iPhone Programming
CMSC 498I – Fall 2010

Objective-C
Adam Porter
Topics

- Objective–C in more detail
  - ObjC types
  - Properties (ObjC 2.0)
  - Memory management
  - Defining Classes
  - Runtime Features
  - Advanced Topics
Types

• Any Basic C Type
  - int, unsigned
  - char, char *

• BOOL – valid values are YES, NO
  - Underlying type is a char

• NSInteger
  - A C typedef: underlying type is a 32 or 64 bit integer
  - so APIs can use system appropriate int storage
Base Types

• SEL
  - Encoding of an Obj-C method name
  - e.g. SEL cmd = @selector(setStringValue:);

// easily tell all my subviews to display
NSArray *mySubviews = // get the list of subviews
[mySubviews performSelector: @selector(setNeedsDisplay)];

// check to see if an object implements a method
if ([anObj respondsToSelector: @selector(delayedUpdate:)]) {
  [anObj performSelector: @selector(delayedUpdate:)
      withObject: mySubviews
      afterDelay: 1.0f /* seconds */];
}
Tangent...

- Previous slide: “respondToSelector”

- When would you not know if an object doesn’t?
  - May have an untyped object
    - Allows you to not care about the object type, just about it’s ability
    - Allows for “optional” methods – delegation...
  - Compatibility
    - In a single source base, check if a feature exists
      - E.g. Your code runs on 2.0, and 3.0 and a certain method only exists on 3.0...
Base Types

• SEL is just a ‘char*’

(gdb) p (char *)@selector(setNeedsDisplay)
$1 = 0x7fff886d1378 “setNeedsDisplay”

( can be useful to know this when debugging...)
Base Types

• NSObject
  - Base class of all Cocoa framework objects
  - Also a protocol (Objective-C itself does not require NSObject)

• id
  - Essentially a pointer to an arbitrary object, like a (void*)
  - Dynamic typing
  - Runtime dispatch will figure out what type it is...
About @

- Special token in Objective-C
  - @interface, @implementation
  - @"This is a constant string"
  - @property, @synthesize
  - @try, @catch, @finally
  - @protocol
  - @selector
  - @class
Hidden Parameters

• Undeclared items you can use in a method

• id self
  - The object receiving the ObjC message
  - Like “this” in C++ or Java

• SEL _cmd
  - The encoding of the received method’s name

• Side note:
  - Messages actually pass along these two items as parameters

  • typedef id (*IMP)(id, SEL, ...);
Naming Conventions

• Camel case

- (void)doSomethingWithFoo:(int)foo andBar:(int)bar;

-NSNotificationCenter

- UITableView

• Prefixes used to delineate namespaces

- UITableView – UIKit Framework

- NSString – Foundation framework

- So, don’t use NS, UI in your code...
ObjC 2.0

- Declared Properties
- Performance improvements
- Fast iteration constructs (in later lectures)
- Garbage collection (not available on iPhone OS…)

Tuesday, September 7, 2010
Properties

• A property is an attribute of an instance accessible using getter and setters
  - Encourages encapsulation
    • Access through methods instead of direct instance variable

• ObjC 2.0 – Properties
  - Syntax for declaring properties with explicit behaviors
  - Compiler can create the implementation for you
  - Can access property using convenient “dot” syntax
Method Declarations

- Example: Class representing information about a person
- Accessors declared as methods

@interface Person : NSObject {
  NSString *name;
}

- (NSString *)name;
- (void)setName:(NSString *)name;
@end
Property Declaration

- Accessors declared a ObjC 2.0 properties
  - Specify property type
  - Specify accessors behave in the “attributes” clause

```
@interface Person : NSObject {
    NSString *name;
}
@property (/* attributes */) NSString *name;
@end
```
Property Attributes

@property (<attributes>) NSString *name;

• Writability
  - readwrite (default), readonly

• Setter semantics (details given later)
  - assign (default), retain, copy

• Atomicity
  - atomic(default), nonatomic
### Public and Private

- Declare public properties and attributes in the header

```cpp
// Person.h
@interface Person : NSObject { }
@property (copy, readonly) NSString *socialSecurityID;
@end
```

- Declare private attributes elsewhere

  - For example, you can declare different attributes for the same properties by redeclaring inside a “unnamed category”

```cpp
// Person.m
@interface Person ()
@property (copy, readwrite) NSString *socialSecurityID;
@end
```
Using Properties

