Learning Objective-C: A Primer

The Objective-C language is a simple computer language designed to enable sophisticated object-oriented programming. Objective-C extends the standard ANSI C language by providing syntax for defining classes, methods, and properties, as well as other constructs that promote dynamic extension of classes. The class syntax and design are based mostly on Smalltalk, one of the first object-oriented programming languages.

If you have programmed with object-oriented languages before, the following information should help you learn the basic syntax of Objective-C. Many of the traditional object-oriented concepts, such as encapsulation, inheritance, and polymorphism, are all present in Objective-C. There are a few important differences, but those differences are called out in this article and more detailed information is available if you need it.

If you have never programmed using an object-oriented language before, you need to have at least a basic understanding of the associated concepts before proceeding. The use of objects and object-oriented constructs is fundamental to the design of iPhone applications, and understanding how they interact is critical to creating your applications. For an overview of object-oriented concepts, see Object-Oriented Programming with Objective-C. In addition, see Cocoa Fundamentals Guide for information about the object-oriented design patterns used in Cocoa.

For a more detailed introduction to the Objective-C language and syntax, see The Objective-C 2.0 Programming Language.

Objective-C: A Superset of C

Objective-C is a superset of the ANSI version of the C programming language and supports the same basic syntax as C. As with C code, you define header files and source files to separate public declarations from the implementation details of your code. Objective-C header files use the file extensions listed in Table 1.

Table 1 File extensions for Objective-C code

Extension	Source type
`.h`	Header files. Header files contain class, type, function, and constant declarations.
`.m`	Source files. This is the typical extension used for source files and can contain both Objective-C and C code.
`.mm`	Source files. A source file with this extension can contain C++ code in addition to Objective-C and C code. This extension should be used only if you actually refer to C++ classes or features from your Objective-C code.

When you want to include header files in your source code, you can use the standard `#include` compiler directive but Objective-C provides a better way. The `#import` directive is identical to `#include`, except that it makes sure that the same file is never included more than once. The Objective-C samples and documentation all prefer the use of `#import`, and your own code should too.

Strings

As a superset of C, Objective-C supports the same conventions for specifying strings as C. In other words, single characters are enclosed by single quotes and strings of characters are surrounded by double quotes. However, most Objective-C frameworks do not use C-style strings very often. Instead, most frameworks pass strings around in `NSString` objects.

The `NSString` class provides an object wrapper for strings that has all of the advantages you would expect, including built-in memory management for storing arbitrary-length strings, support for Unicode, `printf`-style formatting utilities, and more. Because such strings are used commonly though, Objective-C provides a shorthand notation for creating `NSString` objects from constant values. To use this shorthand, all you have to do is precede a normal, double-quoted string with the `@` symbol, as shown in the following examples:

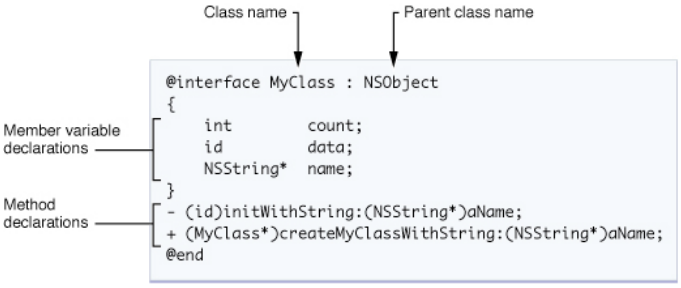
```
NSString*  myString = @"My String\n";
NSString*  anotherString = [NSString stringWithFormat:@"%d %s", 1, @"String"];

// Create an Objective-C string from a C string
NSString*  fromCString = [NSString stringWithCString:"A C string" encoding:NSASCIIStringEncoding];
```

Classes

As in most other object-oriented languages, classes in Objective-C provide the basic construct for encapsulating some data with the actions that operate on that data. An object is simply a runtime instance of a class, and contains its own in-memory copy of the instance variables declared by that class and pointers to the methods of the class.

The specification of a class in Objective-C requires two distinct pieces: the interface and the implementation. The interface portion contains the class declaration and defines the instance variables and methods associated with the class. The implementation portion contains the actual code for the methods of the class. Figure 1 shows the syntax for declaring a class called `MyClass`, which inherits from the `NSObject` base class. The class declaration always begins with the `@interface` compiler directive and ends with the `@end` directive. Following the class name (and separated from it by a colon) is the name of the parent class. The instance (or member) variables of the class are declared in a code block that is delineated by braces (`{` and `}`). Following the instance variable block is the list of methods declared by the class. A semicolon character marks the end of each instance variable and method declaration.

