



Apple Developer Connection
Recommended Title

2nd Edition
Covers Mac OS X 10.2

Learning

Cocoa *with* Objective-C



O'REILLY®

James Duncan Davidson & Apple Computer, Inc.

SECOND EDITION

Learning Cocoa with Objective-C

James Duncan Davidson and Apple Computer, Inc.

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Object-Oriented Programming with Objective-C

Object-oriented programming isn't rocket science, but you can't learn it overnight either. There is a lot of terminology—composed of words like “encapsulation” and “polymorphism” and phrases like “is-a” and “has-a”—that goes with the territory. The concepts behind these terms are not terribly complicated, but they can be hard to explain. Like most useful fields of study, you must work with it a while before it all clicks together in your mind. As each concept becomes clear, you will gain a deeper understanding of the subject. That said, you don't have to understand everything about object-oriented programming on the first pass to make good use of the concepts.

In this chapter, we present the object-oriented concepts that matter most when working with Cocoa, along with quite a bit of hands-on practice using those concepts. If this is the first time you've approached object-oriented programming, read carefully, but don't worry if you don't get everything at first. Just remember to flip back to this part of the book later if something didn't sink in. If you already know a bit about object-oriented programming, then you should treat this as a refresher and see how Objective-C's implementation of the object-oriented concepts with which you are familiar works.

Introducing Objects

Procedural programming divides the programming problem into two parts: data and operations on that data. Because all of the functionality of a procedural program works on the same set of data, the programmer must be very careful to manipulate the data of a program in such a way that the rest of the program can work correctly. He must be aware of the entire program at a low level of abstraction so as not to introduce errors. As a procedural program grows in size, the network of interaction between procedures and data becomes increasingly complex and hard to manage.

Object-oriented programming (OOP), first developed in the 1960s,* restructures the programming problem to allow for a higher level of abstraction. It groups operations and data into modular units called *objects*. These objects can be combined into structured networks to form a complete program, similar to how the pieces in a puzzle fit together to create a picture. In contrast to procedural programming's focus on the interaction between data and functions, the design of objects and the interactions between those objects become the primary elements of object-oriented program design.

By breaking down complex software projects into small, self-contained, and modular units, object orientation ensures that changes to one part of a software project will not adversely affect other portions of the software. Object orientation also aids software reuse. Once functionality is created in one program, it can easily be reused in other programs.

Programming with objects is quite like working with real-world objects. Take an iPod, for example. It embodies both state and behavior. When you operate it, you don't necessarily care how it works, as long as it works in the way that you expect. As long as your iPod plays music when you tell it to and synchronizes your music collection with iTunes when you plug it into your computer, you're happy. Object-oriented programming brings this same level of abstraction to programming and helps remove some of the impediments to building larger systems. To enjoy listening to music, you don't have to know that iTunes and your iPod use the MP3 format; you just put a CD into your computer and import the music into your collection. iTunes and your iPod work together to download the music from your computer when you plug in the iPod. Figure 3-1 shows these components working together.

Classes of Objects

In the real world, there are often many objects of the same kind, or *type*. My iPod is just one of many iPods that exist in the world. In the lingo of object-oriented programming, each iPod is an *instance*. An instance of an object has its own state and leads an existence independent of all other instances. My iPod probably has a very different collection of music than yours does.† But just as all iPods have the same set of buttons—allowing the same set of operations (play, stop, etc.)—all instances of a particular object expose the same functionality to the outside world.

You specify an object by defining its *class*. Think of a class as a blueprint for making object instances. It provides all the information needed to build new instances of an object. Each class defines the internal variables that hold the data of an object instance and the ways, or *methods*, by which that data can be manipulated. These

* SIMULA I and SIMULA 67 were the first two object-oriented programming languages. They were designed and built by Ole-Johan Dahl and Kristen Nygaard in Norway between 1962 and 1967.

† There's even a decent chance that you might not like the music on my iPod, and vice versa.

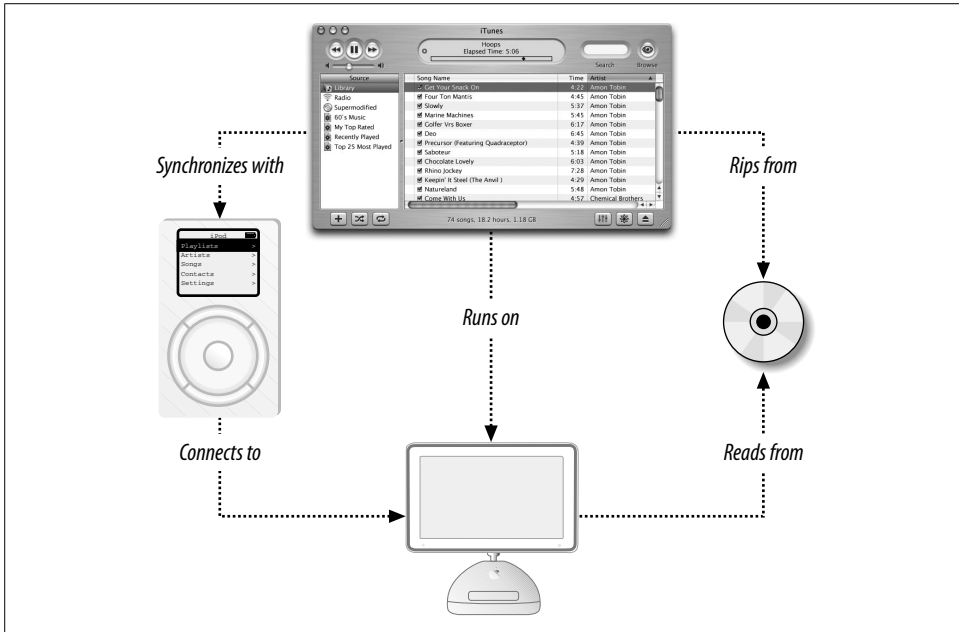


Figure 3-1. Real-world objects interacting together

methods define the *interface* of the object. The interface is how other objects are allowed to use it.

On the back of every iPod is the phrase “Designed by Apple in Cupertino. Assembled in Taiwan.” This is a useful analogy for thinking about how classes and objects relate to each other. In its corporate offices in California, Apple defined how an iPod operates and what kinds of data it can store. Apple shipped those definitions to the factory in Taiwan that now creates many unique instances of an iPod to ship to customers around the world. When you create a class, you create a definition from which the *runtime* (the layer of software that enables the object-oriented system to run) can create any number of objects (see Figure 3-2).

