes above.

CONVENIENCE INITIALIZERS

```
convenience init(title: String, posterPath: String, averageRating: Double, voteCount: Int) {
    let rating = Rating(averageRating: averageRating, voteCount: voteCount)
    self.init(title: title, posterPath: posterPath, rating: rating)
}
```

You may be familiar with **convenience** and **designated** initializers in Obj-C.

Swift takes this concept further by formalizing it and actually enforcing it at compile time. If you don't call a designated initializer from a convenience initializer you will receive an error.

TIP

Notice how we never had to declare an initializer for our Rating struct. Swift creates initializers for structs automatically and includes the properties as parameters.

USING PLACEHOLDER DATA

CREATING OUR COLLECTION VIEW

```
final class CollectionViewController: UIViewController {
  @IBOutlet weak var collectionView: UICollectionView!
  @IBOutlet weak var activityIndicator: UIActivityIndicatorView!

  // We will later be fetching our movies from the web
  // so we went to use a var such that we can update our movies after load
  // We need to have a value, so we default to an empty array
  var movies: [Movie] = []

  func loadPlaceholderData() {
      let goneGirl = Movie(title: "Gone Girl", posterPath: "gonePoster", averageRating: 8.1, voteCount: 845)
      let guardians = Movie(title: "Guardians of the Galaxy", posterPath: "guardiansPoster", averageRating: 9.2, voteCount: 1026)
      let theHobbit = Movie(title: "The Hobbit", posterPath: "hobbitPoster", averageRating: 7.4, voteCount: 1343)
      let theInterview = Movie(title: "The Interview", posterPath: "guardiansPoster", averageRating: 6.3, voteCount: 824)

      self.movies = [goneGirl, guardians, theHobbit, theInterview]
  }
}
```

TIP

We can work with data locally like this to prototype our app before connecting to the API. There is also a great service for creating mock apis for testing: apiary.io

CLASS INHERITANCE

```
final class CollectionViewController: UIViewController {
  //...
  override func viewDidLoad() {
      super.viewDidLoad()
      self.loadPlaceholderData()
  }
}
```

To inherit from a class in Swift we follow the class name with a : then the parent class name.

When we want to override a method on the parent we use the override modifier prior to declaring the function. (don't forget to call super!)

EXTENSIONS

```
extension CollectionViewController: UICollectionViewDelegate, UICollectionViewDataSource {
    ...
}
```

Categories in Obj-C were typically used to extend Cocoa classes to add some extra functionality.

Extensions perform the same function in Swift, but also have almost all of the same capabilities of the main class declaration as well. As a result extensions can be used to better organize our own classes.

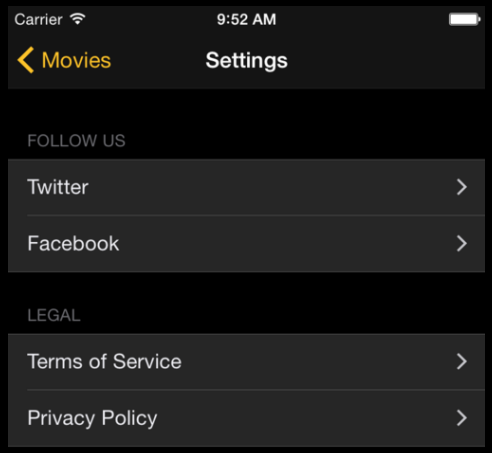
NOTE

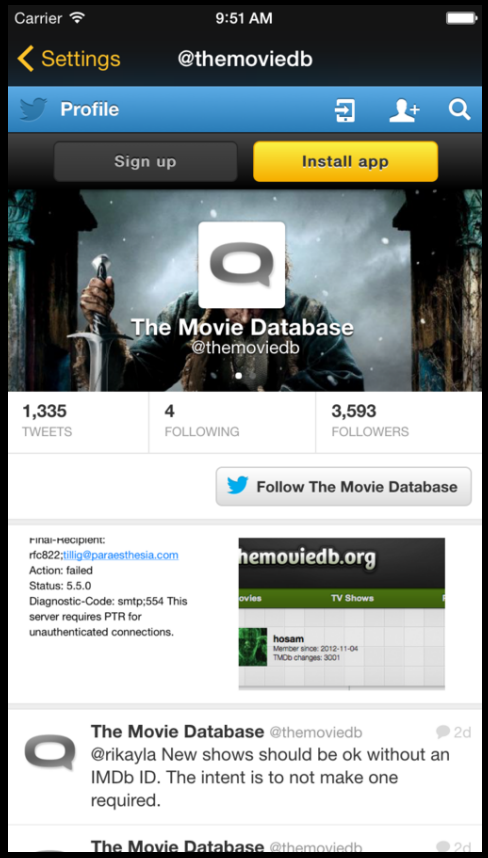
Extensions don't allow stored properties, but they do allow computed properties.

```
var computedProperty: Int {
    return self.myVar + 1
}
```

