Stanford CS193p
Developing Applications for iPhone 4, iPod Touch, & iPad
Fall 2010
Today

More Foundation Framework
NSArray, NSDictionary, NSSet
Enumeration
Property Lists
User Defaults

More Objective-C
Allocating and Initializing objects
Memory Management

Demo
NSDictionary / Enumeration
NSNumber
Introspection
**Foundation Framework**

**NSArray**
Ordered collection of objects.
Immutable. That's right, you cannot add or remove objects to it once it's created.
Important methods:
- `(int)count;`
- `(id)objectAtIndex:(int)index;`
- `(void)makeObjectsPerformSelector:(SEL)aSelector;`
- `(NSArray *)sortedArrayUsingSelector:(SEL)aSelector;`
- `(id)lastObject; // returns nil if there are no objects in the array (convenient)`

**NSMutableArray**
Mutable version of NSArray.
- `(void)addObject:(id)anObject;`
- `(void)insertObject:(id)anObject atIndex:(int)index;`
- `(void)removeObjectAtIndex:(int)index;`
**NSDictionary**

Hash table. Look up objects using a key to get a value.

Immutable. That’s right, you cannot add or remove objects to it once it’s created.

Keys are objects which must implement `- (NSUInteger)hash` & `- (BOOL)isEqual:(NSObject *)obj`

Keys are usually `NSString` objects.

Important methods:
- `- (int)count;`
- `- (id)objectForKey:(id)key;`
- `- (NSArray *)allKeys;`
- `- (NSArray *)allValues;`

**NSMutableArray**

Mutable version of `NSDictionary`.

- `- (void)setObject:(id)anObject forKey:(id)key;`
- `- (void)removeObjectForKey:(id)key;`
- `- (void)addEntriesFromDictionary:(NSDictionary *)otherDictionary;`
**Foundation Framework**

**NSSet**

Unordered collection of objects.

Immutable. That's right, you cannot add or remove objects to it once it's created.

Important methods:
- (int)count;
- (BOOL)containsObject:(id)anObject;
- (id)anyObject;
- (void)makeObjectsPerformSelector:(SEL)aSelector;
- (id)member:(id)anObject;  // uses isEqual: and returns a matching object (if any)

**NSMutableSet**

Mutable version of NSSet.

- (void)addObject:(id)anObject;
- (void)removeObject:(id)anObject;
- (void)unionSet:(NSSet *)otherSet;
- (void)minusSet:(NSSet *)otherSet;
- (void)intersectSet:(NSSet *)otherSet;
Enumeration

Looping through members of a collection in an efficient manner
Language support using for-in (similar to Java)

Example: NSArray of NSString objects

```c
NSArray *myArray = ...;
for (NSString *string in myArray) {
    double value = [string doubleValue]; // crash here if string is not an NSString
}
```

Example: NSSet of id (could just as easily be an NSArray of id)

```c
NSSet *mySet = ...;
for (id obj in mySet) {
    // do something with obj, but make sure you don’t send it a message it does not respond to
    if ([obj isKindOfClass:[NSString class]]) {
        // send NSString messages to obj with impunity
    }
}
```
Enumeration

Looping through the keys or values of a dictionary

Example:

```objective-c
NSDictionary *myDictionary = ...;
for (id key in myDictionary) {
    // do something with key here
    id value = [myDictionary objectForKey:key];
    // do something with value here
}
```
The term “Property List” just means a collection of collections. Specifically, it is any graph of objects containing only the following classes: NSArray, NSDicionary, NSNumber, NSString, NSDate, NSData.

An NSArray is a Property List if all its members are too. So an NSArray of NSString is a Property List. So is an NSArray of NSArray as long as those NSArray’s members are Property Lists.

An NSDicionary is one only if all keys and values are too. An NSArray of NSDicionaries whose keys are NSStrings and values are NSNumbers is one.

Why define this term? Because the SDK has a number of methods which operate on Property Lists. Usually to read them from somewhere or write them out to somewhere.

[plist writeToFile:(NSString *)path atomically:(BOOL)]; // plist isNSArray or NSDicionary
Other Foundation

NSUserDefaults

Lightweight storage of Property Lists.
It's basically an NSDictionary that persists between launches of your application.
Not a full-on database, so only store small things like user preferences.