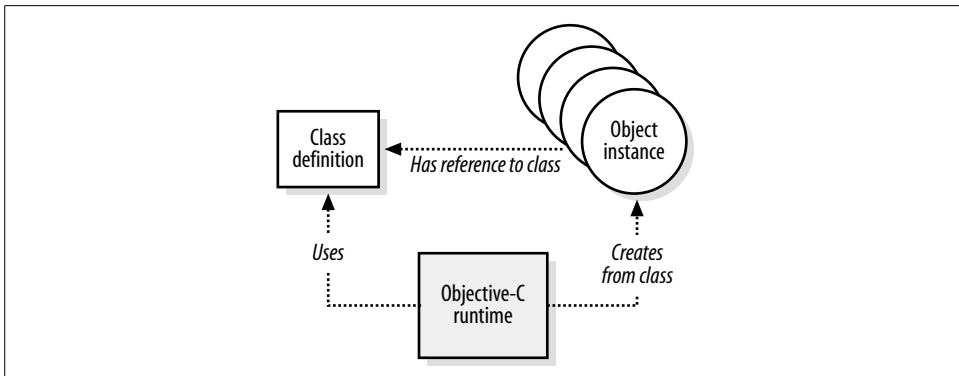


Figure 3-2. Runtime creates object instances from a class

In Objective-C, classes are more than just blueprints. They are actually first-class objects themselves that can have methods associated with the class and not with its instances. These are called *class methods*. Every object created has a reference to its own class. The iPod analogy starts to get a bit stretched here, but imagine that each iPod had a reference to the plans on which it was based and could consult them at any time. This is sort of what it means for an object to look up its class object any-time it needs to do so.

Inheritance

We've defined a class to be a definition, or blueprint, from which object-oriented instances are created. An iPod is an instance of the iPod class. But classes themselves can be defined as specializations of other classes. For example, if you didn't know what an iPod was, you would probably understand if I told you that it was a handheld MP3 player. In fact, all handheld MP3 players share a certain number of characteristics. Like an iPod, a Rio can hold and play MP3 files downloaded from a computer. It can't hold as many songs as the iPod, but at least some of the functionality is the same.



The iPod is actually much more than a portable MP3 player. It's also a bootable FireWire drive that can hold any kind of data that you want it to hold. People are finding some pretty creative uses for it beyond playing music. In Objective-C, objects that can perform other functions can declare that they obey a particular *protocol*, or way of behaving. We'll talk more about protocols and how they can be used effectively in Chapter 9.

Object-oriented programming lets us collect similar functionalities of different classes and group them into a common parent class through *inheritance*. We can say that an iPod and a Rio are both types of MP3 players. If we define a common MP3Player class, we can gather certain aspects common to both devices into one class, as shown in Figure 3-3.

The iPod and Rio classes are both *subclasses* of the MP3Player class. Likewise, the MP3Player class is the *superclass* of the iPod and Rio classes. Each subclass inherits state (in the form of variable definitions) and functionality from the superclass. In this case, both players inherit the same basic functions (play, stop, fast forward, etc.), but have very different underlying implementations. The iPod uses a high-capacity hard drive while the Rio uses flash memory.

Creating a new class is often a matter of specialization. Since the new class inherits all of its superclass's behavior, you don't need to reimplement the things that work in the way that you want. The subclass merely extends the inherited behavior by adding new methods and any variables needed to support the additional methods. A subclass can alter superclass behavior by overriding an inherited method, reimplementing

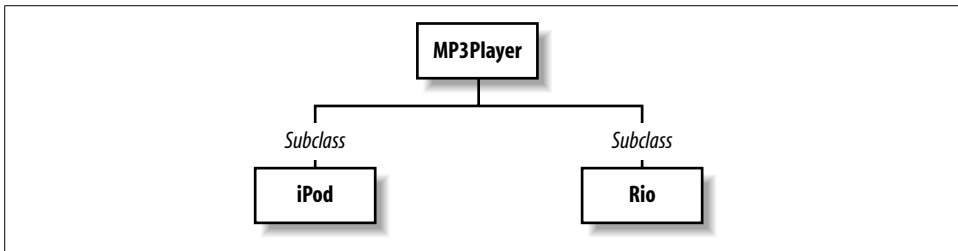


Figure 3-3. Class hierarchy for the MP3Player class

menting the method to achieve a behavior different from the superclass’s implementation.

With Objective-C, a class can have any number of subclasses, but only one superclass.* This means that classes are arranged in a branching hierarchy with one class at the top—the root class that has no superclass—as shown in Figure 3-4.

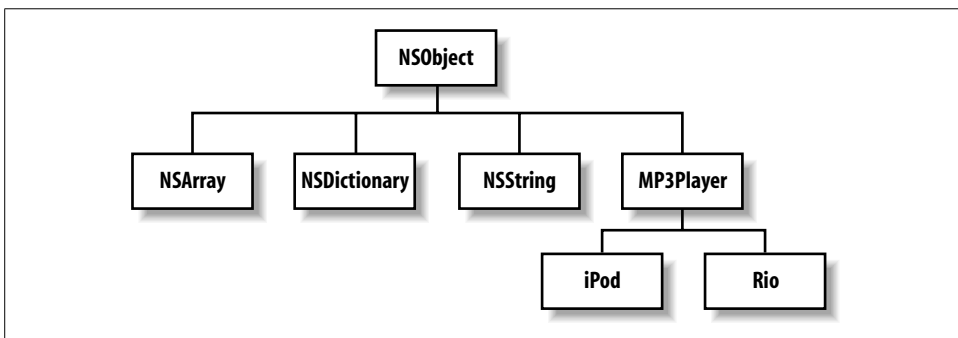


Figure 3-4. The root class in Objective-C

NSObject is the root class of this hierarchy. From NSObject, other classes inherit the basic functionality that lets them work in the system. The root class also creates a framework for the creation, initialization, deallocation, introspection, and storage of objects.

As noted earlier, you often create a subclass of another class because that superclass provides most, but not all, of the behavior that you require. A subclass can have its own unique purpose that does not build on the role of an existing class. To define a new class that doesn’t need to inherit any special behavior other than the default behavior of objects, you make it a subclass of NSObject.

* Some object-oriented programming languages, such as C++, allow classes to inherit functionality from more than one superclass. This ability, known as multiple-inheritance, can often lead to more problems than it solves. Objective-C provides protocols (discussed in Chapter 9) to provide some of the benefits of sharing behavior (but not implementation) across the class hierarchy.



