Modern Objective-C: Recent Changes in API and Convention

Jeff Forbes - 02/06/13

Welcome

- Who's this talk for?
  - New ObjC developers
  - Old ObjC developers (whom haven't touched the language in a while)
- Assumption: completed a 'Hello World' application

Overview

- Style and Convention Changes
  - Scoping your variables
  - Automatic Synthesis
  - ObjC Literals/Subscripting
- Grand Central Dispatch (GCD)

Style and Convention Changes

OSX 10.1

iOS6

Objective C is Old

- Using the base of C and principles SmallTalk, a language was born
- ObjC is a relatively thin wrapper around C
  - Foundation (ObjC) vs CoreFoundation (C)
  - Toll-Free bridging (CFString = NSString)
- All objects are essentially a objc_struct

Creating an Interface

```
@interface Person : NSObject {
  NSString* firstName;
  NSString* lastName;
  int gender;
}
@end
```

Getters/Setters?

Creating an Interface

```
@interface Person : NSObject {
  NSString* firstName;
  NSString* lastName;
  int gender;
}
```

```
- (void)setFirstName:(NSString*)aFirstName;
- (NSString*)firstName;
- (void)setLastName:(NSString*)aLastName;
- (NSString*)lastName;
- (void)setGender:(int)gender;
- (int)gender;

@end
```

Creating an Interface

```
@interface Person : NSObject {
  NSString* firstName;
  NSString* lastName;
  int gender;
}
```

```
- (void)setFirstName:(NSString*)aFirstName;
- (NSString*)firstName;
- (void)setLastName:(NSString*)aLastName;
- (NSString*)lastName;
- (void)setGender:(int)gender;
- (int)gender;

@end
```

This totally sucks - this isn't Java so we can do better!

@property

@property

- A way to automate the process of creating getters/setters
- Introduced in Objective-C 2.0 (otherwise known as The Modern Runtime)
- Even more useful now (with LLVM 4.2)

Fixing our Interface

```
@interface Person : NSObject {
  NSString* firstName;
  NSString* lastName;
  int gender;
}
```

```
- (void)setFirstName:(NSString*)aFirstName;
- (NSString*)firstName;
- (void)setLastName:(NSString*)aLastName;
- (NSString*)lastName;
- (void)setGender:(int)gender;
- (int)gender;

@end
```

Fixing our Interface

```
@interface Person : NSObject {
  NSString* firstName;
  NSString* lastName;
  int gender;
}

@property(retain,nonatomic) NSString* firstName;
@property(retain,nonatomic) NSString* lastName;
@property(assign,nonatomic) int gender;

@end


@implementation Person
@synthesize firstName, lastName, gender;
@end
```

Fixing our Interface

```
@interface Person : NSObject {
  NSString* firstName;
  NSString* lastName;
  int gender;
}

@property(retain,nonatomic) NSString* firstName;
@property(retain,nonatomic) NSString* lastName;
@property(assign,nonatomic) int gender;

@end


@implementation Person
@synthesize firstName, lastName, gender;
@end
```

Fixing our Interface

```
@interface Person : NSObject

@property(retain,nonatomic) NSString* firstName;
@property(retain,nonatomic) NSString* lastName;
@property(assign,nonatomic) int gender;

@end

@implementation Person
@synthesize firstName=_firstName
@synthesize lastName=_lastName
@synthesize gender=_gender;
@end
```

underscore indicates instance variable

Fixing our Interface

```
@interface Person : NSObject

@property(retain,nonatomic) NSString* firstName;
@property(retain,nonatomic) NSString* lastName;
@property(assign,nonatomic) int gender;

@end

@implementation Person
@synthesize firstName=_firstName
@synthesize lastName=_lastName
@synthesize gender=_gender;
@end
```

LLVM is smart enough to now create your ivar and know that you want to prefix it with _!

Fixing our Interface

```
@interface Person : NSObject

@property(retain,nonatomic) NSString* firstName;
@property(retain,nonatomic) NSString* lastName;
@property(assign,nonatomic) int gender;

@end

@implementation Person
@end
```

What about this retain/nonatomic stuff?

First thing: everyone clear on obj-c memory management?