SWITCHES & TUPLES IN ACTION

CREATING OUR SETTINGS AREA





DELEGATES & PROTOCOLS

```
extension WebViewController: UIWebViewDelegate {
    func webViewDidStartLoad(webView: UIWebView) {
        self.activityIndicator.startAnimating()
    }

    func webViewDidFinishLoad(webView: UIWebView) {
        self.activityIndicator.stopAnimating()
    }
}
```

The delegate pattern was very common in Obj-C, and its the same with Swift. A protocol allows us to require a class to have certain methods and properties (by adopting the protocol).

Since we know a class adopting a protocol will have certain properties and methods we can create a variable to hold an instance of the class that we can send messages to - this is a delegate.

In Swift we adopt a protocol just like we inherit from a class - following the class name and ":". We just need to make sure we also set the delegate.

TUPLES


```
let home = (221, "Baker Street")
println(home.0)
println(home.1)

let (number, street) = home
println(number)

let address = (number: 221, street: "Baker Street")
println(address.street)

let three: (Int, Int, Int) = (1,2,3)
```

Why Tuples?

WHY TUPLES?

We can achieve similar functionality to tuples by using arrays, dictionaries, classes, or structs. So why use them?

Vs Dictionaries/Arrays:

- Provides named values
- Easier means of representing data (in some cases)

Vs Classes/Structs:

- Don't need to explicitly define the type
- Includes index notation

```
array[0]
tuple.0

array.myProperty // Nope
dictionary.myProperty // Nope

tuple.myProperty
```

```
tuple.0
myClass.0 // Nope

class Address {
   let number: Int
   let street: String
   init...
}

let home = Address.init(...)
let home = (number: 221, street:"Baker Street")
```

AN EXAMPLE

A CLEAN STATIC (OR SEMI-STATIC) TABLE

```
let path = (indexPath.section, indexPath.row)

switch path {
case (0, 0):
   cell.textLabel?.text = "Bug Report"
case (0, 1):
   cell.textLabel?.text = "Contact Us"
case (0, 2):
   cell.textLabel?.text = "Twitter"
case (1, 0):
   cell.textLabel?.text = "Terms of Service"
case (1, 1):
   cell.textLabel?.text = "Privacy Policy"
default:
   cell.textLabel?.text = "Error!"
}
```

NOTE

Tuples, like structs, are a value type - meaning that the values are copied when they're assigned rather than passing a reference.

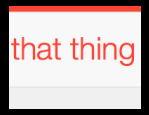
ACT I COMPLETE!

BREAK & REVIEW TIME

Ahead of the Game?

↓

RESOURCES



THAT THING IN SWIFT

thatthinginswift.com

Examples of common Objective-C patterns translated into Swift.

SWIFT DEVELOPER WEEKLY

http://swiftdevweekly.co

A weekly email newsletter that contains interesting curated links about Swift development.

SWIFT TIPS

https://github.com/jbrennan/swift-tips

A Github repo containing a list of useful Swift tips. If you think it can be improved you can make a pull request!

ACT II

OPTIONALS, ENUMS, PROTOCOLS

SEGUES & EMPTY STATES

CREATING OUR DETAIL VIEW

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showMovieView" {
        let movieController = segue.destinationViewController as DetailViewController
        let indexPath = self.collectionView.indexPathsForSelectedItems().first as NSIndexPath?
        if let selectedPath = indexPath {
            let movie = self.movies[selectedPath.row]
            movieController.movie = movie
        }
    }
}
```

```
private func loadMovieInfo() {
    // Setup our empty state (how the screen should look while loading)
    let ellipsis = "..."
    self.backdropImageView.image = nil
    self.titleLabel.text = ellipsis
    self.navigationItem.title = ellipsis
    self.ratingLabel.text = ellipsis
    self.summaryLabel.text = ellipsis
    self.releaseDateLabel.text = ellipsis
    self.genresLabel.text = ellipsis
    self.runtimeLabel.text = ellipsis

    if let movie = self.movie {
        self.titleLabel.text = movie.title
        self.navigationItem.title = movie.title
    }
}
```

OPTIONALS

In Objective-C you always have the option of using *nil* instead of the actual object. As a result you need to perform proper *nil* checking and handling or risk bugs/crashing.

As you've likely noticed variables and constants in Swift by default cannot be *nil*. This is another of Swift's safety features - if you are using the standard types you never have to worry about issues from *nil* values.

