Memory Management

448460-1
Fall 2011
09/22/2011
Kyoung Shin Park
Multimedia Engineering
Dankook University

Overview

- Allocating and Initializing objects
- Autorelease
- Protocols

Creating Objects

Creating Objects

- 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 * person = nil;
Person = [[Person allloc] init];
```

Implementing your own init method

```
#import "Person.h"

@implementation Person
-(id) init {
    // allow superclass to initialize its state first
    if (self = [super init]) {
        // initialize our subclass here
        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

```
- (id) init {
        return [self initWithName:@"No Name"];
}
- (id) initWithName: (NSString *) name {
        return [self initWithName:name age:0];
}
```

Initializing Object: A subclass of UIView

- UIView's designated initializer
  - You cannot create a UIView without specifying an initial rectangle for it

```
-(id) initWithFrame: (CGRect) aRect { // designated initializer
        if (self = [super initWithFrame:aRect]) {
                // initialize my subclass here
        }
        return self;
}
-(id) initToFit: (Shape *) aShape { // convenience initializer
        CGRect fitRect = [MyView sizeForShape:aShape];
        return [self initWithFrame:fitRect];
}
UIView *view = [[UIView alloc] initWithFrame:myFrame];
```

Getting Objects

- But alloc/init is not the only way to get an object
  - Plenty of classes will give you an object if you ask for one

```
NSString *view = [[UIView alloc] initWithFrame:myFrame];
NSArray *keys = [dictionary allKeys];
NSString *lowerString = [string lowercastString];
NSNumber *n = [NSNumber numberWithFloat:42.0];
NSDate *date = [NSDate date];
```

Finishing Up With an Object

- Who frees the memory for all of these objects?

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

[person setName:@"Jimmy Jones"];
[person setAge:32];

[person castBallot];
[person doSomethingElse];

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

No garbage collection on the iOS platforms, yet.
The answer? Reference counting

Memory Management

- 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

	Allocation	Destruction
C	malloc	free
Objective-C	alloc	dealloc

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 in which **-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];   // retain count=1
[person setName:@"Jimmy Jones"];
[person setAge:32];

[person castBallot];
[person doSomethingElse];

// release person makes retain count=0
[person release];   // person will be destroyed here
```

Reference counting in action

```
Person * person = [[Person alloc] init]; // retain count=1

[person retain];                         // retain count=2

[person release];                        // retain count=1

[person release];  // retain count=0, -dealloc automataically called

person = nil;

[person doSomething];                    // No effect
```

Implementing a –dealloc method

```
#import "Person.h"
@implementation Person
- (int) 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 –[person dealloc]
  - You only deal with **alloc, copy, retain, release**

Object Ownership

```
#import <Foundation/Foundation.h>
@interface Person : NSObject
{
        NSString * name;       // Person class "owns" the name
        int age;
}
- (NSString *) name;
- (void) setName: (NSString *) newName;
- (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 copy];
                // name has 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!
                // We can't execute our responsibility to release it!
}
@end
```

Returning a newly created object

```
-(NSString *) fullName {
        NSString *result;
        result = [[NSString alloc] initWithFormat:@"%@ %@",
                                        firstName, lastName];
        [result release]; // WRONG! Result is released too early!
        return result;    // Method returns bogus value
}
@end
```

Returning a newly created object

```
-(NSString *) fullName {
        NSString *result;
        result = [[NSString alloc] initWithFormat:@"%@ %@",
                                        firstName, lastName];
        // We have now executed our responsibility to release.
        // But it won't actually happen until the caller of this method
        // has had a chance to send retain to result if they want.
        [result autorelease];
        return result;   // JUST RIGHT! Result is released, but not
                         // right away. Caller gets valid object and
                         // could retain if needed
}
@end
NSString *personFullname = [Person fullName];
[personFullname retain];
```

Collections and autorelease

```
@implementation MyObject
-(NSArray *) coolCats {
        // loading up an array or dictionary to return to a caller
        NSMutableArray *resultValue;
        resultValue = [[NSMutableArray alloc] init];
        [returnValue addObject:@"Steve"];
        [returnValue addObject:@"Anna"];
        [returnValue addObject:@"David"];
        [returnValue autorelease];
        return returnValue;
}
@end
```

But there's a better way.
Get an autoreleased NSMutableArray in the first place.
Then fill it up and return it.