@property keywords

- **retain**
  - setThing: will call release on current var, retain on the incoming
- **assign**
  - No retain - useful to avoid retain cycles (also for primitives like int, float, etc)
- **copy**
  - performs [obj copy] (assumes a +1 retain count)
- **readwrite** - requires a getter/setter (attr_accessor)
- **readonly** - requires a getter (attr_reader)

@property keywords (cont)

- **nonatomic**
  - no mutex on get/set
- **atomic**
  - uses a spinlock (mutex) to protect access to variable
  - NOT THREAD SAFE unless the variable is immutable
- **strong** - same as retain (used in ARC mode, use this instead of retain)
- **weak** - same as assign EXCEPT with zeroing reference when object is deallocated (also, use this in ARC mode)

Default: (retain/assign,readwrite,atomic)

Public/Private Variables

```
@interface PeopleListController :
UIViewController<UITableViewDataSource,
UITableViewDelegate>
{
  IBOutlet UITableView* _tableView;
  NSFetchedResultsController* _resultsController;
  NSManagedObjectContext* _context;
}
@end
```

Public/Private Variables

Old Way

```
@interface PeopleListController :
UIViewController<UITableViewDataSource,
UITableViewDelegate>
{
  IBOutlet UITableView*  tableView;
  @private
  NSFetchedResultsController* _resultsController;
  NSManagedObjectContext* _context;
}
@end
```

Any access of these variables outside of this class will throw a compiler error. Default is @protected

Public/Private Variables

What's the Problem?

- People look at your headers to see what functionality you expose
- Seeing those ivars there adds unnecessary noise and impacts readability

Solution: remove them from your headers!

PeopleListController.h

```
@interface PeopleListController :
UIViewController<UITableViewDataSource,
UITableViewDelegate>
@property(retain,nonatomic) IBOutlet UITableView*
tableView;
@end
```

(made IBOutlet a property, removed private variables)

PeopleListController.m

```
@interface PeopleListController()
@property(retain,nonatomic) NSFetchedResultsController*
resultsController;
@property(retain,nonatomic) NSManagedObjectContext*
context;
@end
```

```
@implementation PeopleListController
   //your implementation goes here
@end
```

Now, our instance variables are nicely protected!

PeopleListController.m

```
@interface PeopleListController()
@property(retain,nonatomic) NSFetchedResultsController*
resultsController;
@property(retain,nonatomic) NSManagedObjectContext*
context;
@end

@implementation PeopleListController
   //your implementation goes here
@end
```

But what is this syntax?

ObjC Categories

- Analogue - Ruby Modules
- Allow you to add code to any class at runtime
- Beware symbol clashes (cannot have 2 methods named the same!)
- Code on previous slide was a special category called a ‘class extension’

Example - Map

```
//NSString+Map.h
typedef (id)(^ElementOperation)(id);
@interface NSString(Map)
- (NSArray*)map:(ElementOperation)opBlock;
@end

//NSString+Map.m
@implementation NSString(Map)
- (NSArray*)map:(ElementOperation)opBlock
{
    NSMutableArray* retVal = [NSMutableArray array];
    for( id obj in self ){
        [retVal addObject:opBlock(obj)];
    }
    return retVal;
}
@end
```

Example - Map

```
//NSString+Map.h
typedef (id)(^ElementOperation)(id);
@interface NSString(Map)
- (NSArray*)map:(ElementOperation)opBlock;
@end

//NSString+Map.m
@implementation NSString(Map)
- (NSArray*)map
{
    NSMutableArray* retVal = [NSMutableArray array];
    for( id obj in self ){
        [retVal addObject:opBlock(obj)];
    }
    return retVal;
}
@end
```

Moral: Objective C is very flexible!

Literals - the best thing

How it used to be

- Create a number object
  - `[NSNumber numberWithInt:1];`
- Create an array
  - `[NSArray arrayWithObjects: @”a”, @”b”, @”c”, nil]`
- Create a Dictionary
  - `[NSDictionary dictionaryWithObjectsAndKeys: @”obj1”, @”key1”, @”obj2”, @”key2”, nil];`

How it used to be

- Create a number object
  - `[NSNumber numberWithInt:1];`
- Create an array
  - `[NSArray arrayWithObjects: @”a”, @”b”, @”c”, nil]`
- Create a Dictionary
  - `[NSDictionary dictionaryWithObjectsAndKeys: @”obj1”, @”key1”, @”obj2”, @”key2”, nil];`

THIS IS
INFURIATING

"I'm sorry. Mr. Broadbank seems to have stepped away from his desk."