Inheritance is a powerful concept—one that many people new to object-oriented programming tend to use too much. Used inappropriately, it can lead to fragile software. In Cocoa, it’s often easier to use a new set of classes from a new class than to use inheritance. This is called *object composition*. As you work through this book, you’ll see many examples of object composition.

Creating and Using Objects

Now that we’ve introduced a few object-oriented concepts, we are going to dive into some simple code exercises to show how to apply this knowledge. The following steps will guide you:

1. In Project Builder, create a new Foundation tool (File → New Project → Tool → Foundation Tool) project named “objects”, and save it in your *~/LearningCocoa* folder.
2. Next, modify the *main.m* file, located in the “Source” group, so that it looks like Example 3-1. The Foundation tool project template automatically generates some of this code. The lines that you need to add are shown in **boldface** type.

Example 3-1. Creating objects

```
int main (int argc, const char * argv[]) {
    NSAutoreleasepool *pool = [[NSAutoreleasepool alloc] init];

    NSObject * object; // a
    object = [NSObject alloc]; // b
    object = [object init]; // c
    NSLog(@"Created object: %@", object); // d

    [pool release];
    return 0;
}
```

Here’s what the code that we added does:

- a. Declares a variable named `object` of type `NSObject`. You should recognize this as a regular C pointer.
- b. Creates a new object of type `NSObject` and assigns it to the `object` variable. The `alloc` method reserves (or allocates) memory space for the object and returns a pointer to that space. We’ll explain more about methods in just a bit.
- c. Before an object is used in any way, it must be initialized. This `init` call initializes the object so it can be used. The `init` method returns a fully initialized object ready for use. Since it is possible that the `init` method will return a different object, we assign the return to the `object` variable again.

- d. Prints a representation of the object to the console using a `printf` style format string with a `%@` token, indicating that the `svalue` of the object given after the format string should be printed.

There's actually a bit more going on in this code than what we've described. However, we'll fill in the missing pieces as we go to avoid introducing too many concepts at once.

Format String Tokens

There are several methods in Cocoa, such as `NSLog` and `[NSString stringWithFormat]`, that can use format strings with a list of arguments. These format strings can contain all of the normal `printf`-style tokens, as well as a Cocoa-specific token for objects. You'll find the following tokens to be useful:

`%@`

Print as an object

`%d` or `%i`

Print as a signed decimal

`%o`

Print as an unsigned octal

`%s`

Print as a string

`%u`

Print as an unsigned decimal

`%x`

Print as an unsigned hexadecimal

In addition to the tokens in this short list, you can add all sorts of modifiers to control precisely how values are printed. See the `printf` manpage for complete information about format strings. You can access the manpage in either of the following ways:

- Open a Terminal window and enter `man printf` at the prompt.
- In Project Builder, use the Help → Open man page... menu item.

3. Build and run the program. You should see something like this on the console:

```
2002-06-11 23:17:16.181 objects[477] Created object: <NSObject: 0x5ae90>
```

This tells us that we created an object of type `NSObject` that is located at the memory address `0x5ae90`. This isn't the most exciting information that could be printed, and it certainly won't win any user-interface awards, but it shows us that objects are being created in the system by the runtime.



As a Cocoa programmer, you probably won't ever make direct use of the memory location of the object instances you create. But under the hood, Cocoa uses this information to locate and manipulate objects that you reference in code.

Since objects should never be used without proper allocation and initialization, Objective-C programmers tend to combine the methods into one line as shown in Example 3-2. Replace lines a, b, and c from Example 3-1 with the single bolded line in Example 3-2.

Example 3-2. Combing object allocation and initialization

```
int main (int argc, const char * argv[]) {
    NSAutoreleasepool *pool = [[NSAutoreleasepool alloc] init];

    NSObject * object = [[NSObject alloc] init];
    NSLog(@"Created object: %@", object);

    [pool release];
    return 0;
}
```

This shortens the allocation and initialization of an object to one line, ensuring that everything works properly, even in the case where the `init` method of a class returns a different object than originally allocated. We will use this style of object creation throughout the rest of the book.

Working with Multiple Objects

Working with multiple object instances of the same class is easy, as long as you keep the references to different objects distinct.

1. Edit the code in the project's *main.m* file as shown in Example 3-3.

Example 3-3. Working with multiple objects

```
int main (int argc, const char * argv[]) {
    NSAutoreleasepool *pool = [[NSAutoreleasepool alloc] init];

    NSObject * object1 = [[NSObject alloc] init];
    NSObject * object2 = [[NSObject alloc] init];
    NSLog(@"object1: %@", object1);
    NSLog(@"object2: %@", object2);

    [pool release];
    return 0;
}
```

2. When built and run, the program will print something similar to the following:

```
2002-06-11 15:59:29.716 objects[370] object1: <NSObject: 0x4ce90>  
2002-06-11 15:59:29.717 objects[370] object2: <NSObject: 0x4b410>
```

This example shows that two object instances of NSObject have been allocated, and they occupy two different locations in memory.

Methods and Messages

In our discussion about objects so far, we've been using (and promised to explain) the term *method*. Methods are structured like C functions and can be thought of as procedures; but, instead of being global in nature, they are procedures associated with and implemented by the object's class.

There are two kinds of methods: *class methods* and *instance methods*. Class methods are scoped to the class itself and cannot be called on instances of the class. The alloc method is an example of a class method. Instance methods, on the other hand, are scoped to object instances. The init method is an example of an instance method that is called on an instance of an object returned by the alloc method.

To call a method, send an object a *message* telling it to apply a method. All those square brackets that you have seen in the code are message expressions that result in methods being called. Figure 3-5 shows the various parts of a basic message.

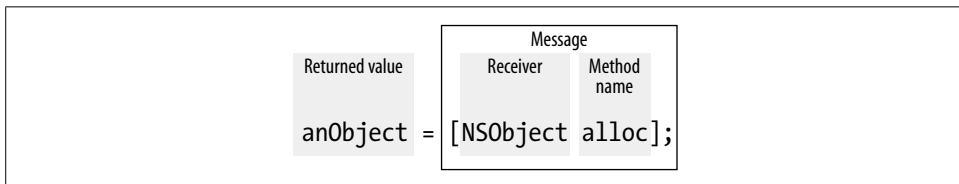


Figure 3-5. Objective-C message expression

In this figure, the message is the expression enclosed in square brackets to the right of the assignment operator (equals sign). The message consists of an object, known as a *receiver*, and the name of a method to call on that object. In this case, the object is the NSObject class, and the method to be called is the alloc method. In response to receiving this message, the NSObject class returns a new instance of the class that will be assigned to the variable anObject.

