iOS Dev Accelerator
Week 1 Day 1

- Intro to Course
- MVC & Single Responsibility
- Type Methods
- JSON
- Bundles
- TableView/Cells Review
- Inheritance

Course Schedule

	Monday	Tuesday	Wednesday	Thursday	Friday
Morning	Previous Week's HW Final Discussion	Lab Time	Lab Time	Lab Time	Lecture Job Hunt/Guest Speakers
Afternoon	Lecture	Lecture	Lecture	Lecture	Lab Time

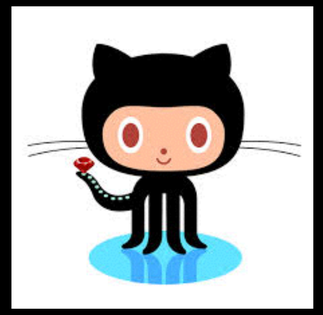
Course Work

- Each week we create a separate standalone app. All the lectures and homework are based around each app's features.
- There is new homework every day, but you will submit your homework only once at the end of each week, by Sunday morning at 9am.
- You will be a given a score based on how many of the required features you got implemented, if your app builds, are you following best practices, etc

Weeks 1–4 Outline

Week 1 - Twitter Clone

Week 2 - Photo Filtering



Week 3 - Github Client

Week 4 - First Project Week

Weeks 5–8 Outline

Week 5 - Objective-C Location Aware Reminders

Week 6 - Core Data Hotel Management

Week 7 - Objective-C StackOverflow Client

Week 8 - 2nd Project Week

Passing this class

- 3 things you must do to pass the class
    1. Complete all homework assignments with a final average of over 90%
    2. Pass a final whiteboard exam (you get more than one try)
    3. At least one app submitted to the app store, but hopefully two

class chat: codefellows.slack.com
(#sea-d34-ios channel)

class repo: https://github.com/codefellows/sea-d34-iOS

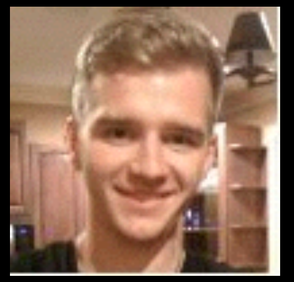
Door Codes
Main Door: *9821
Garage: 3142
The East: 1337
1st Floor: 4242
2nd Floor: 9001
The easy: 101010

