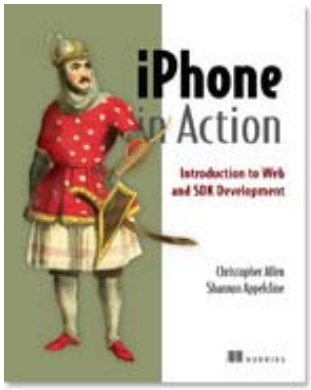
MANNING PUBLICATIONS

SDK Programming for Web Developers

Excerpted from



iPhone in Action

EARLY ACCESS EDITION

Introduction to Web and SDK Development

Christopher Allen and Shannon Appelcline

MEAP Release: May 2008
Softbound print: December 2008 (est.) | 350 pages
ISBN: 193398886X

This excerpt is taken from forthcoming book *iPhone in Action* by Christopher Allen and Shannon Appelcline (Manning Publications). It teaches web developers what they need to know about the iPhone SDK and the Object C language by leveraging their knowledge of web programming languages, such as PHP.

This chapter covers:

- Understanding the unique features of a more complex programming language like C
- Describing the specific paradigm of an object-oriented programming language
- Learning about the MVC Architectural Pattern

We've spent the last five chapters talking about how to program great web pages and applications using familiar tools such as HTML, JavaScript, and PHP. However, as we discussed in chapter 2, web development isn't the end-all and be-all of iPhone programming. There are some programs that will just be better suited for native programming on the iPhone. Apple provides a set of tools for doing this sort of development called the SDK (or software development kit). It includes a set of programs, frameworks, and toolsets which we'll meet fully in the next chapter. It also depends upon a particular programming language: Objective-C.

If you've never worked with Objective-C, don't panic. This chapter is intended to bootstrap you up from your experiences with web development (which we assume includes some sort of web-based programming language, such as PHP, Ruby on Rails, or Perl) so that you'll be prepared to work with Objective-C when you encounter it in the next chapter.

We'll do this by touching upon three major topics. First we'll talk about C, which is a more complex and rigorous programming language than many of the somewhat freeform web-based languages. It's also the core foundation of Objective-C. Then we'll talk about object-oriented programming, which is the style of programming used by Objective-C. Finally we'll hit upon MVC, an architectural model used by many different programming languages, including Objective-C. If you're already familiar with some of these concepts, just skip over the relevant sections.

However, before we start this whirlwind tour, we'll offer one caveat: none of these short overviews can possibly do justice to the topics. There are complete books on each of these topics, and if you feel like you need more information, you should pick one up. Though we ultimately can't be complete, it is our intention to provide you with enough of a gloss that you can not only understand the code of part 3 of this book, but also dive right in yourself by tweaking and ultimately building upon the copious examples that we provide.

8.1 An Introduction to C

The syntax of C will look a lot like whatever language you're familiar with. However, it may vary from your web language of choice in how it deals with some big picture areas, namely: declarations, memory management, file structure, and compilation.

We've summarized all these ideas in table 8.1, but we're going to talk about each of them in turn at more length. We'll hit some minor syntax along the way, but for the most part, we'll save technical details of that sort until we talk about Objective-C in chapter 9.

Table 8.1 The rigorous style of C requires you to think about a few new programming topics.

C Concept	Summary
Declaration & Typing	You must declare variable types You must declare function argument types and return type You may need to repeat these declarations in a header file
Memory Management	You may sometimes need to explicitly manage the memory usage of your variables
Pointers	Some variables are represented as pointers to spaces in memory
File Structure	Programs are divided between source (.c) and header (.h) files
Directives	Precompiler commands are marked with the # sign. This includes the #include directive which incorporates header files into source files

Compiling	Your code is turned into a machine-readable format when you compile it, not at run-time

If you want more information on C, the definitive reference is *The C Programming Language, Second Edition*, by Brian W. Kernighan and Dennis M. Ritchie.

8.1.1 Declarations & Typing

C is generally a more rigorous programming language than some of the very casual languages found on the web. That means there's a little bit more time spent saying what you're going to do before you do it.

The purpose of this is not only to make it easier for a computer to understand and efficiently run your program (which was more important in 1972, when C was first invented, than it is today), but also to make it easier to catch errors (which is still important today, alas). If you tell the computer what you're going to do, then it can give you a warning if that isn't quite what happens.