```
var string: String = nil // Error
```

So what about when you can't declare a value upfront? Like if you're fetching data from an API for example? We use an optional.

```
var string: String? = nil // Okay
```

DECLARING OPTIONALS

Declaring an optional is as simple as adding a ? to the end of any type.

```
var string: String? = "Optional" // Value wrapped in optional {Some:"Optional"}
```

We can continue to assign to it just like before.

```
string = "Changed!"
```

But what happens when we try to call methods on it?

```
string.uppercaseString // Error: String? doesn't have method...
```

How do we get the *String* back out of the optional so we can work with it?

UNWRAPPING OPTIONALS

The "safe" way:

```
if let unwrappedString = string {
   unwrappedString.uppercaseString
}
```

The "unsafe" ways (forced and implicit unwrapping):

```
string!.uppercaseString //Forced unwrap
```

```
var string: String! //Implicit unwrap
```

NOTE

Forced unwrapping can be useful (or even necessary) at times but it entirely strips the safety provided by optionals and should be used with caution. Especially with implicitly unwrapped optionals as they look just like other variables!

OPTIONAL CHAINING

Optional chaining provides us a more concise way of working with optionals without the need for an if/let block every time.

```
var optionalString: String? = "Big gulps huh?"

let newString = optionalString?.uppercaseString
```

The *?* at the end of *optionalString* triggers the optional chaining, which will check the contents of the optional and only execute *uppercaseString* if it is not nil.

NOTE

If an optional in an optional chain is *nil* then the expression will return *nil*. This means newString above is actually an optional also, because its possible for this expression to return *nil* (if *optionalString* turned out to be *nil*).

ENUMS

STARTING OUR TMDB STORE

```
enum MovieList {
   case Popular
   case Upcoming
   case NowPlaying
   case TopRated

   static let allLists = [Popular, Upcoming, NowPlaying, TopRated]

   var queryPath: String {
       switch self {
       case .Popular:
           return "popular"
       case .Upcoming:
           return "upcoming"
       case .NowPlaying:
           return "now_playing"
       case .TopRated:
           return "top_rated"
       }
   }

   var listName: String {
       switch self {
       case .Popular:
           return "Popular"
       case .Upcoming:
           return "Upcoming"
       case .NowPlaying:
           return "Now Playing"
       case .TopRated:
           return "Top Rated"
       }
   }
}
```

PROTOCOLS (CONT.)

OUR FIRST PROTOCOL

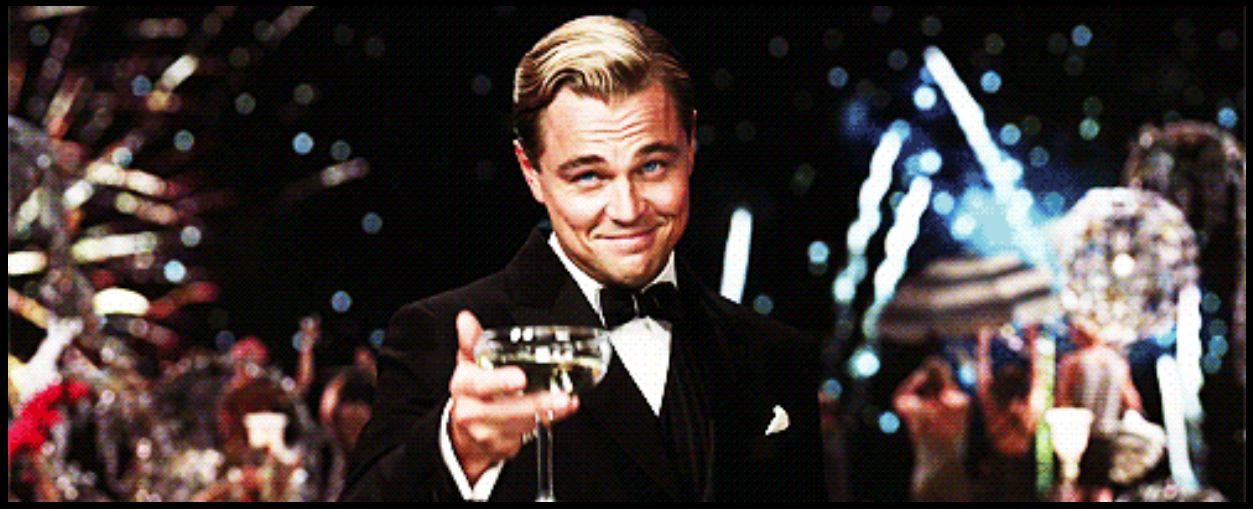
```
protocol FilterViewDelegate {
   var activeList: TMDB.MovieList { get }
   func filterDidChange(list: TMDB.MovieList)
}
```

