iOS Dev Accelerator
Week 1 Day 1

- Structs vs Classes
- MVC Review
- Single Responsibility
- Type Methods
- JSON - Serializing & Parsing
- Bundles
- TableView/Cells Review

Pimp your Xcode

Structs and Value Types

Structs and Classes

- In Objective-C, Structs are a 2nd class citizen compared to Classes.
- Because of this, the Cocoa & Cocoa Touch programming industry has largely ignored Structs, especially when creating the business logic of apps.
- But with Swift, Structs have much of the same functionality as Classes, which means there are now many more opportunities to use Structs in your apps instead of regular old Classes.

Swift Structs

- Structs have the following features in Swift:
  - Define properties to store values
  - Define methods to provide functionality
  - Define subscripts
  - Definit initializers
  - Can be extended
  - Can conform to protocols

Structs are Value Types

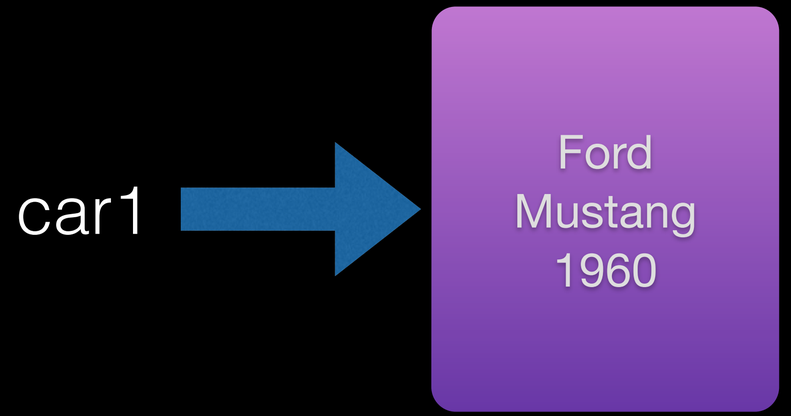
- Structs (and enums) are value types in Swift.
- A value type is a type whose value is copied when it is assigned to a variable or constant, or passed to a function
- Any value type properties the Struct contains are also copied.
- This is a primary difference from Classes, which are passed by references.
- Lets take a look.

Classes are Passed by Reference

```
class Car {
  let model : String
  let make : String
  let year : Int

  init(model: String, make: String, year: Int) {
    self.model = model
    self.make = make
    self.year = year
  }
}

let car1 = Car(model: "Mustang", make: "Ford", year: 1960)
```



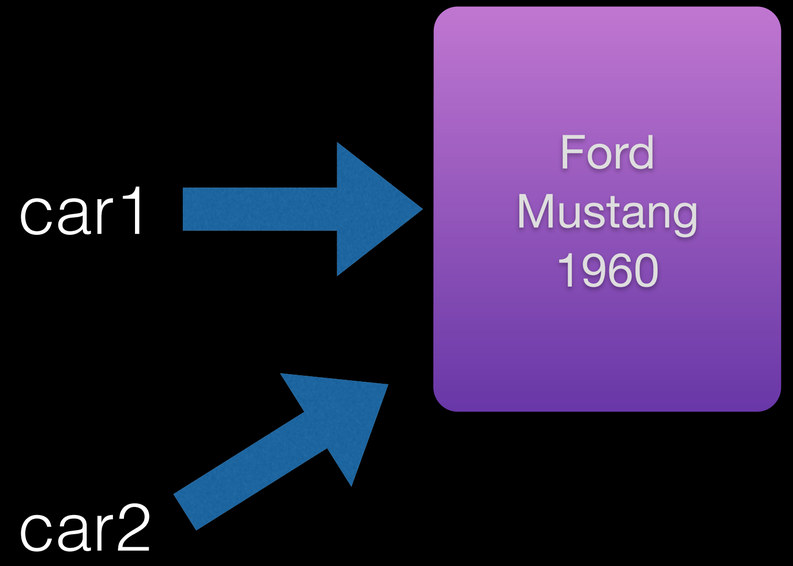
Classes are Passed by Reference

```
class Car {
  let model : String
  let make : String
  let year : Int

  init(model: String, make: String, year: Int) {
    self.model = model
    self.make = make
    self.year = year
  }
}

let car1 = Car(model: "Mustang", make: "Ford", year: 1960)

let car2 = car1
```

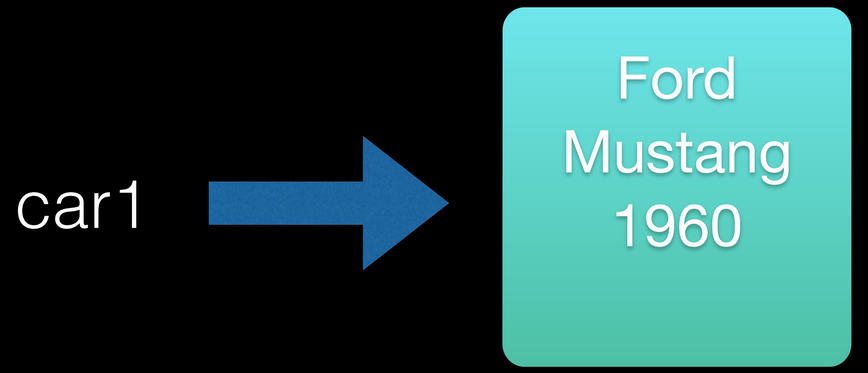


- This means any changes made to car2 are also made to car1, since they point to the same Car instance.
- This is useful in some situations, but what if car1 doesn't want those changes?