Tim Hise, Nordstrom

Michael Babiy, Getty Images

Chris Meehan, HCL Technologies

Rich Lichkus, Reveal



Ivan Lesko, General UI

Lauren Lee, Urban Spoon

Anton Rivera, Big Fish Games

Andrea Silkey, Disney

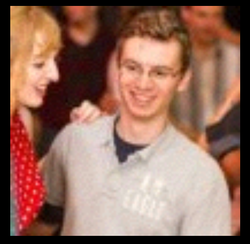
Brian Radebaugh, Nordstrom

Chris Cohan, Eigital

Steven Stevenson, Belief

Dan Fairbanks, Fresh Consulting

Zuri Biringer, Siren



Andrew Rodgers, L4 Mobile

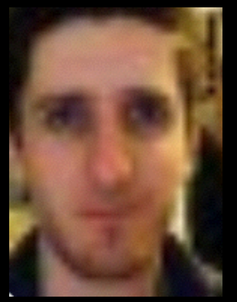
Reed Sweeney, LIFFFT

Matt Remick, Nordstrom

Spencer Fornaciari, Blank Check Labs

Ryo Tulman, Muegello

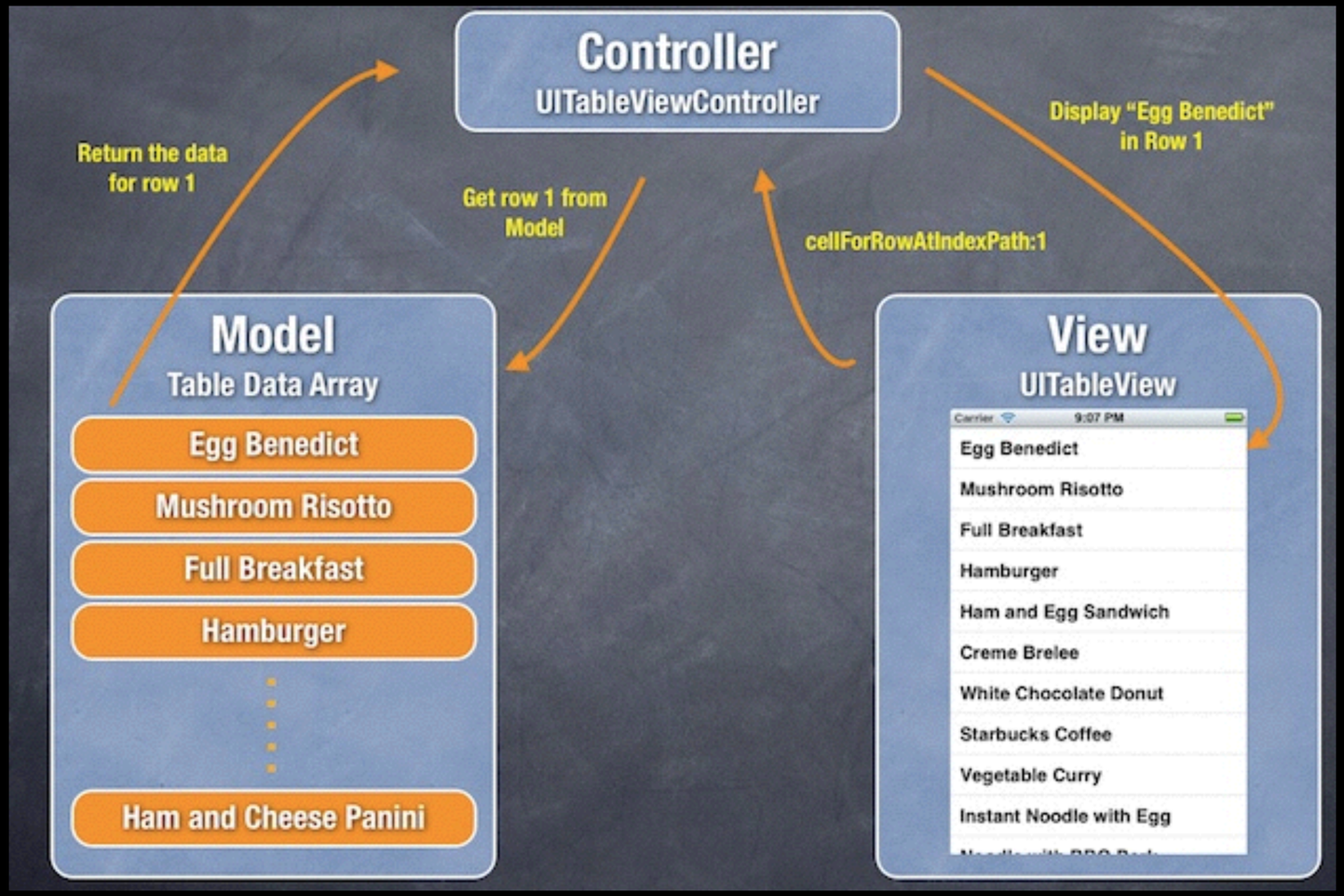
Christian Hansen, National Center of Telehealth



Jeff Schwab, Felt

Pimp your Xcode

Lectures



MVC
(Model-View-Controller)

MVC Facts

- Introduced in the 70's with the Smalltalk programming language.
- Didn't become a popular concept until the late 80's
- The MVC pattern has spawned many evolutions of itself, like MVVM (Model-View-ViewModel)
- MVC is very popular with web design and applications. It's not just for mobile or desktop.

So what is MVC?

- MVC is simply the separation of **M**odel, **V**iew, and **C**ontroller.
- It is a separation of concerns for your code. Being able to separate out these components makes your code easier to read, re-use, test, think about, and discuss.
- The **Model layer** is the data of your app, the **View layer** is anything the user sees or interacts with, and the **Controller layer** mediates between the two.

The differing roles

- The model captures the behavior of the application in terms of its problem domain. This is independent of the interface. The model manages the data, logic, and rules of the app.
- The view layer is any output that is presented to the user in some form of an interface (buttons, screens, colors)
- The controller accepts user actions from the view layer and updates the model layer, or is notified of changes by the model layer and then updates the view layer.

