iOS Foundations II
Session 8

- Previous Class Review
- Custom Table View Cells
- For loops
- Data Persistence
- More Debugging Tips
- What’s Next?

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

Data Persistence

	Core Data	NSKeyedArchiver	NSUserDefaults
Entity Modeling	Yes	No	No
Querying	Yes	No	No
Speed	Fast	Slow	Slow
Serialization Format	SQLite, XML, or NSData	NSData	Binary Plist
Migrations	Automatic	Manual	Manual
Undo Manager	Automatic	Manual	Manual

Sourced from NSHipster

Data Persistence

	Core Data	NSKeyedArchiver	NSUserDefaults
Persists State	Yes	Yes	Yes
Pain in the Ass	Yes	No	No

Sourced from NSHipster

Demo

NSKeyedArchiver

NSKeyedArchiver

- NSKeyedArchiver/Unarchiver serializes NSCoding compliant classes to and from a data representation on disk.
- **Classes you want to serialize with NSKeyedArchiver must conform to the NSCoding protocol.**
- Once you have done that, it is as simple as calling archive and unarchive on NSKeyedArchiver/Unarchiver to load your object graph
- The amazing part of NSKeyedArchiver is that your object graph is saved and loaded as your custom model types. You don't have to recreate all of your model objects when you load them, like we had to do with the plist.

NSCoding Protocol

- The NSCoding protocol is a very simple protocol, it only has 2 methods:
  - init(Coder)
  - encodeWithCoder()
- Your class that conforms to NSCoding must also inherit from NSObject
- The implementation of these two methods is very much just boilerplate code, as you will see in the next slide.

NSCoding Protocol

```
//first required method is the init with coder, this is used internally by
    NSKeyedUnarchiver to load your objects from the archived data
required init(coder aDecoder: NSCoder) {
    self.firstName = aDecoder.decodeObjectForKey("firstName") as String
    self.lastName = aDecoder.decodeObjectForKey("lastName") as String
    if let decodedImage = aDecoder.decodeObjectForKey("image") as? UIImage {
        self.image   = decodedImage
    }
}

//the other required method, used to encode your objects into the archive file by
    NSKeyedArchiver
func encodeWithCoder(aCoder: NSCoder) {
    aCoder.encodeObject(self.firstName, forKey: "firstName")
    aCoder.encodeObject(self.lastName, forKey: "lastName")
    if self.image != nil {
        aCoder.encodeObject(self.image!, forKey: "image")
    }
}
```

Saving/Loading from disk

- When you use NSKeyedArchiver and NSKeyedUnarchiver, you are saving an archive file to disk. To do this, you will need a path that you want to save to.
- Each app has a separate ‘sandbox’ (or directory) for storing data. iOS keeps the apps separated from each other and the OS for security reasons.
- The sandbox contains a number of different sub directories, and the one we are allowed to write to is called the documents directory.

Getting the path

- We can use the function ‘NSSearchPathForDirectoriesInDomains() to get a path to the documents directory.
- It takes 3 parameters:

1. An enum for which directory you are looking for (we will use .DocumentsDirectory since we want the documents directory!)
2. The domain mask (will always use .UserDomainMask)
3. expandTilde Boolean (will always use true)

Example of a save

```
func saveToArchive() {
    //get path to documents directory
    let documentsPath = NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .
        UserDomainMask, true)[0] as String
    //archive
    NSKeyedArchiver.archiveRootObject(self.people, toFile: documentsPath + "/
        archive")
}
```

Example of a load

```
func loadFromArchive() {

    //get path to your app's documents directory in its sandbox
    let documentsPath = NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .
        UserDomainMask, true)[0] as String
    //attempt to unarchive your object graph
    if let peopleFromArchive = NSKeyedUnarchiver.unarchiveObjectWithFile
        (documentsPath + "/archive") as? [Person] {
        //stored the data we just unarchived into this proper   ty
        self.people = peopleFromArchive
        //this is great, it loaded our stuff
    }
}
```

Demo

Debugging

Debugging

- Having the feature you just built work perfectly on the first time you try it pretty much never happens. Everyone makes mistakes all the time.
- When things go wrong, which they will, instead of immediately tweaking the code, try debugging to compare the actual results vs the intended results.
- So how do you debug?

Breakpoints

- A breakpoint is a way to pause the execution of your app at a specific line
- You set breakpoints by clicking in the left hand margin of your code:

```
21      override func viewDidLoad() {
22        super.viewDidLoad()
23
24        self.tableView.dataSource = self
25        self.tableView.delegate = self
26        let cellNib = UINib(nibName: "TweetTableViewCell", bundle: NSBundle.
         mainBundle())
```

- After you have set your breakpoint, there is no need to recompile your app. If your app has to run that code again, the breakpoint will hit and your execution will pause.