Structs are Passed by Copy

```
struct Car {
    let model : String
    let make : String
    let year : Int
}

let car1 = Car(model: "Mustang", make: "Ford", year: 1960)
```

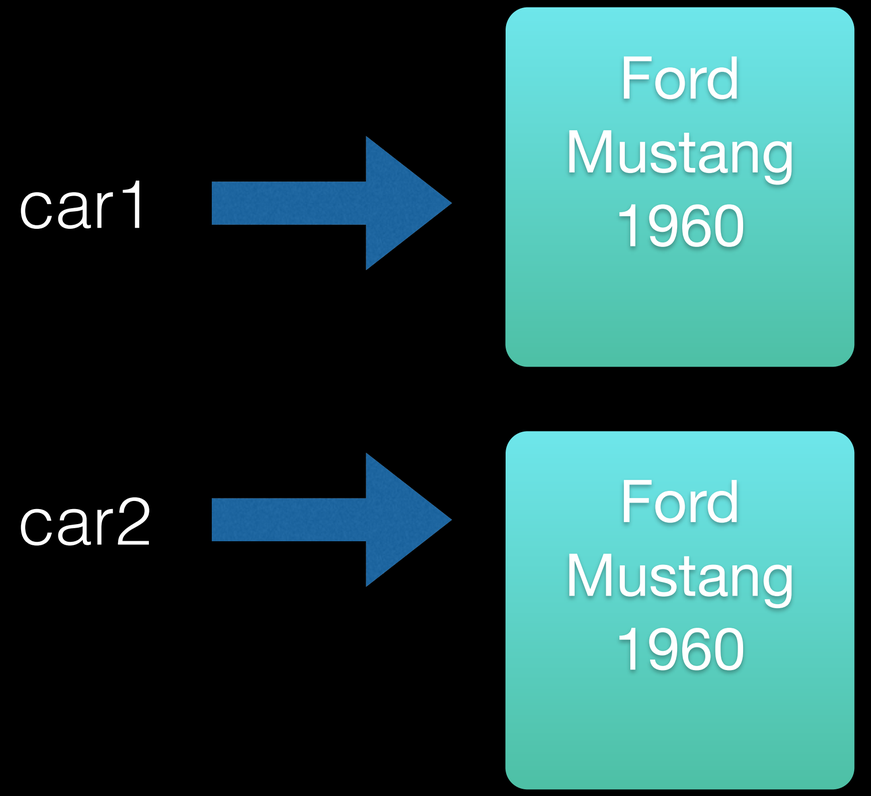


(The struct automatically gives us a constructor called the *memberwise initializer)*

Structs are Passed by Copy

```
struct Car {
    let model : String
    let make : String
    let year : Int
}

let car1 = Car(model: "Mustang", make: "Ford", year: 1960)
let car2 = car1
```



- When you use the assignment operator on two structs, it copies the struct on the right hand side.
- Now changes made to car2 have no effect on car1

