HIT3329 / HIT8329
Creating Data Driven Mobile Applications

Lecture 2
The Objective-C Way

Presented by Paul Chapman

Adjunct Industry Fellow F-ICT
Director, Long Weekend LLC
Swinburne University of Technology

Last Lecture Reviewed

1. What This Course Is About
2. The Learning Curve
3. Objective-C
4. iOS Developer Tools

Any Questions?

Reminder: You Really Should Know These

Object Oriented Fundamentals:

- Classes
- Instances
- Differentiating class & instance methods
- Instance variables (ivars)
- Class inheritance
- Superclass
- Subclassing
- Interfaces/Protocols

What's On For Today?

1. NSObject
2. Properties
3. Memory Management

1.0 NSObject

- Base class for all ObjC object types
- Dynamic typing
- Methods for:
    - object creation → alloc/init
    - object cleanup → dealloc
    - print description → description

☆ Trivia Alert ☆
Apple purchased NeXTStep, Steve Job's post-Apple company, upon his return to Apple Computer in 1996. NeXTStep's technology is the basis for Mac OS X and therefore iOS. *NSObject* means *'NeXTStep Object'*.

1.1 Common Data Types in ObjC

1. Everything is an object (almost)
2. Primitive C data types are all supported
3. Objects are mutable or immutable (editable after creation or not)
4. Objects are mostly passed by pointer

1.2 Common Data Types in ObjC

- NSString
- NSMutableString
- NSArray
- NSMutableArray
- NSDictionary
- NSMutableDictionary
- NSInteger **Transparent wrapper for C integers*
- NSNumber **Container object for C primitive number types*

1.3 Wrapped Data Types

- Not all ObjC data types are rich objects
- Some wrap underlying C data types
- For example:
  - NSInteger is a wrapper for int/long
  - CGFloat is a wrapper for float
  - BOOL is a wrapper for bool

1.4 Benefit of Wrappers

Wrapped Data Types are *architecture independent* versions of common C types.

Therefore we don't have to worry about 32 or 64bit architectural concerns, which is very convenient.

1.5 Creating String Objects

```
/* Create NSString pointer */
NSString *myString;
myString = [[NSString alloc] init]; //empty string

/* Create Immutable String Object */
myString = [NSString stringWithString:@"Hello, World"];
myString = @"Hello, World"; // statically allocated

/* Create Immutable String Object */
NSInteger myNumber = 1;
myString = [NSString stringWithFormat:
                        @"Number is %d", myNumber];
```

1.6 Reassigning A String Pointer

You can't edit an immutable string, but you can assign a new object to the pointer.

```
/* Assign new object to existing pointer */
NSString *myString = @"My first string!";
myString = @"Now point to this string";
```

1.7 Silent Nils

Any message sent to a nil pointer is ignored:

```
NSMutableString *myStr = nil;
[myStr setString:@"Allo world!"];
```

The setString: message is ignored, doesn't crash!

1.8 Nil Initialisation

It's good practice to *nil initialise* pointers.

```
NSString *myStr = nil;
```

This prevents your code from crashing when trying to access uninitialised pointers.

1.9 Creating Number Objects

An *NSInteger* is a scalar integer, not an object

```
NSInteger myInt = 10;
int myInt       = 10; // equivalent on 32-bit systems
```

An *NSNumber* is an object for containing number data

```
NSNumber *myNum;
myNum = [NSNumber numberWithInt:myInt];
myNum = [NSNumber numberWithInt:42];
myNum = [NSNumber numberWithFloat:99.9];
```

Remember: *NSInteger* is not an NSObject and it's not passed as an object pointer

1.10 Number Object Get Methods

Here are some common get methods:

```
[myNum intValue];    // return numeric data as integer
[myNum stringValue]; // return numeric data as string
[myNum floatValue];  // return numeric data as float
```

1.11 NSNumber: Conclusions

THE GOOD

- *NSNumber* is a container for storing numeric data in an object

THE BAD

- Not bad, just misunderstood

THE UGLY

- Verbose, hard to read ... you'll get used to it!