Read and write via a shared instance obtained via class method standardUserDefaults
[[NSUserDefaults standardUserDefaults standardUserDefaults] setArray:rvArray forKey:@“RecentlyViewed”];

Sample methods:
- (void)setDouble:(double)aDouble forKey:(NSString *)key;
- (NSInteger)integerForKey:(NSString *)key; // NSInteger is a typedef to 32 or 64 bit int
- (void)setObject:(id)obj forKey:(NSString *)key; // obj must be a Property List
- (NSArray *)arrayForKey:(NSString *)key; // will return nil if value for key is not NSArray

Always remember to write the defaults out after each batch of changes!
[[NSUserDefaults standardUserDefaults standardUserDefaults] synchronize];
Creating Objects

It is a two step operation to create an object in Objective-C

- Allocating (almost always done with the `NSObject` class method `alloc`)
- Initializing (done with a method that starts with the four letters `init`)

- `alloc` makes space in heap for the class’s instance variables
  - Also sets them all to zero, so instance variables which are object pointers start out `nil`

- Each class has a “designated” `initialize` method
  - `NSObject` is `init` (by convention, all `init` methods start with the four letters `init`)
  - When you subclass an object, you must call your superclass’s designated initializer from your designated initializer (and make sure that it did not fail and return `nil`)
  - Your designated initializer should only take arguments that are required for the proper initialization of your class (i.e. if you can have sensible defaults for things, do it)
  - You can have other initializers (convenience initializers) which take other arguments, but those should always call your designated initializer (so that subclasses of yours will work properly)
  - All `init` methods should be typed (in their declaration) to return `id` (not statically typed)
  - Callers should statically type though, e.g., `MyObject *obj = [[MyObject alloc] init];`
Example: A direct subclass of `NSObject`

We use a sort of odd-looking construct to ensure that our superclass *init*ed properly. Our superclass’s designated initializer can return `nil` if it failed to initialize. In that case, our subclass should return `nil` as well.

Here are two versions. Either is okay. The one on the right looks weird because it assigns a value to `self`, but it’s legal.

```objective-c
@implementation MyObject
- (id)init
{
    if ([super init]) {
        // initialize our subclass here
        return self;
    } else {
        return nil;
    }
}
@end
```

```objective-c
@implementation MyObject
- (id)init
{
    if (self = [super init]) {
        // initialize our subclass here
    }
    return self;
}
@end
```
Example: A subclass of CalculatorBrain with a convenience initializer

Imagine that we enhanced CalculatorBrain to have a list of “valid operations.”
We’ll allow the list to be nil which means that all operations are valid.
It might be nice to have a convenience initializer to set that array of operations.
We’d want to have a @property to set the array as well, of course.
Our designated initializer, though, is still init (the one we inherited from NSObject).

@implementation CalculatorBrain
- (id)initWithValidOperations:(NSArray *)anArray
{
    self = [self init];
    self.validOperations = anArray; // will do nothing if self == nil
    return self;
}
@end

Note that we call our designated initializer on self, not super!
We might add something to our designated initializer someday and we don’t want to have to go back and change all of our convenience initializers too.
Also, only our designated initializer should call our super’s designated initializer.
Example: A subclass of `CalculatorBrain` with designated initializer

Let's change our requirements so that there must be at least one valid operation specified. Now we need a new designated (not convenience) initializer that forces that list to be non-empty. This is bad design, by the way. The other assumption about valid operations was better.

```objective-c
@implementation CalculatorBrain
- (id)initWithValidOperations:(NSArray *)anArray
{
    if (self = [super init]) {
        if ([anArray count]) {
            self.validOperations = anArray; // probably should also check validity
        } else {
            self = nil;
        }
    }
    return self;
}
@end
```

Note that we are now calling our `super`'s designated initializer.

```objective-c
- (id)init { return nil; } // ugh! but necessary since validOperations can't be nil
@end
```
Another example: A subclass of `UIView`

`UIView`’s designated initializer is

```
- (id)initWithFrame:(CGRect)aRect;
```

Thus you cannot create a `UIView` without specifying an initial rectangle for it.

