

Author: [Drew McCormack](#)

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

In the [first installment](#) of this series on Cocoa programming for scientists, you got your first glimpse of Objective-C code. Hopefully you are now sufficiently recovered from the shock of that encounter to start digging deeper. In this part, we'll take a closer look at the object oriented building blocks of Objective-C: classes and objects.

Object-Oriented Programming and Dead Horses

Many scientists have a natural aversion to the object oriented (OO) style of programming. Some argue that it results in 'slow' code, others that it hides the subtleties of an algorithm in little black boxes where you can't look. While these sorts of arguments are rooted to some degree in fact, they are generally exaggerated, or miss the point entirely. In fact, object oriented programming (OOP) has been a raging success outside scientific circles, and with good reason. Heck, even [Fortran](#) now supports it! So it's time to stop flogging a long since deceased horse, and try to come to terms with the new paradigm. I'm here to help, and who knows, you may even like it.

Classes vs Types

OOP may seem like a completely different animal when you first encounter it, but after a while, you realize it actually still has four legs, a head, and a tail. It is actually a natural evolution of the procedural programming model most scientists are already familiar with.

Take the *class*, for example. Classes are the basic building blocks of an OO program, but they are really just like structs on steroids. In C, a struct defines a data type that contains variables. For example, you might find the following in a molecular simulation package:

```
struct Atom {
    double mass;
    double position[3];
}
```

Atom is a type containing the variables `mass` and `position`. Together, these variables constitute an Atom.

In Objective-C, Atom could be a class.

```
@interface Atom : NSObject {
    double mass;
    double position[3];
}
@end
```

As you can see, this is not dissimilar. The syntax now includes the `@interface` and `@end` tokens, and a different class (`NSObject`) also makes an appearance, but otherwise things are pretty much the same.

So where are the steroids referred to earlier? Well, the first is hidden in the rather innocent looking `NSObject` reference. This is the *super class* of `Atom`. `Atom` gets to reuse all the stuff defined in

the `NSObject` class for free — `Atom` *inherits* everything in `NSObject`. We'll discuss this in more detail later in the series when we cover Inheritance and Polymorphism.

The second enhancement is actually not very clear from this example, but it has to do with the space between the closing curly brace and the `@end` keyword. In this space you can declare *methods*, which are basically functions that belong to the class. If the variables in a class define its state, the methods define its behavior.

So a class is really a high-level type — a type with state *and* behavior. Not only that, but one class can inherit from another, getting lots of its variables and methods for free. Hopefully, as this course progresses, you will begin to appreciate how useful these seemingly innocuous attributes of classes really are.

Class Interface

To make this discussion of classes more concrete, we are going to write a simple one that adds numbers together. The class will be called `AdditionOperator`, and its sole purpose will be to sum two decimal numbers.

To begin with, we define the *interface block* of the class, which declares its variables and methods.

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

@interface AdditionOperator : NSObject {
    double leftValue, rightValue;
}

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

@end
```

The interface of a class gets declared in a header file, in the great tradition of C-based languages. In this case, the code above would be entered into a file called 'AdditionOperator.h'.

The code itself begins by importing the Foundation framework, which we met for the first time last week. Foundation contains most of the Cocoa classes that are not directly related to the user interface. It contains classes for strings, for example, and containers like arrays and dictionaries. The `NSObject` class is also contained in Foundation, and that is why it is needed here. In practice, you always need to include either the Foundation framework, or the all-encompassing Cocoa framework.

To use a framework, you use the `#import` preprocessor directive. This is an Objective-C extension to the C language preprocessor. In C, you use the `#include` directive to import a file, but this does not prevent a conflict from arising if a piece of code ends up being included twice. The `#import` directive does prevent this, by first checking if the code in question has already been included, and, if so, skipping over it.

Note that when you use the `#import` directive with a framework, you use triangular brackets, and enter the included file after the name of the framework itself. This is because Mac OS X uses a *two-level namespace*. The idea is to avoid naming conflicts: Imagine that you had another framework

that also included a header file called 'Foundation.h'. This would conflict with the 'Foundation.h' file in the Foundation framework, and the preprocessor wouldn't know which file to import. To resolve this, the name of the framework is given in the import directive.

Now we get to the class interface itself. `AdditionOperator` is a *subclass* of `NSObject`; it inherits every variable and method contained in `NSObject`. Just about all Cocoa classes ultimately inherit from `NSObject`, as we'll discuss in detail later. `NSObject` includes various methods, including some for initialization and deletion of objects, and others for memory management. We'll meet these in due course.

The variables in a class are known as *instance variables*, or *ivars* for short. This is because they belong to an object, or *instance*, of the class in question. In the case of `AdditionOperator`, there are two: `leftValue` and `rightValue`. These are designed to store the left and right values in an addition expression like '5 + 2'. In this example, `leftValue` would be 5, and `rightValue` would be 2.

`AdditionOperator` defines two new methods: `initWithLeftValue:andRightValue:` and `evaluate`. (I say 'new methods' because the class also includes all of the methods defined in `NSObject`.) The first of these methods is an *initializer*, which sets up a new object in a valid state when it is first created. An initializer is always called when creating a new object in Objective-C. The second method is invoked to calculate the value of the operator, that is, to add the left and right value together.

The syntax of the method declarations may have you a bit perplexed. Let's consider the initializer in order to clarify things:

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

It begins with a hyphen. This indicates that the method is an *instance method*, and belongs to an object (instance) of the class, rather than to the class itself. This may not make sense now, but it should become clear later. If you want to write a method that belongs to the class, ie that is shared by all objects of that class, you use a '+' symbol instead of a hyphen.

The naming of the method uses the segmented style that was first introduced in the first tutorial. Hopefully you are somewhat used to it by now. The difference here is that the type of each argument is included in parentheses after the colons. This indicates that the arguments `lv` and `rv` should be of the type `double`, a double precision number.

You'll also notice that the type `id` is given in parentheses just after the hyphen. This is the return type of the method. If there is no return value, you enter `void` here, similar to plain C. The type `id` is special in Objective-C; it is used to mean an object of any class.

Class Implementation

The header file defines the public interface of a class, its variables and methods. The implementation of the methods is found in a second file, with the extension 'm'. For `AdditionOperator`, the following implementation appears in the file 'AdditionOperator.m'.