Collections and autorelease

```
@implementation MyObject
-(NSArray *) coolCats {
        NSMutableArray *resultValue;
        // use the method array which returns
        // an autoreleased NSMutableArray
        resultValue = [NSMutableArray array];
        [returnValue addObject:@"Steve"];
        [returnValue addObject:@"Anna"];
        [returnValue addObject:@"David"];
        // Now we don't need this autorelease.
        return returnValue;
}
@end
```

But there's an even better way.
Use collection classes **"create with" methods**.

Collections and autorelease

```
@implementation MyObject
-(NSArray *) coolCats {
        // returns an autoreleased NSArray of the items
        return [NSArray arrayWithObjects:@"Steve",
                                @"Anna", @"David", nil];
}
@end
```

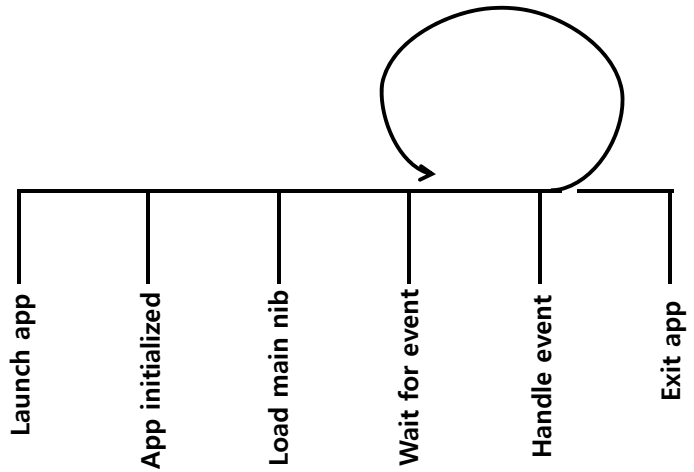
- Other convenient create with methods (all return autoreleased objects):

```
[NSString stringWithFormat: @"Meaning of %@ is %d", @"Life", 42];
[NSDictionary dictionaryWithObjectsAndKeys: kim, @"TA", cho, @"student", nil];
[NSArray arrayWithContensOfFile: (NSString *) path];
```

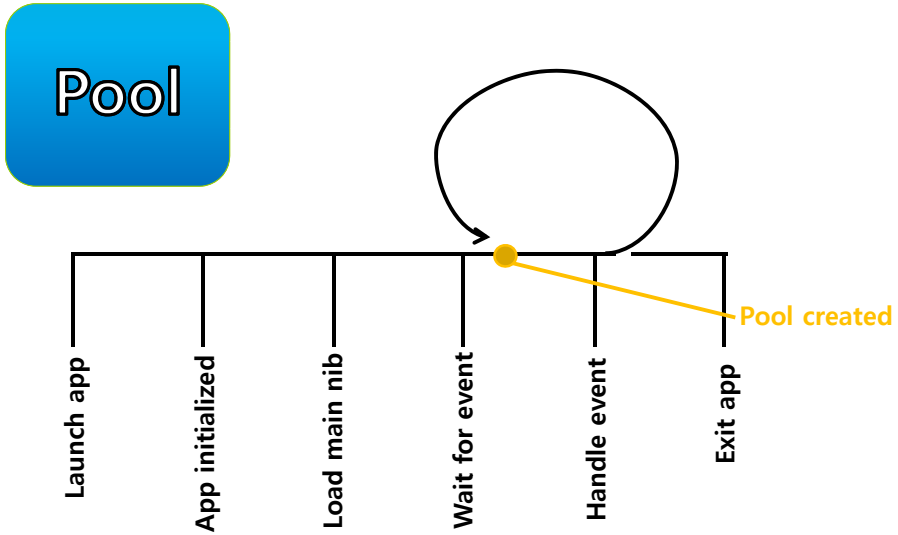
How does autorelease work?

- **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.

