

Author: [Drew McCormack](#)

Website: <http://www.macanics.net>

How did we get created? When will we be destroyed? Why am I so forgetful lately? Important questions all. Unfortunately we will not answer any of them today (or in the future for that matter), but they are related to the topic of this installment of the Cocoa for Scientists series: object life-cycle. How does an object get born? When does it get deleted, and how does it deallocate itself? And, last but by no means least: Why am I so forgetful lately...no...how does a Cocoa program manage its memory use?

Allocating and Initializing Objects

[Last time](#) we learned about classes. Classes are like a template that can be used to create objects. A class describes the data and methods that objects of that type contain. For example, Steve Jobs is human (some may argue that he is super-human — or sub-human, depending on your viewpoint — but let's leave that to one side). In an application, we may have a class `Human`, and Steve Jobs could be an instance or object of that class. The class would detail what an object should include, such as a first and last name. By *instantiating* that class, you can create different objects representing different human beings.

One cool thing about being a programmer is that you get to play God. You get to say when something comes into existence, and when it is obliterated. In Cocoa, you create a new object by allocating memory for it to use, and then initializing its instance variables. In many other languages, these two things are rolled into one, but in Cocoa they are distinct phases.

Let's see how this works by returning to the simple `AdditionOperator` class from the [last tutorial](#), in particular the place in the main function where the operator was created:

```
AdditionOperator *operator =  
    [[AdditionOperator alloc] initWithLeftValue:4.0  
    andRightValue:5.0];
```

In order to create an `AdditionOperator` object, first a call is made to `alloc`, and then a call is made to `initWithLeftValue:andRightValue:`. `alloc` can be found in the superclass, `NSObject`. All it does is allocate memory for the new object. You don't really need to know how it does this, and you virtually never need to write your own `alloc` method. You just need to know to call it every time you need a new instance of a class.

`alloc` returns a chunk of memory big enough to store the data of the `AdditionOperator`, but it doesn't give that data any value. To initialize the instance variables, an *initializer* method is called, in this case `initWithLeftValue:andRightValue:`.

Last time I glossed over the initializer, but this time I want to do it justice. Here is the typical form of an initializer:

```
-(id) initWithLeftValue:(double)lv andRightValue:(double)rv {  
    if ( self = [super init] ) {  
        leftValue = lv;  
        rightValue = rv;  
    }  
    return self;  
}
```

The perceptive ones in the audience may point out that this is not exactly the same as the initializer from the last tutorial. True. Last time I just wanted to include something easy to understand, but this time I want to present a real initializer, as you would find it in a typical Cocoa application.

The difference with the former version of the class lies in the `if` block. Cocoa is a framework built upon conventions. If you stick to the conventions, you will be fine; if you flout them, it will hurt. This is an example of a Cocoa convention: An initializer will return the constant `nil` — which is similar to a zero, or the C constant `NULL` — whenever something goes wrong in the initialization. So, in general, you should check the return value to see if it is `nil` after you call an initializer.

It is important that you always call the superclass (via the `super` variable) in your own initializers. Why? Because you have to remember that you have inherited all the instance variables and methods of the superclass. If you don't *chain* to the superclass, the data in that class will remain uninitialized.

But that doesn't explain everything about this `if` block. First, the `self` variable, which you will recall is a pointer to the object being initialized, is assigned to the return value of invoking the `init` method of the superclass (`NSObject`), and then the value of `self` is tested to see if it is `nil`. The reason `self` gets assigned in this way is that Cocoa has a second convention that says that an initializer is allowed to throw away the originally allocated object, and allocate a new one. We will not discuss why you would do this at this stage, but given that it is allowed, the assignment of `self` to the return value of the super class initializer has you covered. (There is actually [some debate](#) about whether this convention makes any sense, but given it has Apple's endorsement, we will stick with it here.)

If `super`'s `init` method succeeds, the `self` variable will not be `nil`, and the block of code after the `if` will get executed. This block initializes the two instance variables of the `AdditionOperator` object. The last line of the initializer returns the `self` object. An initializer always returns the object it initializes. Note that if something had gone wrong in the superclass initializer, `self` would be `nil`, and `nil` would also be the return value of `initWithLeftValue:andRightValue:`.

