

Author: [Drew McCormack](#)

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

Any Objections to Objective-C?

When I recently joined MacResearch, I had to decide what contribution I could make to the site and its content. In the past few years I've regularly written [online articles](#) on Cocoa development, which culminated in a [book](#), so a component of Cocoa was not out of the question. But I'm also a [Research Scientist](#) (Theoretical Chemistry), code daily in dinosaurs like Fortran and C, and have even written a couple of [short courses](#) on various aspects Scientific Programming. What I am yet to do is develop a course on Cocoa for Scientists, and by a marvelous juxtaposition of events, I am now in a position to do just that.

There are lots of places to learn Cocoa, not the least of which is Apple's own [documentation](#), but there isn't any material specifically targeted at Scientists and Scientific Programmers. In my experience, Scientists require special treatment when it comes to learning technologies like Cocoa, because they can't afford to waste too much time on such frivolities — after all, there is Science to be done — and many find it difficult to grasp some of the concepts introduced because they are so deeply entrenched in the practices of yesteryear (Who can spell 'Fortran 77?'). It's nobody's fault — it's just the way it is.

In this series of tutorials, I want to introduce Apple's flagship development frameworks in a way that will appeal more to someone brought up on a staple diet of Fortran or C, rather than Java or C#. I will relate concepts foreign to many scientific developers, like Object Orientation (OO), back to procedural languages like Fortran, in order to emphasize the motivations behind the language design, and highlight the benefits — and potential drawbacks — that arise.

Of course, the language Objective-C is only one aspect of Cocoa, some would even say the most minor aspect. The rest comprises of the frameworks themselves, which allow you to develop flashy graphical interfaces to impress your friends, and maybe even facilitate some useful research. We'll get to that too, but unlike the vast majority of Cocoa introductions, we'll start where computational scientists are most comfortable: on the command line. Later, we'll move into Apple's

Integrated Development Environment (IDE), Xcode, and start to develop graphical programs.

The examples I'll be presenting will be necessarily simple, but I will target the types of applications that Scientists are most interested in writing, rather than ones that might interest the neighbor's kid. Topics along the way will include: parsing/importing data; presenting data in tables and plots; visualization; and wrapping existing legacy tools in a graphical interface.

I hope to keep the tutorials rather short. None of us have much time to spare, and you are more likely to be able to digest a number of concise tutorials than one long monster. The tutorials should appear every 2–3 weeks, depending on how much time I can free up. (In case you didn't realize, MacResearch is run on a voluntary basis.)

Objective-C, Ugly? I Object!

Today, I want to start by introducing you to an ugly duckling: Objective-C. When you first come across a piece of Objective-C, it may appear very ugly indeed, simply because it doesn't follow some of the conventions adopted in C-based programming languages that you might be more used to, like C++ and Java. What I want to assure you is that Objective-C is in fact a beautiful Swan, and after a few weeks playing with it, I'm sure you will agree — they all do in the end.

Objective-C is actually a superset of the C language, a Frankensteinian cross with the object-oriented language Smalltalk. Like Mary Shelley's monster, the syntax can be a bit frightening to begin with, but after a couple of days, you will wonder why other languages don't do it like Objective-C. Rather than tell you about it, though, I'll just show you instead. Here is our first Objective-C program:

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

// -----
// Program that sums a series of whitespace delimited numbers passed in via
// standard input.
// -----
int main(int argc, char *argv[])
{
    int retCode = 0;
    NSAutoreleasePool *pool = [NSAutoreleasePool new];
```

```
// Get file handle for standard input,
// and read in a string of numbers

NSFileHandle *fileHandle = [NSFileHandle fileHandleWithStandardInput];
NSData *inputData = [fileHandle readDataToEndOfFile];
NSString *inputString =
    [[NSString alloc] initWithData:inputData
encoding:NSUTF8StringEncoding];

// Split the string on whitespace, and convert to numbers
NSArray *tokens = [inputString componentsSeparatedByString:@" "];
double sum = 0.0;
unsigned int i;
for ( i = 0; i < [tokens count]; ++i ) {
    sum += [[tokens objectAtIndex:i] doubleValue];
}