The MVC layout of our Week 1 App

Model Layer

Tweet

User

TweetJSON Parser

UserJSON Parser

TwitterService

makes network calls

Controller Layer

HomeTimeLine ViewController

Tweet ViewController

UserTimeLine ViewController

View Layer

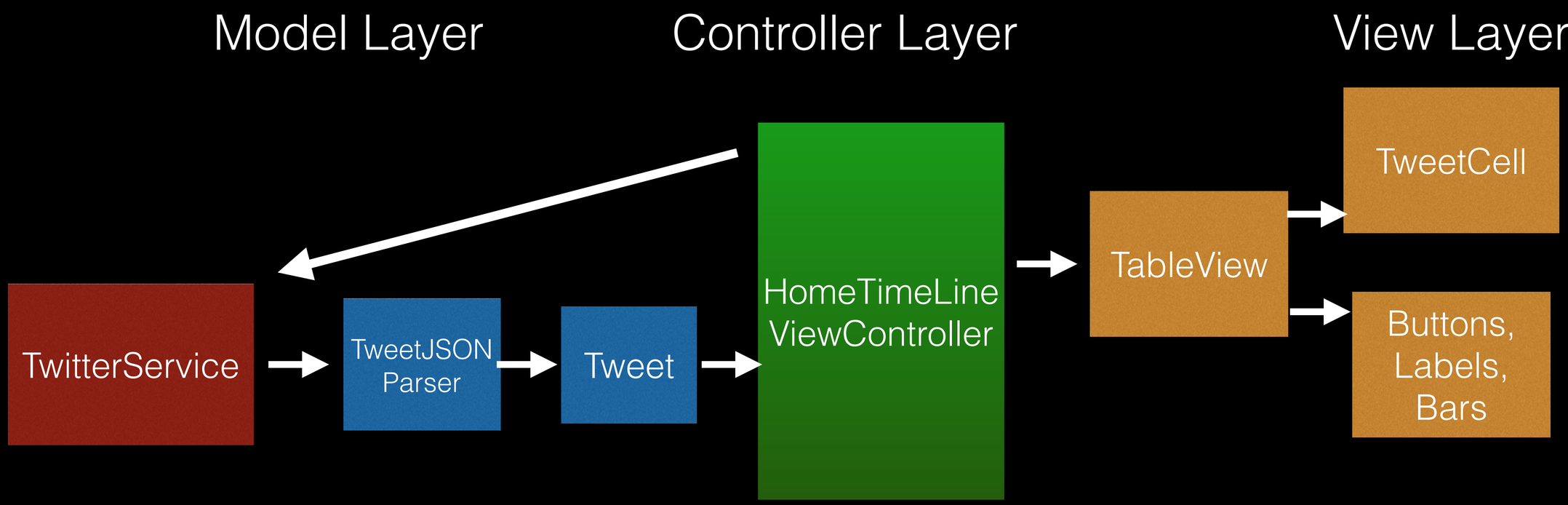
Storyboard

TweetCell.xib

TweetCell

Buttons, Labels, Bars

One Scene's MVC



Demo

Single Responsibility

- The Single Responsibility principle is this:
  - **A class should only have one reason to change**
- What this means is that each class you design should really only have one responsibility, which means it will only need to change if the details of that one responsibility changes.
- For example, our Tweet model class in our Twitter app should only need to change if we need to change what we display to the user
- Or our TweetJSONParser class only needs to change if the JSON that is retrieved from the web service is changed.

Single Responsibility Benefits

- Keeps your classes focused and concise. SRP can take a view controller down from 1000 lines to 100 lines.
- Makes it easier to Test. We will cover BDD and TDD later in the course, but designing your classes with SRP in mind will make writing tests much easier!
- If a class has more than one responsibility, that increases that chance that more than one developer/team will be working on the same class at the same time, which can introduce problems into your code base that are not easily fixed.

Type Methods

Type Methods

- A Type method is a method that operates on types rather than instances of the type.
- Type methods in Swift are just like Class methods in Objective-C.
- Type methods can be defined for classes, structs, and enums.
- Type methods are very commonly used as ‘convenience allocators’, often referred to as factory methods.
- Anytime a method does not need any ‘state’ (aka properties) to execute, making it a class method is probably a good idea.
- We will use type methods to create Tweet objects from JSON data.

Type Methods

- In Swift, type methods on a class are defined with the class keyword:

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

As you can see, type methods are called with dot syntax, just like regular instance methods

Demo

JSON

JSON

- “JavaScript Object Notation” (but it really is language independent)
- “open standard format that uses human readable **text to transmit data objects** consisting of attribute-value pairs”
- Used primarily for communication between the server and client
- Is a more popular alternative to XML
- The official internet media type of JSON is application/json (more on what this is later in the course)

JSON data types

- **Number** : Decimal number that makes no distinction between an integer and float
- **String** : A sequence of zero or more unicode characters. Delimited with double quotation marks, and escaped with a backslash
- **Boolean** : True or false vales
- **Array**: An ordered list of zero or more values, can be of any type. Array's use [] square bracket notation with elements separated by a comma.
- **Object** aka **dictionary** : Unordered associative collection. Use {} curly bracket notation with pairs separated by a comma. Within each pair separation is established with a : colon. All keys must be strings and must be unique within that object.
- **null** : empty value.

JSON Example

```
{ "users":[
        {
            "firstName":"Ray",
            "lastName":"Villalobos",
            "joined": {
                "month":"January",
                "day":12,
                "year":2012
            }
        },
        {
            "firstName":"John",
            "lastName":"Jones",
            "joined": {
                "month":"April",
                "day":28,
                "year":2010
            }
        }
    ]}
```

the root data structure of this JSON is a dictionary denoted by the { bracket

first item in the dictionary is an array that is paired with the key “users”. arrays are denoted by the square brackets []

the array contains dictionaries separated by a comma

end of array and end of root dictionary

JSON is everywhere!

Facebook API JSON Response


```
{
   "data": [
      {
         "id": "X999_Y999",
         "from": {
            "name": "Tom Brady", "id": "X12"
         },
         "message": "Looking forward to 2010!",
         "actions": [
            {
               "name": "Comment",
               "link": "http://www.facebook.com/X999/posts/Y999"
            },
            {
               "name": "Like",
               "link": "http://www.facebook.com/X999/posts/Y999"
            }
         ],
         "type": "status",
         "created_time": "2010-08-02T21:27:44+0000",
         "updated_time": "2010-08-02T21:27:44+0000"
      },
```

JSON is everywhere!