Here’s an example of creating a `UIView` (note the statically typed variable) ...

```
UIView *view = [[UIView alloc] initWithFrame:myFrame]; // no compiler warning (just like cast)
```
Initializing Objects

A Tale of Two Initializers

Here are two different implementations of a subclass of `UIView`.
One keeps `initWithFrame:` as its designated initializer & adds a convenience initializer `initWithFit:`.

```objective-c
@implementation MyView
-(id)initWithFrame:(CGRect)aRect {
    if (self = [super initWithFrame:aRect]) {
        // initialize my subclass here
    }
    return self;
}
-(id)initToFit:(Shape *)aShape {
    CGRect fitRect = [MyView sizeForShape:aShape];
    return [self initWithFrame:fitRect];
}
@end
```
Initializing Objects

A Tale of Two Initializers

Here are two different implementations of a subclass of UIView:

One keeps initWithFrame: as its designated initializer & adds a convenience initializer initToFit:
The other makes initToFit: the subclass’s designated initializer (initWithFrame: is a convenience)

@implementation MyView
- (id)initWithFrame:(CGRect)aRect {
    if (self = [super initWithFrame:aRect]) {
        // initialize my subclass here
    }
    return self;
}

- (id)initToFit:(Shape *)aShape {
    CGRect fitRect = [MyView sizeForShape:aShape];
    if (self = [super initWithFrame:fitRect]) {
        // initialize my subclass here
    }
    return self;
}
@end

@implementation MyView
- (id)initWithFrame:(CGRect)aRect {
    return [self initToFit:[self defaultShape]];
}

- (id)initToFit:(Shape *)aShape {
    CGRect fitRect = [MyView sizeForShape:aShape];
    if (self = [super initWithFrame:fitRect]) {
        // initialize my subclass here
    }
    return self;
}
@end

@implementation MyView
- (id)initWithFrame:(CGRect)aRect {
    return [self initToFit:[self defaultShape]];
    }

- (id)initToFit:(Shape *)aShape {
    CGRect fitRect = [MyView sizeForShape:aShape];
    if (self = [super initWithFrame:fitRect]) {
        // initialize my subclass here
    }
    return self;
}
@end
A Tale of Two Initializers

Here are two different implementations of a subclass of UIView.
One keeps initWithFrame: as its designated initializer & adds a convenience initializer initWithFrame:.
The other makes initWithFrame: the subclass's designated initializer (initWithFrame: is a convenience)

@implementation MyView
- (id)initWithFrame:(CGRect)aRect {
    if (self = [super initWithFrame:aRect]) {
        // initialize my subclass here
    }
    return self;
}
- (id)initWithToFit:(Shape *)aShape {
    CGRect fitRect = [MyView sizeForShape:aShape];
    return [self initWithFrame:fitRect];
}
@end

@implementation MyView
- (id)initWithFrame:(CGRect)aRect {
    return [self initWithToFit:[self defaultShape]];
}
- (id)initWithToFit:(Shape *)aShape {
    CGRect fitRect = [MyView sizeForShape:aShape];
    if (self = [super initWithFrame:fitRect]) {
        // initialize my subclass here
    }
    return self;
}
@end

Note that both versions call UIView's designated initializer.
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 *newDisplay = [display.text stringByAppendingString:digit];
NSArray *keys = [dictionary allKeys];
NSString *lowerString = [string lowercaseString];
NSNumber *n = [NSNumber numberWithFloat:42.0];
NSDate *date = [NSDate date]; // returns the date/time right now

Who frees the memory for all of these objects?
Not just the above, but the alloc/init ones too.

Sorry, no garbage collection
Not on the iOS platforms anyway.

The answer?
Reference Counting
Reference Counting

How does it work?
- Simple set of rules everyone must follow.

You take ownership for an object you want to keep a pointer to
- Multiple owners for a given object is okay (common).

When you’re done with an object, you give up that ownership
- There’s a way to take “temporary” ownership too.

When no one claims ownership for an object, it gets deallocated
- After that point, your program will crash if that object gets sent a message!
Object Ownership

When do you take ownership?
You immediately own any object you get by sending a message starting with `new`, `alloc` or `copy`. The most common one of these is, of course, the combination of `alloc` followed by `init...`. If you get an object from any other source you do not own it, but you can take ownership by sending it the `NSObject` message `retain`.