1.12 Two Stage Object Creation

Design Pattern

- One Stage Creation (most languages)

```
SomeClass *myObject = SomeClass.new();
```

- Two Stage Creation (ObjC)

```
SomeClass *myObject = [[SomeClass alloc] init];
```

#1: alloc - allocates memory for the object
#2: init - creates the object

1.13 Why Two Stages?

- Allows for flexible memory allocation
- Avoids having too many initializers
- Simplifies creation and temporary instances

Ref: "Cocoa Design Patterns: Two-Stage Creation" by Erik M. Buck and Donald A. Yacktman
http://www.informit.com/articles/article.aspx?p=1398610

1.14 Creating NSArrays

```
// Create some objects to place in the NSArray
NSString *str1  = @"apple";
NSString *str2  = @"android";
NSNumber *num1 = [NSNumber numberWithInt:100];

// NSArray is a collection of object pointers
// NB: Array is 'nil' terminated
NSArray *myArray = [[NSArray alloc] initWithObjects: str1, str2, num1,
nil];

// A mutable (editable) array
NSMutableArray *myMutableArray = [[NSMutableArray alloc]
initWithObjects: str1, str2, num1, nil];
```

Nil Terminator

1.15 Rules About Arrays

Rule #1: NSArrays only store objects subclassing NSObject

Rule #2: C data types (scalar types) are not objects and cannot be stored in NSArrays (or any collection object, actually)

1.16 NSArray Convenience Methods

```
// More convenient NSArray creation
NSArray *myArray = [NSArray arrayWithObjects: str1, str2, num1, nil];

// More convenient NSMutableArray creation
NSMutableArray *myMArray = [NSMutableArray array];
[myMArray addObject:str1];
```

Any object can be stored in NSArrays, even other NSArrays

```
// Nested NSArray creation
NSArray *myNestedArray = [NSArray arrayWithObjects: myMArray, str1,
str2, num1, nil];
```

1.17 Non-Objects in NSArray FAIL

```
// Rookie Mistake: putting scalar NSInteger in NSArray
NSInteger myInt = 10;
[myMutableArray addObject:myInt]; // "EXEC_BAD_ACCESS"
```

This fails because NSInteger is not a real NSObject, it transparently wraps scalar integer data types.

1.18 Non-Objects in NSArrays SUCCESS

NSArrays store non-objects (e.g. integers, floats, pointers, structs) by wrapping in container objects.

```
NSNumber *myNum = [NSInteger numberWithInt:myInt];
[myMutableArray addObject:myNum]; // NICE!
```

Non-objects can also be wrapped using *NSValue*

1.19 Static & Dynamic Typing

- Static Typing
  All object types are known at compile time and checked by compiler

- Dynamic Typing
  All object types are NOT known at compile time so are *not* checked by compiler

1.20 The Anonymous Type: id

The 'id' anonymous data type makes dynamic typing possible in ObjC

Example 1: `id myObject;`

Defines a pointer to ANY ObjC object, but without compile time checking (it might point to garbage!)

Example 2: `NSObject *myObject2;`

Compiler will check it points to NSObject or object descended from NSObject.

Source: http://unixjunkie.blogspot.com/2008/03/id-vs-nsobject-vs-id.html

1.21 Anonymous Data Type - id

Example 3: **`id<NSObject> myObject2;`**

A pointer to the id type and compiler checks the object conforms to the NSObject protocol.

2.0 Declared Properties

What are Declared Properties?

- Declares the publicly-accessible instance variables of a class
- Introduced in ObjC 2.0
- Automagic creation of setters & getters

Why Declared Properties?