Modifying your Breakpoint

- There are a few things you can do to your breakpoints after you have placed it:
- Click it once to disable it. It will look faded out.
- To remove it, drag it off of the ‘gutter’
- You can right click a breakpoint and choose from a few options (the ones above, plus edit breakpoint)

Conditional Breakpoint

- Using a conditional breakpoint, you can designate specific conditions where the breakpoint should pause execution.
- In addition, you can specify actions to take place upon the breakpoint being triggered.
- These actions can be a wide range of things: AppleScripts, Capturing OpenGL frames, Log or speak a message, Execute Shell command, play a sound.

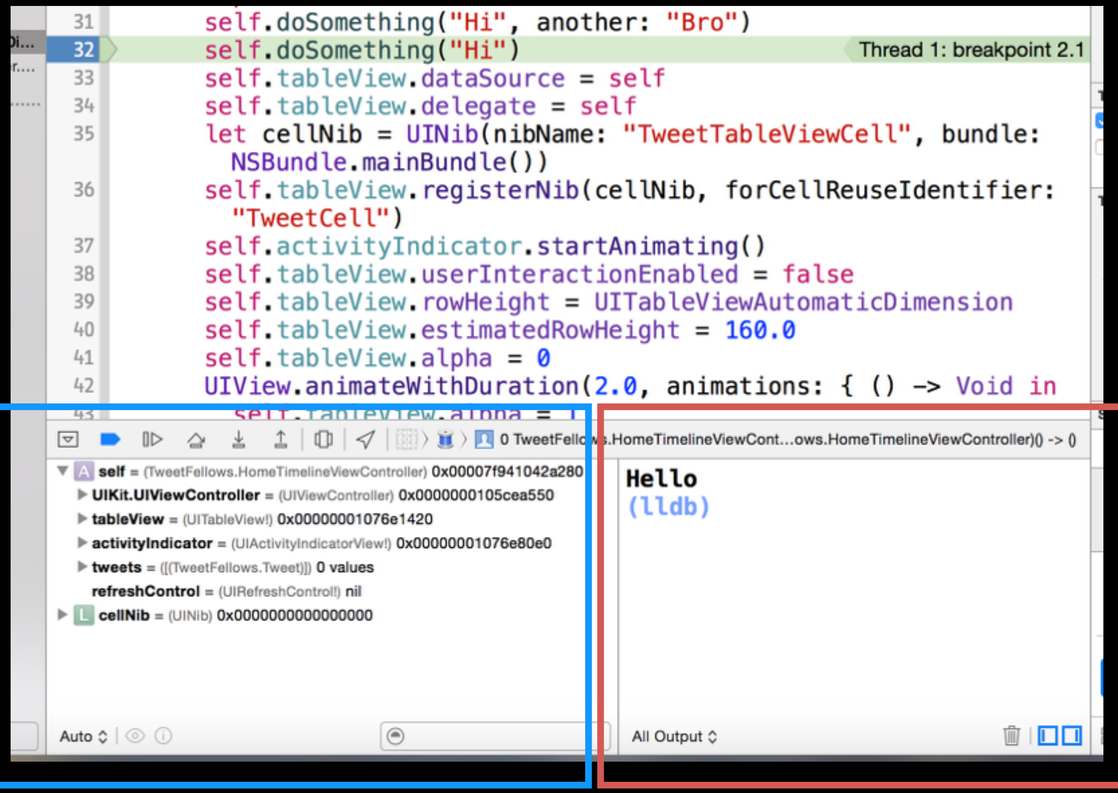
Exception Breakpoints

- Exception breakpoints are super useful and super simple to set.
- These special breakpoints allow you to pause execution of your app when an exception is raised, not caught.
- Which is to say, before your app actually crashes because of the exception.
- This is helpful because often times Xcode does not produce a very helpful debug statement when your app crashes.

Demo

Debug Area

- Once your application is paused because of a breakpoint, you can then do the actual debugging using the Debug Area at the bottom of Xcode.
- The two different views are the variable view and debugger console



Variable View

Debugger Console

Variable View

- The Variable View allows you to inspect the value of a variable to help uncover problems in your code
- The Variable View allows you to see all variables in the current scope at the time of the paused execution
- You can specify which items you want to see by using the little popup menu at the bottom:
  - Auto: Displays recently accessed Variables
  - Local: Displays local variables
  - All: Displays all variables

Demo

Debugger Console & LLDB

- The Debugger Console is a great tool to use for debugging.
- It is made possibly by LLDB.
- LLDB is an open source debugger that comes bundled inside Xcode and lives in the debugger console
- When you add breakpoints, you are actually telling LLDB when it should pause execution of the app.

print

- The print command allows you print the value of a variable
- You can also use p for short.

po

- The po command (print object) prints the result of calling description on the object