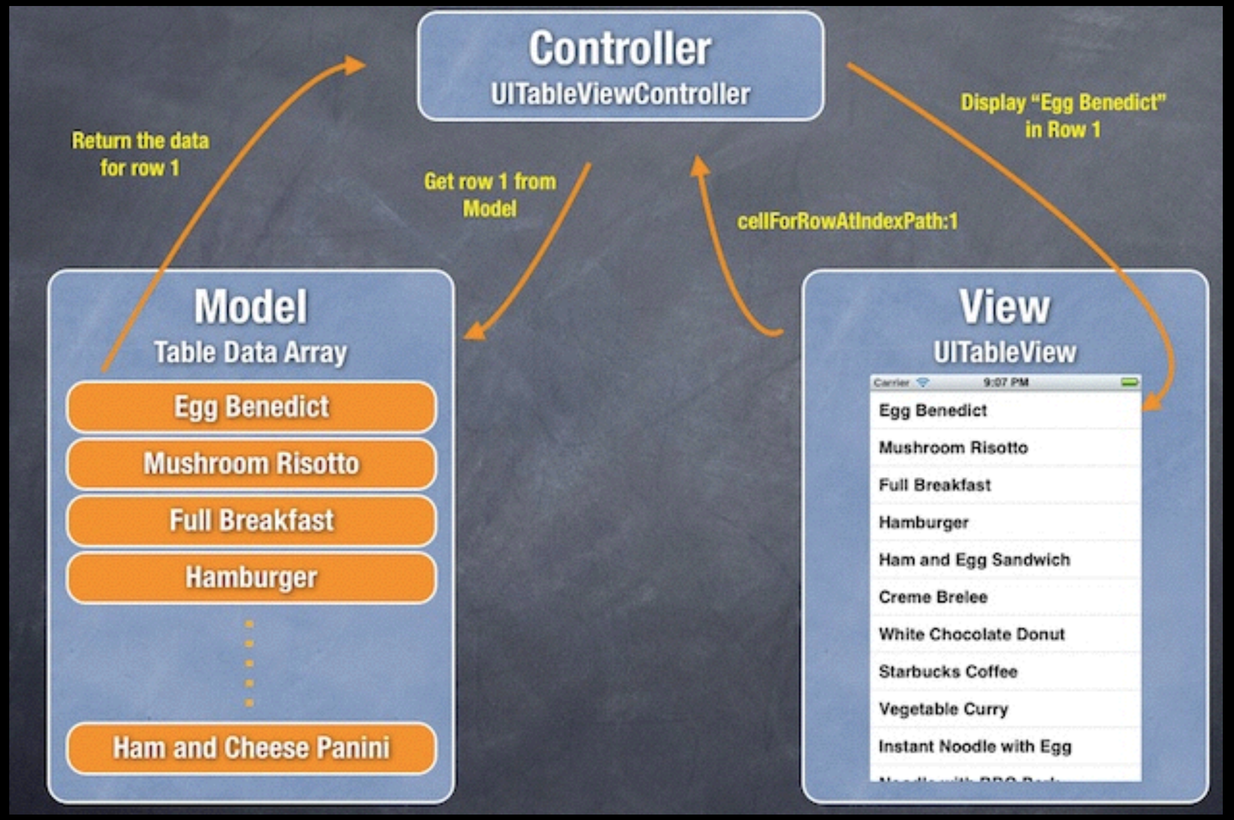
Structs are safer than Classes

- Because they are passed by copy instead of reference, Structs are safer to use than Classes.
- In particular, this makes them much, much more thread safe.
- Because when you pass a Struct around to other threads, you are passing a copy, not a reference! So two threads can’t be read/writing to the same struct at the same time.
- We will talk much more in depth about threading and safety tomorrow and throughout the course.

So when to use Structs vs Classes

- This is a tough question to answer, it largely depends on your personal programming philosophy.
- However, here is some guidance from iOS/Mac developers I really respect:
    1. **If it doesn't make sense for an entity to be copied, use a Class.** (it makes sense for a car to be copied (like a car factory), but doesn't make sense for a UIWindow to be copied, or a Person to be copied)
    2. **If the entity you defined has all value type properties, it should probably be a Struct.**
    3. **If the entity needs to be passed by reference, use a class, and if it needs to be passed by copy, use a struct.**
    4. **If the entity will be heavily involved with multithreading, use a struct.**

Struct Demo



MVC (Model-View-Controller) Review

MVC is all about Seperation & Communication

- 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.
- With this separation, communication becomes key in keeping everything loosely coupled and organized.

