

## Announcements

- **Discuss Final Project Ideas on Wednesday**
  - Final Project teams will consist of 4 – 5 people
    - No teams of 1, 2, or 3 people

## Today's Topics

- **Testing in iOS**
- **Objective-C Language**
- **Common Foundation Classes**

## Objective-C

- **Strict superset of C**
  - Mix C with ObjC
  - Or even C++ with ObjC (usually referred to as ObjC++)
- **A very simple language, but some new syntax**
- **Single inheritance, classes inherit from one and only one superclass**
- **Protocols define behavior that cross classes**
  - A protocol is a collection of methods grouped together
    - Indicates that a class implements a protocol
    - Similar to interfaces in Java

## Syntax Additions

- **Small number of additions**
- **Some new types**
  - Anonymous object
  - Class
  - Selectors (covered later)
- **Syntax for defining classes**
- **Syntax for message expressions**

# OOP with ObjC

## Classes and Objects

- **In Objective-C, classes and instances are both objects**
  - Class is the blueprint to create instances
- **Classes declare state and behavior**
- **State (data) is maintained using instance variables**
- **Behavior is implemented using methods**
- **Instance variables typically hidden**
  - Accessible only using getter/setter methods

## OOP From ObjC Perspective

- **Everybody has their own spin on OOP**
  - Apple is no different
- **For the spin on OOP from an ObjC perspective:**
  - Read the “Object-Oriented Programming with Objective-C” document:
    - [https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/OOP\\_ObjC/Introduction/Introduction.html](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/OOP_ObjC/Introduction/Introduction.html)

## Messaging syntax

## Message syntax

- [receiver **message**]
- [receiver **message:argument**]
- [receiver **message:arg1 andArg:arg2**]

## Class and Instance Methods

- **Instances respond to instance methods**
  - (id)init;
  - (float)height;
  - (void)walk;
- **Classes respond to class methods**
  - +(id)alloc;
  - +(id)person;
  - +(Person \*)sharedPerson;

## Message examples

```
Person *voter; //assume this exists
[voter castBallot];

int theAge = [voter age];
[voter setAge:21];

if ([voter canLegallyVote]) {
    // do something voter
}

[voter registerForState:@"CA" party:@"Independant"];

NSString *name = [[voter spouse] name];
```

## Method definition examples (.h)

```
Person *voter; //assume this exists
-(void)castBallot;
[voter castBallot];

-(int)age;
int theAge = [voter age];

-(void)setAge: (int)age;
[voter setAge:21];

-(BOOL)canLegallyVote;
if ([voter canLegallyVote]) {
    // do something voter
}

-(void)registerForState: (NSString*) state party: (NSString*)party;
[voter registerForState:@"CA" party:@"Independant"];

-(Person*)spouse;
-(NSString*)name;
NSString *name = [[voter spouse] name];
```

## Additional Example

- **Method calls**
  - [receiver message:arg1 andArg:arg2]

Consider a Shape class with a method called defineShape

```
-(void) defineShape:(int) sides height:(int) myHeight width:(int) myWidth {  
    int tempSides = sides;  
    int tempHeight = myHeight;  
    //set shape properties....  
}
```

```
Shape *basicShape;
```

```
[basicShape defineShape:4 height:3 width: 2]; //correct call
```

```
[basicShape defineShape:4]; //calls something else (or issues warning)
```

```
[basicShape defineShape:4:3:2]; //warning no matching signature, fail when running
```

I could create:

```
-(void) defineShape: (int) sides : (int) myHeight : (int) myWidth {  
  
}
```

I could then call:

```
[basicShape defineShape:4:3:2];
```

Considered a bad idea... the parameters are not clearly labeled

```
-(void) defineShape:(int) sides height:(int) myHeight width:(int) myWidth {
```

Good!

```
-(void) defineShape: (int) sides : (int) myHeight : (int) myWidth {
```

Bad!

## Terminology

- **Message expression**

[receiver method: argument]

- **Message**

[receiver method: argument]

- **Method**

– The code selected by a message

## Dot Syntax

- **Objective-C 2.0 introduced dot syntax**

- **Convenient shorthand for invoking accessor methods**

```
float height = [person height];
```

```
float height = person.height;
```

```
[person setHeight:newHeight];
```

```
person.height = newHeight;
```

- **Follows the dots...**

```
[[person child] setHeight:newHeight];
```

// **exactly the same as**

```
person.child.height = newHeight;
```



## Objective-C Types

## Dynamic and Static typing

- **Dynamically-typed object**  
**id anObject**
  - Just id
  - Not id \* (unless you really, really mean it...)
- **Statically-typed object**  
**Person \*anObject**
- **Objective-C provides compile-time, not runtime, type checking**
- **Objective-C always uses dynamic binding**
  - determining the exact implementation of a request based on both the request (operation) name and the receiving object at the run-time

## The null object pointer

- **Test for nil explicitly**

```
if (person == nil)
    return;
```

- **Or implicitly**

```
if (!person) return;
```

- **Can use in assignments and as arguments if expected**

```
person = nil;
[person name];
```

## BOOL typedef

- **When ObjC was developed, C had no boolean type (C99 introduced one)**

- **ObjC uses a typedef to define BOOL as a type**

```
BOOL flag = NO;
```

- **Macros included for initialization and comparison: YES and NO**

```
if (flag == YES)
if (flag)
if (!flag)
if (flag != YES)
flag = YES;
flag = 1;
```

## Class Introspection

- You can ask an object about its class

```
Class myClass = [myObject class];  
NSLog(@"My class is %@", [myObject className]);
```

- Testing for general class membership (subclasses included):

```
if ([myObject isKindOfClass:[UIControl class]]) {  
    // something  
}
```

- Testing for specific class membership (subclasses excluded):

```
if ([myObject isKindOfClass:[NSString class]]) {  
    // something string specific  
}
```