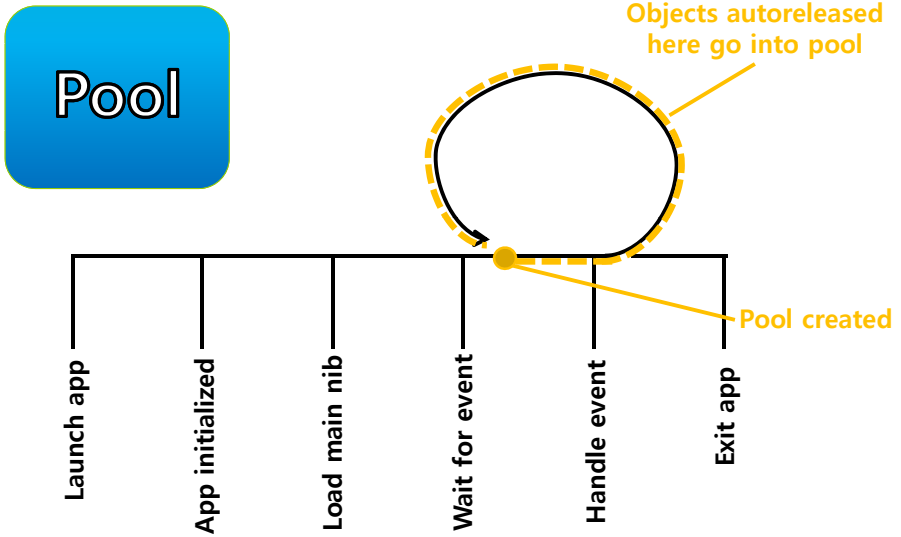
Autorelease Pools



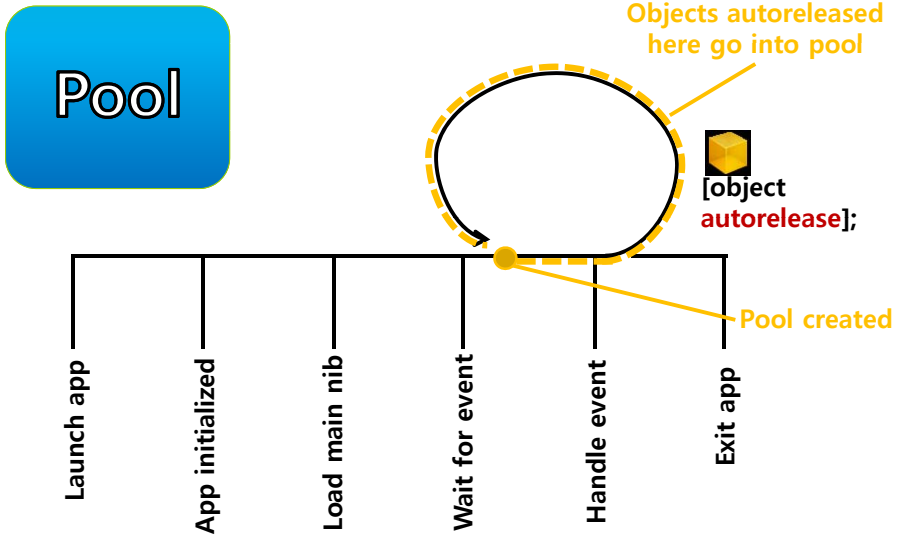
Autorelease Pools



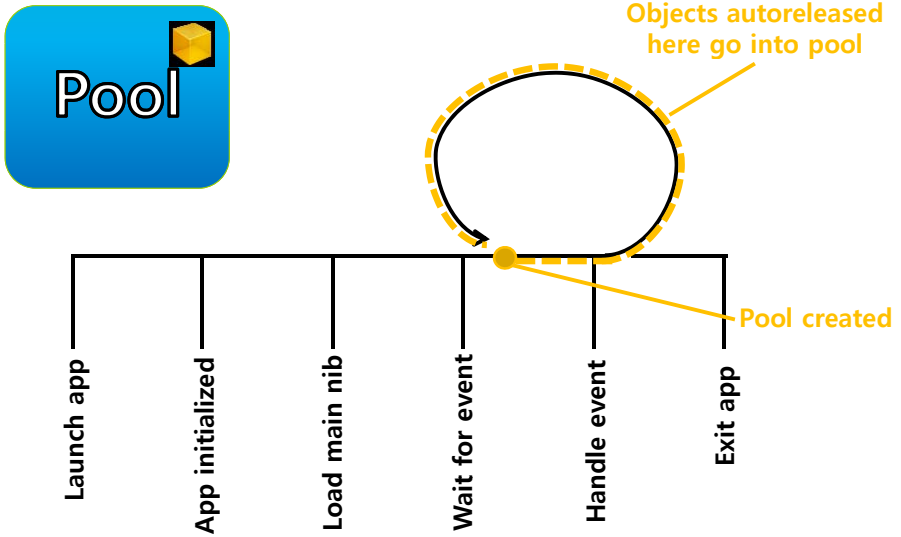
Autorelease Pools



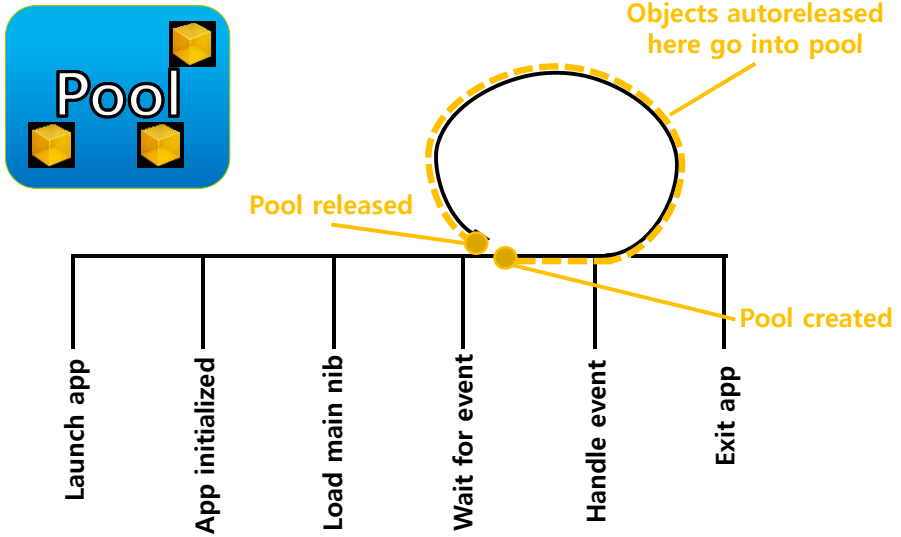
Autorelease Pools



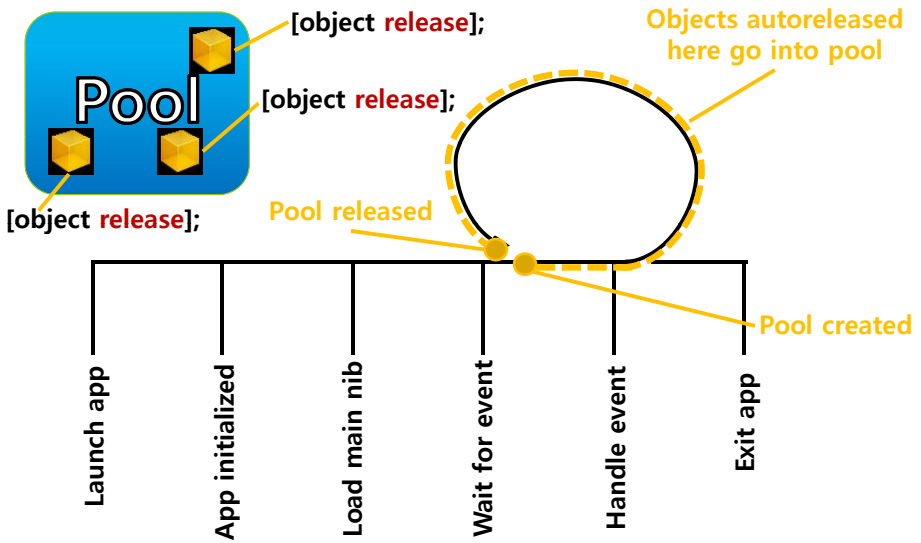
Autorelease Pools



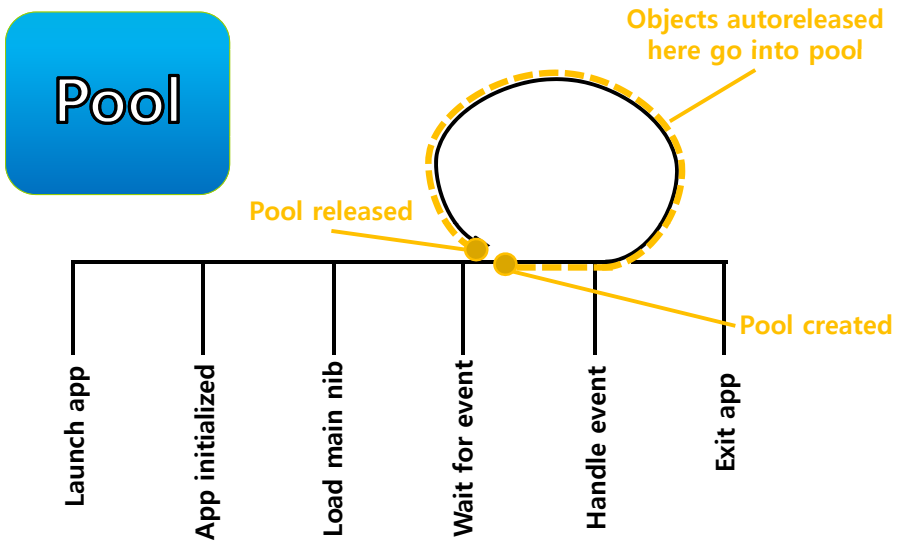
Autorelease Pools



Autorelease Pools



Autorelease Pools



Other Ownership Rules

- Collections take ownership when an object is added to them
  - NSArray, NSDictionary, NSSet release ownership when an object is removed.
- Think of @"string" as autoreleased
  - In reality, they are constants, so retain and release have no effect on them.
- NSString objects are usually sent copy rather than retain
  - That gives you an immutable version of the string to hold on to.