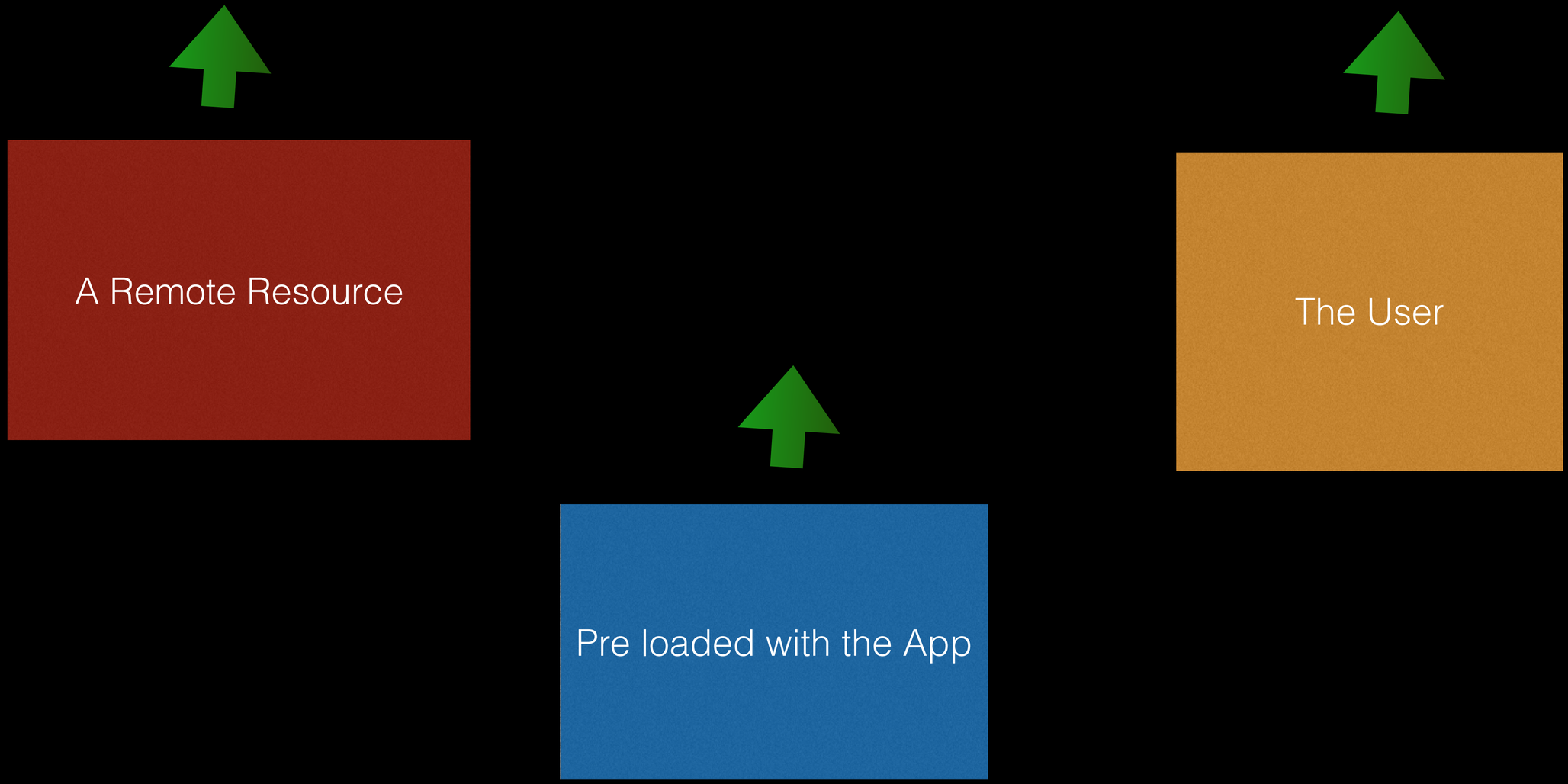
The differing roles

- 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.

Wheres the data?

- Mobile apps are all about:
    1. Showing information to the user
    2. Allowing the user to interact with this information
- So a great question to always be asking is, where is the data coming from?

It can come from many places



The MVC layers of our Week 1 App

Model Layer

- Tweet
- User
- TweetJSON Parser
- UserJSON Parser
- TwitterService

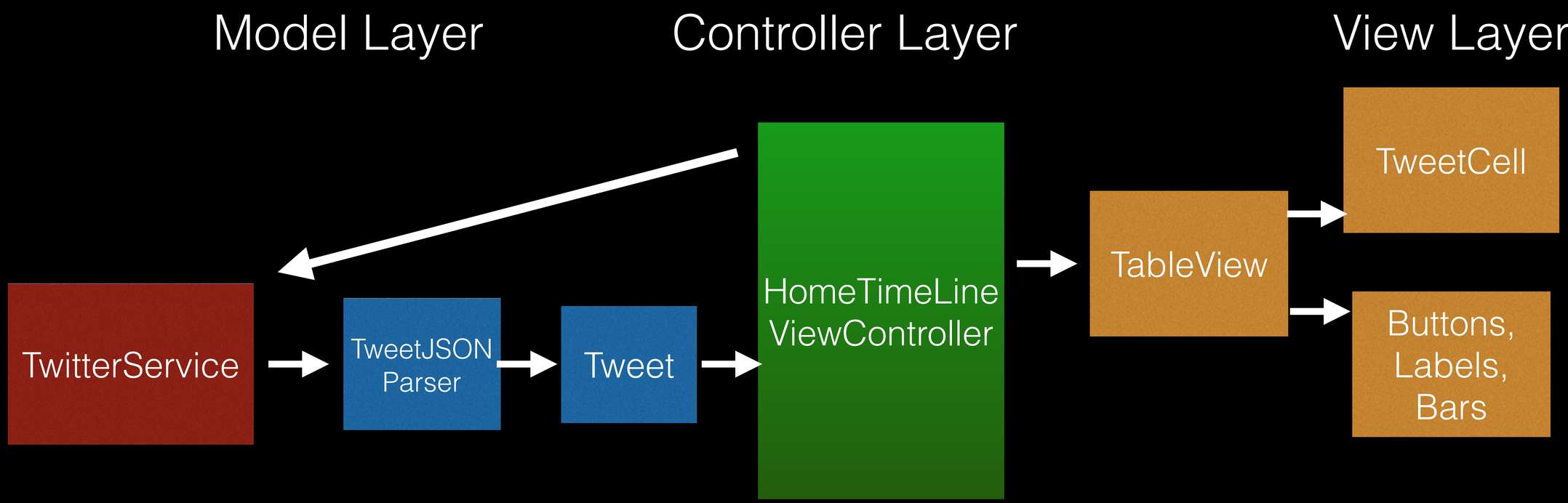
Controller Layer

- HomeTimeLine ViewController
- Tweet ViewController
- UserTimeLine ViewController

View Layer

- Storyboard
- TweetCell.xib
- TweetCell
- Buttons, Labels, Bars

One Scene's MVC (seperation and communication!)



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 (or struct) 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 struct in our Twitter app should only need to change if we need to change how we display a tweet to 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 200 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 identified.