## Working with Objects

## Identity versus Equality

- **Identity**—testing equality of the pointer values

```
if (object1 == object2) {  
    NSLog(@"Same exact object instance");  
}
```

- **Equality**—testing object attributes

```
if ([object1 isEqual: object2]) {  
    NSLog(@"Logically equivalent, but may be different  
    object instances");  
}
```

## -description

- **NSObject implements -description**  
-(NSString \*)description;
- **Objects represented in format strings using %@**
- **When an object appears in a format string, it is asked for its description**
  - [NSString stringWithFormat: @"The answer is: %@", myObject];
- **You can log an object's description with:**
  - NSLog([anObject description]);
- **Your custom subclasses can override description to return more specific information**
- **Similar to a toString() in Java**

## Foundation Classes

## Foundation Framework

- Value and collection classes
- User defaults
- Archiving
- Notifications
- Undo manager
- Tasks, timers, threads
- File system, pipes, I/O, bundles

## NSObject

- **Root class**
- **Implements many basics**
  - Memory management
  - Introspection
  - Object equality

## NSString

- **General-purpose Unicode string support**
  - Unicode is a coding system which represents all of the world's languages
- **Consistently used throughout Cocoa Touch instead of “char \*”**
- **The most commonly used class**
- **Easy to support any language in the world with Cocoa**

## String Constants

- **In C constant strings are**
  - “simple”
- **In ObjC, constant strings are**
  - @“just as simple”
- **Constant strings are**

```
NSString *aString = @"Hello World!";
```

## Format Strings

- **Similar to printf, but with %@ added for objects**

```
NSString *aString = @"Johnny";  
NSString *log = [NSString stringWithFormat: @"It's %@", aString];
```

- **log would be set to**

- It's 'Johnny'

- **Also used for logging**

```
NSLog(@"I am a %@, I have %d items", [array className], [array count]);
```

- **would log something like:**

- I am a NSArray, I have 5 items

## NSString

- Often ask an existing string for a new string with modifications

```
-(NSString *)stringByAppendingString:(NSString *)string;  
-(NSString *)stringByAppendingFormat:(NSString *)string;
```

```
-(NSString *)stringByDeletingPathComponent;
```

- Example:

```
NSString *myString = @"Hello";  
NSString *fullString;  
fullString = [myString stringByAppendingString:@" world!"];
```

- fullString would be set to
  - Hello world!

## NSString

- Common NSString methods

```
-(BOOL)isEqualToString:(NSString *)string;  
-(BOOL)hasPrefix:(NSString *)string;  
-(int)intValue;  
-(double)doubleValue;
```

- Example:

```
NSString *myString = @"Hello";  
NSString *otherString = @"449";  
if ([myString hasPrefix:@"He"]) {  
    // will make it here  
}  
if ([otherString intValue] > 500) {  
    // won't make it here  
}
```



## NSMutableString

- subclasses NSString
- Allows a string to be modified
- Common NSMutableString methods

+ (id)string;

- (void)appendString:(NSString \*)string;

- (void)appendFormat:(NSString \*)format, ...;

```
NSMutableString *newString = [NSMutableString string];  
[newString appendString:@"Hi"];  
[newString appendFormat:@", my favorite number is: %d",[self favoriteNumber]];
```

## Collections

- Array - ordered collection of objects
- Dictionary - collection of key-value pairs
- Set - unordered collection of unique objects
  
- Common enumeration mechanism
- Immutable and mutable versions
- Immutable collections can be shared without side effect
  - Prevents unexpected changes
  - Mutable objects typically carry a performance overhead

## NSArray

- **Common NSArray methods**

```
+ arrayWithObjects:(id)firstObj, ...; // nil terminated!!!  
-(unsigned)count;  
-(id)objectAtIndex:(unsigned)index;  
-(unsigned)indexOfObject:(id)object;
```

- **NSNotFound returned for index if not found**

```
NSArray *array = [NSArray arrayWithObjects:@"Red", @"Blue",  
@"Green",nil];  
  
if ([array indexOfObject:@"Purple"] == NSNotFound) {  
    NSLog(@"No color purple");  
}
```

## NSMutableArray

- **NSMutableArray subclasses NSArray**

- So, everything in NSArray

- **Common NSMutableArray Methods**

```
+ (NSMutableArray *)array;  
- (void)addObject:(id)object;  
- (void)removeObject:(id)object;  
- (void)removeAllObjects;  
- (void)insertObject:(id)object atIndex:(unsigned)index;
```

```
NSMutableArray *array = [NSMutableArray array];  
[array addObject:@"Red"];  
[array addObject:@"Green"];  
[array addObject:@"Blue"];  
[array removeObjectAtIndex:1];
```

## NSDictionary

- **Common NSDictionary methods**

```
+ dictionaryWithObjectsAndKeys:(id)firstObject, ...;  
-(unsigned)count;  
-(id)objectForKey:(id)key;
```

- **nil returned if no object found for given key**

```
NSDictionary *colors =  
[NSDictionary dictionaryWithObjectsAndKeys:@"Red", @"Color 1",  
@"Green", @"Color 2", @"Blue", @"Color 3", nil];  
  
NSString *firstColor = [colors objectForKey:@"Color 1"];  
  
if ([colors objectForKey:@"Color 8"]) {  
    // won't make it here  
}
```

## NSMutableDictionary

- **NSMutableDictionary subclasses NSDictionary**

- **Common NSMutableDictionary methods**

```
+ (NSMutableDictionary *)dictionary;  
- (void)setObject:(id)object forKey:(id) key;  
- (void)removeObjectForKey:(id)key;  
- (void) removeAllObjects;
```

```
NSMutableDictionary *colors = [NSMutableDictionary dictionary];  
[colors setObject:@"Orange" forKey:@"HighlightColor"];
```

## NSSet

- **Unordered collection of distinct objects**
- **Common NSMutableSet methods**

```
+ initWithObjects:(id)firstObj, ...; // nil terminated  
- (unsigned)count;  
- (BOOL)containsObject:(id)object;
```