- Creating setters & getters manually is a chore
- In ObjC all object properties are accessible, use properties to define what is publicly available

2.1 Declaring Properties

In Header File (.h):

```
@interface SomeClass : NSObject {
    NSString *name;
    NSInteger yearStarted;
    NSString *description;
    NSString *someDelegateProperty;
@private
    BOOL *_somePrivateFlag;
}
@property (nonatomic, retain) NSString* name;
@property (nonatomic, assign) NSInteger yearStarted;
@property (nonatomic, readonly) NSString* description;
@property (readwrite, assign) NSString *aDelegateProp;

@end
```

2.2 Declaring Properties Simplified

In Header File (.h):

```
@interface SomeClass : NSObject {
    // Creating ivars for properties is redundant
    // The @property & @synthesize directives create
    // the corresponding ivars for us.
    //
@private
    BOOL *_somePrivateFlag;
}
@property (nonatomic, retain) NSString* name;
@property (nonatomic, assign) NSInteger yearStarted;
@property (nonatomic, readonly) NSString* description;
@property (readwrite, assign) NSString *aDelegateProp;

@end
```

2.3 Generating Accessors

In Implementation File (.m):

```
@implementation SomeClass
@synthesize name;               // one per line OK
@synthesize description;      // multiple lines OK
@synthesize someDelegateProp; // order doesn't matter
@synthesize yearStarted;

@end
```

The @synthesize directives generate setters and getters corresponding to the properties declared in the .h file.

Further Reading: *http://www.theocacao.com/document.page/510*

2.4 Accessors = Syntactic Sugar

The self. accessors are syntactic sugar. They're equivalent to sending an object a set/get message.

```
// uses "self." setter
self.myStringProperty = @"foo";
// sends a "set" message
[self setMyStringProperty: @"foo"];
```

2.5 Accessors = Syntactic Sugar

These lines pre-compile to identical code. However dot notation is generally easier to read.

```
// Compare these ...
self.window.rootViewController = self.myViewCtrl;

// Messages can be even more dense than this!
[[self window] setRootViewController:[self myViewCtrl]];
```

2.6 Setter & Getter Message Syntax

Like Java, setters methods in ObjC include the verb 'set':

```
// "set" myStringProperty
[self setMyStringProperty: @"bar"];
```

Unlike Java, getter methods do not include the 'get' verb. You simply call the property name:

```
// "get" myStringProperty
[self myStringProperty];
```

NB: ObjC and Cocoa have different naming conventions for methods & properties. Learn to use them appropriately.

2.7 Declared Property Parameters

We will review retain/assign/copy after completing the next section.

@property (<parameters>) <type> <name>;

Parameters control how properties are accessed and assigned.

assign, retain, copy

- **retain** - retains a pointer to assigned object
- **assign** - assign value to ivar directly (commonly used with primitive/scalar data types, delegates or objects owned by others)
- **copy** - make a copy of the object assigned

atomic, nonatomic

- **atomic** (default) - thread safe (rarely needed)
- **nonatomic -** usually use this one

readwrite, readonly

- **readwrite** (default) - override with 'readonly'
- **readonly** - prevents assignment (does not synthesize setter method)

See: http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocProperties.html

Source: "The Internet"

3.0 iOS Memory Management

Manual Memory Management

- Memory management is done by ***reference counting***
- Simpler than C (no malloc'ing)

Why No Garbage Collection?

- Mobile devices have limited CPU/memory
- GC can be resource intensive and slow the phone
- Manual memory management is a compromise

3.1 The "Impossible" Phone

When the iPhone was announced, Research In Motion - creators of Blackberry, thought such a device was "impossible".

According to sources, they believed Apple "was lying" and such a phone would not have the battery life to do what they claimed.

When they finally got hold of an iPhone, they opened it and found a battery with a tiny circuit board attached!

Source: http://www.redmondpie.com/blackberry-maker-rim-thought-apple-was-lying-about-iphone-in-2007/

3.2 Rules of Memory Management

1. You own any object you create
2. You take ownership of an object using retain
3. When no longer needed, you must *relinquish ownership* of an object you own (release)
4. You must not *relinquish ownership* objects you do not own

Source: http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html

3.3 Reference Counting

The Retain, Release Cycle

- Each object has a **retain count**
- Tracks how many other objects expect it to *stick around*
- Call object's **retain** method to say *"hey, stick around"*
- Call object's **release** method to say *"I'm done with you"*
- When an object's **retain count** equals 0, it is deallocated from memory

NOTE: Fully released objects not be immediately deallocated!