Instagram API JSON Response


```
{
    "data": [{
        "distance": 41.741369194629698,
        "type": "image",
        "users_in_photo": [],
        "filter": "Earlybird",
        "tags": [],
        "comments": { ... },
        "caption": null,
        "likes": { ... },
        "link": "http://instagr.am/p/BQEEq/",
        "user": {
            "username": "mahaface",
            "profile_picture":
"http://distillery.s3.amazonaws.com/profiles/profile_1329896_75sq_1294131373.jpg",
            "id": "1329896"
        },
        "created_time": "1296251679",
        "images": {
            "low_resolution": {
                "url":
"http://distillery.s3.amazonaws.com/media/2011/01/28/0cc4f24f25654b1c8d655835c58b850a_6.jpg",
                "width": 306,
                "height": 306
            },
```

JSON is everywhere!

Github API JSON Response


```
[
  {
    "id": 1296269,
    "owner": {
      "login": "octocat",
      "id": 1,
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "",
      "url": "https://api.github.com/users/octocat",
      "html_url": "https://github.com/octocat",
      "followers_url": "https://api.github.com/users/octocat/followers",
      "following_url": "https://api.github.com/users/octocat/following{/other_user}",
      "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
      "organizations_url": "https://api.github.com/users/octocat/orgs",
      "repos_url": "https://api.github.com/users/octocat/repos",
      "events_url": "https://api.github.com/users/octocat/events{/privacy}",
      "received_events_url": "https://api.github.com/users/octocat/received_events",
      "type": "User",
      "site_admin": false
    },
    "name": "Hello-World",
    "full_name": "octocat/Hello-World",
    "description": "This your first repo!",
```

JSON workflow in your app

1. Make a network call and receive a response back form the server
2. Serialize the JSON data contained in the response into objects (dictionaries, arrays, strings,etc)
3. Parse through the serialized JSON objects and create your model objects

Network/JSON workflow in your app

2 The server receives your request, processes it, and sends response back

1 Some event happens in your app (launch, button press) and a network request is made

3 Your app receives the response, checks for errors, and then serializes the JSON ‘data payload’ of the response into objects

4 Your app parses through the objects and creates model objects from the JSON objects

5 The newly created model objects are shown to the user via the view layer

Today’s JSON workflow in your app

1

Load the local JSON from our Bundle

2

serialize the JSON ‘data payload’ of the response into objects

3

Your app parses through the objects and creates model objects from the JSON objects

4

The newly created model objects are shown to the user via the view layer

JSON Parsing – interpreting JSON in your code

- You may eventually use third party frameworks to simplify your JSON parsing, but first you need to understand how to write your own parsing.
- It's conceptually similar to parsing through a plist.
- Use the NSJSONSerialization class to convert raw JSON data you get back from a network service to objects (Dictionaries, Arrays, Strings, etc) and vice versa.

NSJSONSerialization

`+ JSONObjectWithData:options:error:`

Returns a Foundation object from given JSON data.

Declaration

SWIFT

```swift
class func JSONObjectWithData(_ data: NSData!,
                              options opt: NSJSONReadingOptions,
                                error error: NSErrorPointer) -> AnyObject!
```

OBJECTIVE-C

```objc
+ (id)JSONObjectWithData:(NSData *)data
                 options:(NSJSONReadingOptions)opt
                   error:(NSError **)error
```

Parameters

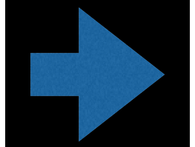
data	A data object containing JSON data.
opt	Options for reading the JSON data and creating the Foundation objects. For possible values, see NSJSONReadingOptions.
error	If an error occurs, upon return contains an `NSError` object that describes the problem.

NSJSONSerializationOptions

- NSJONReadingMutableContainers : specifies that arrays and dictionaries are created as mutable objects
- NSJSONReadingMutableLeaves : Specifies that leaf strings in the JSON Object graph are created as instances of NSMutableString
- NSJSONReadingAllowFragments : Specifies that the parses should allow top level objects that are not instance of NSArray or NSDictionary
- NSJSONWritingPrettyPrinted: Specifies that the JSON data generated should use white space to make it more human readable.

Parsing JSON

```
{
  "name": "Brad Johnson",
  "favoriteTeam": "Seahawks",
  "favoriteLanguage": "Swift"
}
```



```
if let filePath = NSBundle.mainBundle().pathForResource("test", ofType:
  "json") {
  if let data = NSData(contentsOfFile: filePath) {
    var jsonError : NSError?
    if let jsonObject = NSJSONSerialization.JSONObjectWithData(data,
      options: nil, error: &jsonError) as? [String : AnyObject] {
      if let name = jsonObject["name"] as? String{
        println(name)
      }
      if let favoriteTeam = jsonObject["favoriteTeam"] as? String {
        println(favoriteTeam)
      }
      if let favoriteLanguage = jsonObject["favoriteLanguage"] as?
        String {
        println(favoriteLanguage)
      }
    }
  }
}
```

Welcome to the pyramid of optional doom

JSON and Optionals

- So why are optionals all over JSON parsing? Well, its because pulling things out of dictionaries and arrays is pretty dangerous.
- The thing you expect to be there, might not be there! Either because you used the wrong key, or because the server gave you back something you weren't expecting.
- Once you get over how many if-lets you are going to type, it actually becomes comforting, knowing your app wont crash if the JSON and your code doesn't line up.
- In Swift 1.2, you can nest optional bindings, which will improve this situation.

