iOS Dev Accelerator
Week 6 Day 3

- Unit Testing
- TDD
- NSFetchedResultsController

Unit Testing

Unit Testing

- In computer programming, unit testing is a software testing method by which individual units of source code are tested to determine whether they are fit to use.
- The necessity for writing tests for iOS applications is constantly debated
- Objective-C and Swift are both strongly, statically typed, which helps avoid a lot of bugs a more dynamic language, like javascript, need tests to avoid.
- Also Xcode has compile time code checking and wonderful autocomplete, text editors do not out of the box.
- Also a large part of an iOS app is the user interface, which is hard to test and usually not worth writing tests for.
- Alas, there a number of benefits to writing tests, and lots of companies use testing, so you need to know it.

Benefits of Unit Testing

- Unit testing helps you design your app during the early stages
- You can test out the functionality of your app's logic without having to rig up any dummy UI or navigating through your app's screens.
- Unit testing gives you confidence when adding new features. Unit tests can tell you which old features were broken by new code.
- Unit tests help projects with multiple developers keep everyone on the same page and helps each developer make changes independently with confidence of not breaking things.

Xcode and Testing

- So how do we write tests for our iOS apps?
- There a ton of third party testing frameworks and libraries available to us
- But theres a testing framework built right into Xcode (since Xcode 5) called XCTest

Test Class

- Best practice is to have a test class for every custom class in your app.
- A test class inherits from XCTestCase.
- The test class will automatically have a setup() and teardown() method (more on this in a few slides)
- After those come the actual Test methods.
- Test Methods must always return void, and start with the prefix ‘test’ and take in no parameters.
- Each method can be run individually, or the entire test class can be run, or all your test classes can be run in the entire project (considered your ‘test suite’).

Demo

FIRST

- Good unit tests follow FIRST
  - **F**ast: If your tests are slow, no one will run them
  - **I**solated: Any test should function properly when its run by itself, or before or after other tests, and tests should never share the same data or depend on each others results.
  - **R**epeatable: You should get the same results back when running a test multiple times against the same code base
  - **S**elf-verifying: The tests themselves should always reports success or failure, you should never have to check some file or log
  - **T**imely: While its great to write tests after you have already written the code, its ideal to write the test before writing the code, and the test will function as a specification for the functionality you are trying to develop. This is the basis of TDD, which we will talk about next.

CoreData and testing

- Core Data is an interesting case for testing
- Since working with core data usually means working with a database, to keep our tests **Isolated** we would have to be constantly clearing our database out to make sure each test gets a clean database.
- But doing the manual cleaning of the database wouldn't be very **Fast** since database operations are pretty slow, relatively speaking.
- The solution is to use a core data stack that is configured for in-memory use only, instead of a SQLite-backed store.

Demo

Test Methods

- Each method that begins with “test” is recognized as an actual test to run
- Running a test will evaluate any assertions within, and determine if the test should **Pass** or **Fail**

```
func testOnePlusOneEqualsTwo() {
    XCTAssertEqual(1 + 1, 2, "One plus one should equal two")
}
```

Asserting the truth

- Assertions are about what *you* expect to have. They are assertions about the state of a running program. They are facts you are trying to prove correct. It’s not a guess.