  - The method copy in NSMutableString returns an NSString, not an NSMutableString.
  - Of course, you still release it when you are done with it.
- You should release an object as soon as possible

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 to name to remain valid!
[name retain];
// ...
// Eventually, we'll release it (maybe in our –dealloc?)
[name release];
```

Property Memory Management Policy

- What about **@property**?
  - It's important to understand who owns an object returned from a getter or passed to a setter.
  - Getter methods usually return instance variables directly.
  - What if we use @synthesize to implement our setter? Is retain automatically sent? No
- There are three options for setters made by @synthesize

```
@property (assign) NSString *name; // pointer assignment
@property (retain) NSString *name; // retain called
@property (copy) NSString *name; // copy called
```

Property Memory Management Policy

- Example:

```
@property (retain) NSString *name; // retain called
@synthesize // will create a setter equivalent to this..

-(void) setName: (NSString *)aString {
    [name release]; // @synthesize will release the previous object
    name = [aString retain]; // then, retain the new one
}
```

Property Memory Management Policy

- Example:

```
@property (copy) NSString *name;
@synthesize // will create a setter equivalent to this..

-(void) setName: (NSString *)aString {
    [name release]; // still release before copying
    name = [aString copy];
}
```

Property Memory Management Policy

- Example:

```
@property (assign) NSString *name;
@synthesize // will create a setter equivalent to this..