// Print output
NSLog(@"The sum of the numbers was %g.", sum);

// Release objects
[inputString release];

[pool release];
return retCode;
}
```

Rather than overwhelming you with the complexities of the Xcode development environment, and to show that there is nothing particularly magic about Objective-C, we are going to compile this with the standard GCC compiler in Terminal. Our objective, see, is to not be all at sea with Objective-C. Capische?

Before we do though, you will need to install the Xcode development tools if you do not already have them. The GCC compiler is included in the install. The developer tools installer can be found on a Mac OS X install disk, or can be [downloaded](#) from Apple's developer site. (Be warned: the download is several hundred megabytes!)

With the developer tools installed, use your favorite text editor to copy the code above into a file called `Adder.m`. Then, drop into Terminal, and issue this command to compile an executable:

```
gcc -ObjC -o adder Adder.m -framework Foundation
```

Now run the newly formed executable by issuing the command `./adder`. Enter a line of decimal numbers separated by whitespace, and then hit

Control-D to close the standard input pipe. You should see the sum of the numbers printed out. Here is an sample session:

```
./adder
1.24 2.35 245.1
2006-08-18 06:41:21.461 adder[14463] The sum of the numbers was 248.69.
```

Compiling Objective-C Code

Now let's go over what just happened, beginning with the compile command. Normally you would never have to type such a command when developing with Cocoa, because Xcode would do that for you, but I wanted to demonstrate that compiling an Objective-C program is actually very similar to compiling a C program. You need to include the `-objc` flag; add a `.m` extension to each source file, rather than `.c`; and link the Foundation framework. (If you have never compiled for Mac OS X before, you may not be familiar with the `-framework` link option. It is just the equivalent of the library link option `-l`, but for frameworks.)

The Foundation framework is part of Cocoa. You can see that in the program above it is imported using the `#import` preprocessor directive, which is an Objective-C extension to the standard C directives. The Foundation framework provides the *classes* that are used in `Adder.m`, like `NSString` and `NSData`. (If you're wondering, the `NS` prefix is used throughout Cocoa to avoid naming conflicts with other frameworks. There is some contention as to where it comes from, but many believe it is an abbreviation of NeXTStep, the forebear of Mac OS X.)

Introducing Classes and Objects

Classes are similar to structs in C, or user-defined types in Fortran 90, but are more powerful. We will leave most discussion of classes for later, but for now, you should know that each class describes a type that can contain data, in the form of variables, and can have *methods*, which are similar to functions and subroutines in Fortran and C. The difference is that the methods belong to a particular class.

In C and Fortran 90, you can define variables of a given type. For example, a C program might have code like this

```
struct Cow {
    float weight;
```

```
};  
struct Cow c;
```

which in Fortran 90 might look like this

```
type Cow  
  real :: weight  
end type  
type (Cow) :: c
```

These are both examples of *instantiation* of a given type. You are creating a variable or *instance*, called `c`, of a given type, `Cow`.

You can do the same thing in Objective-C with classes. When you instantiate a class, you get an instance or *object*. An example of an instantiation from `Adder.m` is

```
NSAutoreleasePool *pool = [NSAutoreleasePool new];
```

Don't worry at this point about what the class `NSAutoreleasePool` actually does, but concentrate instead on the object creation process. The variable in this case is called `pool`. (Note that it is actually a pointer to an object, rather than an object itself; this is always the way objects are used Objective-C.) `pool` gets assigned to an object that is created by calling a method of the `NSAutoreleasePool` class called `new`.

We will once again have to defer any further discussion of object instantiation to a later date. For now it is enough to know that a class describes the data corresponding to a particular type, and its methods. And, when you instantiate a class, you get an object.

Introducing Messaging and Methods

To finish off this tutorial, I want to focus on *messaging*. You have just seen an example of messaging in the line of code above. Messaging involves sending an object or class a message, and is similar to, though not identical to, calling a function or subroutine in Fortran or C. In Objective-C, the syntax of sending a message makes use of square braces, like in this example from `Adder.m`:

```
NSArray *tokens = [inputString componentsSeparatedByString:@" "];
```

The message is called `componentsSeparatedByString:`, and it is being sent to the object `inputString`. (Note that the colon is part of the name; we will discuss this shortly.) The `@" "` represents a literal string in Objective-C, in this case a blank character. This code is sending the message `componentsSeparatedByString:` to the object `inputString`, and passing it a string containing a space character as argument.

In Fortran 90, you would do something similar like this:

```
character(len=100) :: tokens(10)
character(len=100) :: inputString
inputString = "Hello there"
tokens = ComponentsSeparatedByString(inputString, " ")
```

The square braces turn things on their head somewhat. Notice that the object `inputString` is passed as the first argument to the function in the Fortran 90 code. In Objective-C, because the function — or *method* as it is known in the object-oriented terminology — belongs to the object `inputString`, you don't have to explicitly pass it in — it is passed automatically.

Things get even more strange when you have more than one argument to pass. Take this code from `Adder.m`, for example:

```
NSString *inputString = [[NSString alloc] initWithData:inputData
encoding:NSUTF8StringEncoding];
```

First of all, there are two messages here, one nested in the other. The first is simple: `alloc`. It has no arguments, and does not have a colon as a result. In Objective-C, there is one colon per argument.

The second message is more complex: The message is sent to the return value of the `alloc` method. The second message is called `initWithData:encoding:`. Such a name is called a *selector* in Objective-C; it is a label for a method. There are two colons included, and two arguments corresponding to those colons: `inputData` and `NSStringEncoding`.

What is probably most confusing, though, is that the name is split up into segments, one for each colon. To clarify this, let's take a look at what the analogous function call would be in a C program:

```
NSString *inputString = initWithData_encoding_( AllocString(), inputData,  
NSUTF8StringEncoding );
```

Because colons cannot appear in C function names, I have used underscores instead. Note how the function name is concatenated together in the C version — and most other programming languages — whereas in the Objective-C case it is mixed with the arguments.

This is probably the single most difficult thing to grasp when first learning Objective-C. It is foreign to most people, and takes some getting used to. But let me assure you, you will get used to it, and it won't take very long. And when you do, you will start to appreciate some of the advantages of this system, as I will now explain.

Firstly, I should point out that splitting up the name is actually optional. You could choose to use a C-like method naming scheme, and write code like this:

```
NSString *inputString = [[NSString alloc] initWithData_encoding_:inputData  
:NSUTF8StringEncoding];
```

With this scheme, which is perfectly legal Objective-C, the name has been conglomerated at the beginning. Apart from the use of square braces, and colons in place of commas, this code looks quite a bit like the C version. So C-like naming is possible, but is unconventional in Objective-C code. I should emphasize that this method is not equivalent to the method given earlier — they are not interchangeable. The point I want to make is simply that different naming schemes are possible, and you can even use naming schemes similar to those from other languages.

The segmented naming approach of Objective-C is actually more flexible than schemes used by other languages. You have the freedom to segment your method names, better describing the arguments. Code becomes self-documenting, meaning it is easier to read, and you don't need to write as many comments. You also don't need to lookup the arguments of a given method as often. When programming in C or Fortran, it is common to have to lookup a routine to remind oneself how the arguments should be ordered. In Objective-C, it is generally obvious:

```
[NSString stringWithContentsOfFile:path encoding:NSUTF8StringEncoding  
error:error];
```

It reads like a sentence and, as we'll see in a future tutorial, with Xcode's code completion you don't need to type every letter.

Finishing Up

That's it for now. If you only take one thing away from this installment, make it the Objective-C messaging syntax. Think about it over the few weeks before our next tutorial, and get used to the idea. This is what scares most people initially, but there is a rationale behind it, it is just not the approach most are used to. Once you have the messaging syntax under your belt, the rest should be a breeze.

If you are concerned that there are still many things in `Adder.m` that you don't understand, don't worry: we will tackle them all in good time. Today's lesson was designed to start getting you used to how Objective-C looks, and not much more than that.

Next time we'll delve further into the language, taking a closer look at how messaging works, and how you write classes.