First, we see this rigor in the *typing* of variables. Before you're allowed to use a variable in C, you must say how it's going to be used. For example this says that the variable `n` will be used as an integer:

```
int n;
```

Second, we see it in functions, where you must declare not only what types of variables you'll be passing a function, but also what type of variable you're going to return. This is all done as part of the line of code that kicks off the function. For example the following function takes two floating point numbers and returns a floating point number:

```
float divide(float numerator, float divisor) {
```

These variable and function *declarations* often get done a second time as part of a header file, which is a topic that we're going to return to momentarily.

8.1.2 Memory Management & Pointers

You didn't have to worry at all about how memory was used to store your active data in most web-based languages. Conversely, in C if you're dynamically changing your data during your program's run-time, you often do. This is usually done through a function called `malloc()`, a cousin to the `alloc` message we'll meet in the SDK.

When you specifically allocate memory, you also have to deallocate it when you're done. If you don't, your program will "leak," which means that it will gradually increase its memory footprint over time due to memory that's been "lost." As we'll see, Objective-C has a very specific method for managing memory recovery.

When you allocate memory you end up working with memory addresses that lead to your data, rather than the data itself. This is also by default the case with some sorts of data, such as strings. To address this, C introduces the concept of a *pointer*, wherein a variable refers to a memory address which itself points to your data. When this is the case you can *dereference* the memory address, and thus access your data, with the * character.

For example the following would define a pointer to an integer:

```
int *bignumber;
```

The good news is that in Objective-C you won't really have to worry about pointers, because those details will (once more) be hidden. However, when you initially declare variables, you'll almost always do so with a `*`; you'll constantly be creating pointers to objects.

8.1.3 File Structure & Directives

When you look at the file structure of a complex program, you'll see that it includes a variety of files with .c and .h suffixes. The .c files include all the source code: the various functions that you're used to using in a program, split up in a (hopefully) rational way.

The .h (or header) files, meanwhile, are the tools that allow you to easily integrate the source code from one .c file into the rest of your program. They contain all the variable declarations that you want to make available outside of specific functions. In addition, they contain *function prototypes*. These are declarations for your functions that effectively describe a protocol for using them:

```
float divide(float numerator, float divisor);
```

Just as header declarations make a variable available outside of its own function, function prototypes make a function available outside of its own file.

In order to use a header file, you need the capability to include one file inside another. For example, if you want to access some of the global variables or some of the functions of file2.c in file1.c, you do so by incorporating file2.c's header file. You do this by inserting an include command into file1.c:

```
#include "file2.h"
```

The appropriate file is then inserted as part of the C preprocessor's work. This is what's known as a *compiler directive*, or just a *macro*.

As we'll see, Objective-C replaces .c files with .m or .mm files and replaces the `#include` directive with `#import`, but the actual ideas are the same.

8.1.4 Compiling

The final difference between C and most web-based programming languages is that you must *compile* it. This means that the human-readable source code is turned into machine-readable object code. The same thing actually happens to your web programs, but whereas they compile at run-time, C instead compiles in advance, providing for more efficient program startup. C compilation can be done by a command line program (like "cc" or "gcc") or by some fancy integrated development environment (like Xcode, which we'll meet shortly).

Because C programs tend to include many files, they need special instructions to tell them how to put all the code together. This is most frequently done with a *makefile*, though integrated environments might have their own ways to list what should be used, possibly shielding the user entirely from worrying about this.

And that's C in a nutshell—or at least a look at its most notable features that may differ from your web programming experience.

8.2 An Introduction to Object Oriented Programming

C is fundamentally a procedural language, as are early web-based languages like Perl and PHP (though that's changing for both through current releases). You make calls to functions that do the work of your program. Object-oriented programming (OOP) moves away from this old paradigm. Specifically, there are no longer separate functions and variables; instead data and commands are bound into a more cohesive whole.

As with our previous section, we've created a summary chart of the major elements of object-oriented programming, which you can find in table 8.2.

Table 8.2 Object-oriented programming introduces a number of new concepts.