Figure 1 A class declaration



Listing 1 shows the implementation of `MyClass` from the preceding example. Like the class declaration, the class implementation is identified by two compiler directives—here, `@implementation` and `@end`. These directives provide the scoping information the compiler needs to associate the enclosed methods with the corresponding class. A method's definition therefore matches its corresponding declaration in the interface, except for the inclusion of a code block.

Listing 1 A class implementation

```
@implementation MyClass

- (id)initWithString:(NSString *) aName
{
    if (self = [super init]) {
        count count = 0;
        data = nil;
        name = [aName copy];
        return self;
    }
}

+ (MyClass *)createMyClassWithString: (NSString *) aName
{
    return [[[self alloc] initWithString:aName] autorelease];
}
@end
```

Note: Although the preceding class declaration declared only methods, classes can also declare properties. For more information on properties, see "Properties".

When storing objects in variables, you always use a pointer type. Objective-C supports both strong and weak typing for variables containing objects. Strongly typed pointers include the class name in the variable type declaration. Weakly typed pointers use the type `id` for the object instead. Weakly typed pointers are used frequently for things such as collection classes, where the exact type of the objects in a collection may be unknown. If you are used to using strongly typed languages, you might think that the use of weakly typed variables would cause problems, but they actually provide tremendous flexibility and allow for much greater dynamism in Objective-C programs.

The following example shows both strongly and weakly typed variable declarations for the `MyClass` class:

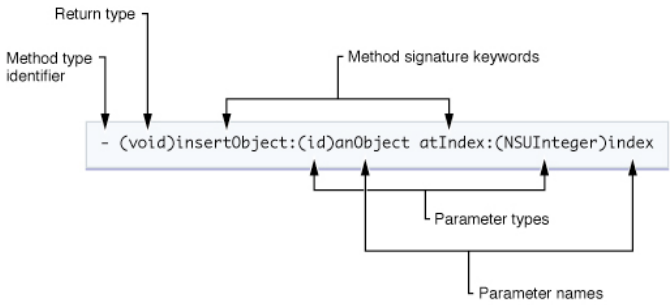
```
MyClass*  myObject1;    // Strong typing
id        myObject2;    // Weak typing
```

Methods

A class in Objective-C can declare two types of methods: instance methods and class methods. An instance method is a method whose execution is scoped to a particular instance of the class. In other words, before you call an instance method, you must first create an instance of the class. Class methods, by comparison, do not require you to create an instance, but more on that later.

The declaration of a method consists of the method type identifier, a return type, one or more signature keywords, and the parameter type and name information. Figure 2 shows the declaration of the `insertObject:atIndex:` instance method. The declaration is preceded by a minus (-) sign, which indicates that this is an instance method. The method's actual name (`insertObject:atIndex:`) is a concatenation of all of the signature keywords, including colon characters. The colon characters declare the presence of a parameter. If a method has no parameters, you omit the colon after the first (and only) signature keyword. In this example, the method takes two parameters.

Figure 2 Method declaration syntax



When you want to call a method, you do so by "messaging" the corresponding object. The message in this case is the method signature, along with the parameter information the method needs. All messages you send to an object are dispatched dynamically, thus facilitating the polymorphism behavior of Objective-C classes. In other words, if a subclass defines a method with the same signature as one of its parent classes, the subclass receives the message first and can opt to forward the message (or not) to its parent.

Messages are enclosed by brackets ([and]). Inside the brackets, the object receiving the message is on the left side and the message (along with any parameters required by the message) is on the right. For example, to send the `insertObject:atIndex:` message to an object in the `myArray` variable, you would use the following syntax:

```
[myArray insertObject:anObj atIndex:0];
```

To avoid declaring numerous local variables to store temporary results, Objective-C lets you nest messages. The return value from each nested message is used as a parameter, or as the target, of another message. For example, you could replace any of the variables used in the previous example with messages to retrieve the values. Thus, if you had another object called `myAppObject` that had methods for accessing the array object and the object to insert into the array, you could write the preceding example to look something like the following:

```
[[myAppObject getArray] insertObject:[myAppObject getObjectToInsert] atIndex:0];
```

Although the preceding examples sent messages to an instance of a class, you can also send messages to the class itself. When messaging a class, the method you specify must be defined as a class method instead of an instance method. You can think of class methods as something akin to (but not exactly like) static members in a C++ class.