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 the way to go.
- We will use type methods to create Tweet instances from JSON data.

Type Methods

struct Tweet

var text : String
var date : NSDate
var author : String

instance Tweet

var text = “I love TaylorSwift!! #music”
var date = 3/4/15
var author = “brad”

instance Tweet

var text = “seahawks are the best”
var date = 3/32/15
var author = “brad”

instance Tweet

var text = “coding is great!”
var date = 3/21/15
var author = “tyler”

Type Methods

struct Tweet

var text : String
var date : NSDate
var author : String

static func generateESTDate {
……
}

instance Tweet

var text = “I love TaylorSwift!! #music”
var date = 3/4/15
var author = “brad”

instance Tweet

var text = “seahawks are the best”
var date = 3/32/15
var author = “brad”

instance Tweet

var text = “coding is great!”
var date = 3/21/15
var author = “tyler”

a type method is not accessible via an instance of the type! This is the direct opposite of regular methods, which we can call instance methods

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. And they can have parameters and return values just like normal

Type Methods

- And on Structs, they are defined with the static keyword

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    static func unlockLevel(level: Int) {
        if level > highestUnlockedLevel {
          highestUnlockedLevel = level }
    }
    static func levelIsUnlocked(level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }
```

Demo

JSON

JSON

- “JavaScript Object Notation” (but its not just for javascript!)
- **“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, IN ALL STACKS/PROGRAMMING LANGUAGES
- 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. In iOS programming, if you want to put null inside a collection use NSNull class

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. Dictionaries are denoted by the { bracket

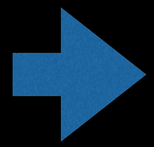
first item in the dictionary is an array that is paired with the key “users”. Arrays are denoted by the square brackets []

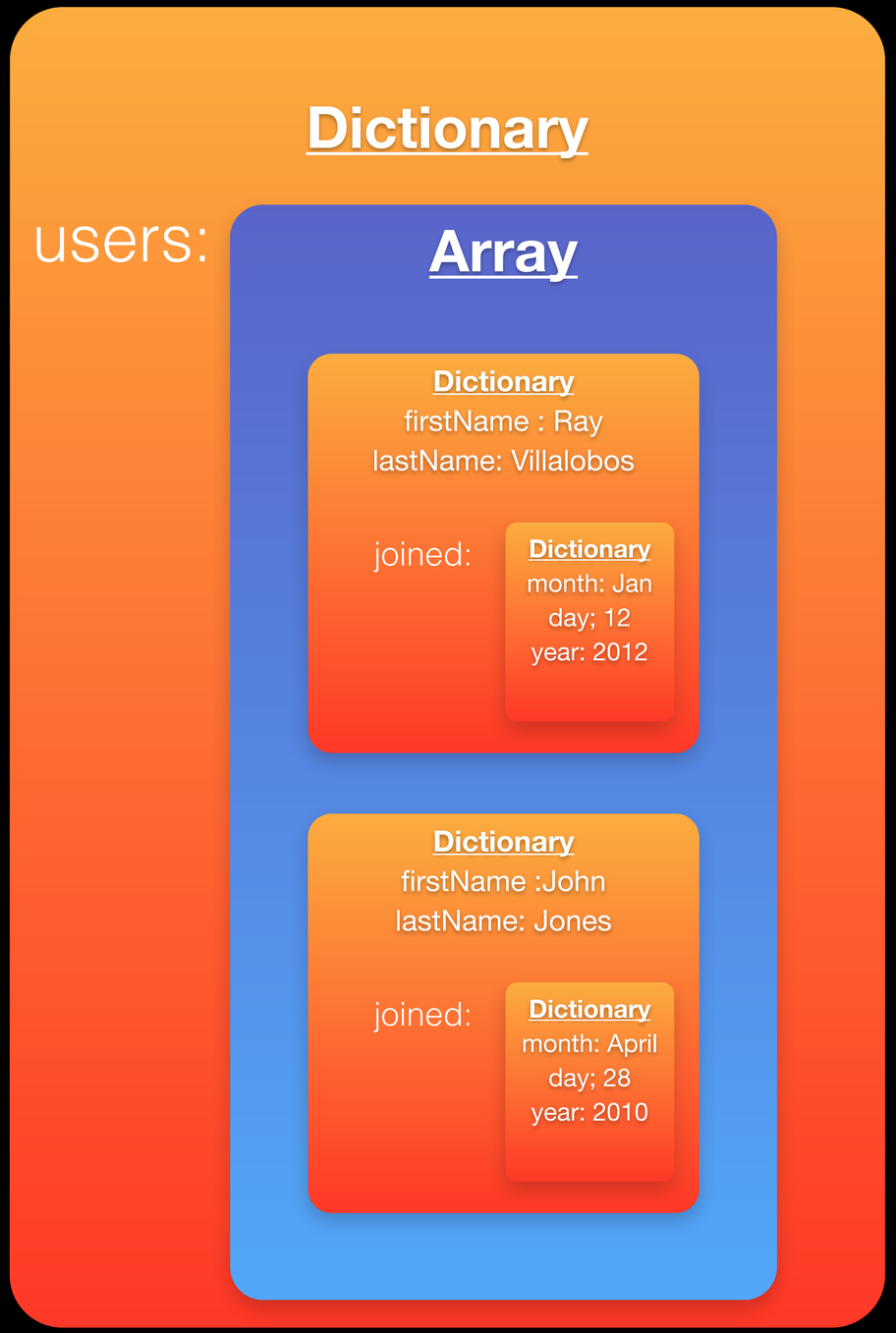
the array contains dictionaries separated by a comma

end of array and end of root dictionary

JSON Example

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





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!",
```