OOP Concept	Summary
Class	A collection of variables and functions that work together
Framework	A collection of class libraries
Inheritance	The way in which a subclass gets variables and functions from its parent
Message	A call sent to an object, telling it to execute a function
Object	A specific instance of a class
Subclass	A descendent of a class, with some features in common and some variance

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides is an influential book which generally talks about OOP, then specifically covers some design patterns usable in it. There are many other books on the topic.

8.2.1 Objects and Classes

The central concept in object-oriented programming is (as you might guess) the *object*. Think of it as a super-variable, or (if you prefer) as a concrete, real-world thing—which is what's actually being modeled in the OOP paradigm

An object combines values (which Objective-C calls *instance variables* and *properties*) and functions (which Objective-C calls *methods*). These variables and methods are intrinsically tied together. The variables describe the object while the methods give ways to act upon it (such as creating, modifying, or destroying the object).

Objects are specific *instances* of *classes*. The class is where the variable and method descriptions actually appear. An individual object then takes all of that information, and starts setting its own version of the variables as it sees fit.

Classes gain power because they support *inheritance*. That means that you can subclass an existing class. Your subclass starts off with all the default variables and methods for its class, but you can now supplement or even *override* them.

It's frequent for a subclass to override a parent method by first calling the parent method, then doing a few things to vary how it works. For example if you had a class that

represented an eating utensil, it might include a simple method to transfer solid food from your plate to your mouth. If you created a subclass for a spoon, it could include a new method that worked on liquids.

Once you've created a hierarchy of classes and subclasses, you can store them away in a *class library*. When you put several of those together you have a *software framework*, and that's what we'll be working with in the SDK, as Apple has provided numerous frameworks to make your programming of the iPhone easier.

8.2.2 Messaging

If the object is the OOP equivalent of the variable, then the *message* is the OOP equivalent of the function. In order to get something done in an object-oriented program, you send a message to a specific object that asks it to execute a specific method. The object then does so internally, using its own variables and reporting its results out to the calling object.

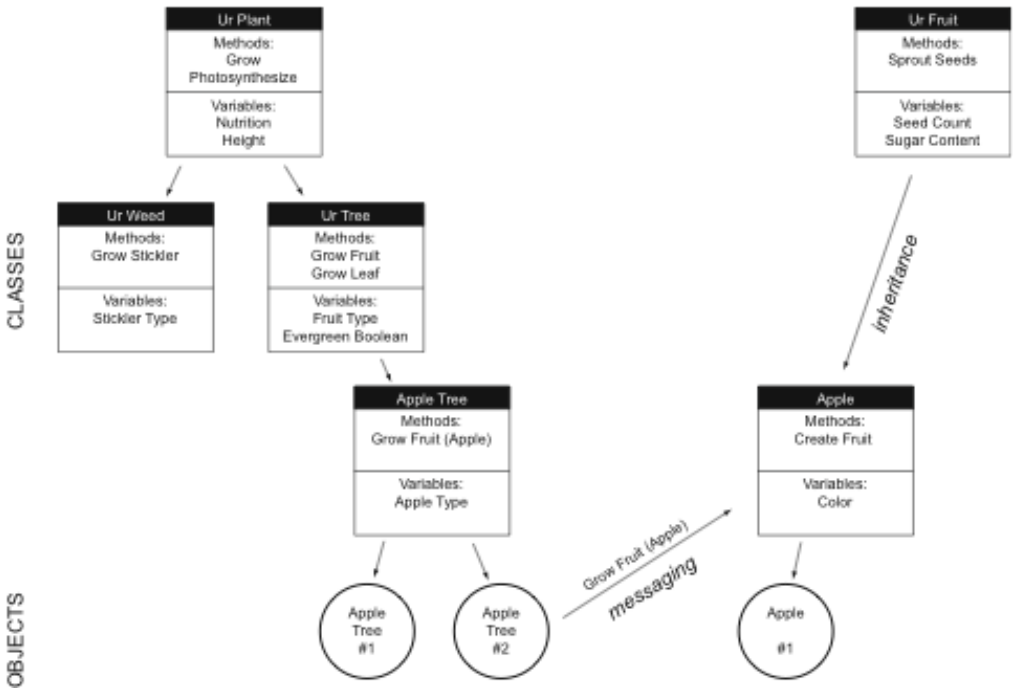
One of the most frequent types of messages that you'll see in object-oriented programming is a call to a method that looks at or changes an object's variables. A *getter* is an *accessor* that looks at data, while a *setter* is a *mutator* that changes data (though Apple actually calls both accessors in some of its documentation).