Now, less painful!

- Create a number object
  - `@1`
- Create an array
  - `@[@"a", @"b", @"c"]`
- Create a Dictionary
  - `@{@"key1" : @"obj1", @"key2" : @"obj2"}`

These expand out at compile time to perform the exact same functionality

Subscripting!

- LLVM 4.2 also introduces subscripting

NSDictionary

```
NSDictionary* dict =
@{@”key1” : @1, @”key2” : @2 };

NSLog(@”%@”, dict[@”key1”];

<prints 1>
```

NSArray

```
NSArray* array = @[ @1, @2, @3];

NSLog(@”%@”, array[0];

<prints 1>
```

You can add this support to any classes you want!

Methods you have to implement

```
// To add array style subscripting:
- (void)setObject:(id)obj atIndexedSubscript:
(NSUInteger)idx; // setter
- (id)objectAtIndexedSubscript:
(NSUInteger)idx; // getter

// To add dictionary style subscripting
- (void)setObject:(id)obj forKeyedSubscript:(id
<NSCopying>)key; // setter
- (id)objectForKeyedSubscript:(id)key;  // getter
```

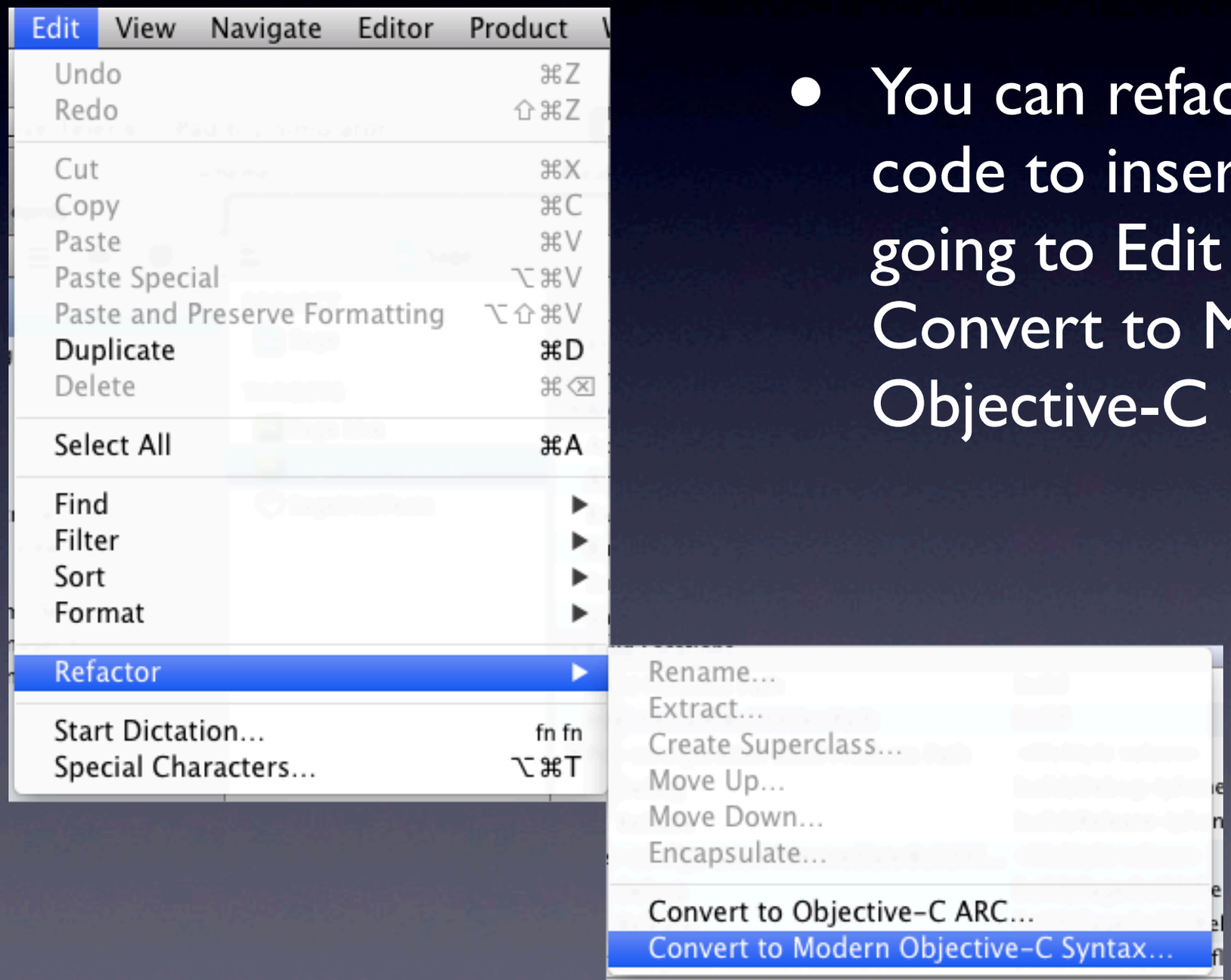
NSDictionary Subclass

Don't allow NSNull to be set on a dictionary! A lot of JSON parsers do this and it's super annoying

```
@interface JFDictionary : NSDictionary
@end
```

```
@implementation JFDictionary
- (void)setObject:(id)obj forKeyedSubscript:(id
<NSCopying>)key
{
  if([obj isEqual:[NSNull null]]) return;
}
- (id)objectForKeyedSubscript:(id)key {
  return [super objectForKeyedSubscript:key];
}
@end
```

XCode will help you



- You can refactor all old code to insert literals by going to Edit -> Refactor -> Convert to Modern Objective-C Syntax

Grand Central Dispatch (GCD)

What is GCD?

- Addition of closures/blocks to C
  - Syntax: ^{ // do some code }
- Defines structures for doing tasks asynchronously using queues (dispatch queues)
- Super awesome

Dispatch Queues

- Serial Queue (iOS5 and above)
  - All blocks will execute serially
  - Good for supporting multithreading (instead of mutexes)
- Global Concurrent Queues
  - Queues that execute blocks on *n* threads with a given priority
- Concurrent Queues (iOS6 and above)