## NSMutableSet

- **NSMutableSet subclasses NSMutableSet**
  - **Common NSMutableSet methods**
- ```
+ (NSMutableSet *)set;  
- (void)addObject:(id)object;  
- (void)removeObject:(id)object;  
- (void)removeAllObjects;  
- (void)intersectSet:(NSSet *)otherSet;  
- (void)minusSet:(NSSet *)otherSet;
```

## Enumeration

- **Consistent way of enumerating over objects in collections**
- **Use with NSArray, NSDictionary, NSSet, etc.**

```
NSArray *array = ... ; // assume an array of People objects
```

```
// old school
Person *person;
int count = [array count];
for (i = 0; i < count; i++) {
    person = [array objectAtIndex:i];
    NSLog([person description]);
}
```

```
// new school
for (Person *person in array) {
    NSLog([person description]);
}
```

## Other Classes

- **NSData / NSMutableData**
  - Arbitrary sets of bytes
- **NSDate / NSCalendarDate**
  - Times and dates

## Methods and Selectors

## Terminology

- **Message expression**

[receiver method: argument]

- **Message**

[receiver method: argument]

- **Selector**

[receiver method: argument]

- **Method**

The code selected by a message

## Methods, Messages, Selectors

- **Method**

- Behavior associated with an object

```
-(NSString *)name
{
    // Implementation
}
-(void)setName:(NSString *)name
{
    //Implementation
}
```

## Methods, Selectors, Messages

- **Selector**

- Name for referring to a method
- Includes colons to indicate arguments
- Doesn't actually include arguments or indicate types

```
SEL mySelector = @selector(name);
```

```
SEL anotherSelector = @selector(setName:);
```

```
SEL lastSelector = @selector(doStuff:withThing:andThing:);
```

## Methods, Messages, Selectors

- **Message**

- The act of performing a selector on an object
- With arguments, if necessary

```
NSString *name = [myPerson name];
```

```
[myPerson setName:@"New Name"];
```

## Selectors identify methods by name

- **A selector has type SEL**

```
SEL action = [button action];  
[button setAction:@selector(start:)];
```

- **Conceptually similar to function pointer**

- **Selectors include the name and all colons, for example:**

```
(void)setName:(NSString *)name age:(int)age;
```

- **Would have a selector:**

```
SEL sel = @selector(setName:age:);
```



## Working with selectors

- You can determine if an object responds to a given selector

```
id obj;

SEL sel = @selector(start:);

if ([obj respondsToSelector:sel]) {
    [obj performSelector:sel withObject:self];
    //equivalent to [obj start:self];
    //For multiple arguments use ... withObject: withObject:
}
```

- This sort of introspection and dynamic messaging underlies many Cocoa design patterns

```
-(void)setTarget:(id)target;
-(void)setAction:(SEL)action;
```

## More Info on Selectors

- Selectors are unique identifiers that replace the name of methods when compiled
- Compiler writes each method name into a table and associates it with this unique id (the selector)
- The compiler assigns all method names a unique selector or SEL (the selector type)
  - Every “method name” whether it is a part of your class or another class has an entry in that table with a unique selector value

- More information at:

<https://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocSelectors.html>

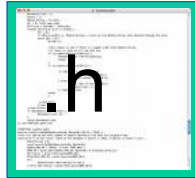
# Custom Classes

## Design Phase

- **Create a class**
  - Person
- **Determine the superclass**
  - NSObject (in this case)
- **What properties should it have?**
  - Name, age, whether they can vote
- **What actions can it perform?**
  - Cast a ballot

## Defining a class

### A public header and a private implementation



Header file



Implementation file

## Class interface declared in header file

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}

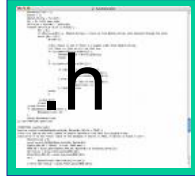
// method declarations
- (NSString *)name;
- (void)setName:(NSString *)value;

- (int)age;
- (void)setAge:(int)age;
- (BOOL)canLegallyVote;
- (void)castBallot;

@end
```

## Defining a class

### A public header and a private implementation



Header file



Implementation file

## Implementing custom class

- Implement setter/getter methods
- Implement action methods

## Class Implementation

```
#import "Person.h"

@implementation Person

-(int)age {
    return age;
}

-(void)setAge:(int)value {
    age = value;
}
//... and other methods

@end
```

## Calling your own methods

```
#import "Person.h"

@implementation Person

-(BOOL)canLegallyVote {
    return ([self age] >= 18);
}

-(void)castBallot {
    if ([self canLegallyVote]) {
        // do voting stuff
    } else {
        NSLog(@"I'm not allowed to vote!");
    }
}

@end
```

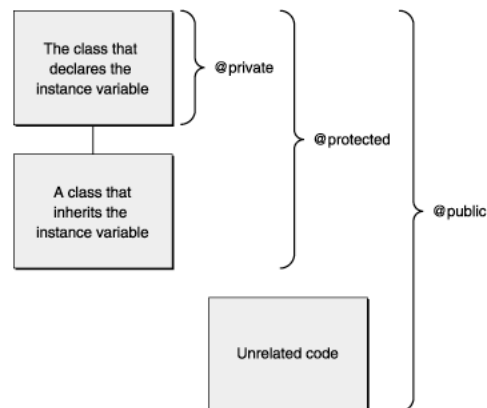
## Superclass methods

- As we just saw, objects have an implicit variable named “self”
  - Like “this” in Java and C++
- Can also invoke superclass methods using “super”

```
-(void)doSomething {  
    // Call superclass implementation first  
    [super doSomething];  
  
    // Then do our custom behavior  
    int foo = bar;  
    // ...  
}
```

## Public and Private Instance Variables

- By default all variables are protected



## Instance Variable Protection

```
@interface Worker : NSObject
{
    char *name; //actually protected

    @private
    int age;
    char *evaluation;