-(void) setName: (NSString *)aString {
    name = aString; // No release here because we never retain or
                    // copy. This means that if that object is
                    // retained by all other owners, we'll have
                    // a bad pointer. So, we only use assign if the
                    // passed object essentially owns the object
                    // with the property.
}
```

Protocols

Protocols

- Similar to **@interface**, but no implementation
  - Protocols is defined in the header file (e.g. Thing.h)

```
@protocol Thing
-(void) doSomething;  // implementers must implement this
@optional
-(int) getSomething;   // do not need to implement this
@required
-(NSArray *) getManySomethings: (int) howMany; // must implement
@end
```

Protocols

- Classes them implement it
  - They must proclaim that they implement it in their @interface

```
@interface MyClass : NSObject <Thing>
...
@end
```

- You must implement all non-@optional methods
- Can now declare id variables with added protocol requirement

```
id <Thing> obj = [[MyClass alloc] init]; // compiler will love this
id <Thing> obj = [NSArray array]; // compiler will not like this one
```

- Also can declare arguments to methods to require protocol

```
-(void) giveMeThingObject:(id <Thing>) anObjectImplementingThing;
```

Protocols

- Just like static typing, this is all just compiler-helping-you stuff
- Think of it as documentation for your method interfaces
- Number one use of protocols in iOS is **delegate** and **dataSource**
  - The delegate or dataSource is always defined as an assign @property

```
@property (assign) id <UISomeObjectDelegate> delegate;
```

Protocols

```
@protocol UIScrollViewDelegate
@optional
-(void) scrollViewWillBeginDragging: (UIScrollView *) scrollView
-(void) scrollViewDidEndDragging: (UIScrollView *) scrollView
          willDecelerate: (BOOL) decelerate
...
@end
@interface UIScrollView: UIView
@property (assign) id <UIScrollViewDelegate> delegate;
@end
@interface MyViewController: UIViewController <UIScrollViewDelegate>
...
@end
MyViewController @myVC = [[MyViewController alloc] init];
UIScrollView *scrollView = ...;
scrollView.delegate = myViewController;  // compiler won't complain
```

Memory Management Recap

Memory Management Recap

- **alloc/copy/new**
  - If it's an instance variable, probably you're looking at dealloc
  - If it's a local variable, **release** it as soon as you're done using it (at least just before return)
  - If it's a local variable that you are returning, **autorelease** it.
- Watch out when you set a variable multiple times
  - Common mistake: Assigning a new value to an instance variable without **releasing** old value
  - Fix: Use an **@property (retain)** and dot notation to set that instance variable in you code
- **Go immutable!**
  - It's simpler and less error-prone than **allocing** a mutable object, modifying it then **releasing** it
- Try to use methods other than alloc/copy/new
- But don't abuse autorelease by creating a huge pool

Memory Management Recap

Fine:
```
- (NSDictionary *)testVariableValues {
    NSMutableDictionary *returnValue = [[NSMutableDictionary alloc] init];
    [returnValue setObject:[NSNumber numberWithFloat:3.5] forKey:@"x"];
    [returnValue setObject:[NSNumber numberWithFloat:23.8] forKey:@"y"];
    return [returnValue autorelease];
}
```
Better:
```
- (NSDictionary *)testVariableValues {
    NSMutableDictionary *returnValue = [NSMutableDictionary dictionary];
    [returnValue setObject:[NSNumber numberWithFloat:3.5] forKey:@"x"];
    [returnValue setObject:[NSNumber numberWithFloat:23.8] forKey:@"y"];
    return returnValue;
}
```
Best:
```
- (NSDictionary *)testVariableValues {
    return [NSDictionary dictionaryWithObjectsAndKeys:
                                [NSNumber numberWithFloat:3.5], @"x",
                                [NSNumber numberWithFloat:23.8], @"y", nil];
}
```

Memory Management Recap

Bad:
```
- (NSString *) tenThousandGs {
    NSString *s = "";
    for (int i = 0; i<10000; i++) s = [s stringByAppendingString:@"G"];
    return s;
}
```
Fine:
```
- (NSString *) tenThousandGs {
    NSMutableString *s = [[NSMutableString alloc] init];
    for (int i = 0; i<10000; i++) s = [s stringByAppendingString:@"G"];
    return [s autorelease];
}
```
Best:
```
- (NSString *) tenThousandGs {
    NSMutableString *s = [NSMutableString string];
    for (int i = 0; i<10000; i++) [s appendString:@"G"];
    return s;
}
```

References

- Lecture 3 Slide from iPhone Application Development (Winter 2010) @Stanford University