It's important to use accessors and mutators because they support the core OOP ideal of *encapsulation*. The actual variables in objects are hidden away from the rest of the world, freeing up that global name space, which otherwise could become quite cluttered. If one object uses a `foo` variable that no longer impacts the use of a `foo` variable by someone else. Instead, each variable can only be accessed by the methods of its own class.

Some OOP languages support two different types of messages. You might see calls to *class methods* (where a special class object does general stuff like create an object) or to *instance methods* (where an instance object acts in some way). As we'll see, this is the case with Objective-C.

Figure 8.1 combines many of the ideas that we've talked about so far into a single diagram using a vegetative example.

Figure 8.1 Inheritance and messaging combine to form an intricate network of objects in Object Oriented Programming.



When we look at the classes that are built into the iPhone OS, we'll see that there are even more levels of inheritance and overall a much larger web of classes and messages.

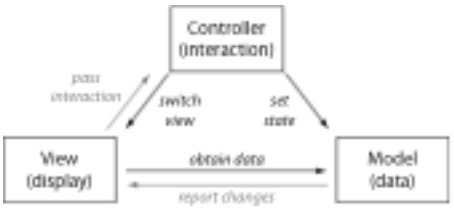
8.3 The Model-View-Controller (MVC) Pattern

Programming languages innately have their own philosophies and models that underlie how they work. Encapsulation and inheritance are two of the philosophies that are critical to OOP. A philosophy that's critical to good Objective-C programming is the Model-View-Controller (MVC) architectural pattern.

This method of software design goes back to 1979, when Trygve Reenskaug—then working on Smalltalk at Xerox PARC—described it. It's widely available today in OOP frameworks (including the iPhone's Cocoa Touch) and as libraries for other programming languages.

The MVC pattern breaks a program into three parts. The *model* is the data at the heart of a program. Meanwhile, the view and the controller together comprise the presentation layer of your application. The *view* is essentially the user interface side of things, while the *controller* sits between the view and the model, accepting user input and modifying the other elements appropriately. Figure 8.2 displays what this model looks like.

Figure 8.2 The MVC model covers how user input and other changes affect your program's design.



From the diagram you can see this core ideal of the controller accepting input and making changes, but note that there will also be direct interaction between the model and the view.

Dynamic web design offers a simple example of an MVC pattern. The model represents your data, usually stored in a database or XML. The view is your HTML page itself. Finally, the controller is your JavaScript, PHP, Perl, or Ruby on Rails code, which accepts the input, kicking out HTML code on one side and modifying the database on the other.

Within Objective-C, you'll see an even more explicit interpretation of MVC. There are objects specifically called views and view controllers. If you're following good Objective-C programming practice, you'll make sure your controllers are accepting input from your view and doing the work themselves.

8.4 Summary

We think it's entirely possible for someone without object-oriented experience (and even without C experience) to make the transition from creating iPhone-based web pages to creating iPhone-based native apps. We also think there are very good reasons for doing so. As we said back in chapter 2, the SDK and web development can each do different things well, and it's always best to use the right tool for the job at hand.

However, we won't promise that it'll be easy. The whole idea of objects replacing procedural calls is a pretty big switch in how you do things when programming. When you meet actual Objective-C code, you'll also see that even with the simplest program you're going to have several different files to work with that each serve very different purposes.

Fortunately you're going to have three terrific advantages on your side. First, you're going to have access to the SDK's programming tools, Xcode and Interface Builder. The first will constantly offer you whatever documentation you need for the objects, methods, and properties you're using, while the second will provide you with a simple, graphical way to create objects. Second, you're going to have Objective-C itself on your side. Though its code looks a bit different from most programming languages you're liable to have worked with, its code is simple, elegant, and overall quite easy to read. Third, you're going to be able to use the iPhone OS's enormous library of frameworks, making your code even shorter and simpler thanks to many years of development on Apple's part.

You're going to be pretty amazed at what you can do in just a few lines of code.