```
var delegate: FilterViewDelegate?
```

```
extension CollectionViewController: FilterViewDelegate {
   func filterDidChange(list: TMDB.MovieList) {
       self.activeList = list
       self.navigationItem.title = list.listName
       self.collectionView.setContentOffset(CGPoint(x: 0, y:-self.collectionView.contentInset.top), animated: false)
       self.refreshMovieResults()
   }
}
```

ACT II COMPLETE!

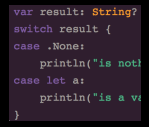
BREAK & REVIEW TIME



Ahead of the Game?

↓

RESOURCES



SWIFT OPTIONALS MADE SIMPLE

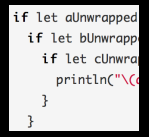
appventure.me/2014/06/13/swift-optionals-made-simple/

A common-sense explanation of optionals in Swift.

UNDERSTANDING OPTIONAL CHAINING

http://nomothetis.svbtle.com/understanding-optional-chaining

A helpful article for understanding the various methods for unwrapping optionals, and the mechanics involved.



TEARING DOWN SWIFT'S OPTIONAL PYRAMID OF DOOM

http://www.scottlogic.com/blog/2014/12/08/swift-optional-pyramids-of-doom.html

When working with optionals in Swift you'll often come across situations where you need to unwrap multiple values prior to performing some logic. This post offers a potential solution.

ACT III

NETWORKING, CLOSURES, GENERICS

FETCHING FROM THE NET

CONTINUING OUR TMDB STORE

```
class func fetchMovieList(list: MovieList, completion: MoviesCompletion) {
   let urlString = "\(TMDB_API_BASE_URL)/movie/\(list.queryPath)?api_key=\(TMDB_API_KEY)"
   let request = NSURLRequest(URL: NSURL(string: urlString)!)

   NSURLConnection.sendAsynchronousRequest(request, queue: NSOperationQueue.mainQueue()) {
       (response:NSURLResponse!, data:NSData!, error:NSError!) -> Void in
       let json = JSON(data: data)

       if let results = json["results"].array {
           var moviesArray: [Movie] = []

           for jsonMovie in results {
               if let movieResponse = MovieResponse(json: jsonMovie) {
                   let movie = Movie(TMDBResponse: movieResponse)
                   moviesArray.append(movie)
               }
           }

           completion(result: Result.Success(moviesArray))
           return
       }

       completion(result: Result.Failure)
   }
}
```

A VERY SIMPLE GENERIC

SUPER SWEET ERROR HANDLING

```
enum Result<T> {
   case Success(T)
   case Failure
}
```

```
switch result {
    case .Success(let results):
       self.movies = results
       self.collectionView.reloadData()
    case .Failure:
       // Handle error
    }
}
```

CLOSURES

LIKE BLOCKS BUT WAY LESS ANNOYING

```
// Known arguments (common use case)
func applyMultiplication(value: Int, multFunction: Int -> Int) -> Int {
   return multFunction(value)
}

applyMultiplication(2, {value in
   value * 3
})

// Using shorthand we reference by position
applyMultiplication(2, {$0 * 3})

// When last parameter we can omit )
applyMultiplication(2) {$0 * 3}
```

TYPEALIAS

JUST LIKE IT SOUNDS

```
typealias MoviesCompletion = (result: Result<[Movie]>) -> Void
typealias MovieCompletion = (result: Result<Movie>) -> Void
```

Provides a custom shorthand for a given type -
great for closures and other lengthy types.

THE JSON NIGHTMARE

OPTIONALS GONE INSANE

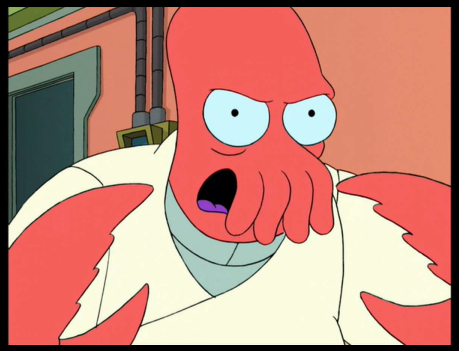
```
let jsonObject : AnyObject! = NSJSONSerialization.JSONObjectWithData(dataFromTwitter, optic
if let statusesArray = jsonObject as? NSArray{
   if let aStatus = statusesArray[0] as? NSDictionary{
       if let user = aStatus["user"] as? NSDictionary{
           if let userName = user["name"] as? NSDictionary{
               //Finally We Got The Name