You typically use class methods as factory methods for creating new instances of the class or for accessing some piece of shared information associated with the class. The syntax for a class method declaration is identical to that of an instance method, with one exception. Instead of using a minus sign for the method type identifier, you use a plus (+) sign.

The following example demonstrates the use of a class method as a factory method for a class. In this case, the `arrayWithCapacity:` method is a class method on the `NSMutableArray` class that allocates and initializes a new instance of the class and returns it to your code.

```
NSMutableArray*   myArray = nil;    // nil is essentially the same as NULL

// Create a new array and assign it to the myArray variable.
myArray = [NSMutableArray arrayWithCapacity:0];
```

Properties

Properties are a convenience notation used to replace accessor method declarations. Properties do not create new instance variables in your class declaration. They are simply a shorthand for defining methods that access existing instance variables. Classes that expose instance variables can do so using the property notation instead of using getter and setter syntax. Classes can also use properties to expose “virtual” instance variables—that is, pieces of data that are computed dynamically and not actually stored in instance variables.

Practically speaking, properties reduce the amount of redundant code you have to write. Because most accessor methods are implemented in similar ways, properties eliminate the need to provide a distinct getter and setter method for each instance variable exposed in the class. Instead, you specify the behavior you want using the property declaration and then synthesize actual getter and setter methods based on that declaration at compile time.

You include property declarations with the method declarations in your class interface. The basic definition uses the `@property` compiler directive, followed by the type information and name of the property. You can also configure the property with custom options, which define how the accessor methods behave. The following example shows a few simple property declarations:

```
@property BOOL flag;
@property (copy) NSString* nameObject;  // Copy the object during assignment.
@property (readonly) UIView* rootView;  // Create only a getter method.
```

Another benefit of properties is that you can use dot syntax when accessing them in your code, as shown in the following example:

```
    myObject.flag = YES;
    CGRect   viewFrame = myObject.rootView.frame;
```

Although the object and property names in the preceding example are contrived, they demonstrate the flexibility of properties. The dot syntax actually masks the corresponding set of method calls. Each readable property is backed by a method with the same name as the property. Each writable property is backed by an additional method of the form `set`*PropertyName*`:`, where the first letter of the property name is capitalized. (These methods are the actual implementation of properties and are the reason you can include property declarations for attributes of your class that are not backed by instance variables.) To implement the preceding code using methods instead of properties, you would write the following code:

```
    [myObject setFlag:YES];
    CGRect   viewFrame = [[myObject rootView] frame];
```

For information on how to declare properties in your own classes, read "Properties" in The Objective-C 2.0 Programming Language.

Protocols and Delegates

A protocol declares methods that can be implemented by any class. Protocols are not classes themselves. They simply define an interface that other objects are responsible for implementing. When you implement the methods of a protocol in one of your classes, your class is said to conform to that protocol.

In iPhone OS, protocols are used frequently to implement delegate objects. A delegate object is an object that acts on behalf of, or in coordination with, another object. The best way to look at the interplay between protocols, delegates, and other objects is to look at an example.

The `UIApplication` class implements the required behavior of an application. Instead of forcing you to subclass `UIApplication` to receive simple notifications about the current state of the application, the `UIApplication` class delivers those notifications by calling specific methods of its assigned delegate object. An object that implements the methods of the `UIApplicationDelegate` protocol can receive those notifications and provide an appropriate response.

The declaration of a protocol looks similar to that of a class interface, with the exceptions that protocols do not have a parent class and they do not define instance variables. The following example shows a simple protocol declaration with one method:

```
@protocol MyProtocol
- (void)myProtocolMethod;
@end
```

In the case of many delegate protocols, adopting a protocol is simply a matter of implementing the methods defined by that protocol. There are some protocols that require you to state explicitly that you support the protocol, and protocols can specify both required and optional methods. As you get further into your development, however, you should spend a little more time learning about protocols and how they are used by reading "Protocols" in The Objective-C 2.0 Programming Language.

For More Information

The preceding information was intended primarily to familiarize you with the basics of the Objective-C language. The subjects covered here reflect the language features you are most likely to encounter as you read through the rest of the documentation. These are not the only features of the language though, and you are encouraged to read more about the language in The Objective-C 2.0 Programming Language.