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 models 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 workflow in your app

- Once you have made your request and gotten your response, theres 2 distinct steps:

    1. **Serialize** the JSON data contained in the body of the response into native types (dictionaries, arrays, strings,etc)
    2. **Parse** through the serialized JSON objects and create your model objects with data received

JSON Serializing

- Use the **NSJSONSerialization** class to convert raw JSON data you get back from a network service to native types (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

In general you won’t need these, but you can treat it as a case-by-case basis with the different web API’s you end up using.

Demo

Parsing JSON

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

Optional Binding FTW
(For the Win)

```
if let jsonDictionary = NSJSONSerialization.JSONObjectWithData
  (data, options: nil, error: &error) as? [String : String],
let name = jsonDictionary["name"],
let favoriteTeam = jsonDictionary["favoriteTeam"],
let favoriteLanguage = jsonDictionary["favoriteLanguage"]
{

  //do something with our newly parsed information

}
```

JSON and Optionals

- So why are optionals all over JSON parsing?
- Because pulling things out of dictionaries 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.

Optional Binding Review

- 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 that is unwrapped.
- The syntax of an optional binding:

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

using optional binding

```
func printTitle(title : String?) {
  if let title = title {
    println(title)
  }
}
```

always make the name of the optional binding the same as the optional you are testing

not using optional binding

```
func printTitle(title : String?) {
  if title != nil {
    println(title!)
  }
}
```

always avoid doing this if you can

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.

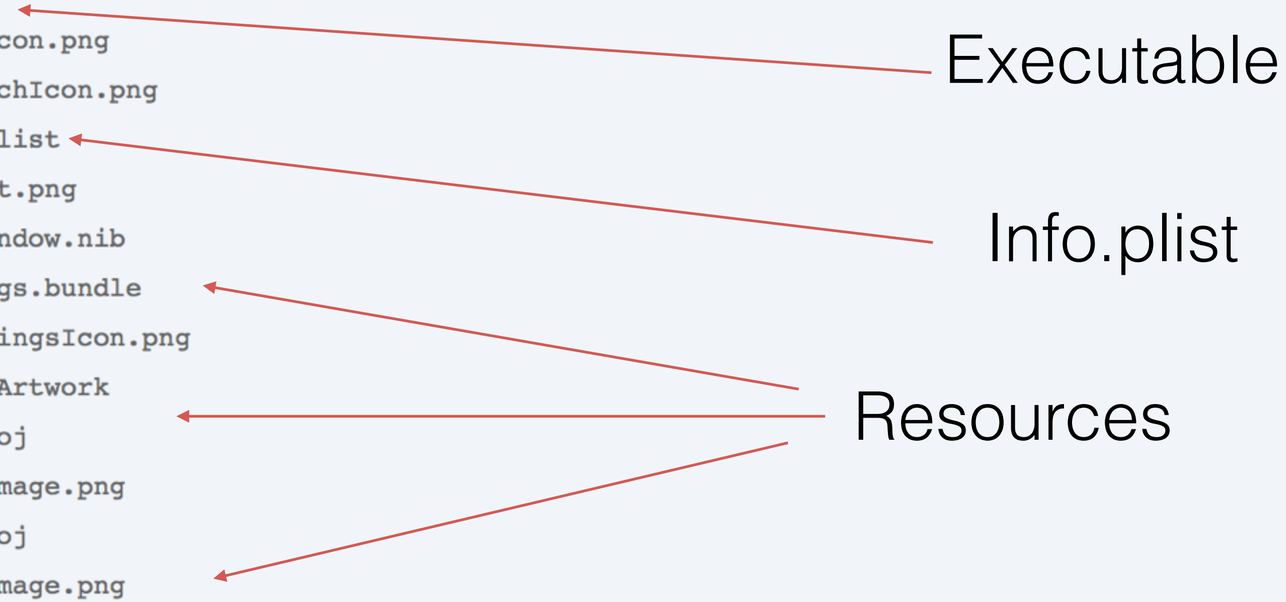
Application Bundle

- 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:)