Arguments in Messages

The message in Figure 3-5 calls a method that doesn't take any arguments. Like procedures, methods can receive multiple arguments. In Objective-C, every message

argument is identified with a label (a colon-terminated keyword), which is considered part of the method name. Figure 3-6 shows a message with a single argument.

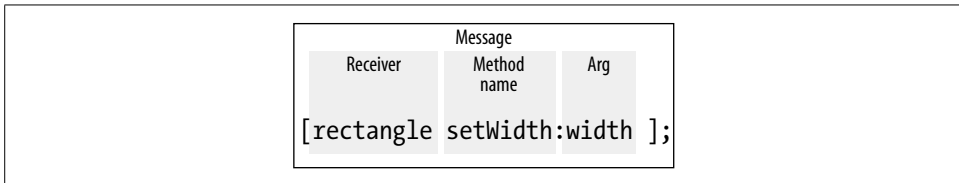


Figure 3-6. Objective-C message expression with a single argument

In this figure, the message tells the runtime to call the `setWidth:` method and pass it the argument `width`. Notice that a colon terminates method names that take an argument, while method names that don't take an argument (like the `alloc` method in Figure 3-5) don't have a colon.

Figure 3-7 shows a multiple-argument message. Here, the message and arguments are used to set the width and height of the rectangle object to `width` and `height`, respectively. This method is called the `setWidth:height:` method.

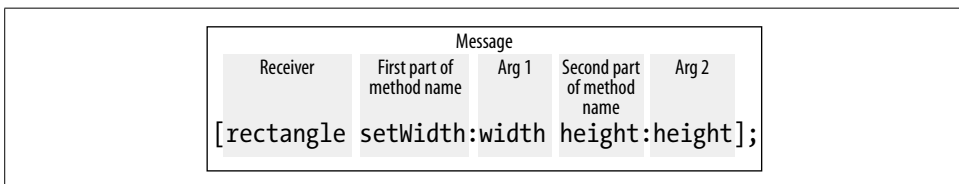
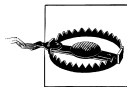


Figure 3-7. Objective-C message expression with multiple arguments



Note that `setWidth:height:` refers to one method, not two. It will call a method with two arguments. When using this method, you must pass in two arguments, labeled and formatted as in Figure 3-6. If you haven't used Smalltalk or one of its derivatives, you will find this practice strange at first, but you'll soon appreciate the readability it imparts to code.

Nested Messages

Figure 3-8 shows nested messages. By enclosing one message within another, you can use a returned value as an argument without having to declare a variable for it. The innermost message expression is evaluated first, resulting in a return object. Then the next nested message expression is evaluated using the object that was returned in the inner expression as the receiver of the second message. We saw this in action in Example 3-2 when we combined the `alloc` and `init` methods of `NSObject`.

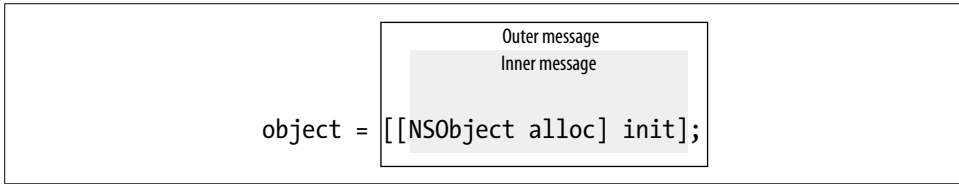
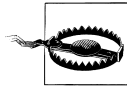


Figure 3-8. Nested Objective-C messages



Nested messages work only when the inner expression returns an object value. If the inner expression returns something else (for example, an `int`) then a crash will result at runtime. This is because messages can be passed only to objects, not to primitive types.

How Messaging Works

The `NSObject` class ensures that every object in the system has an instance variable named `isa`. This variable points to the class that defines how the object works. In addition, every class object has a reference to its superclass. This relationship is illustrated in Figure 3-9.

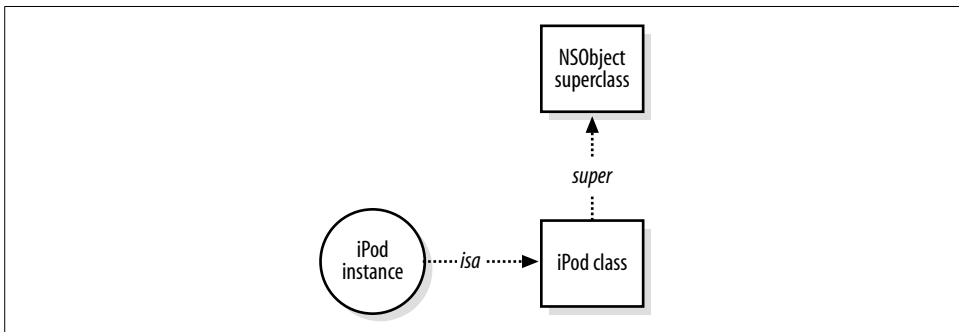


Figure 3-9. Instances have a reference to their class.

The class object contains quite a bit of information about the internals of the class and how it works. Part of this information is a method lookup table that maps *selectors* to methods, as shown in Figure 3-10.

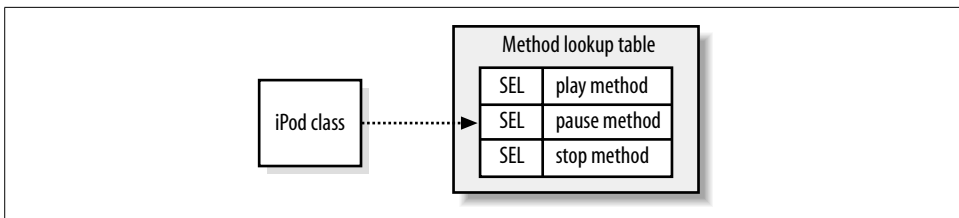


Figure 3-10. Method lookup table

A selector (defined as the SEL type in Objective-C) is a compiler-assigned code that identifies a method to the runtime. When you send a message to an object, the compiler actually creates code to perform a call to an Objective-C-defined function, which uses the selector information to perform a dynamic method lookup at runtime. For more details about how this functionality works underneath the hood, read *Inside Mac OS X: The Objective-C Language*, located in the `/Developer/Documentation/Cocoa/ObjectiveC` folder.