```
#import "AdditionOperator.h"
```

```
@implementation AdditionOperator
```

```
-(id) initWithLeftValue:(double)lv andRightValue:(double)rv {
```

```

    [super init];
    leftValue = lv;
    rightValue = rv;
    return self;
}

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

@end

```

This file begins with another import, that of the class header. This is needed so that the compiler knows about the class interface as it processes the implementation. Note that when importing one of your own headers, you use quotation marks, rather than the triangular brackets used for framework header files.

Analogous to the `@interface` block, the implementation of a class is contained between `@implementation` and `@end` tokens. Each of the methods declared in the header file — and often many that aren't — are defined in the implementation block. (An example of when a method can appear in the implementation block and not in the interface block is when it gets declared in the superclass. This is known as *overriding* a method, and will be discussed later in the series.)

The method implementations can make reference to the arguments they are passed, of course, but also to the class instance variables. For example, in the initializer method, the instance variables `leftValue` and `rightValue` are assigned the same values as the arguments `lv` and `rv`, respectively. The instance variables belong to the object that is being initialized; the object can be referred to in any instance method using the special variable `self`. (For those familiar with C++ or Java, `self` is equivalent to the `this` pointer.) In setting the instance variables, it would be just as valid to write

```

self->leftValue = lv;
self->rightValue = rv;

```

because `leftValue` and `rightValue` are actually contained in the object `self`, but as long as there is no ambiguity in the variable names, you do not need to explicitly do this.

Aside from the special `self` variable, which is magically passed to any instance method for you, there is also `super`. `super` is a bit like `self` — it is a pointer to the object that owns the method — but there is one major difference: `super` is used to access variables and methods in the superclass.

In the initializer above, the method `init` is invoked for `super`. This means that the `init` method in `NSObject` will be invoked. Even if there was an implementation of `init` in the `AdditionOperator` class — which there isn't — it would not be invoked, because when you use `super` you are saying that you want to look in the superclass, not the immediate class.

Testing It Out

To test this simple class, generate the `AdditionOperator` files (ie `'AdditionOperator.h'` and `'AdditionOperator.m'`), and then create a file called `'main.m'` with the following contents:

```
#import "AdditionOperator.h"

int main() {
    AdditionOperator *operator = [[AdditionOperator alloc]
initWithLeftValue:4.0
    andRightValue:5.0];
    NSLog(@"4.0 + 5.0 is %f", [operator evaluate]);
    [operator release];
    return 0;
}
```

This creates an `AdditionOperator` object with left value 4.0, and right value 5.0, and prints out the result of evaluating the operator. Don't concern yourself too much with things you don't understand in the main function, because they will be dealt with in the coming weeks.

To compile and run this, issue the following commands:

```
gcc -ObjC AdditionOperator.m main.m -framework Foundation
./a.out
```

The output should look like similar to this:

```
2006-10-31 14:24:07.758 a.out[1470] 4.0 + 5.0 is 9.000000
```

Stay Tuned

Today you got your first glimpse of the internals of a class. Next time we will look in more detail at how classes work, from initializing objects to inheritance. In the meantime, try to get your head around some of the concepts introduced today. OO is a foreign concept for most scientists, but if you stick at it, it won't be long before it all fits together. In the beginning it seems to just be made up of lots of jargon and seemingly abstract concepts, but there is a reason to the madness, and it won't be long before it starts falling into place, and 'invoking the designated initializer of a superclass' will become second nature.