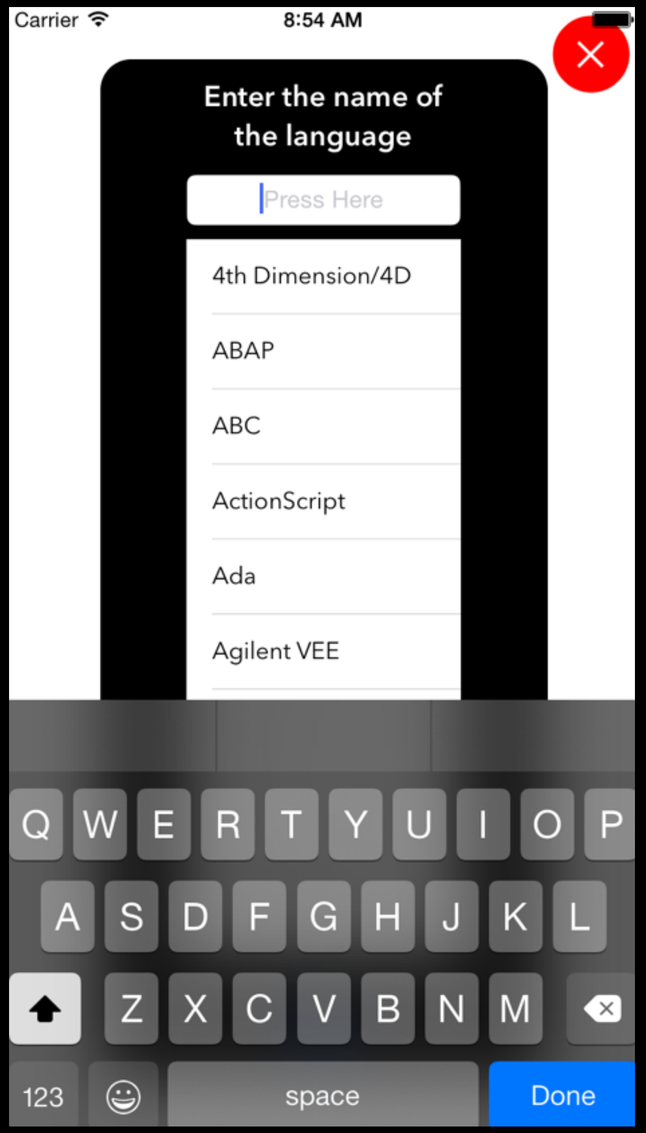
UITableView Review

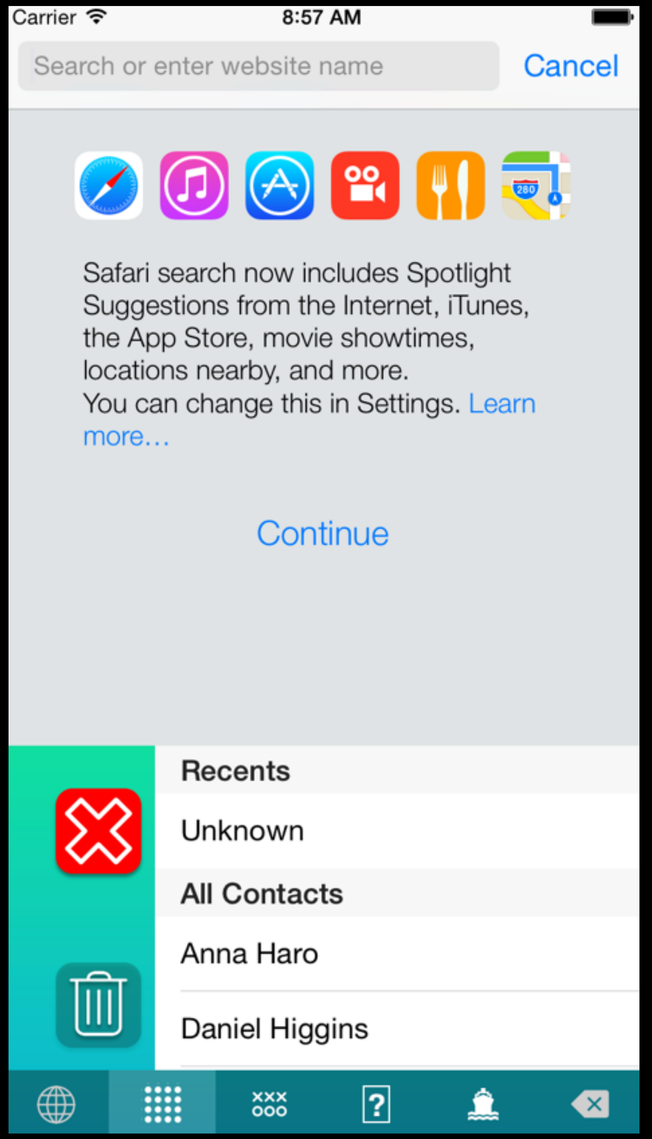
TableViews

- “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.

TableView Review Fact #1

- **A table view is a subclass of UIView.**
- Views can take any shape and size, and go anywhere on screen.
- So a table view doesn't have to be full screen.





