iOS Foundations II
Session 7

- Custom Table View Cell
- For loops
- Property List
- Multiple Optional Bindings
- Access Control

Custom UITableViewCell

TableViewCell Review

- 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. contentView itself is a read only property, you cant set it to be a different view.

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.

Custom UITableViewCell

- Creating and laying out your own custom UITableViewCell is a relatively straight forward workflow:
  - Create a new class that is a subclass of UITableViewCell
  - In your storyboard, set your prototype tableview cell to be your new custom class
  - Drag out any interface elements you want onto your prototype cell, and then harness the badass power of auto layout to make it look amazing
  - Create outlets to each element in your custom class's implementation
  - Refactor your cellForRow:AtIndexPath: method on your tableview's datasource to use your new class

Demo

For loops

For Loops in Swift

- Swift provides all the familiar control flow from other C-like languages.
- For loops, while loops, if statements, and switch statements all fall under the category of control flow
- We will focus on for loops today, in particular the different ways Swift allows you to format and implement for loops.

For Loops in Swift

- Swift provides 2 types of for loops:
  - The regular ‘for loop’ performs a set of statements until a specific condition is met, typically by incrementing a counter each time the loop ends
  - The ‘for-in loop’ performs a set of statements for each item in a range,sequence,collection, or progression

Regular For Loop

- Here is an example of a regular for loop in Swift:

```
for var index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
```

- So here is the general form of a for loop in Swift:

```
for initialization; condition; increment {
    statements
}
```

Demo

For-in Loop

- You use the for-in loop to iterate over collection of items, such as items in an array:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    println("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

- You can even do it on a string!

```
for character in "Hello" {
    println(character)
}
```

Demo

Which one do I use?

- Most devs prefer to use the for-in loop vs the regular C style for loop. It looks cleaner and doesn't require any extra variable initialization or conditionals that may be prone to off-by-one errors.
- Of course, if you have no collection to loop through, you should the C style for loop.
- An advantage of the C style for loop is that you can keep track of the index you are on for each iteration of the loop.

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.

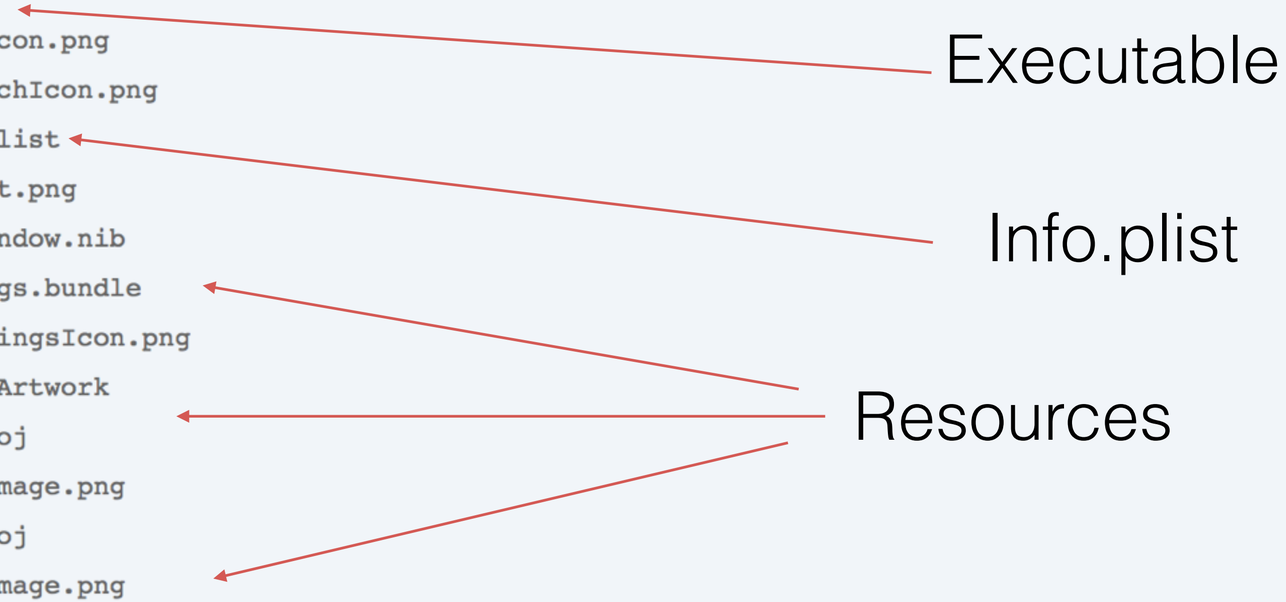
Application Bundle

- 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. Not very common.
- You can get the path for a resource by using pathForResource(ofType:)