    @protected
    id job;
    float wage;

    @public
    id boss;
}
```

Objective C methods are all public (a way around this...)  
<http://iphonedevopertips.com/objective-c/private-methods.html>

## Private Methods

- Private methods in Objective-C are declared in your .m file

```
@interface Worker ()
- (void) myPrivateMethod1;
- (int) anotherPrivateMethod: (NSString *) myStringArgument
@end

@implementation Worker
-(void) myPrivateMethod1 {
    //Do something here
}
-(int) anotherPrivateMethod: (NSString *) myStringArgument {
    return 0;
}
- (void) somePublicMethod {
}
@end
```

# Object Lifecycle

# Object Lifecycle

- **Creating objects**
- **Memory management**
- **Destroying objects**



## Object Creation

- **Two step process**

- allocate memory to store the object
- initialize object state

- **+alloc**

- Class method that knows how much memory is needed

- **-init**

- Instance method to set initial values, perform other setup

## Create = Allocate + Initialize

```
Person *student = nil;
```

```
student = [[Person alloc] init];
```

Or

```
Person *student = nil;
```

```
student = [Person alloc];
```

```
student = [student init];
```

## Implementing your own -init method

```
#import "Person.h"

@implementation Person

-(id)init {
    // allow superclass to initialize its state first
    self = [super init];
    if (self != nil) {
        age = 0;
        name = @"Bob";

        // do other initialization...
    }
    return self;
}

@end
```

## Multiple init methods

- **Classes may define multiple init methods**
  - (id)init;
  - (id)initWithName:(NSString \*)name;
  - (id)initWithName:(NSString \*)name age:(int)age;
- **Less specific ones typically call more specific with default values**
  - Designated Initializers

```
- (id)init { return [self initWithName:@"Bob"];
}
- (id)initWithName:(NSString *)name {
    return [self initWithName:name age:0];
}
```

## Finishing Up With an Object

```
Person *person = nil;

person = [[Person alloc] init];

[person setName:@"Alan Cannistraro"];
[person setAge:29];
[person setWishfulThinking:YES];

[person castBallot];

// What do we do with person when we're done?
```

## Two flavors of Memory Management

- **Automatic Reference Counting (ARC)**
  - Full support starting in iOS 5
- **Manual Reference Counting**
  - Original Objective C design
- **Choose one or the other**
  - Do not attempt to use both in the same .m file

## Why learn both methods?

- Many of the tutorials and examples on the web were created pre-ARC
- A solid understanding of manual reference counting makes ARC easier to understand
- Xcode can run into problems with migrating existing code to use ARC

## (Manual) Memory Management

|             | Allocation | Destruction |
|-------------|------------|-------------|
| C           | malloc     | free        |
| Objective-C | alloc      | dealloc     |

- **Calls must be balanced**
  - Otherwise your program may leak or crash
- **However, you' ll never call -dealloc directly**
  - One exception, we' ll see in a bit...

## (Manual) Reference Counting

- **Every object has a retain count**
  - Defined on NSObject
  - As long as retain count is > 0, object is alive and valid
  
- **+alloc and -copy create objects with retain count == 1**
- **-retain increments retain count**
- **-release decrements retain count**
  
- **When retain count reaches 0, object is destroyed**
- **-dealloc method invoked automatically**
  - One-way street, once you're in -dealloc there's no turning back

## Balanced Calls

```
Person *person = nil;
person = [[Person alloc] init];

[person setName:@"John Smith"];
[person setAge:29];
[person setWishfulThinking:YES];

[person castBallot];

// When we're done with person, release it
[person release]; // person will be destroyed here
```

## Reference counting in action

```
Person *person = [[Person alloc] init];
```

- Retain count begins at 1 with +alloc

```
[person retain];
```

- Retain count increases to 2 with –retain

```
[person release];
```

- Retain count decreases to 1 with –release

```
[person release];
```

- Retain count decreases to 0, -dealloc automatically called

## Messaging deallocated objects

```
Person *person = [[Person alloc] init];
```

```
// ...
```

```
[person release]; // Object is deallocated
```

```
[person doSomething]; // Crash!
```

## Messaging deallocated objects

```
Person *person = [[Person alloc] init];  
// ...  
[person release]; // Object is deallocated  
  
person=nil;  
  
[person doSomething]; // No effect
```

## Implementing a -dealloc method

```
#import "Person.h"  
  
@implementation Person  
  
- (void)dealloc {  
    // Do any cleanup that's necessary  
    // ...  
  
    // when we're done, call super to clean us up  
    [super dealloc];  
}  
@end
```

## Object Lifecycle Recap

- Objects begin with a retain count of 1
- Increase and decrease with `-retain` and `-release`
- When retain count reaches 0, object deallocated automatically
- You *never* call `dealloc` explicitly in your code
  - Exception is calling `-[super dealloc]`
  - You only deal with `alloc`, `copy`, `retain`, `release`

## Object Ownership

```
#import <Foundation/Foundation.h>

@interface Person : NSObject
{
    // instance variables
    NSString *name; //Person class "owns" the name
    int age;
}

// method declarations
-(NSString *)name;
-(void)setName:(NSString *)value;

-(int)age;
-(void)setAge:(int)age;

-(BOOL)canLegallyVote;
-(void)castBallot;

@end
```



## Object Ownership

```
#import "Person.h"
@implementation Person

- (NSString *)name {
    return name;
}

- (void)setName:(NSString *)newName {
    if (name != newName) {
        [name release];
        name = [newName retain];
        // name's retain count has been bumped up by 1
    }
}

@end
```

## Object Ownership

```
#import "Person.h"
@implementation Person

- (NSString *)name {
    return name;
}

- (void)setName:(NSString *)newName {
    if (name != newName) {
        [name release];
        name = [newName copy];
        // name has a retain count of 1, we own it
    }
}