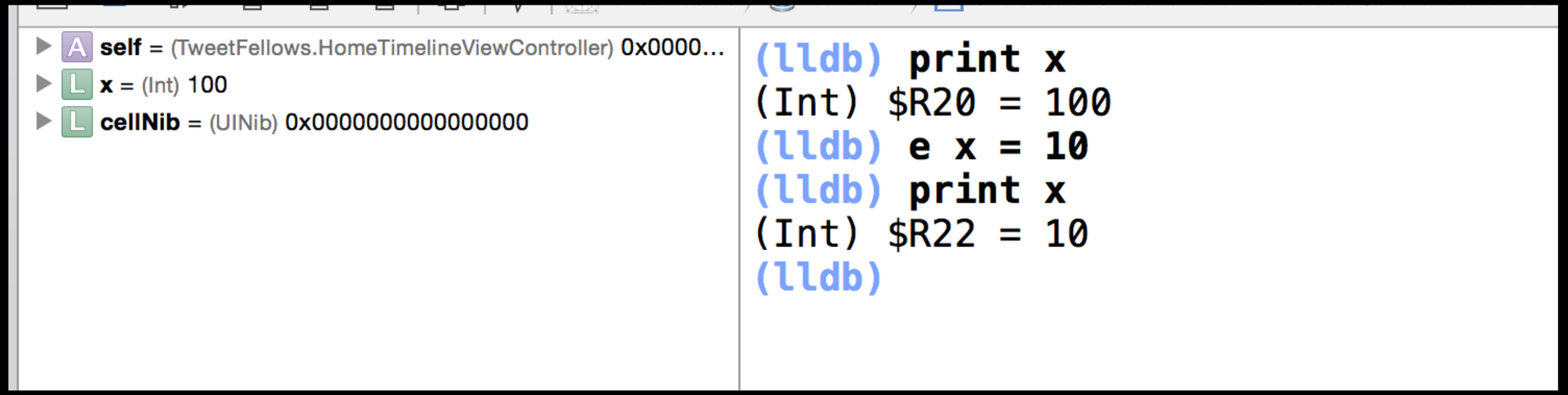
```
(lldb) print self.view
(UIView!) $R13 = Some {
  Some = 0x00007f89d8c75240 {
    UIKit.UIResponder = {
      ObjectiveC.NSObject = {}
    }
  }
}
(lldb) po self.view
<UIView: 0x7f89d8c75240; frame = (0 0; 375 667); autoresize
+H; layer = <CALayer: 0x7f89d8c614b0>>

(lldb) po x
10

(lldb) print x
(Int) $R16 = 10
(lldb)
```

expression

- The expression command lets you modify the value held by a variable from the debug console.
- It doesn't just modify the variable in the debug console, it actually modifies the value in the program!
- Also can be shorted to just e



Demo

Flow Control

- There are 4 buttons in the debug bar that you can use to control the flow of the program while paused for a breakpoint:

Continue Step Over Step Into Step Out

Flow Control

- Continue: Un-pauses the program, allowing it to continue executing normally. It will continue forever or until it hits the next breakpoint.
- Step Over: Executes a line of code as if it were a blackbox. If the line you are at is a function call, it will not go inside that function. Instead it will just execute the function and keep going.
- Step In: Steps inside the function call of the current line in order to debug or examine its execution.
- Step Out: Use this if you want to leave the current function you are in. It will execute until it hits the first return statement. If you accidentally Step In, just use Step Out to get out.

Demo

What we learned

OOP

Abstraction
Classes Objects
Inheritance

Swift

Syntax Optionals
Types Conditionals
Optional Bindings

UIKit

UIView UILabel
UITextField
UIImageView
UITableView
UIViewController
UITableViewCell
UINavigationController

Programming Design Patterns

Delegation MVC
Target-Action

Xcode

Navigation Segues
Storyboards
Debugging
Auto Layout

Source Control

git github

Natural Next Topics

- **Networking:** Performing HTTP Requests with Apple provided classes (NSURLSession) and Closures. Parsing The JSON in the response.
- **More Interface Stuff:** Size Classes and iPad, Animations, Collection views, Programmatic Layout
- **Core Data:** Persistent Store, MOM, Managed Object Context, Core Data Stack
- **Read Apple's HIG** (Human Interface Guidelines)

Where to go from here

- Keep coding. If you stop, your coding dreams will die (jk…..not)
- If you enjoyed learning about iOS and this course, consider:
  - Taking the Dev Accelerator (Apply now!)
  - Building your own App and submitting it
- Consider that many of the apps on the app store are more simple than the app we created in this course
- Avoid the path of least resistance (just reading about programming instead of actually programming)
- Try another programming language. Once you have the basic concepts of objects, methods, variables, parameters, etc — learning a different programming language is considerably less difficult