So, you're new at iOS Dev?

A Grab Bag of need to know topics.

Jeff Forbes - Mar. 14, 2013

Who is this for?

- Really new developers
    - (a hello world helps)
- Somewhat new developers
    - Lots of clarification on how Cocoa handles Memory Management, etc
- Anyone who want to Unit Test their ObjC code

Outline

- Memory Management in Cocoa
  - Traditional Reference Counting
  - ARC (Automated Reference Counting)
- Handling Asynchronous Networking
- Primer on Unit Testing using XCode/ SenTestingKit
- Random Q&A

Memory Manangent in Objective-C

(or, we're not in GC land anymore!)

Memory Management Options

- Traditional Retain/Release
  - Manual memory management using reference counting
  - The way to manage memory in ObjC for ~30 years
- Garbage Collection - OSX Only
  - Traditional generational garbage collector
  - Introduced in Objective-C 2.0 - OSX Leopard (10.5)
- Automated Reference Counting (ARC)
  - Works as described - automates the traditional retain/ release
  - Introduced in iOS5/OSX Lion

Memory Management Options

- Traditional Retain/Release
    - Manual memory management using reference counting
    - The way to manage memory in ObjC for ~30 years
- ~~Garbage Collection - OSX Only~~ DEPRECATED!!
    - ~~Traditional generational garbage collector~~
    - ~~Introduced in Objective-C 2.0 - OSX Leopard (10.5)~~
- Automated Reference Counting (ARC)
    - Works as described - automates the traditional retain/release
    - Introduced in iOS5/OSX Lion

Traditional Retain/ Release

Reference Counting

- Objective-C objects have a lifetime that is dictated by its retainCount
- Objects take ownership of objects (say, a contract) which cause the retainCount to increment
- If an object is finished with that object, it decrements this count
- When the count is 0, the system will free this memory

Retain/Release

- Retain an object - [object retain] (+1)
- Release an object - [object release] (-1)
- Autorelease an object - [object autorelease]
    - (we'll talk about this later)
- When [object retainCount] == 0, the object's dealloc method is called and subsequently freed

Convention

These return a +1 retainCount:

- `NSArray *array = [[NSArray alloc] init];`
- `NSArray *array2 = [array copy];`
- `NSArray *array3 = [NSArray new];`
  - (this is a macro for [[NSArray alloc] init])

Class methods generally return an autoreleased object:

- `NSArray *array4 = [NSArray arrayWithArray:array3]`

When to retain?

- Anything that you need to hang around for the lifetime of your object
- Generally, your class should never retain something more than once
- Failure to retain memory may result in a dangling pointer (non arc mode), which is the primary cause of EXC_BAD_ACCESS
- You should always trace the lifecycle of major objects to ensure they dealloc

Retain Cycles

- If you have 2 classes with dependencies on each other, and they both retain each other, you have a retain cycle
    - Both classes result with a retaincount of 1 and live forever on the island of misfit allocations
- This can be avoided by creating a weak reference to one of the 2 classes.
- Delegates are always weak for this reason!

Example

$$\begin{tikzcd}\text{Class A} \arrow[r] & \text{Class B} \arrow[r, shift right] & \text{Class C} \arrow[l, shift right]\end{tikzcd}$$

```
@interface ClassA
@property(retain,nonatomic) ClassB* classB;
@end

@interface ClassB
@property(retain,nonatomic) ClassC* classC;
@end

@interface ClassC
@property(retain,nonatomic) ClassB* classB;
@end
```

If ClassA releases, it should release ClassB. Unfortunately, ClassB doesn't release because ClassC holds a strong reference to it!

Example

$$\begin{tikzcd}\text{Class A} \arrow[r] & \text{Class B} \arrow[r, shift right] & \text{Class C} \arrow[l, dashed, shift right]\end{tikzcd}$$

```
@interface ClassA
@property(retain,nonatomic) ClassB* classB;
@end

@interface ClassB
@property(retain,nonatomic) ClassC* classC;
@end

@interface ClassC
@property(assign,nonatomic) ClassB* classB;
@end
```

Now, they will deallocate as expected!

...Autorelease Pool?