@end
```

## Releasing Instance Variables

```
#import "Person.h"
@implementation Person

- (void)dealloc{
    //Do any cleanup that's necessary
    [name release];

    // when we're done, call super to clean us up
    [super dealloc];
}

@end
```

## Autorelease

## Returning a newly created object

```
-(NSString *)fullName {
    NSString *result;

    result = [[NSString alloc] initWithFormat:@"%@" "%@",
        firstName, lastName];

    return result;
}
```

- **Wrong: result is leaked!**

## Returning a newly created object

```
-(NSString *)fullName {
    NSString *result;

    result = [[NSString alloc] initWithFormat:@"%@" "%@",
        firstName, lastName];

    [result release];
    return result;
}
```

- **Wrong: result is released too early!**
- **Uncertain what method returns**

## Returning a newly created object

```
-(NSString *)fullName {  
    NSString *result;  
  
    result = [[NSString alloc] initWithFormat:@"%@ %@",  
        firstName, lastName];  
  
    [result autorelease];  
    return result;  
}
```

- **Just right: result is released, but not right away!**
- **Caller gets valid object and could retain if needed**

## Autoreleasing Objects

- **Calling -autorelease flags an object to be sent release at some point in the future**
- **Let's you fulfill your retain/release obligations while allowing an object some additional time to live**
- **Makes it much more convenient to manage memory**
- **Very useful in methods which return a newly created object**

## Method Names & Autorelease

- **Methods whose names includes alloc or copy return a retained object that the caller needs to release**

```
NSMutableString *string = [[NSMutableString alloc] init];
```

```
// We are responsible for calling -release or -autorelease  
[string autorelease];
```

- **All other methods return autoreleased objects**