Objective-C–Defined Types

So far, we’ve talked about a few of Objective-C’s built-in types, such as SEL. Before we continue, Table 3-1 lists the set of Objective-C–defined types.

Table 3-1. Objective-C–defined types

Type	Definition
id	An object reference (a pointer to its data structure)
Class	A class object reference (a pointer to its data structure)
SEL	A selector (a compiler-assigned code that identifies a method name)
IMP	A pointer to a method implementation that returns an id
BOOL	A Boolean value, either YES or NO
nil	A null object pointer, (id)0
Nil	A null class pointer, (Class)0

The `id` type can be used to type any kind of object, class, or instance. In addition, class names can be used as type names to type instances of a class statically. A statically typed instance is declared as a pointer to an instance of its class or to an instance of any class from which it inherits.

Creating New Classes

When you want to create a new kind of object, you define a new class. A class is defined in two files. One file, the *header file* (`.h`), declares the variables and methods that can be invoked by messages sent to objects belonging to the class. The other file is the *implementation file* (`.m`), which actually implements the methods declared by the header file, as well as the private implementation details of the class. The interface defined in the header file is public. The implementation is private and can be changed without affecting the interface or the way the class is used.

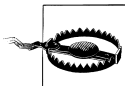
To show how to create a new class, we will model songs that would go into an MP3 player. Don’t get too scared yet; we’re not actually going to write the MP3 player itself.

1. To get started, create a new Foundation Tool in Project Builder (File → New Project → Tools → Foundation Tool) named “songs”, and save it in your `~/LearningCocoa` folder.
2. Define a header for our song class. Choose File → New File, then select Objective-C class as the file type, as shown in Figure 3-11.



Figure 3-11. New File Assistant

3. Name the file `Song.m`, as shown in Figure 3-12. Make sure that the “Also create “`Song.h`” checkbox is clicked. This creates the header file for the application’s interface.



Be careful not to confuse this use of the word *interface* with the term *Graphical User Interface*. This use of the word refers to how components talk, or know, about each other and doesn’t refer to how users will interact with the program.

When you finish, Project Builder should look something like Figure 3-13. If `Song.h` and `Song.m` are not in the Source category of files, you can simply drag them there. (Hint: Use the black insertion indicator that appears in the outline view to guide you as you drag.) Where they appear doesn’t matter to Project Builder, but keeping things neat and tidy will help you, especially on larger projects.



Figure 3-12. New Objective-C class Assistant

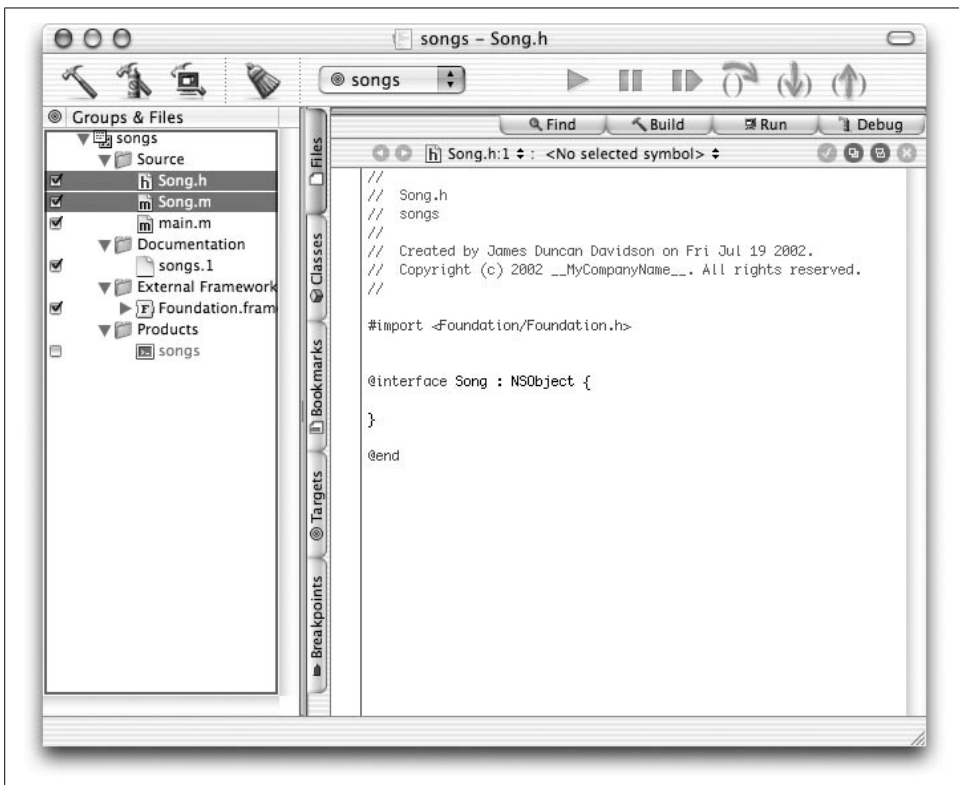


Figure 3-13. Newly created class in Project Builder

By creating the class header and implementation files, we have a start on a class that can be used in the rest of the program, including our main function in the *main.m* file. Project Builder creates a basic *Song.h* header file for you. A new class is declared in the header file with the `@interface` compiler directive. In this case, the directive is the following:

```
@interface Song : NSObject
```

This indicates that we are defining a class called *Song* that inherits from *NSObject*. The colon indicates the inheritance. The rest of the file is left for us to complete. All the instance variables used by the class are declared between the brackets. All the methods of the class are declared between the end bracket and the `@end` directive.

Defining the Song Interface

Edit the *Song.h* file as shown in Example 3-4. Once again, the lines that you need to add are shown in **boldface** type.