So who owns an object you get not from `new`, `alloc` or `copy`?
The object you got it from will own it temporarily until the call stack unwinds (more on this later). Or, the object you got it from owns it and is going to live long enough for you to `retain` it.

How do you give up ownership when you are done?
Send the object the `NSObject` message `release`. Do not send `release` to an object you do not own. This is very bad.
So how does this “temporary ownership” thing work?
If you want to give someone an object with the “option” for them to take ownership of it, you must take ownership of it yourself, then send the object the message `autorelease` (or obtain a temporarily owned object from somewhere else, modify it, then give it away). Your ownership will “expire” at some future time (but not before the current event is finished). In the meantime, someone else can send `retain` to the object if they want to own it themselves.

Best understood by example

```c
-Money *showMeTheMoney:(double)amount {

}
```
**Temporary Ownership**

So how does this “temporary ownership” thing work?
If you want to give someone an object with the “option” for them to take ownership of it,
you must take ownership of it yourself, then send the object the message `autorelease`
(or obtain a temporarily owned object from somewhere else, modify it, then give it away).
Your ownership will “expire” at some future time (but not before the current event is finished).
In the meantime, someone else can send `retain` to the object if they want to own it themselves.

Best understood by example

```objective-c
-(Money *)showMeTheMoney:(double)amount {
    Money *theMoney = [[Money alloc] init:amount];
}
```

We now own `theMoney` because we got it using the method `alloc`. We are responsible for sending it `release`. 
So how does this “temporary ownership” thing work?
If you want to give someone an object with the “option” for them to take ownership of it, you must take ownership of it yourself, then send the object the message `autorelease` (or obtain a temporarily owned object from somewhere else, modify it, then give it away). Your ownership will “expire” at some future time (but not before the current event is finished). In the meantime, someone else can send `retain` to the object if they want to own it themselves.

Best understood by example

```c
-(Money *)showMeTheMoney:(double)amount {
    Money *theMoney = [[Money alloc] init:amount];
    return theMoney;
}
```

Oops! We just `returned` it. We can’t execute our responsibility to `release` it!
Temporary Ownership

So how does this “temporary ownership” thing work?

If you want to give someone an object with the “option” for them to take ownership of it, you must take ownership of it yourself, then send the object the message `autorelease` (or obtain a temporarily owned object from somewhere else, modify it, then give it away).

Your ownership will “expire” at some future time (but not before the current event is finished). In the meantime, someone else can send `retain` to the object if they want to own it themselves.

Best understood by example

- (Money *)showMeTheMoney:(double)amount {
  Money *theMoney = [[Money alloc] init:amount];
  [theMoney autorelease];
  return theMoney;
}

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 `theMoney` if they want.
So how does this “temporary ownership” thing work?
If you want to give someone an object with the “option” for them to take ownership of it, you must take ownership of it yourself, then send the object the message autorelease (or obtain a temporarily owned object from somewhere else, modify it, then give it away). Your ownership will “expire” at some future time (but not before the current event is finished). In the meantime, someone else can send retain to the object if they want to own it themselves.

Best understood by example
- (Money *)showMeTheMoney:(double)amount {
  Money *theMoney = [[Money alloc] init:amount];
  [theMoney autorelease];
  return theMoney;
}

Caller
Money *myMoney = [bank showMeTheMoney:4500.00];
[myMoney retain];
Loading up an array or dictionary to return to a caller
Imagine you have a method that returns an array of something.

@implementation MyObject

- (NSArray *)coolCats
{
    NSMutuableArray *returnValue = [[NSMutuableArray alloc] init];
    [returnValue addObject:@"Steve"];  
    [returnValue addObject:@"Ankush"];  
    [returnValue addObject:@"Sean"];  
    return returnValue;
}
@end

Bad! We can't release this now.
Loading up an array or dictionary to return to a caller

Imagine you have a method that returns an array of something.

```objc
@implementation MyObject

-(NSArray *)coolCats
{
    NSMutableArray *returnValue = [[NSMutableArray alloc] init];
    [returnValue addObject:@"Steve"];  
    [returnValue addObject:@"Ankush"];  
    [returnValue addObject:@"Sean"];  
    [returnValue autorelease];
    return returnValue;
}
@end
```

All fixed!
Loading up an array or dictionary to return to a caller
Imagine you have a method that returns an array of something.