```
NSMutableString *string = [NSMutableString string];
```

```
// The method name doesn't indicate that we need to release it  
// So don't - we're cool!
```

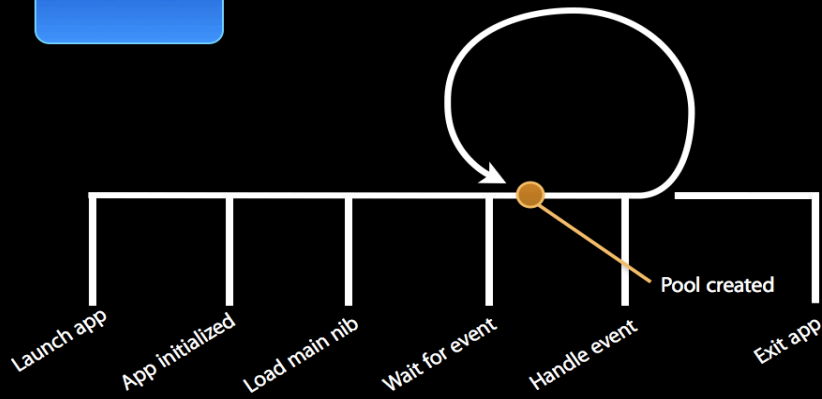
- **This is a convention**
  - follow it in methods you define

## How does -autorelease work?

- **Object is added to current autorelease pool**
- **Autorelease pools track objects scheduled to be released**
  - When the pool itself is released, it sends -release to all its objects
- **UIKit automatically wraps a pool around every event dispatch**

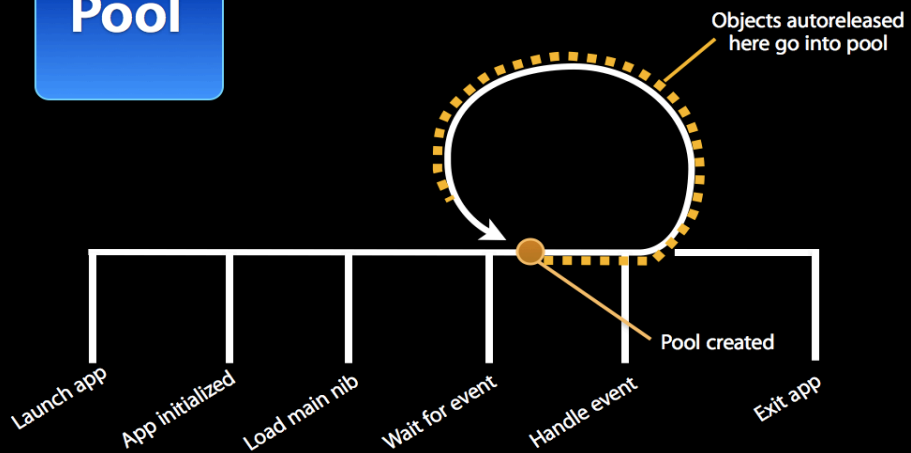
## Autorelease Pools (from cs193p slides)

Pool

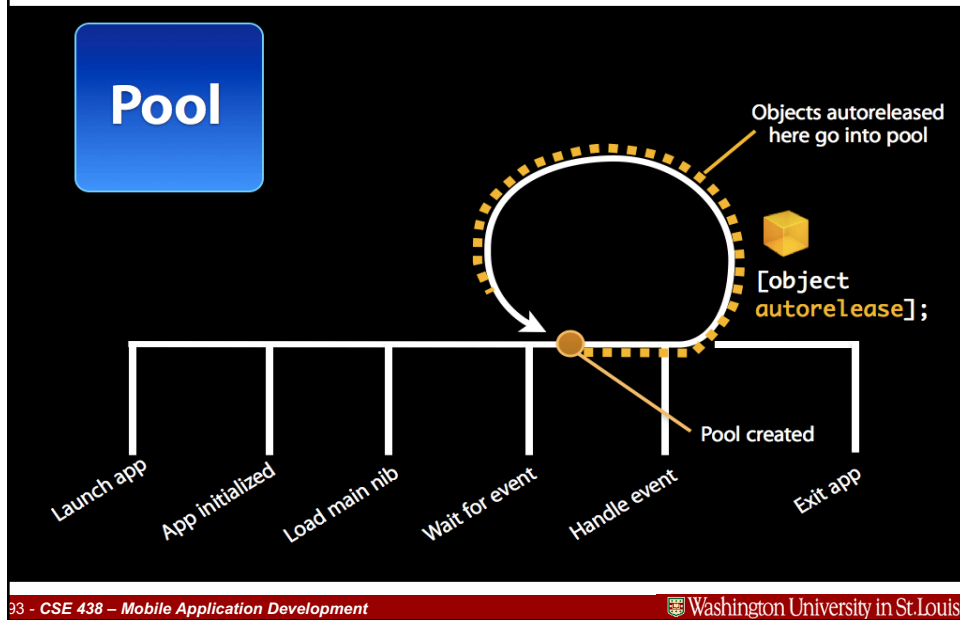


## Autorelease Pools (from cs193p slides)

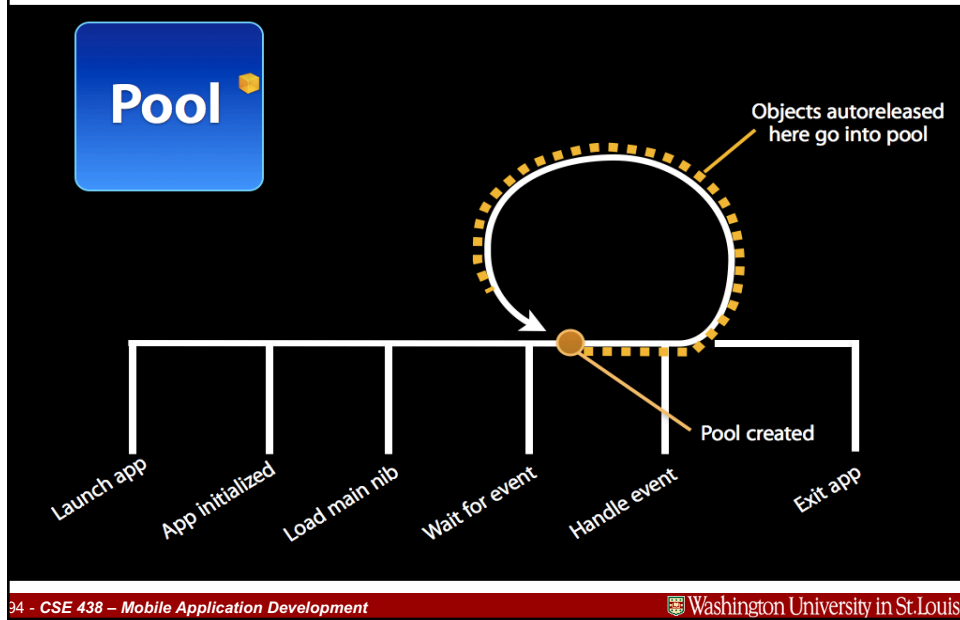