Demo

plist (property list)

- Apple-flavored XML
- Keys must be strings
- Values must be NSCoding compliant (more on this next lecture)
- Load from your bundle or from the web
- Root-level object is typically a Dictionary or Array

Xcode GUI format

Key	Type	Value
▼ Root	Array	(3 items)
▼ Item 0	Dictionary	(2 items)
FirstName	String	Jimmy
LastName	String	Graham
▼ Item 1	Dictionary	(2 items)
LastName	String	Wilson
FirstName	String	Russell
▼ Item 2	Dictionary	(2 items)
LastName	String	Johnson
FirstName	String	Brad

Raw format

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <dict>
    <key>FirstName</key>
    <string>Jimmy</string>
    <key>LastName</key>
    <string>Graham</string>
  </dict>
  <dict>
    <key>LastName</key>
    <string>Wilson</string>
    <key>FirstName</key>
    <string>Russell</string>
  </dict>
  <dict>
    <key>LastName</key>
    <string>Johnson</string>
    <key>FirstName</key>
    <string>Brad</string>
  </dict>
</array>
</plist>
```

Demo

plist Workflow

1. Get the URL of the plist file living in your bundle
2. Read the plist into an Array or Dictionary using class methods on the Objective-C collection types, based on the root object of the plist, but cast it back as a Swift collection type
3. Parse the data in the plist into your model objects

Demo

Multiple Optional Bindings

- New with Swift 1.2, you can now have multiple optional bindings in the same if-let statement.
- The multiple bindings are comma seperated
- They are evaluated in turn, and if any of the attempted bindings are nil, the entire if-let evaluates to false.

Demo

Access Control

- Access Control is a universal programming concept that allows developers to restrict access to parts of their code from code in other source code files or modules
- It allows you to hide implementation details of your code, and specify a designated interface through which your code should be interacted with

Access Control in Swift

- In swift, you can assign specific access levels to individual types (classes, structures, enumerations)
- You can also assign specific access levels to properties, methods, inits, and subscripts of those types.
- You can also restrict protocols, global constants/variables/functions (not very common)

Access Control in Swift

- Swift provides default access control levels to all of your code, which usually work well for your typical app projects
- If you are writing a regular, single-target app, you don’t actually need to specify access controls at all if you don’t want to.
- But if you are working in a team, access control is a very valuable tool.
- It’s also a bit more explicit Objective-C.

Swift Modules and Source Files

- Swift's access control model is based around on the concept of modules and source files
- A **module** is a single unit of code distribution. Like an app or a framework.
- A **source file** is a single Swift file within a module. It is a single file within an app or framework

Access Levels

- Swift provides 3 different access levels for your code:
- **Public** : enables entities to be used within any source file from their defining module (app or framework), and also in a source file from another module that imports the defining module.
- **Internal** : enables entities to be used within any source file from their defining module, but NOT in any source file outside of that module. This is the default access level.
- **Private** : restricts use of an entity to its own defining source file. Use this to hide implementation details of a specific piece of functionality.

Demo

Encapsulation

- In computer science and object oriented programming, encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.
- Usually only the object's own methods can directly inspect or manipulate its data.
- When a programming language has features to help you with encapsulation, it is usually referred to as access control.
- In Objective-C, the separation of .h and .m does most of the work for us.

Encapsulation Benefits

- Encapsulation provides a number of benefits:
  - **Hides complexity:** Think of a car. A car has super complex internals. But we as users only interface with simple controls, like the wheel and ignition. Imagine if you had to actually know exactly how a car works to use it. We can consider those wheel and ignition controls its public interface.
  - **It helps us achieve loose coupling:** Encapsulation helps us decouple modules that comprise our app, allowing them to be developed, used, tested, and understood in isolation.

Demo