@implementation MyObject

- (NSArray *)coolCats
{
    NSMutableArray *returnValue = [[NSMutableArray alloc] init];
    [returnValue addObject:@"Steve"];        
    [returnValue addObject:@"Ankush"];       
    [returnValue addObject:@"Sean"];         
    [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.
Loading up an array or dictionary to return to a caller

Imagine you have a method that returns an array of something.

```c
@implementation MyObject

-(NSArray *)coolCats {
    NSMutableArray *returnValue = [[NSMutableArray alloc] init];
    [returnValue addObject:@“Steve”];
    [returnValue addObject:@“Ankush”];
    [returnValue addObject:@“Sean”];
    [returnValue autorelease];
    return returnVal;
}
@end
```

But there’s a better way.

Get an autoreleased `NSMutableArray` in the first place.
Then fill it up and return it.
Loading up an array or dictionary to return to a caller

Imagine you have a method that returns an array of something.

```swift
@interface MyObject
@property (nonatomic, strong) NSArray *coolCats;
@end

@implementation MyObject
- (NSArray *)coolCats {
    NSArray *returnValue = [NSArray array];
    [returnValue addObject:@"Steve"];  
    [returnValue addObject:@"Ankush"];  
    [returnValue addObject:@"Sean"];  
    [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.
Loading up an array or dictionary to return to a caller
Imagine you have a method that returns an array of something.

@implementation MyObject
-

NSMutableArray *returnValue = [NSMutableArray array];
[returnValue addObject:@“Steve”];
[returnValue addObject:@“Ankush”];
[returnValue addObject:@“Sean”];

return returnValue;
@end

But there’s a better way.
Get an autoreleased NSMutableArray in the first place.
Then fill it up and return it.
Loading up an array or dictionary to return to a caller
Imagine you have a method that returns an array of something.

@implementation MyObject

-(NSArray *)coolCats
{
NSMutableArray *returnValue = [NSMutableArray array];
[returnValue addObject:@"Steve"];  
[returnValue addObject:@"Ankush"];  
[returnValue addObject:@"Sean"];  
return returnValue;
}

@end
Collections and autorelease

Loading up an array or dictionary to return to a caller
Imagine you have a method that returns an array of something.

@implementation MyObject

- (NSArray *)coolCats
{
    NSMutableArray *returnValue = [NSMutableArray array];
    [returnValue addObject:@"Steve"];  
    [returnValue addObject:@"Ankush"]; 
    [returnValue addObject:@"Sean"]; 
    return returnValue;
}

@end

But there's an even better way!
Use collection classes "create with" methods.
Loading up an array or dictionary to return to a caller
Imagine you have a method that returns an array of something.

@implementation MyObject

- (NSArray *)coolCats
{
    NSMutableArray *returnValue = [NSMutableArray array];
    [returnValue addObject:@“Steve”];
    [returnValue addObject:@“Ankush”];
    [returnValue addObject:@“Sean”];
    return [NSArray arrayWithObjects:@“Steve”, @“Ankush”, @“Sean”, nil];
}
@end

But there's an even better way!
Use collection classes “create with” methods.
Loading up an array or dictionary to return to a caller

Imagine you have a method that returns an array of something.

```objective-c
@implementation MyObject
-
    (NSArray *)coolCats
    {
        return [NSArray arrayWithObjects:@"Steve", @"Ankush", @"Sean", nil];
    }
@end
```

Other convenient create with methods (all return autorelease objects):