Pool



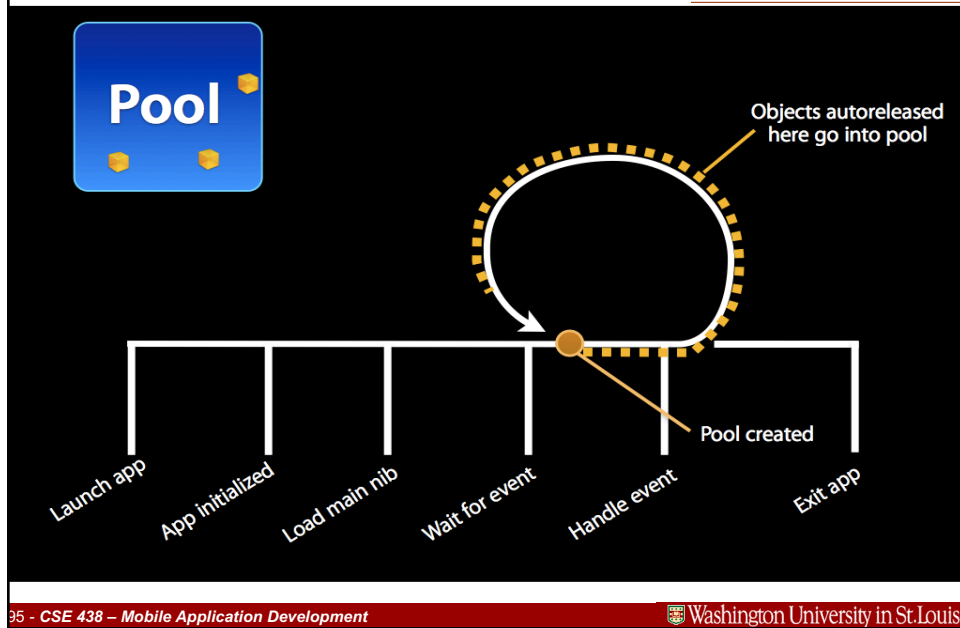
## Autorelease Pools (from cs193p slides)



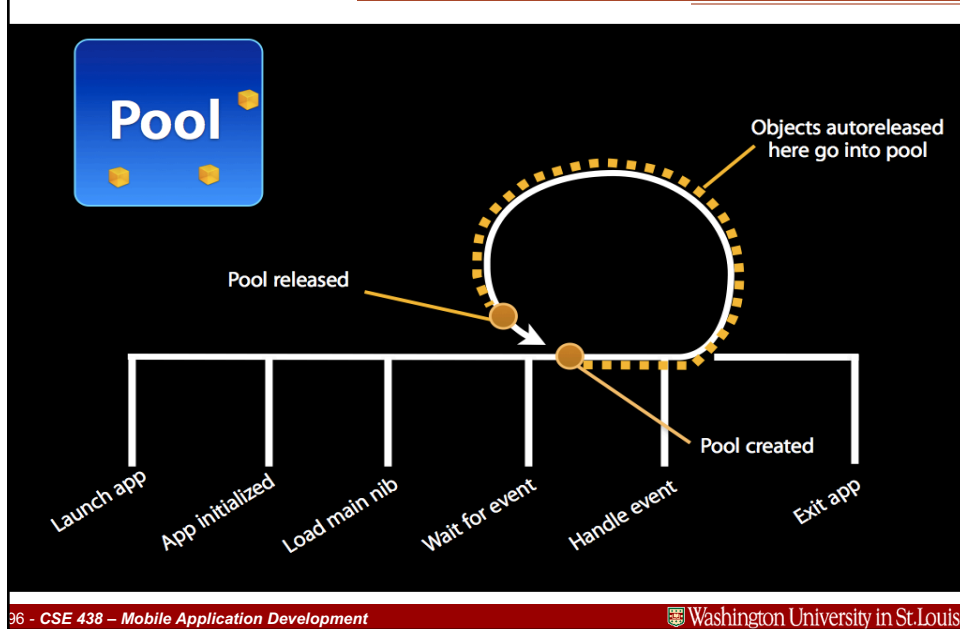
## Autorelease Pools (from cs193p slides)



## Autorelease Pools (from cs193p slides)

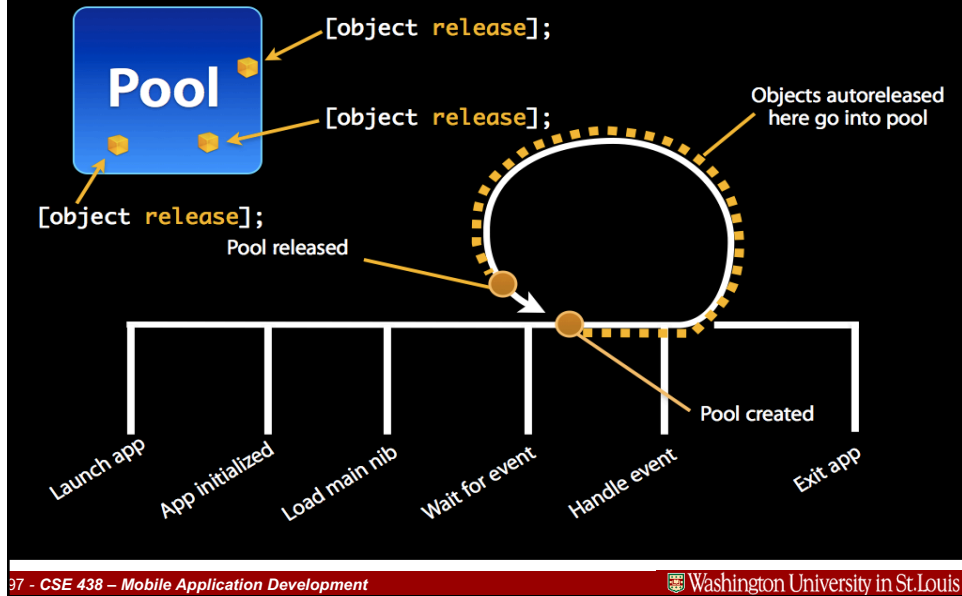


## Autorelease Pools (from cs193p slides)

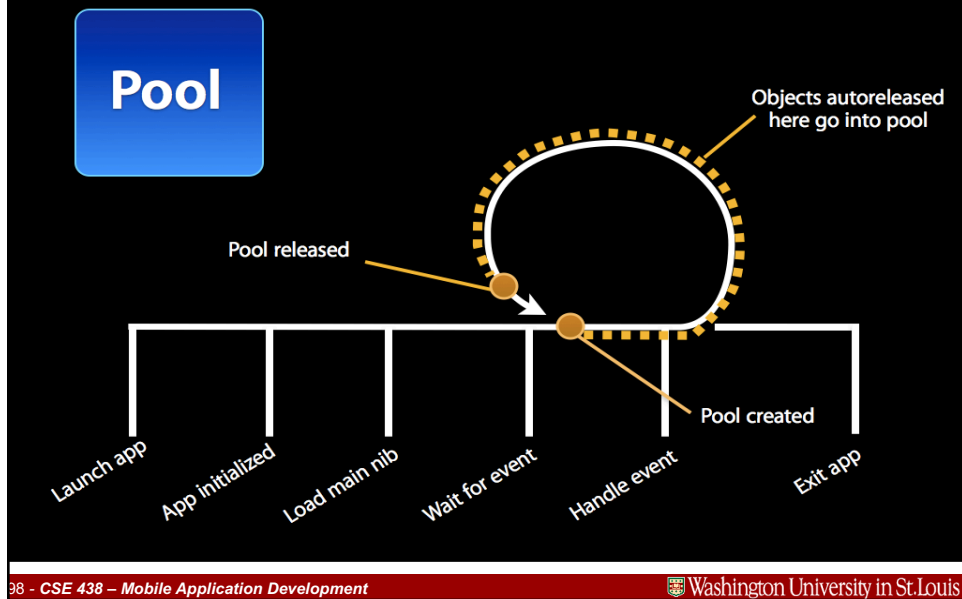




## Autorelease Pools (from cs193p slides)



## Autorelease Pools (from cs193p slides)



## Hanging Onto an Autoreleased Object

- **Many methods return autoreleased objects**
  - Remember the naming conventions...
  - They're hanging out in the pool and will get released later
- **If you need to hold onto those objects you need to retain them**
  - Bumps up the retain count before the release happens