Example 3-4. Song.h interface

```
@interface Song : NSObject {  
    NSString * name; // a  
    NSString * artist; // b  
}  
- (NSString *)name; // c  
- (void)setName:(NSString *)newName; // d  
- (NSString *)artist; // e  
- (void)setArtist:(NSString *)newArtist; // f  
@end
```

Here's what the additional code does:

- a. Declares the `name` variable that will point to an object of type `NSString`.
- b. Declares the `artist` variable that will point to an object of type `NSString`.
- c. Declares an instance method, named `name`, that returns a pointer to an `NSString` object when called.
- d. Declares an instance method, named `setName:`, that takes a pointer to an `NSString` object as an argument. The minus sign at the start of the method declaration indicates that it's an instance method, as opposed to a class method. This method will be used to set the name of the song that the object represents. The method does not return anything, so we declare it to return `void`.
- e. Declares an instance method, named `artist`, that returns a pointer to an `NSString` object when called.
- f. Declares an instance method, named `setArtist:`, that takes a pointer to an `NSString` object as an argument. This method will be used to set the artist of the song that the object represents. Once again, this method does not return anything, so we declare it to return `void`.



In our example so far, we've only defined instance methods using the minus sign (-). Class methods, such as the alloc method of NSObject, are defined in code using the plus sign (+).

Properties and Accessor Methods

Notice that in our *Song.h* file we have two methods for each variable, which are named using a simple pattern. The variable is known as a *property*. The methods are known as *accessor methods*. The method that is named after the property returns the value of the property to the caller. The method named `setProperty` changes the property. In the case of our *Song* class, the `name` method gets the name of the *Song*, and the `setName` method sets it. Hiding access to the properties of the object is called *encapsulation*; this allows you to change the implementation of how variables are stored inside the object without potentially breaking code elsewhere.

Defining the Song Implementation

Now that we have defined the interface, we actually need to fill in the implementation of the class. Take a look at the *Song.m* file. You will notice that it imports *Song.h*, which is the application's interface. There are also two compiler directives, `@implementation Song` and `@end`.

Add the code shown in Example 3-5 between the `@implementation` and `@end` directives.

Example 3-5. The Song.m implementation

```
#import "Song.h"

@implementation Song

- (NSString *)name // a
{
    return name; // b
}

- (void)setName:(NSString *)newName // c
{
    [newName retain]; // d
    [name release]; // e
    name = newName; // f
}

- (NSString *)artist // g
```

Example 3-5. The *Song.m* implementation (continued)

```
{
    return artist;                                // h
}

- (void)setArtist:(NSString *)newArtist          // i
{
    [newArtist retain];                           // j
    [artist release];                             // k
    artist = newArtist;                           // l
}
```

@end

Here's what the additional code does:

- a. Declares the `name` method that returns an `NSString` return value.
- b. Returns the `NSString` object associated with the `name` instance variable.
- c. Declares the `setName:` method that takes a single `NSString` argument.
- d. Sends the `retain` message to the `newName` object. This tells the object that we intend to keep a reference to it. This is part of Cocoa's memory management that will be described in depth in Chapter 4.
- e. Sends the `release` message to the `name` object. If the `name` object is not pointing to an `NSString` object (if it is pointing to `nil`), then this message will not do anything. However, if `name` had been set on this `Song` object before, this message would tell the `NSString` object that we were not interested in it anymore.
- f. Sets the `name` variable to point to the `NSString` object to which `newName` points.
- g. Declares the `artist` method that returns an `NSString` return value.
- h. Returns the `NSString` object associated with the `artist` instance variable.
- i. Declares the `setArtist:` method that takes a single `NSString` argument.
- j. Sends the `retain` message to the `newArtist` object, telling it that we are interested in keeping a reference to it.
- k. Sends a `release` message to the existing object to which our `artist` variable points, if any.
- l. Sets the `artist` variable to point to the `NSString` object to which `newArtist` points.

Using the `Song` Class

Now, we need to edit the main function in the `main.m` file, so we can do something with the `Song` class.

1. Edit the *main.m* file to match Example 3-6.

Example 3-6. Using the *Song* class

```
#import <Foundation/Foundation.h>
#import "Song.h" // a

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Song * song1 = [[Song alloc] init]; // b
    [song1 setName:@"We Have Explosive"];
    [song1 setArtist:@"The Future Sound of London"];

    Song * song2 = [[Song alloc] init]; // c
    [song2 setName:@"Loops of Fury"];
    [song2 setArtist:@"The Chemical Brothers"];

    NSLog(@"Song 1: %@", song1); // d
    NSLog(@"Song 2: %@", song2);

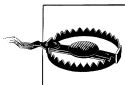
    [pool release];
    return 0;
}
```

Here's what the additional code does:

- a. Imports the *Song.h* interface file, so we can use the *Song* class.
 - b. Allocates, initializes, and sets the name (*setName*) and artist (*setArtist*) of *song1*. The *alloc* and *init* methods work just the same as they did with *NSObject*, since *Song* inherits them from *NSObject*.
 - c. Allocates, initializes, and sets the name and artist of *song2*.
 - d. Prints the *song1* and *song2* objects, so we can see them.
2. Build and run the program. You should see something like this on the console:

```
2002-06-11 22:05:11.866 songs[7058] Song 1: <Song: 0x50f30>
2002-06-11 22:05:11.867 songs[7058] Song 2: <Song: 0x4f4b0>
```

You will recognize that this output is similar to the output that was printed from the *NSObject* object instances. This is because the *NSLog* method actually calls the *description* method on an object, as defined by the *NSObject* class. To change this to print something a bit more user-friendly, we somehow need to redefine what the *description* method prints.



If you get a compiler error saying that the *song2* variable is undeclared, chances are that you are not using Mac OS X 10.2 (Jaguar). This book makes use of many of the new features of Jaguar, including support for the C99 standard in GCC 3.1.

Overriding Methods

A subclass can not only add new methods to the ones it inherits from a superclass; it can also replace, or *override*, an inherited method with a new implementation. No special syntax is required; just reimplement the method in the subclass's implementation file.

Overriding methods doesn't alter the set of messages that an object can receive. It alters the method implementation that will be used to respond to those messages. This ability for each class to implement its own version of a method is known as *polymorphism*.

1. Edit the *Song.m* file as shown in Example 3-7 to add the description method.

Example 3-7. Adding the description method

```
#import "Song.h"

@implementation Song

- (NSString *)name
{
    return name;
}

- (void)setName:(NSString *)newName
{
    [newName retain];
    [name release];
    name = newName;
}

- (NSString *)artist
{
    return artist;
}

- (void)setArtist:(NSString *)newArtist
{
    [newArtist retain];
    [artist release];
    artist = newArtist;
}

- (NSString *)description // a
{
    return [self name]; // b
}

@end
```

The code we added performs the following tasks:

- a. Declares the `description` method that overrides the method by the same name in the `NSObject` class. We don't need to declare this method in the `Song.h` interface file, as it is already part of the interface declared by `NSObject`.
 - b. Returns the name of the song as its description, using the special `self` variable that points to the object under operation. We could have just returned the variable directly from this method, but using the `[self name]` message means that if the internal implementation of the `Song` class changes, this method will work correctly with no additional work.
2. Build and run the program. You should see the following output on the console:

```
2002-06-11 22:32:20.435 songs[7096] Song 1: We Have Explosive
2002-06-11 22:32:20.436 songs[7096] Song 2: Loops of Fury
```

Overriding the `description` method allows us to assign much more meaningful strings for output than `NSObject`'s default class name and memory address output.