Optional Binding + Down-casting

- You are going to use *Optional Binding and Down-casting* a lot when parsing through JSON.
- Optional Binding can be used to find out if an optional contains a value of a certain type, and if it does then it makes that value available as a temporary constant.

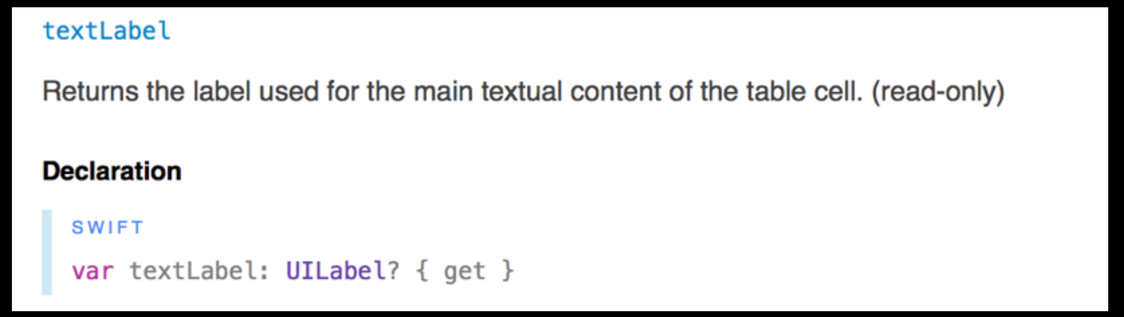
Optionals Review

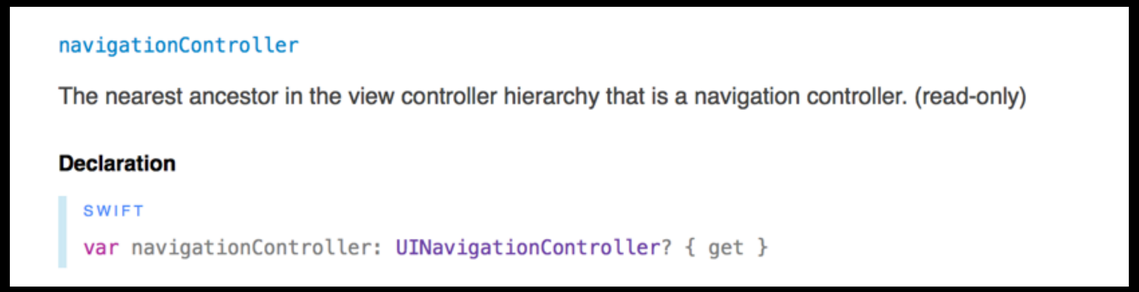
- You use optionals in situations where a value may be absent. An optional says:
  - There is a value, and it equals x
  - or
  - There isn't a value at all
- A variable is marked as optional by adding a question mark to the end of its type:

```
var twitterAccount : ACAccount?
```

Optionals Review

- You will see a ton of optionals in Apple's API's, specifically for return values or properties:





- So in your code, you will need to account for the fact that these values can be nil. So how do we do that?

Checking optionals for values

- you can use an if statement to check if an optional contains a value:

```
if convertedNumber != nil {
    println("convertedNumber contains some integer
        value.")
}
```

- Once you know an optional contains a value, you can use the ! to access its underlying value. This is called a **forced unwrapping** :

```
if convertedNumber != nil {
    println("convertedNumber has an integer value
        of \(convertedNumber!).")
}
// prints "convertedNumber has an integer value of
        123."
```

Optional Binding

- You can use **optional binding** to find out wether an optional contains a value,and if so, to make that value available as a temporary constant or variable that is unwrapped.
- The syntax of an optional binding:

```
if let constantName = someOptional {
    statements
}
```