Confused?
Here's another way of looking at it ...

3.4 Canine Memory Management I

Your neighbour asks you to take his dog **Fluffy** for a walk!

3.4 Canine Memory Management I

Your other neighbour asks you to take ***Rambo*** for a walk!

3.5 Canine Memory Management II

First of All

- Your '*object*' is the dog
- Calling **retain** puts a leash on the dog
- Calling **release** takes a leash off

Furthermore

- You can have as many leashes on the dog as you want
- You need at least one leash to ensure he doesn't run away (otherwise you lose your object = crash!)
- When you want to give the dog back to its owner, take all the leashes off (otherwise you will leak memory)

Source: "Cocoa programming for Mac OS X", Aaron Hillegass, Addison-Wesley Professional (2004)

3.6 Canine Memory Management III

	Retain Count
● Neighbour brings you his new dog (alloc/init) handing you the leash already on him	1
● Whilst walking, 2 other people become interested in your dog, they each put leash on him (retain)	3
● The dog is naughty so you decide to take the leash off and let him run away (release)	2
● There are still two leashes on him, so he doesn't go anywhere (but you don't own him)	2
● You later forgive him and put a leash on (retain)	3
● The other people remove their leashes (release x 2)	1
● Later you give him back to his owner (release)	0

Retain count reaches zero
object is deallocated

3.7 Intermittent Crashes

"But It Only Leaks Sometimes!"

If you remove YOUR leash, the dog may or may not wander off when needed (get deallocated). Sometimes memory leaks & crashes seem to "come and go" because the object is deallocated at different times. **But the problem is still there!** Only by putting a leash on him can you ensure he won't wander off unexpectedly.

ObjC Proverb: *Bad memory management is like a torn umbrella in the desert, it only leaks sometimes!*

3.8 Retain / Release Example

Create an object, assign it to a property and then release it

```
// tArray pointer points to new object
// alloc/init returns object, retainCount == 1
NSMutableArray *tArray = [[NSMutableArray alloc] init];

// 'self.' setter syntax, retainCount +1
[self setMyArray: tArray];

// Send release message to our object, retainCount -1
[tArray release];
```

The local var tmpArray no longer owns the array object. myArray now owns it and is responsible for cleaning up.

3.9 Retain/Release with Dot Notation

1. Same as before, looks nicer with dot notation!

```
// tArray pointer points to new object
// alloc/init returns an object with retainCount==1
NSMutableArray *tArray = [[NSMutableArray alloc] init];

// 'self.' setter syntax, retainCount +1
self.myArray = tArray;

// Send release message to our object, retainCount -1
[tArray release];
```

3.10 Retain/Release with Dot Notation

(continued)

// IMPORTANT: The self. notation implicitly retains the // assigned object. You must remove this "leash" later // in the class' dealloc method

2. Same again, more convenient and autoreleased

```
self.myArray = [[[NSMutableArray alloc] init] autorelease];
```

3. Same again, shorter and still autoreleased

```
self.myArray = [NSMutableArray array];
```

3.11 Declared Properties - Retain

Remember this from a few slides back?

assign, retain, copy

- retain - retains a pointer to assigned object
- assign - assign value to ivar directly
- copy - make a copy of the object assigned

```
@property (nonatomic, retain) NSArray *myArray;
```

This declared property uses the retain parameter.