XCTAssertEqualObjects

XCTAssertNotEqualObjects

XCTAssertEqual

XCTAssertNotEqual

XCTAssertGreaterThan

XCTAssertGreaterThanOrEqual

XCTAssertLessThan

XCTAssertLessThanOrEqual

XCTAssertNil

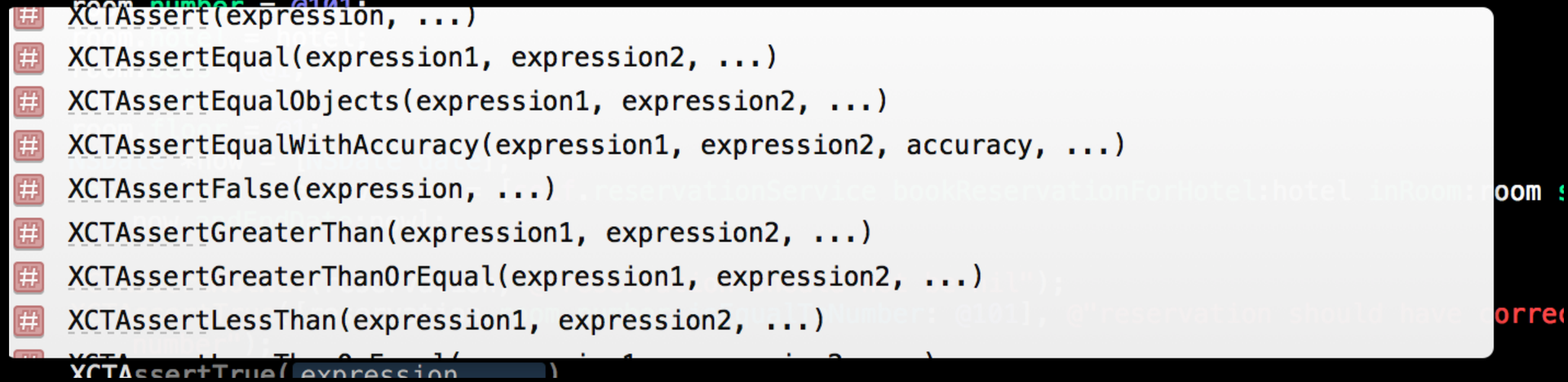
XCTAssertNotNil

XCTAssertTrue

XCTAssertFalse

Assertion Parameters

- The parameters of the assert macros are pretty simple.



- The expressions are just the things you are asserting about.
- The …. is an optional spot for a string that will print when the test fails.

Asserting the truth

- For Boolean tests, use `XCTAssertTrue` and `XCTAssertFalse`
- When testing for equality, use `XCTAssertEqual` or `XCTAssertNotEqual`
- Use `XCTAssertNil` or `XCTAssertNotNil` for asserting existence of nil
- Use `XCTFail` to purposely fail a test. Useful when a feature isn't implemented yet.

The structure of a test

- **Arrange:** Arrange all necessary preconditions and inputs
- **Act:** Act on the object or method under test
- **Assert:** Assert that the expected results have occurred

Demo

Writing good tests

- Remember FIRST, especially Fast and Isolated.
- Remember your are testing one unit, or one piece of code in your app.
- Certain parts of your code will depend on other parts of your code (dependencies), next week we will learn techniques to help write tests around these.
- For example, if you write a test for your JSON parsing, it shouldn't fail because the server gave you back incorrect JSON. Your test needs to isolate just the JSON parsing, if it fails it needs to be because your parsing code is wrong. Not because the internet was down or you were rate limited.

But what do I test?

- There are no hard rules on what you test.
- Here is a great general guideline: you need to test all the different paths of your application. (different paths are like hitting an if/switch/while loops)
- You should have 3-5 test for each public method. You don't need to write tests for private methods specifically. with the idea being your public methods eventually call your private methods, so they are ‘covered’ by tests.
- Basically, test your model!

What not to test

- Don’t test obvious stuff (like testing if 2+2 is 4, or a regular apple provided init)
- Don’t test Apple’s code. You should be testing your own custom logic (which will include Apple’s code, but also your own)
- Don’t test interface stuff (it takes too much time and not worth it)
- Don’t test outside resources like web servers
- You dont need to test view controllers specifically. Any logic inside of them should be handled by model & service classes. Thanks SRP!!!

Test until fear turns into boredom

TDD

Test Driven Development

- **Test First** : Developers are encouraged to write tests before writing code that will be tested.
- **Red, Green, Refactor**: Writing a failing test that encapsulates the desired behavior of the code you have not yet written
- **App Design**: Getting an idea of what the features of the app are how they all fit together, also known as System Metaphor.
- **Independent Tests**: Each class should have an associated unit test class that tests methods of that class. Being unit tests, we want to minimize the dependencies of the class we are testing . To help with this, we can use **mock objects**.

TDD Example

- The author of “Test-Driven iOS Development” uses a stack overflow client, somewhat similar to ours, to teach how TDD can work with an iOS app.
- Lets take a look at some of his examples

TDD

- He starts off with a Topic model class
- Before even creating the class, he creates a test class for it and writes a single test that wont even compile:

```
- (void)testThatTopicExists {
    Topic *newTopic = [[Topic alloc] init];
    STAssertNotNil(newTopic,
        @"should be able to create a Topic instance");
}
```

TDD

- He then creates the class and imports Topic.h into the test class and the test now passes!
- He then writes a test to test an initializer that takes in a string to be used as the topics name:

```
- (void)testThatTopicCanBeNamed {
    Topic *namedTopic = [[Topic alloc] initWithName: @"iPhone"];
    STAssertEqualObjects(namedTopic.name, @"iPhone",
        @"the Topic should have the name I gave it");
}
```

- He then adds the name property to the Topic class and also adds the custom init as well.
- Finally he writes a test to check if a tag can be set as well:

```
- (void)testThatTopicHasATag {
    Topic *taggedTopic = [[Topic alloc] initWithName: @"iPhone"
        tag: @"iphone"];
    STAssertEqualObjects(taggedTopic.tag, @"iphone",
        @"Topics need to have tags");
}
```