Calling Superclass Methods

Sometimes, in a method that overrides a superclass's method, calling the functionality in the superclass's method can be useful. To do this, you can send a message to `super`, a special variable in the Objective-C language. When you send a message to `super`, it indicates that an inherited method should be performed, rather than the method in the current class.

For example, if we wanted to print the same information that the `NSObject` class prints in the `description` method, we could implement our `description` method as follows:

```
- (NSString *)description
{
    return [super description];
}
```

If we were to make this change, we would see the following output:

```
2002-06-11 22:37:03.997 Songs[7115] Song 1: <Song: 0x53100>
2002-06-11 22:37:03.998 Songs[7115] Song 2: <Song: 0x530a0>
```

Since this defeats the purpose of overriding the `description` method, we're not going to add this implementation to our `Song` class. If you experiment with this, be sure to set it back to `return [self description]`.

Object Creation

One of a class's primary functions is to create new objects of the type defined by the class. As we've seen, objects are created at runtime in a two-step process that first allocates memory for the instance variables of the new object and then initializes

those variables. We've said this before, but because it's important, we'll repeat it here: an alloc message should *always* be coupled with an init message in the same line of code. The receiver for the alloc message is a class, while the receiver for the init message is the new object instance:

```
TheClass * newObject = [[TheClass alloc] init];
```

The alloc method dynamically allocates memory for a new instance of the receiving class and returns the new object. The receiver for the init message is the new object that was dynamically allocated by alloc. An object isn't ready for use until it is initialized, but it should be initialized only once. The version of the init method defined in the NSObject class does very little. In fact, it simply returns self, a special variable in Objective-C that is defined to point to the object that is called upon by the method.

After being allocated and initialized, a new object is a fully functional member of its class with its own set of variables. The newObject object can receive messages, store values in its instance variables, and so on.

Subclass versions of the init method should return the new object (self) after it has been successfully initialized. If it can't be initialized, the method should release the object and return nil. In some cases, an init method might release the new object and return a substitute. Programs should therefore always use the object returned by init, and not necessarily the one returned by alloc.

Subclass versions of init incorporate the initialization code for the classes from which they inherit through a message to super. When working with classes that inherit from NSObject, a simple call to the superclass init method, as shown in the following code block, is sufficient.

```
- init
{
    [super init];
    /* class-specific initialization goes here */
    return self;
}
```

Note that the message to super precedes the initialization code added in the method. This ensures that initialization proceeds in the order of inheritance.

However, since extending classes other than NSObject may return a different object than that on which the initializer was called, you must be more careful in these cases and use the following code:

```
- init
{
    if (self = [super init]) {
        /* class specific initialization goes here */
    }
    return self;
}
```

Note that this code checks to see if `super` returned an object, or `nil`, before doing any initialization itself. This code will work in any situation; however, none of the classes that we create in this book require these checks.



If you have been observant, you may have noticed that we have used two kinds of syntax to denote comments. The first is the traditional `/* ... */` C-style comment. The second is the newer `//` style comment that continues to the end of the line. You'll see both forms used quite frequently in Objective-C code. There really aren't any guidelines as to which style should be used where. You should simply use whichever works best, given the context of the comment.

Designated initializers

Subclasses often define initializer methods with additional arguments to allow specific values to be set. The more arguments an initializer method has, the more freedom it gives you to determine the character of initialized objects. Classes often have a set of initializer methods, each with a different number of arguments, to set up objects ahead of time with appropriate information. For example, we could define the following initializer method for our `Song` class:

```
- (id)initWithName:(NSString *)newName artist:(NSString *)newArtist;
```

This initializer allows us to create new `Song` objects and set them up with one line of code rather than three. For this to work properly in cases where users of this class don't call this initializer, but simply use the `init` method, we make sure that the `init` method calls this initializer with appropriate default arguments. This method is called the *designated initializer* for the class. The other initialization methods defined in this class invoke the designated initializer through messages to `self`. In this way, all the initializers are chained together. The designated initializer should always call its superclass's designated initializer.



Typically, though not always, the designated initializer is the one with the most arguments. The only way to determine the designated initializer of a class accurately is to read the documentation for the class.

1. To work with designated initializers, edit `Song.h` and add the initializers, as shown in Example 3-8.

Example 3-8. Adding a designated initializer

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

@interface Song : NSObject {
    NSString *name;
    NSString *artist;
}
- (id)initWithName:(NSString *)newName artist:(NSString *)newArtist;
- (NSString *)name;
```

Example 3-8. Adding a designated initializer (continued)

```
- (void)setName:(NSString *)newName;
- (NSString *)artist;
- (void)setArtist:(NSString *)newArtist;
```

```
@end
```

The code we added declares an initializer for our `Song` class that takes the name of the song as well as the artist.

2. Now add the initializer implementations to `Song.m` as shown in Example 3-9.

Example 3-9. Designated initializer implementation

```
#import "Song.h"

@implementation Song

- (id)init                                     // a
{
    return [self initWithName:nil artist:nil];
}

- (id)initWithName:(NSString *)newName artist:(NSString *)newArtist // b
{
    [super init];                             // c
    [self setName:newName];                   // d
    [self setArtist:newArtist];              // e
    return self;                              // f
}

- (NSString *)name
{
    return name;
}

- (void)setName:(NSString *)newName
{
    [newName retain];
    [name release];
    name = newName;
}

- (NSString *)artist
{
    return artist;
}

- (void)setArtist:(NSString *)newArtist
{
    [newArtist retain];
    [artist release];
    artist = newArtist;
}
```

Example 3-9. Designated initializer implementation (continued)

```
- (NSString *)description
{
    return [super description];
}

@end
```

The code we added in Example 3-9 performs the following tasks:

- a. Overrides the `init` method provided by `NSObject`. This overridden method calls the new designated initializer with `nil` string arguments for the name and artist arguments.
 - b. Declares our designated initializer with the same signature we used in *Song.h*.
 - c. Calls the `init` method of the `NSObject` superclass.
 - d. Sets the name of the new object.
 - e. Sets the artist of the new object.
 - f. Returns the freshly initialized object, ready for use.
3. Now, edit the *main.m* file to match Example 3-10.

Example 3-10. Using the designated initializer

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

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Song * song1 = [[Song alloc] initWithName:@"We Have Explosive"
                                                artist:@"The Future Sound of London"];
    Song * song2 = [[Song alloc] initWithName:@"Loops of Fury"
                                                artist:@"The Chemical Brothers"];

    NSLog(@"Song 1: %@", song1);
    NSLog(@"Song 2: %@", song2);

    [pool release];
    return 0;
}
```

In this code, we've simply replaced the longer three lines with our new initializer.