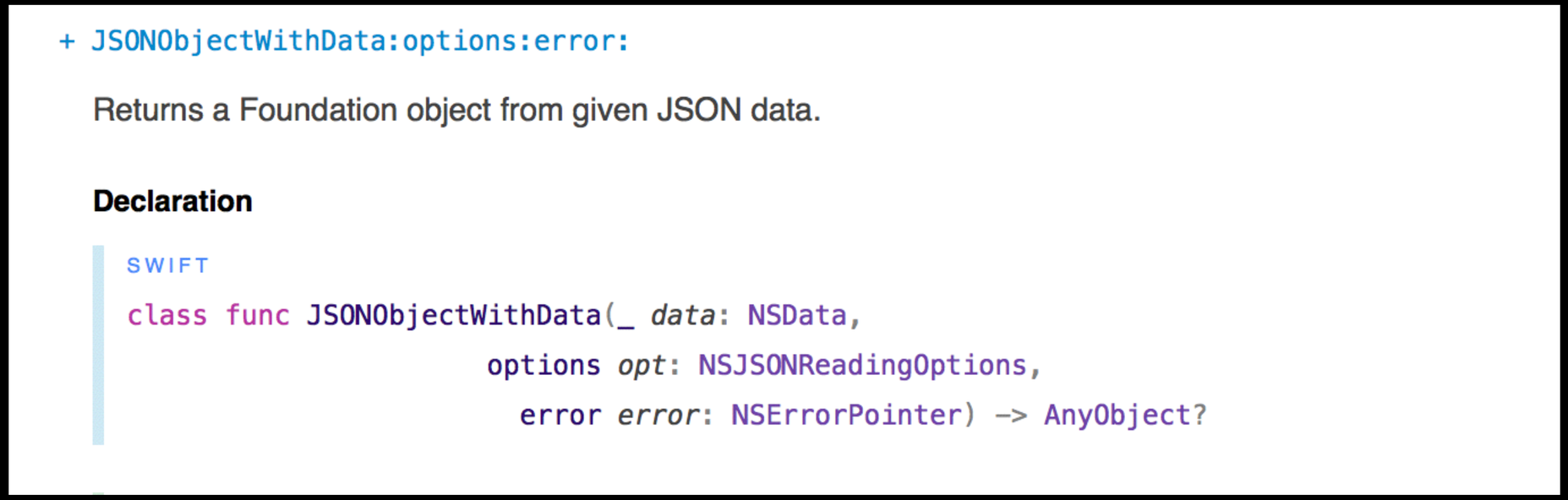
using optional binding

```
func printTitleValue(value : String?) {

    if let title = value {
        println(title)
    }
}
```

not using optional binding

```
func printTitleValue(value : String?) {

    if value != nil {
        let title = value!
        println(title)
    }
}
```

JSON and Optionals



- As you can see here, the NSJSONSerialization method that we use to convert the raw JSON data to foundation objects returns an optional value.
- So we first must check to see if a value was even returned.
- And then we check to see if the value returned matches the type of object we are expecting.

Downcasting

- A constant or variable of a certain type may actually refer to an instance of a subclass behind the scenes.
- When you believe this is true, you can try to downcast to the subclass with the type cast operator **as**
- Since down casting can fail, there are two forms of down casting:
  - optional form: as? - use this when you are not sure if the downcast will succeed. Will return an optional if it worked, nil if it didn't
  - forced form : as (In swift 1.2, this is now as!) use when you know it will work

Downcasting

```
let teams : [String : AnyObject] = ["SEA" : "Seahawks", "SF" : "49ers",
  "ARI" : "Cardinals", "Logos" : [UIImage]() ]


let seahawks = teams["SEA"] //seahawks type is AnyObject?
```

seahawks type is AnyObject? because our teams dictionary holds values of type AnyObject

```
let teams : [String : AnyObject] = ["SEA" : "Seahawks", "SF" : "49ers",
  "ARI" : "Cardinals", "Logos" : [UIImage]() ]


//let seahawks = teams["SEA"] //seahawks type is AnyObject?
let seahawks = teams["SEA"] as? String
```

We can use the optional downcasting to type cast it down to a String?

Downcasting

```
let teams : [String : AnyObject] = ["SEA" : "Seahawks", "SF" : "49ers",
  "ARI" : "Cardinals", "Logos" : [UIImage]() ]


let seahawks = teams["SEAL"] as? String
println(seahawks!) //CRASH
```

But because we left it as an optional, we still have to force unwrap the variable to access its value, which is dangerous because the optional downcast could have failed and returned nil, and force unwrapping a nil is an exception

```
let teams : [String : AnyObject] = ["SEA" : "Seahawks", "SF" : "49ers",
  "ARI" : "Cardinals", "Logos" : [UIImage]() ]


if let seahawks = teams["SEAL"] as? String {
  println(seahawks)
}
```

Wont crash!

So we use optional binding + optional downcasting to help us in these situations

Demo

Bundles

Bundles

- “Bundle are a fundamental technology in OS X and iOS that are used to encapsulate code and resources”
- A bundle is a directory with a standard hierarchical structure that holds code and resource for code.
- Bundles provide programming interfaces for accessing the contents of bundles in your code.