- NSAutoreleasePool is a convenient way to manage short lifetime allocations
  - When the pool is drained, all objects are sent a `release` message
- Convention: class methods that return objects are generally autoreleased
- Every thread has one - it's up to you to create one if you spawn a new thread
- The main thread drains/creates a pool every iteration of the event loop

https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSAutoreleasePool_Class/Reference/Reference.html

When do things get released?

```
@interface NSString
+ (NSString*)stringWithString:(NSString*)str;
@end

@implementation NSString
+ (NSString*)stringWithString:(NSString*)str
{
  NSString* retVal = [[NSString alloc]
  initWithString:str];
  return [str autorelease];
}

@implementation JeffsCoolClass
- (void)doSomeStringThings
{
  NSString* thing = [NSString
  stringWithString:@”Hello”];
  NSLog(@”%@”, thing);
}
@end
```

When do things get released?

```
@interface NSString
+ (NSString*)stringWithString:(NSString*)str;
@end

@implementation NSString
+ (NSString*)stringWithString:(NSString*)str
{
  NSString* retVal = [[NSString alloc]
  initWithString:str];
  return [str autorelease];
}

@implementation JeffsCoolClass
- (void)doSomeStringThings
{
  NSString* thing = [NSString
  stringWithString:@”Hello”];
  NSLog(@”%@”, thing);
}
@end
```

(Thing is released when doSomeStringThings drops out of scope)

Local Autoreleasepools

- Sometimes, you might do a bunch of operations that autorelease

```
NSArray* lines = [bigString componentsSeparatedByString:@”\n”];

NSMutableString* retVal = [NSMutableString string];
NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

for( int i = 0; i < [lines count]; i++ ){
  NSString newLine = [lines[i] uppercaseString];
  [retVal appendFormat:@”%@\n”, newLine];
  if( i % 50 == 0 ){
    [pool drain];
    pool = [[NSAutoreleasePool alloc] init];
  }
}
[pool drain];
```

Sort of clunky: Meet @autoreleasepool

- Once an autoreleasepool is drained, it’s a dead object and must be reallocated
- Sort of clunky -- with ARC and LLVM 4 @autoreleasepool was introduced
- Automatically handles pool for you

@autoreleasepool

```
NSArray* lines = [bigString componentsSeparatedByString:@”\n”];

NSMutableString* retVal = [NSMutableString string];
~~NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];~~

for( int i = 0; i < [lines count]; i++ ){
  NSString newLine = [lines[i] uppercaseString];
  [retVal appendFormat:@”%@\n”, newLine];
  ~~if( i % 50 == 0 ){~~
    ~~[pool drain];~~
    ~~pool = [[NSAutoreleasePool alloc] init];~~
  ~~}~~
}
~~[pool drain];~~
```

@autoreleasepool

```
NSArray* lines = [bigString componentsSeparatedByString:@”\n”];

NSMutableString* retVal = [NSMutableString string];

@autoreleasepool{
  for( int i = 0; i < [lines count]; i++ ){
    NSString newLine = [lines[i] uppercaseString];
    [retVal appendFormat:@”%@\n”, newLine];
  }
}
```

ARC (Automatic Reference Counting)

- ARC does static analysis on your code and inserts retain/release
- @autoreleasepool basically does a local ARC test
- Zeroing weak references (strong/weak properties) minimize programming errors
- Any project can be converted to ARC by going to XCode -> Edit -> Refactor -> Convert to ARC...

Questions?

Asynchronous Networking

- Probably the most difficult thing for a iOS dev to do is handle a lot of outgoing network calls
- Classic situation where operations require priority, cancellation, etc.
- There's a lot of good stuff out there to help with this kind of issue!

Dispatch Queues? Nope!

- Dispatch queues are less suited to finer grained control of operations
- Cannot reshuffle priority of scheduled items in a dispatch queue
- Cannot cancel a block once it is added
- Low visibility on how deep the queue goes, etc.

NSOperation/Queue

- Supports everything we need!
  - Concurrency
  - Cancellation
  - Priority
  - Completion blocks
- Downside: lots of boilerplate/scaffolding

https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/NSOperationQueue_class/Reference/Reference.html

AFNetworking

- AFNetworking solves all these problems for us in a very simple, straightforward fashion
- Allows us to easily create REST clients, all with the flexibility of using NSOperations and queues.
- Really active OSS project!