3.12 Generated Setter Code

The corresponding 'retain' setter code generated at compile time looks like this:

```
-(void) setMyArray:(NSArray *)value
{
  [value retain];       // Object passed is retained +1
  [__myArray release]; // Old object is released -1
  __myArray = value;   // __myArray = compiler created ivar
}
```

3.13 Retain, Release Idiom #1

Create, Use, Release

```
1:  NSObject* object = [[NSObject alloc] init];
2:  // use object //
3:  [object release];
```

1: Creates retained object (alloc/init returns retained obj)
2: Do some stuff with it
3: Release object, retainCount is zeroed, object is deallocated

3.14 Retain, Release Idiom #2

Create, Assign Ownership, Relinquish Ownership

```
1:  NSObject* object = [[NSObject alloc] init];
2:  [array addObject:object];
3:  [object release];
```

1: Creates retained object

2: Add to an array, the array retains object

3: Release object, retainCount is not zero, but we are no longer owner, the array is the owner

3.15 Retain, Release Idiom #3

Using dot notation with alloc/init without leaking.

```
// Assuming this property declaration
@property (nonatomic, retain) NSObject *myProperty;

// LEAKY: alloc/init returns object with retainCount=1
// then self.myProperty "over retains" the object
self.myProperty = [[NSObject alloc] init];

// FIX #1: Use a temporary object pointer
NSObject *tmpObj = [[[NSObject alloc] init];
self.myProperty = tmpObj;
[tmpObj release];

// FIX #2: Use autorelease, a nice one liner
self.myProperty = [[[NSObject alloc] init] autorelease];
```

3.16 Retain, Release Idiom #4

Taking ownership of an autoreleased object.

```
// LEAKS: 'myProperty' has not taken ownership of the
// object referenced so it will go out of scope (leak)
myProperty = [NSString stringWithFormat:@"Hi"];

// FIX: Use synthesized setter, assumes this declaration
// @property (nonatomic,retain) NSString *myProperty;
self.myProperty = [NSString stringWithFormat:@"Hi"];
```

3.17 Dealloc

When your object owns other objects, you must free them when your object is deallocated. Do this in the dealloc method.

```
-(void) dealloc
{
    //release your properties
    [myVar release];
    [myString release];
    [myArray release];

    // call super class' dealloc method
    [super dealloc];
}
```

3.18 Release Using Dot Notation

Since ObjC 2.0 you can do this with synthesized properties.

```
self.myVar = nil;
self.myString = nil;
```

Syntactic sugar again, releases the object + nils out the pointer

3.19 Dot Notation in Dealloc

```
self.myVar = nil;
```

1. Releases the object
2. Nils out the pointer
3. Any messages sent to via the pointer are ignored

Conclusion: Nulling out your properties hides errors.
Is this better or not?

3.20 Autorelease

Autorelease Pools store *temporary objects* to be released "later"

- *Autoreleased* objects usually deallocated at end of event loop
- *Released* objects are usually deallocated immediately
- Add objects to autorelease pool by calling *autorelease*

```
[[[NSMutableArray alloc] init] autorelease];
```

- Autorelease is not Garbage Collection
- Not good for creating large numbers of objects
  (ie. might run out of memory before pool is drained)

Note:

1. Cocoa factory methods return autoreleased objects
2. User factory methods should also return autoreleased objects

3.21 Automatic Reference Counting (ARC)

- Introduced in iOS 5.0 (in beta as of July 2011)
- Choose either ARC or manual memory management

Benefits of ARC

- retain/release calls are automagically inserted at compile time
- Not a garbage collector - no slowdown in performance

Costs of ARC

- Doesn't work on iOS 3.x*
- Need to update existing code**
- Not yet widely tested "in the wild"

** 95%+ of devices are already on iOS4 or higher as of June 2011*
*** There is a tool for migrating existing code in Xcode*

Ref: *http://longweekendmobile.com/2011/09/07/objc-automatic-reference-counting-in-xcode-explained/*

3.22 So How Should I Manage Memory?

Why should we learn manual memory management?

- ARC is new and yet to be widely tested
- ARC is not garbage collection!
- Reference counting underpins ARC
- Make sure you can properly use manual memory management (reference counting)

My Conclusion: In some cases you know better what you want than an automated schema. On a constrained device you should know what's going on "under the hood". If you don't, your app's performance could suffer.

End of Lecture 2

1. Lab Work
2. Assignments
3. Get Funky