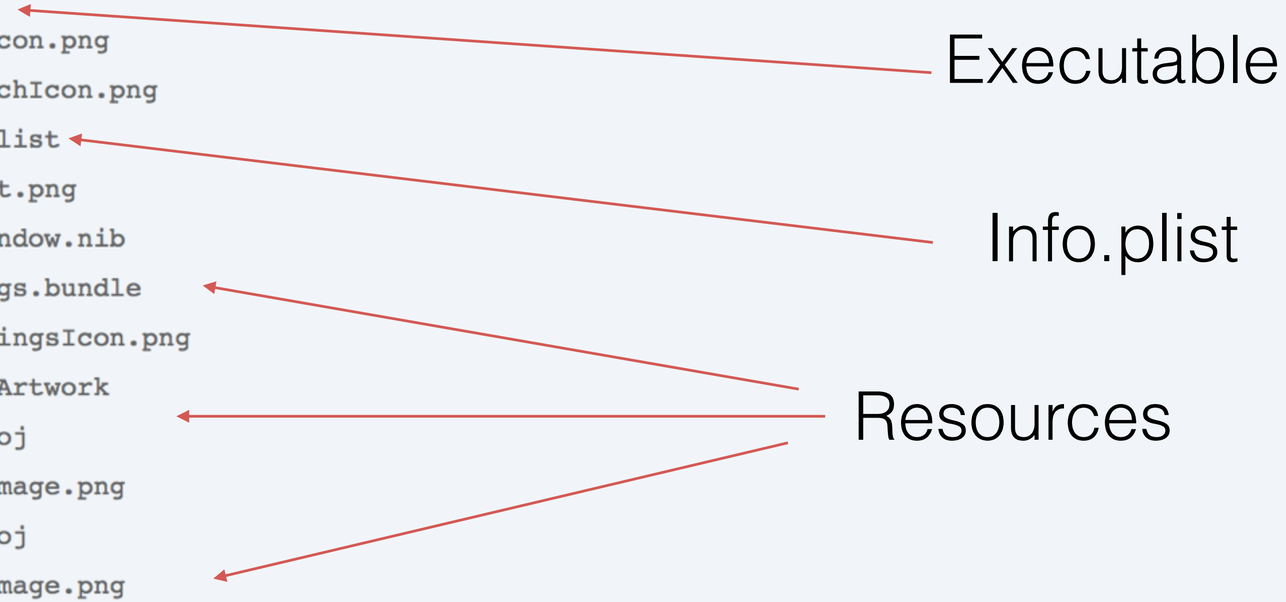
Application Bundle

- There a number of different types of bundles, but for iOS apps the most important is the application bundle.
- The application bundle stores everything your app requires to run successfully.
- Inside of your application bundle lives 4 distinct types of files:
  - Info.plist - a plist file that contains configuration information for your application. The system relies on this to know what your app is.
  - Executable - All apps must have an executable file. This file has the apps main entry point and any code that was statically linked to your app’s target.
  - Resource files - Any data that lives outside your app’s executable file. Images,icons, sounds, nibs, etc. Can be localized.
  - Other support files - Mostly used for mac apps. Plugins, private frameworks, document templates.

Application Bundle

Listing 2-1 Bundle structure of an iOS application

```
MyApp.app
    MyApp
    MyAppIcon.png
    MySearchIcon.png
    Info.plist
    Default.png
    MainWindow.nib
    Settings.bundle
    MySettingsIcon.png
    iTunesArtwork
    en.lproj
        MyImage.png
    fr.lproj
        MyImage.png
```



Bundles in code

- the NSBundle class represents a bundle in code.
- NSBundle has a class method called mainBundle(), which returns the bundle that contains the code and resources for the running app.
- You can also access other bundles that aren't the main bundle by using bundleWithPath() and passing in a path to another bundle.
- You can get the path for a resource by using pathForResource(ofType:)