  - Works like global concurrent queues, except user defined

Things to know

- Dispatch queues can be paused
- Difficult to cancel enqueued blocks
  - Cannot inspect count of blocks on the queue
  - Cannot cancel individual blocks without adding additional infrastructure
- Never use a NSThread inside a block. Ever.

Dispatching Blocks

```
- (void)doSomeStuffToArray:(NSArray*)array
{
  dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE
  _PRIORITY_DEFAULT, 0), ^{
    for( NSObject* obj in array ){
      NSLog(@"%@", obj);
    }
  });
  //method returns before block executes
}

- (void)doSomeStuffToArraySynchronously:(NSArray*)array
{
  dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_
  PRIORITY_DEFAULT, 0), ^{
    for( NSObject* obj in array ){
      NSLog(@"%@", obj);
    }
  });
  //call is synchronous, will block until complete
}
```

Completion Blocks

- A lot of methods you want to execute async and then execute code immediately after it finishes
  - Generally, you would assign a delegate and wait for the data to come back
- Blocks make this a lot easier
- Can define your own block format which makes doing completion blocks clearer

Defining a Custom Block

```
typedef void (^OperationCompletionBlock)(id data, NSError* err);
```

return value　　block name　　block arguments

Defining a Custom Block

```
typedef void (^OperationCompletionBlock)(id data, NSError* err);
```

return value　　block name　　block arguments

Using block declaration

```
- (void)getDataAtURL:(NSURL*)url
          completion:(OperationCompletionBlock)completion;
```

Demo

- Let's implement that.

dispatch_once

- A lot of times you want to set a global variable, but you want to make sure it only happens once.
- Useful for singleton pattern

dispatch_once

```
+ (id)sharedManager
{
    static DataManager* __sharedManager = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _sharedManager = [[self alloc] init];
    });
}
```

Guaranteed to execute only once!

Accessing a protected resource

- Use serial queues
- Demo code!

FIN