```objective-c
[NSString stringWithFormat:@"Meaning of %@ is %d", @"life", 42];
[NSDictionary dictionaryWithObjectsAndKeys:ankush, @"TA", janestudent, @"Student", nil];
[NSArray arrayWithContentsOfFile:(NSString *)path];
```
Other Ownership Rules

Collections take ownership when an object is added to them
NSArray, NSDictionary, NSSet (NSDictionary takes ownership of both keys and values).
They then 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
In other words, on the next line of code after you’ve finished using it.
The longer it stays owned, the more a reader of your code wonders what it’s going to be used for.
Deallocation

What happens when the last owner calls release?
A special method, dealloc, is called on the object & the object’s memory is returned to the heap.
After this happens, sending a message to the object will crash your program.

You should override dealloc in your classes, but NEVER call it!
It only gets called by release when the last owner has released the object.
The one exception about calling it is that you must call [super dealloc] in your dealloc.
Besides that, if you call it in your homework assignments, you will be soundly reproofed.

Example
-(void)dealloc
{
    [brain release];
    [otherObjectInstanceVariable release];
    [super dealloc];
}

Stanford CS193p Fall 2010
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
But for callers this is practically no different than return an `autorelease`d object.
That's because we assume the implementor of the getter isn't going to be `released` immediately.
So the object we got from the getter will be around long enough for us to `retain` it or not.

Consider `UILabel`'s text property in our Calculator
```swift
display.text = [display.text stringByAppendingString:digit];
```
This comes back still owned by `display` (the `UILabel`).
We don't `retain` it because we are not going to keep it.
We're just going to use it to acquire a different `NSString` from `stringByAppendingString:`
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
But for callers this is practically no different than return an autorelease object.
That’s because we assume the implementor of the getter isn’t going to be released immediately.
So the object we got from the getter will be around long enough for us to retain it or not.

Consider `UILabel`'s `text` property in our Calculator
```
display.text = [[display.text stringByAppendingString:digit] autorelease];
```
This comes back still owned by `display` (the `UILabel`).
We don’t retain it because we are not going to keep it.
We’re just going to use it to acquire a different `NSString` from `stringByAppendingString:`.

This is the `NSString` we acquired.
It comes back autorelease.
We’re not going to retain this one either because we’re just going to pass it on to `display`'s `text` property setter method.
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**
But for callers this is practically no different than return an `autorelease`ed object.
That’s because we assume the implementor of the getter isn’t going to be `released` immediately.
So the object we got from the getter will be around long enough for us to `retain` it or not.

Consider UILabel’s text property in our Calculator
```objective-c
display.text = [[display.text stringByAppendingString:digit] autorelease];
```
This is the `NSString` we acquired.
It comes back `autorelease`ed.
We’re not going to `retain` this one either because we’re just going to pass it on to `display’s` text property setter method.
What about `@property` setter methods?
Imagine setting an instance variable via a property.
Did we `retain` that object that we just set to be one of our instance variables?
We certainly should and we certainly could in our implementation of our setter.
But what if we use `@synthesize` to implement our setter? Is `retain` automatically sent? No.

There are three options for setters made by `@synthesize`
- `@property (retain) NSArray *myArrayProperty;`
- `@property (copy) NSString *someNameProperty;`
- `@property (assign) id delegate;`

The first two are straightforward, the third requires thought
The third (assign) means that neither `retain` nor `copy` is sent to the object passed to the setter.
This means that if that object is `released` 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.
Best example: a Controller and its Views (because a View should never outlive its Controller).
A View can have a property (delegate is very common) which is `assigned` to its Controller.
@property

Example

@property (retain) NSString *name;

@synthesize will create a setter equivalent to this ...

- (void)setName:(NSString *)aString
{
    [name release];
    name = [aString retain];
}

Note that @synthesize will release the previous object (if any, could be nil) before setting and retaining the new one.
Example

@property (copy) NSString *name;

@synthesize will create a setter equivalent to this ...

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

Still releasing before copying.
@property

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.
View Hierarchy and Custom Views
Breaking out from just buttons and text fields

Application Lifecycle
From creation through event-handling and delegate method calling

View Controller Lifecycle
Same thing, but for UIViewControllers

Navigation Controllers
Building multi-screen applications
Demo

**Collector**
Collects strings or numbers touched on in the user interface
Reports how many of each happened

**Model**
Collector

**View**
Bunch of random buttons

**Controller**
CollectorViewController

**Watch for ...**
How we use **NSNumber** to wrap primitives (**ints** in this case)
How we use introspection to have an **NSDictionary** with object of different classes as keys
Using properties (with or without associated instance variables)