UITableView and UITableViewCell

TableViews

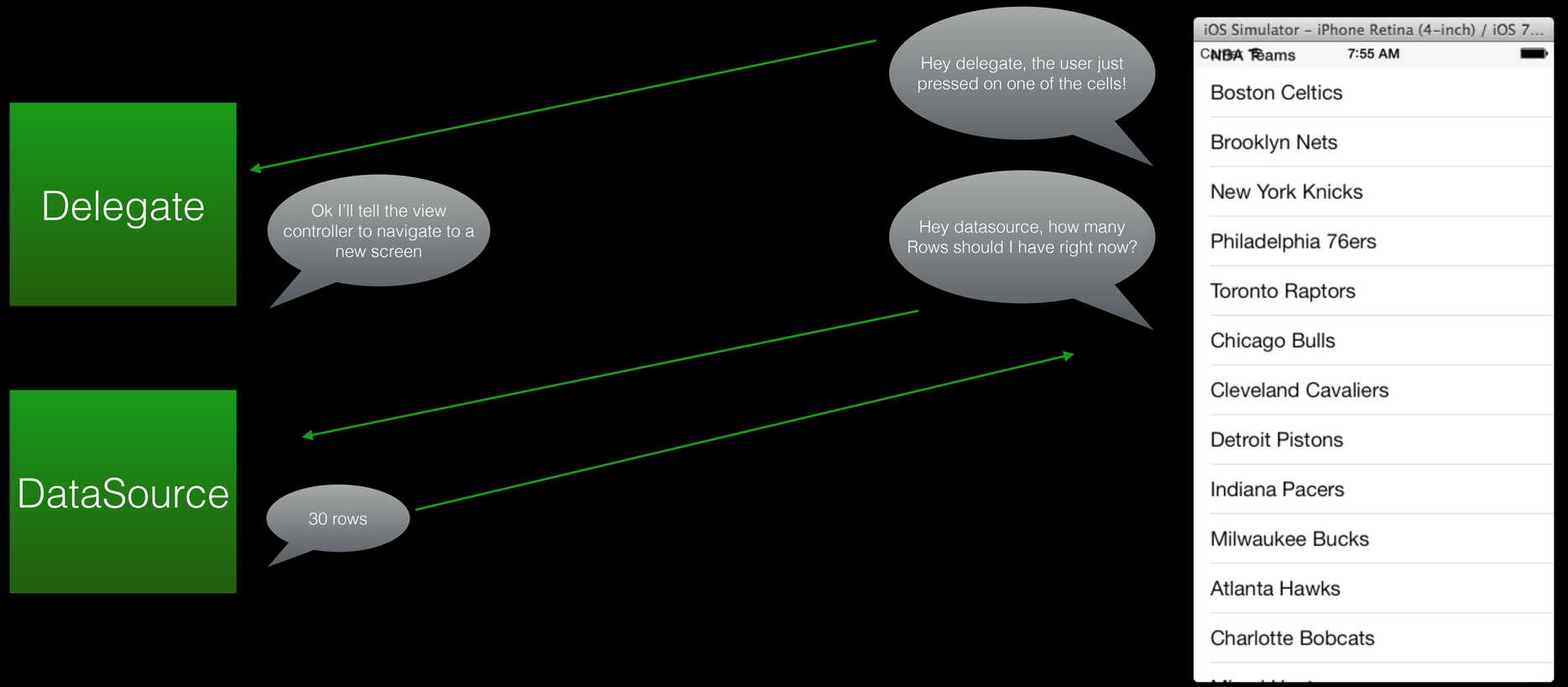
- “A Tableview presents data in a scrollable list of multiple rows that may be divided into sections.”
- A Tableview only has one column and only scrolls vertically.
- A Tableview has 0 through n-1 sections, and those sections have 0 through n-1 rows. A lot of the time you will just have 1 section and its corresponding rows.
- Sections are displayed with headers and footers, and rows are displayed with Tableview Cells, both of which are just a subclass of UIView.

So how do Tableviews work?

- Tableviews rely on the concept of delegation to get their job done.

- Picture time:

TableViews and Delegates



A tableview has 2 delegate objects. One actually called delegate and one called datasource. The datasource pattern is just a specialized form of delegation that is for data retrieval only.

Delegation

- The whole point of delegation is to allow you to implement custom behavior without having to subclass.
- Apple could have designed UITableView's API so you would have to subclass UITableView, but then you would have to understand UITableviews in a lot more detail(which methods can I override? Do I have to call super?)
- Delegates adopt a protocol to declare that they are capable of being the delegate. Think of it like a contract.
- Delegation is used extensively in a large portion of Apple's frameworks.

TableViews

- A tableview requires 2 questions to be answered (aka methods to be implemented) by its delegates and they are all in the datasource.
- `tableView(numberOfRowsInSection:) How many rows am I going to display?`
- `tableView(cellForRowAtIndexPath:) What cell do you want for the row at this index?`
- Number of sections is actually optional, and is 1 by default.

Demo

TableViewCell

- UITableViewCell is a direct subclass of UIView.
- You can think of it as a regular view that contains a number of other views used to display information.
- The ‘Content View’ of a cell is the view that all content of a table view cell should be placed on. Think of it as the default super view of your cell. It contentView itself is read only.

TableViewCell Style

- Setting the style of an instance of UITableViewCell will expose certain interface objects on the cell.
- The default style exposes the default text label and optional image view.
- Right Detail exposes a right aligned detail text label on the right side of the cell in addition to the default text label.
- Left Detail exposes a left aligned detail text label on the right side of the cell in addition to the default text label.
- Subtitle exposes a left aligned label below the default text label.

Creating tableView Cells

- You can instantiate them in code with the initializer init(style: UITableViewCellStyle, reuseIdentifier: String?)
- But usually you will be setting them up in your storyboard or in a xib file.
- If they are in your storyboard, you just have to set their reuse identifier in the identity inspector, and then call dequeueReusableCellWithIdentifier() at the appropriate time.
- Later in the week we will learn how to design and use cells with xibs.

Demo

Inheritance

Inheritance in Swift

- Inheritance is one of the foundations of object oriented programming, and its no different in Swift.
- A class inherits methods and properties from its super class.
- Not only can a subclass access the properties and methods of its super class, it can also override those methods and properties to refine or modify their behavior.
- Xcode provides support for helping you override by checking to make sure the types, parameters, and return values match up.

Base class

- Any class that does not inherit from another class is known as a base class.
- In Objective-C that class is NSObject. Swift has no universal base lass like NSObject
- So any class you create yourself without a super class automatically becomes a base class for you to build upon

Overriding in Swift

- Swift provides the override keyword to help you show clear intent when you are overriding, since accidental overriding can cause hard to diagnose bugs
- Anytime you override a method or property, you can access the super class's version of that method or property simply by accessing it on the variable 'super'