• Can use messaging, or “dot” syntax

• Setting a property

```objective-c
person.name = @”Fred”;
--- OR ---
[person setName: @”Fred”];
```

• Getting a property

```objective-c
NSString *name = person.name;
--- OR ---
NSString *name = [person name];
```
Property Implementation

• Tell compiler who will be implementing the methods

• @synthesize someProperty ...  
  - Tells compiler to generate getter, and setter methods  
  - Creates setter method named: -setSomeProperty:
  - Creates getter method named: -someProperty;

• @synthesize someProperty = someProperty_;  
  - Tells compiler to use ‘someProperty_’ for the storage

• @dynamic  
  - Tells compiler you will provide method implementations later…
Property Implementation

• Example

```Objective-C
@interface Person : NSObject {
    NSString *name_
}
@property (copy, readwrite) NSString *name;
@end
```

• Tell compiler to generate the implementation

```Objective-C
@implementation Person
@synthesize name = name_;  
@end
```
Property Implementation

- `@synthesize`
  - Methods generated only if you don’t provide one

- Example
  - `setName:` is used instead of auto-generated version setter
  - getter automatically generated still

```swift
@implementation Person
@synthesize name = name_;

-(void)setName:(NSString *)name {
    ....
}
@end
```
Property Implementation

• Implementing a property setter yourself

```c
- (void)setName:(NSString *)name {
    if (name_ != name) {
        [name_ release];
        name_ = [name retain];
    }
}
```

• Implementing the getter yourself

```c
- (NSString *)name {
    return name_;}
```
Opinions About Properties

• Use properties whenever possible in declarations
  - Even if not using `@synthesize` to implement
  - Attribute annotations convey useful information

• Avoid using “dot” notation
  - Is “object.result = 42” a simple assignment or method call?
    • Hides the amount of ObjC messaging going on…
    • Can mask performance issues
  - Searching for call sites requires you to run two searches
    “.someProperty”, and “someProperty]”
Memory Management

- You should deallocate objects that are no longer in use
  - But knowing when they are no longer in use can be difficult

- Solutions
  - Don’t worry, be happy!
  - Garbage collection
  - Reference counting
Memory Management

• Garbage collection
  - Automatic reference tracking
  - Safe, but not user-controllable

• Reference counting
  - Manual reference tracking
  - More risky, but user-controllable

• iPhone – reference counting
  - Mac OS – you pick which to use
Reference Counting

• Each object has a “reference count”
  - The number of clients who’ve taken ownership of the object

• When the object is in use
  - Reference count > 0

• When the object is not in use
  - Reference count == 0
  - System can deallocated object
Reference Counting

• Newly allocated objects start with a count of 1
  - [NSString alloc]

•Copied objects start with a count of 1
  - [obj copy] or [obj mutableCopy]

• When using an object – increment reference count
  - [obj retain]

• When you are done – decrement reference count
  - [obj release], or [obj autorelease]

• When “reference count” is 0, object goes away
Reference Counting
// alloc: creates with retain count = 1
Person *person = [[Person alloc] init];
// alloc: creates with retain count = 1
Person *person = [[Person alloc] init];
Reference Counting

// alloc: creates with retain count = 1
Person *person = [[Person alloc] init];

[person retain]; // increment retain count: +1
Person *person = [[Person alloc] init];

[person retain];  // increment retain count: +1
// alloc: creates with retain count = 1
Person *person = [[Person alloc] init];

[person retain];  // increment retain count: +1
[person release];  // decrement retain count: -1
// alloc: creates with retain count = 1
Person *person = [[Person alloc] init];

[person retain];  // increment retain count: +1
[person release];  // decrement retain count: -1
// alloc: creates with retain count = 1
Person *person = [[Person alloc] init];

[person retain]; // increment retain count: +1
[person release]; // decrement retain count: -1
[person release]; // decrement retain count: -1
// alloc: creates with retain count = 1
Person *person = [[Person alloc] init];

[person retain]; // increment retain count: +1
[person release]; // decrement retain count: -1
[person release]; // decrement retain count: -1
@implementation Person

- (void)dealloc { // Called when reference count hits 0
    [name_ release]; // release instance variables
    [super dealloc];
}

@end
@implementation Person

-(void)dealloc { // Called when reference count hits 0
    [name_ release]; // release instance variables
    [super dealloc];
}

@end
Reference Counting

@implementation Person

- (void)dealloc { // Called when reference count hits 0
    [name_ release]; // release instance variables
    [super dealloc];
}