           }
       }
   }
}
```

```
let jsonObject : AnyObject! = NSJSONSerialization.JSONObjectWithData(dataFromTwitter, options: NSJSONReadingOpti
if let userName = (((jsonObject as? NSArray)?[0] as? NSDictionary)?["user"] as? NSDictionary)?["name"]{
 //What A disaster above
}
```

SWIFTYJSON TO THE RESCUE

```
let json = JSON(data: dataFromNetworking)
if let userName = json[0]["user"]["name"].string{
 //Now you got your value
}
```

A third-party library?



Why? There are a number of ways to deal with the JSON issue - but pretty much all of them involve concepts slightly above the level of an intro course.

FAILABLE INITIALIZERS

INSULATING OUR API STORE

```
init?(json: JSON) {
    let jsonId          = json["id"]
    let jsonTitle       = json["title"]

    if let id = jsonId.int {
        self.tmdbId = id
    } else {
        return nil
    }

    if let title = jsonTitle.string {
        self.title = title
    } else {
        return nil
    }
}
```

FETCHING IMAGES

```
private func loadImage(size: String, path: String, completion: ImageCompletion) {
    let url = NSURL(string: "\(TMDB_IMAGE_BASE_URL)/\(size)/\(path)")
    // Another option:
    // let url = NSURL(string: TMDB_IMAGE_BASE_URL + "/" + size + "/" + path")
    if url == nil {
        completion(result: Result.Failure)
        return
    }

    // This method when passed the Main Queue will execute the completion closure on the Main Thread
    // Otherwise we would want to get the Main Thread before executing our completion closure
    // As we are going to be doing our interfaced updates in the completion
    NSURLConnection.sendAsynchronousRequest(NSURLRequest(URL: url!), queue: NSOperationQueue.mainQueue()) {
        (response: NSURLResponse!, data: NSData!, error: NSError!) -> Void in

        if data != nil {
            if let image = UIImage(data: data) {
                completion(result: Result.Success(image))
                return
            }
        }

        // If we make it here something went wrong
        completion(result: Result.Failure)
    }
}
```

HANDLING MISSING DATA

A GRACEFUL VIEW

```
private func updateInterface() {
        let na = "Not Available"

        self.titleLabel.text = self.movie?.title ?? na
        self.ratingLabel.text = self.ratingString(self.movie?.rating) ?? "N/A"
        self.summaryLabel.text = self.movie?.summary ?? "No Synopsis Available."
        self.releaseDateLabel.text = self.movie?.releaseDate ?? na
        self.genresLabel.text = self.genresString(self.movie?.genres) ?? na
        self.runtimeLabel.text = self.minutesString(self.movie?.runtime) ?? na

        self.loadBackdropImage()
}
```

WRAPPING IT UP

ALMOST THERE!

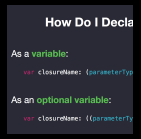
ACT III COMPLETE!

BREAK & REVIEW TIME

Haven't had enough?

↓

RESOURCES



HOW DO I DECLARE A CLOSURE IN SWIFT?

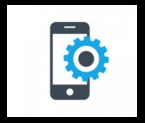
http://goshdarnclosuresyntax.com

A handy reference for the closure syntax in Swift.

NICE WEB SERVICES - SWIFT EDITION

http://commandshift.co.uk/blog/2014/12/28/nice-web-services-swift-edition/

An article that describes an interesting approach to handling web services in Swift. Portions of the networking in the Swift Movies app are based on this approach.



SIMPLE JSON PARSING WITH SWIFT

http://www.atimi.com/simple-json-parsing-swift-2/

You've seen the issues that arise with parsing JSON in Swift and how a library like SwiftyJSON can help solve them. If you want to know to create your own JSON parser, this is a good place to start.

EPILOGUE

WHERE TO GO FROM HERE

Post Comments/Feedback on the Slides

Ask Questions/Start Discussions on the Meetup Group

Check out this list of Swift resources I've gathered for you