https://github.com/AFNetworking/AFNetworking

Simple Example Using AFNetworking

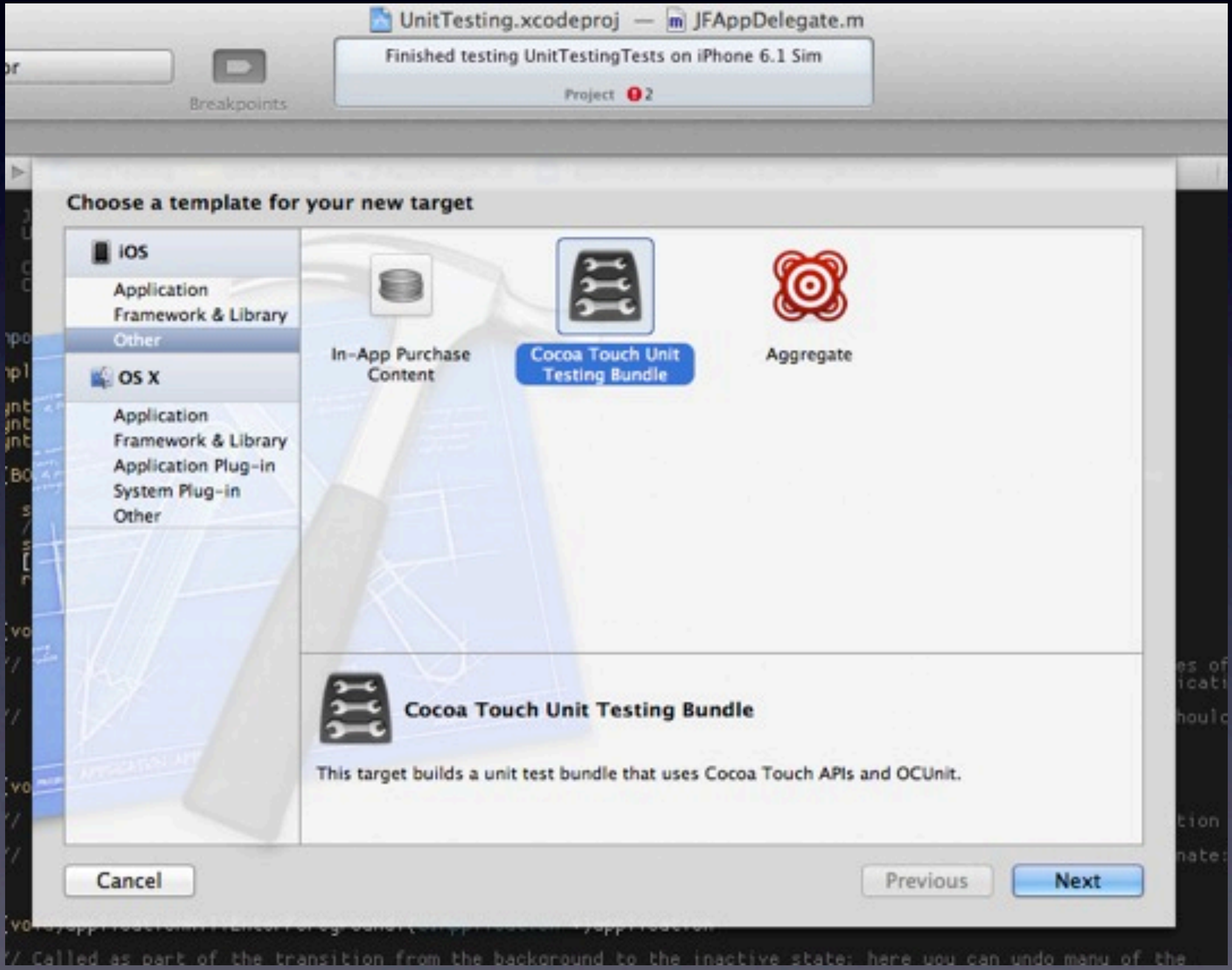
Unit Testing with XCode

Overview

- XCode provides built in tools to Unit Test your code
- OCUnit/SenTestingKit come with every installation
- Good starting point for testing, but there’s a lot more out there (KIF, Telerik, MonkeyTalk, etc)

Adding to your Project

- All you have to do to get started is go to File -> New Target... -> Other -> Cocoa Touch Unit Testing Bundle
- Then you can to go to Product -> Test to kick off the tests



Cocoa Touch Unit Testing Bundle

This target builds a unit test bundle that uses Cocoa Touch APIs and OCUnit.

Test Format

- All test case objects are a subclass of `SenTestCase`
- If they are added to the test target, XCode will automatically run the tests unless explicitly disabled
- All testing methods must start with `test`
- Setup/teardown methods optional

STAssertions

- Testing your expected results involves using the STAssert methods

```
STAssertEquals( value1, value2, faildesc... )

STAssertEqualObjects( value1, value2,
faildesc...)

STAssertNil( value, faildesc...)

STAssertTrue( value, faildesc... )

STFail(faildesc...)
```

http://developer.apple.com/library/mac/#documentation/developertools/Conceptual/UnitTesting/AB-Unit-Test_Result_Macro_Reference/result_macro_reference.html#//apple_ref/doc/uid/TP40002143-CH9-SW1

Example - Let’s write a test

More Stuff

- Mock Framework - OCMock
  - http://ocmock.org
- Integration Testing - KIF
  - https://github.com/square/KIF
- I wish I had more on this, but I ran out of time!

FIN.

- Questions?