@end
@implementation Person

- (void)dealloc { // Called when reference count hits 0
    [name_ release]; // release instance variables
    [super dealloc];
}
@end

Reference Counting
Reference Counting

Once object is deallocated don’t send it messages

// alloc: creates with retain count = 1
Person *person = [[Person alloc] init];

[person retain]; // increment retain count: +1
[person release]; // decrement retain count: -1
[person release]; // decrement retain count: -1

** person’s -dealloc is invoked for you **

[person setName:@”Fred”]; // BOOM goes the dynamite!!!!
Fundamental Rule

• You take ownership of an object if
  - you create it using a method that begins with “alloc” or “new”
  - you create it using a method that contains “copy” anywhere in method name
  - you call retain on it

• Release objects you own when you’re done with them

• Don’t release objects you don’t own!

• There is at least one exception. We’ll cover it in later lectures
## Rule Details

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-retain</td>
<td>increment reference count</td>
</tr>
<tr>
<td>-release</td>
<td>decrement reference count</td>
</tr>
<tr>
<td>-autorelease</td>
<td>decrement reference count, soon…</td>
</tr>
<tr>
<td>-alloc</td>
<td>returns object with +1 reference</td>
</tr>
<tr>
<td>-copy</td>
<td>returns object with +1 reference</td>
</tr>
<tr>
<td>-mutableCopy</td>
<td>“ “</td>
</tr>
<tr>
<td>-copyBlah…</td>
<td>“ “</td>
</tr>
<tr>
<td>-newBlah</td>
<td>“ “</td>
</tr>
</tbody>
</table>
Reference Counting Examples

- Increment reference count

```objective-c
myInstanceVariable_ = [someObject retain];
```

- Decrement reference count

```objective-c
[myInstanceVariable_ release];
myInstanceVariable_ = nil;
```

- Decrement reference count “later”

```objective-c
[anObject autorelease];
```
Autorelease

• When you are done with an object, but need to hand it off to someone else

```c
+(NSString *)displayStringWithSongCount:(NSInteger)ct
{
    ...
    s = [[NSString alloc] initWithFormat:@"%d songs", ct];
    return [s autorelease];
}
```
Autorelease

• Autoreleased objects go into an NSAutoreleasePool

• NSAutoreleasePool sends \texttt{[obj release]} when the pool is released

\begin{verbatim}
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
...
[string autorelease] ...

[pool release]; // ‘string’ is really released here
\end{verbatim}

• Normally whenever your code is executing, system will have set up an autorelease pool for you.
Return Values

• Deciphering Ownership of Returned Objects
  - How do you figure out if you need to retain it?

• Apply the Rules

- (NSString *)name;

+ (Person *)personWithName:(NSString *)name;

- (NSString *)copyName;

- (NSString *)mutableCopy;
Return Values

• Deciphering Ownership of Returned Objects
  - How do you figure out if you need to retain it?

• Apply the Rules

```c
// no special naming rule matches; -- value is autoreleased, you must retain it to have ownership claims to it...
- (NSString *)name;

// Factory without ‘copy/new’ in name; value is autoreleased, ...
+ (Person *)personWithName:(NSString *)name;

- (NSString *)copyName;

- (NSString *)mutableCopy;
```
Return Values

- Deciphering Ownership of Returned Objects
  - How do you figure out if you need to retain it?
- Apply the Rules

```objective-c
// no special naming rule matches; -- value is autoreleased, you must retain it to have ownership claims to it...
-(NSString *)name;

// Factory without ‘copy/new’ in name; value is autoreleased, ...
+(Person *)personWithName:(NSString *)name;

// ‘copy’ in the name, +1 retained value returned
-(NSString *)copyName;

// ‘copy’ in the name, +1 retained value returned
-(NSString *)mutableCopy;
```
Examples

- `myString_`, `myPerson_` are instance variables
  - Need to make sure objects we are using don’t get deallocated!

```swift
// must retain since -name returns an autoreleased object
myPerson_ = [[Person personWithName:@"Chuck"] retain];

// no add. retain needed, “alloc” creates object with +1 count
myPerson_ = [[Person alloc] initWithName:@"Chuck"];

// must retain since -name returns an autoreleased object
myString_ = [[myPerson_ name] retain];

// instead of retain, could use copy...
// don’t need to invoke retain on the copy
myString_ = [[myPerson_ name] copy];
```
NSObject