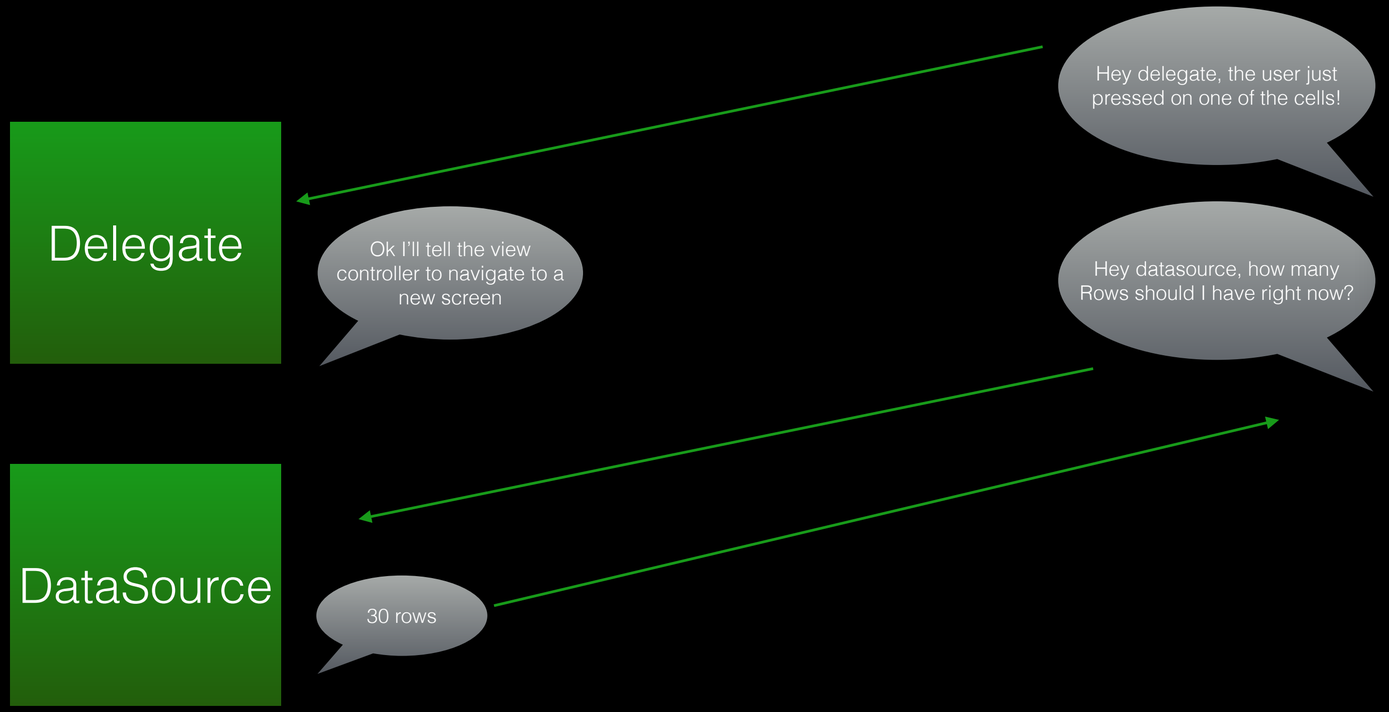
TableView Review Fact #2

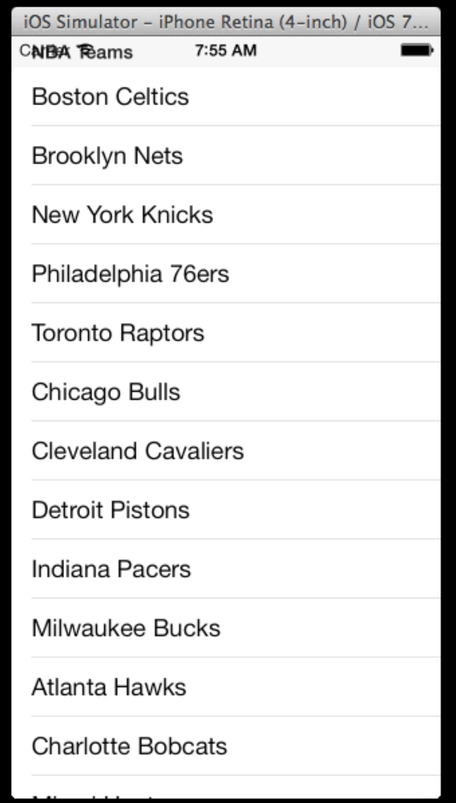
- **Table views rely on the concept of Delegation to function**

- This is a very popular design pattern with apple.

TableView Review Fact #3

- **Table view has two delegate objects. One called delegate, the other called data source**
- The delegate is responsible for all user interaction with the table view, data source is responsible for the supply the table view with data to show.





TableView Review Fact #4

- **The UITableViewDataSource protocol has 2 required methods**
- `tableView(numberOfRowsInSection:)` How many rows am I going to display?
- `tableView(cellForRowAtIndexPath:)` What cell do you want for the row at this index?

TableView Review Fact #5

- **Table views reuse table view cells. So you need to make sure you properly ‘scrub’ their old data before they are shown again.**
- This clean up can take place in your tableView(cellForRowAtIndexPath) method. Most of the time you are already doing it by giving its labels and views new data that overwrite the old data.

Demo