The Designated Initializer

A class can actually have many different initializers. Different initializers may be used for different circumstances. For example, the Foundation class `NSData`, which can be used to store a block of raw data, can be initialized with the data from a file by invoking the initializer `initWithContentsOfFile:`, or with a pre-existing buffer of bytes by invoking `initWithBytes:length:`. This raises the question: Which superclass initializer should you invoke from your initializer method?

Well, Cocoa has an answer for you here too, and it again comes down to — you guessed it — convention. Cocoa has the concept of a designated initializer, which is the initializer you should generally chain to from a subclass. There is no language support for the designated initializer; it is purely a convention, and, as such, typically involves no more than a statement to the effect in the comments or documentation.

One of the more important designated initializers is the `init` method of the `NSObject` class. More often than not, your own classes will inherit from `NSObject`, and just as in `AdditionOperator`, you should chain to that method of the superclass.

Deallocating Objects

The flip side of allocation and initialization is deallocation. What happens when you destroy an object? In Cocoa, when an object is ready to disappear, the method `dealloc` gets called, which is inherited from `NSObject`. Anything that needs to happen when the object is no longer needed should happen here. For example, any memory that has been dynamically allocated for the object should be deallocated. Resources like open files should also be released.

The version of the `AdditionOperator` class from last time didn't need to include a `dealloc` method, because it didn't need to do anything special when deleted, but we can easily modify it so that `dealloc` is needed. For example, let's assume the interface of `AdditionOperator` is like so:

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

@interface AdditionOperator : NSObject {
    NSNumber *leftValue;
    NSNumber *rightValue;
}

-(id)initWithLeftValue:(double)lv andRightValue:(double)rv;
-(double)evaluate;

@end
```

`NSNumber` is a class in the Foundation framework that can be used to store decimal and other types of numbers.

The class implementation would be:

```
@implementation AdditionOperator

-(id)initWithLeftValue:(double)lv andRightValue:(double)rv {
    if ( self = [super init] ) {
        leftValue = [[NSNumber alloc] initWithDouble:lv];
        rightValue = [[NSNumber alloc] initWithDouble:rv];
    }
    return self;
}

-(void)dealloc {
    [leftValue release];
    [rightValue release];
    [super dealloc];
}

-(double)evaluate {
    return [leftValue doubleValue] + [rightValue doubleValue];
}

@end
```

The initializer has been modified to convert the `doubles` passed into `NSNumber` objects. Note that the same allocation and initialization pattern is used to create the `NSNumber`s as is used to create the `AdditionOperator` itself. The `evaluate` method has also been updated slightly, because now it must retrieve the `double` values (using the `doubleValue` method) from the `leftValue` and `rightValue` objects before they can be added.

The biggest change, however, is the introduction of a `dealloc` method. This is needed to destroy the `NSNumber` objects that were allocated in the initializer. If you don't do this, you get a memory leak, with the `NSNumber` objects remaining in existence until the program exits.

The form of the `dealloc` method above is quite typical. Usually, a number of objects are 'released', and then the `dealloc` method of the superclass is invoked so that it can do any cleaning up it needs to. You may be wondering why the `release` method is called for the two `NSNumber`s, rather than the `dealloc` method. This is part of Cocoa's memory management system, which will be explained in detail next time. For now, the short answer is: `release` will call the `dealloc` method for you, but it first checks to make sure no other objects need to use the `NSNumber`s.

Summary of Patterns

That's basically it for this time, but before finishing, I just want to impress on you one more time the basic form of an initializer and a deallocator in Cocoa.

```
-(id)init... {
    // Chain to designated initializer of superclass
    if ( self = [super init] ) {
        ...
    }
    return self;
}

-(void)dealloc {
    ...
    [super dealloc];
}
```

If you can internalize that, you are ready for the next step, Memory Management, which will be the topic of the next discussion.

If you are wondering whether we will ever get to make a Cocoa app with Xcode and Interface Builder, be patient. You can't really do that stuff effectively unless you understand the basics of Cocoa classes. We'll get to graphical interfaces and move into Xcode after a couple more tutorials, and then the fun will really begin.