• Almost every class is a subclass of NSObject

• Your subclasses should implement
  
  - (id)init;
  
  - (void)dealloc;

• You’ll often want to implement
  
  - (id)copy;
  
  - (BOOL)isEqual:(id)other;
  
  - (NSUInteger)hash;
Initialization

- If you do nothing, ivars values default to 0, NO, nil, ...

```c
-(id)init {
    if ((self = [super init])) {

        // initialize instance variables
        ssn_ = -1;
        name_ = nil;
        birthdate_ = nil;

    }
    return self;
}
```
Initializer Idioms

• Designated Initializer
  - Usually the “init” with the most parameters
  - Responsible for initializing of inherited variables with call to “super init”
  - Other init methods funnel down to designated initializer

• Convenience Initializer
  - Instead of alloc/init/autorelease

    s = [[[NSString alloc] initWithFormat:...] autorelease];

    s = [NSString stringWithFormat:...];
Designated Initializer Example

```objective-c
- (id)initWithSongCount:(NSInteger)count
    startIndex:(NSInteger)index
{
    if ((self = [super init])) {
        count_ = count;
        index_ = index;
    }
    return self;
}

- (id)initWithSongCount:(NSInteger)count {
    return [self initWithSongCount:count startIndex:0];
}

- (id)init {
    return [self initWithSongCount:0 startIndex:0];
}
```
Deallocation

• Free resources

• Release hold on instance variables

```c
- (void)dealloc {
    [name_ release];
    [super dealloc];
}
```
Copying

- Often want to copy rather than reference an object

```objective-c
@protocol NSCopying
- (id)copyWithZone:(NSZone *)zone;
@end

@interface Person : NSObject < NSCopying >
@end

@implementation Person
...
- (id)copyWithZone:(NSZone *)zone {
    Person *personCopy = [[Person alloc] init];
    personCopy.name = self.name;
    return personCopy;
}
```
Equality

- `isEqual`: used to compare object
  - when searching in arrays, dictionaries, etc…

- `NSObject`’s implementation uses pointer equality

- If you implement `isEqual`:
  - Equal objects require same `-hash` value
  - Objects with same `-hash` don’t require `isEqual` equality
Advanced Features
Categories

• Extend a class
• Even if you don’t have the code...

@interface NSString (MD5Additions)
- (NSData *)md5Checksum;
@end

@interface NSDate (DisplayAdditions)
- (NSString *)monthAndDayLocalizedString;
@end
Protocols

- Group of methods with special meaning
- Not associated with a particular class

@protocol TableDataProvider

@required
- (NSInteger)countOfTitles;
- (NSString *)titleAtIndex:(NSInteger)index;

@optional
- (BOOL)canSelectRow:(NSInteger)integer;

@end
Protocol Conformance

• Declare

@class MyObject <TableDataProvider>

• Type

- (void)setTableDataProvider:(id <TableDataProvider>)tdp;

• Check

id anObj = ...;
if ([anObj conformsToProtocol:@protocol(TableDataProvider)])

...
Protocol Example

@protocol NSCopying
- (id)copyWithZone:(NSZone *)zone;
@end

@interface Person : NSObject < NSCopying >

@end

@implementation Person

...

- (id)copyWithZone:(NSZone *)zone {
    Person *personCopy = [[Person alloc] init];
    personCopy.name = self.name;
    return personCopy;
}

Introspection

- Ability of objects to reveal information
- Can ask an object “do you implement this?”

```swift
result = // some default value...
SEL selector = @selector(canSelectFoo:);
if ([delegate respondsToSelector:selector]) {
    result = [delegate canSelectFoo: foo];
}
return result;
```

- Can ask an object “what are you?”

```swift
if ([delegate isKindOfClass: ...]) ...
if ([delegate conformsToProtocol: ...]) ...
```
Topics To Explore

• **NSInvocation**
  - object encapsulation of a method invocation

• **Forwarding**
  - ability to handle unimplemented methods
  - forward to other (possibly remote) objects

• **Exception handling**
  - `@try`, `@catch`, `@finally`

• `@synchronized`
Assignment

• ObjC.pdf
  - Chapter 1: Objects, classes and messaging
  - Chapter 2: Defining a class

• CocoaFundamentals.pdf
  - P.77 - 81: How Memory Management Works
  - P.81 - 88: Initializing an Object
  - P.63 - 67: Categories, Protocols, Declared Properties