```
name = [NSMutableString string];
```

```
// We want name to remain valid!
```

```
[name retain];
```

```
// ...
```

```
// Eventually, we'll release it (maybe in our -dealloc?)
```

```
[name release];
```

## Side Note: Garbage Collection

- Autorelease is not garbage collection
- Objective-C on iPhone OS (iOS) does not have garbage collection

# Automatic Reference Counting (ARC)

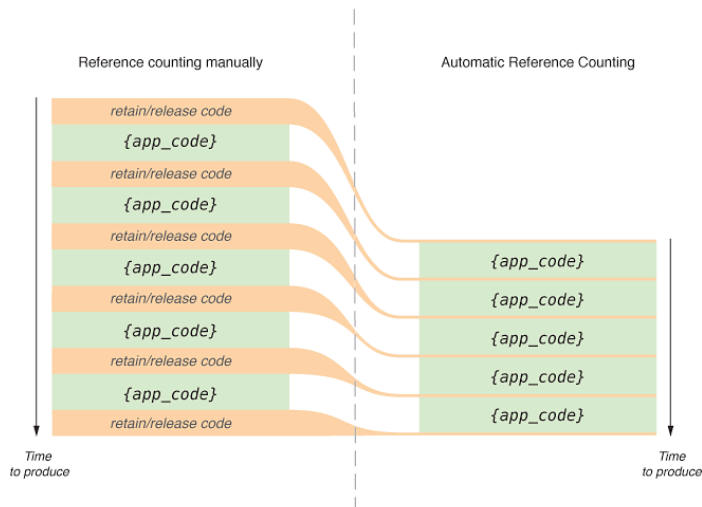
## Automatic Reference Counting (ARC)

- **The new and “improved” way to manage memory**
  - All objects are either strong or weak
- **Strong**
  - Keep me around until I no longer need this memory
- **Weak**
  - Keep me around as long as some other object needs this memory

## Automatic Reference Counting

- **By default all objects allocated when using ARC are strong**
  - `NSNumber *myNumber = [NSNumber alloc] init];`
- **Weak references are often used when pointing to objects on a storyboard**
  - UIButton, UILabel, UIImage
    - These objects are already instantiated when the storyboard loads
    - We just want a pointer to them while they are alive

## Automatic Retain/Release



## Properties

- Provide access to object attributes
- Shortcut to implementing getter/setter methods
- Also allow you to specify:
  - read-only versus read-write access
  - memory management policy

## Defining Properties

```
#import<Foundation/Foundation.h>
@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}

// method declarations
-(NSString *)name;
-(void)setName:(NSString *)value;
-(int)age;
-(void)setAge:(int)age;
-(BOOL)canLegallyVote;

-(void)castBallot;
@end
```

## Defining Properties

```
#import<Foundation/Foundation.h>
@interface Person : NSObject
{
// instance variables
    NSString *name;
    int age;
}
```

```
// method declarations
-(NSString *)name;
-(void)setName:(NSString *)value;
-(int)age;
-(void)setAge:(int)age;
-(BOOL)canLegallyVote;
```

```
-(void)castBallot;
@end
```

## Defining Properties

```
#import<Foundation/Foundation.h>
@interface Person : NSObject
{
// instance variables
    NSString *name;
    int age;
}
```

```
// method declarations
-(NSString *)name;
-(void)setName:(NSString *)value;
-(int)age;
-(void)setAge:(int)age;
-(BOOL)canLegallyVote;
```

```
-(void)castBallot;
@end
```

## Defining Properties

```
#import<Foundation/Foundation.h>
@interface Person : NSObject
{
    // instance variables
    NSString *name;
    int age;
}

// property declarations
@property int age;
@property (copy) NSString *name;
@property (readonly) BOOL canLegallyVote;

-(void)castBallot;
@end
```

## Synthesizing Properties

```
@implementation Person

-(int)age {
    return age;
}

-(void)setAge:(int)value {
    age = value;
}

-(NSString *)name {
    return name;
}

-(void)setName:(NSString *)value {
    if (value != name) {
        [value release];
        name = [value copy];
    }
}

- (BOOL)canLegallyVote { ...
```

## Synthesizing Properties

@implementation Person

```
-(int)age {  
    return age;  
}  
  
-(void)setAge:(int)value {  
    age = value;  
}  
  
-(NSString *)name {  
    return name;  
}  
  
-(void)setName:(NSString *)value {  
    if (value != name) {  
        [value release];  
        name = [value copy];  
    }  
}  
  
-(BOOL)canLegallyVote { ...
```

## Synthesizing Properties

@implementation Person

```
-(int)age {  
    return age;  
}  
  
-(void)setAge:(int)value {  
    age = value;  
}  
  
-(NSString *)name {  
    return name;  
}  
  
-(void)setName:(NSString *)value {  
    if (value != name) {  
        [value release];  
        name = [value copy];  
    }  
}  
  
-(BOOL)canLegallyVote { ...
```



## Synthesizing Properties

```
@implementation Person

@synthesize age;
@synthesize name;
- (BOOL)canLegallyVote {
    return (age > 17);
}

@end
```

## iOS Property Attributes

- Use strong and weak instead of retain and assign

```
@property (retain) NSString *name; // retain called
@property (strong) NSString *name; // new way
```

```
@property (assign) NSString *name; // pointer assignment
@property (weak) NSString *name; // new way
```

## Property Names vs. Instance Variables

- Property name can be different than instance variable

```
@interface Person : NSObject {
    int numberOfYearsOld;
}

@property int age;

@end

@implementation Person

@synthesize age = numberOfYearsOld;

@end
```

## Properties

- Mix and match synthesized and implemented properties

```
@implementation Person

@synthesize age;
@synthesize name;

(void)setAge:(int)value {
    age = value;
}

@end
```

- Setter method explicitly implemented
- Getter method still synthesized

## Properties In Practice

- **Newer APIs use @property**
- **Older APIs use getter/setter methods**
- **Properties used heavily throughout UIKit APIs**
  - Not so much with Foundation APIs
- **You can use either approach**
  - Properties mean writing less code, but “magic” can sometimes be non-obvious

## Further Reading

- **Objective-C 2.0 Programming Language**
  - “Defining a Class”
  - “Declared Properties”
- **Memory Management Programming Guide for Cocoa**