4. Build and run the program. You should see the following familiar output:
- ```
2002-06-11 23:08:07.783 Songs[7195] Song 1: We Have Explosive
2002-06-11 23:08:07.784 Songs[7195] Song 2: Loops of Fury
```



As you can see in Example 3-10, a line of code is often too long to fit on one line. Project Builder has autoindentation functionality to make these constructs look good automatically, so you don't have to type in a bunch of spaces manually. Simply go into Project Builder's preferences, select the Indentation pane, and make sure that the "Syntax-aware indenting" checkbox is checked.

## Object Deallocation

We have a flaw in our very simple program in the form of a memory leak. Of course, this leak is probably not going to hurt anybody—since the program exits so quickly, allowing the operating system to reclaim all the memory belonging to the process—but it doesn't pay to get into bad habits. As well, code has a tendency to be reused in ways that the original author did not expect. Therefore, you should always make a point of cleaning up after your code, no matter how simple it is.

When an object is no longer being used by the program, it must be deallocated. When an object is released, the `dealloc` method (provided by the `NSObject` class) is called, letting it release objects it has created, free allocated memory, and so on. Since our `Song` class has two instance variable objects, they need to be released when an instance of the class is released.

1. To do this, we need to add a `dealloc` method implementation to our `Song` class in the `Song.m` file, as shown in Example 3-11.

*Example 3-11. Adding a deallocation method*

```
#import "Song.h"

@implementation Song

- (id)init
{
 return [self initWithName:nil artist:nil];
}

- (id)initWithName:(NSString *)newName artist:(NSString *)newArtist
{
 [super init];
 [self setName:newName];
 [self setArtist:newArtist];
 return self;
}

- (void)dealloc // a
{
 NSLog(@"Deallocating %@", self); // b
}
```

Example 3-11. Adding a deallocation method (continued)

```
[name release]; // c
[artist release]; // d
[super dealloc]; // e
}

- (NSString *)name
{
 return name;
}

- (void)setName:(NSString *)newName
{
 [newName retain];
 [name release];
 name = newName;
}

- (NSString *)artist
{
 return artist;
}

- (void)setArtist:(NSString *)newArtist
{
 [newArtist retain];
 [artist release];
 artist = newArtist;
}

- (NSString *)description
{
 return [self name];
}

@end
```

The code that we added in Example 3-11 performs the following tasks:

- a. Declares the `dealloc` method. Note that since the `dealloc` method is defined by the `NSObject` class, we don't have to declare it in the *Song.h* header file.
  - b. Prints out a message saying that the object is being deallocated.
  - c. Releases the `name` instance variable.
  - d. Releases the `artist` instance variable.
  - e. Calls `dealloc` on the superclass, allowing the deallocation functionality of the `NSObject` class to operate. When you override the default `dealloc` functionality, you must always be sure to call `dealloc` in the superclass.
2. Edit the *main.m* source file with the changes shown in Example 3-12.

### Example 3-12. Releasing the song objects created

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

int main (int argc, const char * argv[]) {
 NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

 Song * song1 = [[Song alloc] initWithName:@"We Have Explosive"
 artist:@"The Future Sound of London"];
 Song * song2 = [[Song alloc] initWithName:@"Loops of Fury"
 artist:@"The Chemical Brothers"];

 NSLog(@"Song 1: %@", song1);
 NSLog(@"Song 2: %@", song2);

 [song1 release];
 [song2 release];

 [pool release];
 return 0;
}
```

The added code tells the system that we are no longer interested in the `song1` and `song2` variables. Because we are no longer interested, and there are no other objects interested in these variables, they will be deallocated immediately. This will plug up our memory leak, making it a good citizen.

3. Build and run the project. You should see the following output:

```
2002-06-11 23:12:07.783 songs[7200] Song 1: We Have Explosive
2002-06-11 23:12:07.784 songs[7200] Song 2: Loops of Fury
2002-06-11 23:12:07.783 songs[7200] Deallocating We Have Explosive
2002-06-11 23:12:07.784 songs[7200] Deallocating Loops of Fury
```

In Chapter 4, we present the finer details of memory management and explain why the act of releasing an object here calls the `dealloc` method of our `Song` objects.

## Other Concepts

There are some other concepts in object-oriented programming and Objective-C that we haven't explored in depth in this chapter. Before you learn too much about these new concepts, you'll want to practice quite a bit with the concepts that you've already learned. We're telling you about these other concepts now so that when you come to them, you won't be surprised.

### Categories

You can add methods to a class by declaring them in an interface file under a category name and defining them in an implementation file under the same name. The category name indicates that the methods are additions to a class declared elsewhere, not to a new class.

### *Protocols*

Class and category interfaces declare methods that are associated with a particular class—methods that the class implements. Informal and formal protocols, on the other hand, declare methods that are not associated with a class, but which any class—and perhaps many classes—might implement. We'll talk more about protocols in Chapter 9.

### *Introspection*

An object, even one typed as `id`, can reveal its class and divulge other characteristics at runtime. Several introspection methods, such as `isMemberOfClass:` and `isKindOfClass:`, allow you to ascertain the inheritance relationships of an object and the methods to which it responds.

Remember, you can find out much more information about Objective-C and object-oriented programming in the developer documentation installed on your hard drive along with the Developer Tools (*/Developer/Documentation/Cocoa/ObjectiveC*).

## **Exercises**

1. Use the resources in Appendix C, and read the documentation for `NSObject` and `NSString`.
2. Read the documentation for the `NSLog` function.
3. Investigate the `isa` and `self` variables by having the designated initializer of the `Song` class print a description of the class.