- And then writes the code to make that test pass

Red,Green,Refactor

- After taking those test from red to green, now is a great time to see if any refactoring can be done.
- The author concludes that he will never use the init with only the name parameter, so he deletes and refactors his test to only use the initWithName:andTag:
- He also creates a topic property that all the tests can use.

```
- (void)setUp {
    topic = [[Topic alloc] initWithName: @"iPhone" tag: @"iphone"];
}

- (void)tearDown {
    topic = nil;
}

- (void)testThatTopicExists {
    STAssertNotNil(topic, @"should be able to create a Topic instance");
}

- (void)testThatTopicCanBeNamed {
    STAssertEqualObjects(topic.name, @"iPhone",
        @"the Topic should have the name I gave it");
}

- (void)testThatTopicHasATag {
    STAssertEqualObjects(topic.tag, @"iphone",
        @"the Topic should have the tag I gave it");
}
```

Demo

setup() and tearDown()

- XCTestCase subclasses have two default methods
  - `setup()` is called before each test method in an XCTestCase is run
  - `tearDown()` is called after each test method finishes
- Useful for creating common objects that multiple test methods will use
- Common pattern in other unit testing frameworks too.

Demo

NSFetchedResultsController

Fetched Results Controller

- “You use a fetched results controller to efficiently manage the results returned from a Core Data fetch request to provide data for a UITableView object”

NSFetchedResultsController

- You typically have an NSFetchedResultsController property on the view controller that also manages the table view.
- You initialize the fetched results controller with 4 parameters:
  - **required: a fetch request**. must contain at least one sort descriptor for order.
  - **required: a managed object context**
  - optional: a key path that returns the section names. the controller spits the results based on this designated attribute.
  - optional: the name of the cache file the controller should use.

NSFetchedResultsController Modes

- A fetched Results controller has 3 modes:
  - No tracking: delegate is set to nil. the tableview just shows the data from the original fetch.
  - Memory-only tacking: delegate is non-nil and the file cache name is set to nil. Controller monitors objects in its result set and updates the tableview in response to changes.
  - Full persistent tracking: delegate and file cache name are non-nil. Controller tracks and displays changes and maintains a persistent cache of the results.

NSFetchedResultsController Workflow

1. Create a Fetch Request
2. Create the Fetched Results Controller
3. Tell the Fetched Results Controller to performFetch()

- In addition, you will need to modify your numberOfSections, numberOfRowsForSection, and cellForRow methods of your tableview datasource, and implement the NSFetchedResultsControllerDelegate protocol

TableView Datasource modifications

- numberOfSections: return fetchedResultsController.sections.count
- numberOfRows: first grab the section Info (of type NSFetchedResultsSectionInfo) from the fetchedResultsController.sections array. then return the sectionInfo.numberOfObjects
- cellForRow: grab the object for each cell by calling objectAtIndexPath on your fetchedResultsController

Demo

NSFetchedResultsController Section Grouping

- Grouping your results into sections is extremely easy and intuitive with the NSFetchedResultsController
- When you initialize your fetched results controller, one of the parameters is sectionNameKeyPath
- This is whatever attribute you want to create sections for. In our case, hotel name seems like a great fit. We could even do it by floor!
- Make sure to implement tableView:titleForHeaderInSection to give each section a proper name
- **One Gotcha: The first sort descriptor you apply to the initial fetch request must match the key path for the sections!**

Demo

NSFetchedResultsController Cache

- You probably noticed one of the parameters for the NSFetchedResultsController's initializer is a cache name
- The operations NSFetchedResultsController are actually pretty expensive for the device, especially the section separations!
- Using a cache helps alleviate that slowdown if the fetchedResultsController is launched with the same fetch more than once.
- Using a cache is as simple as passing in a string for he cache name with the initializer
- Keep in mind, if you change the fetch at all (entity, sort descriptors) you need to invalidate the cache by either calling deleteCacheWithName: or by reinitializing your fetched Results controller with a new cache name.

Demo

NSFetchedResultsController and monitoring for changes

- The real super power of the NSFetchedResultsController is how it can monitor for changes in the underlying data and automatically reload the interface appropriately
- This is done by the fetched results controller notifying its delegate of any changes that took place.

NSFetchedResultsControllerDelegate

- the fetched results controller notifies its delegate that the controller's fetched results have been changed due to an add, remove, move, or update operations.
- These are the delegate methods you will need to implement:
    - controllerWillChangeContent: - tell your tableview to beginUpdates.
    - controllerDidChangeSection:atIndex:forChangeType: - insert new sections or delete old sections in your tableview
    - controllerDidChangeObject:atIndexPath:ForChangeType: - insert/change/delete/move rows in your tableview
    - controllerDidChangeContent: - tell your tableview to endUpdates.

Demo