The Objective-C Programming Language

2006-01-10

Contents

Tables, Figures, and Listings

Introduction to The Objective-C Programming Language

An object-oriented approach to application development makes programs more intuitive to design, faster to develop, more amenable to modification, and easier to understand. Most object-oriented development environments consist of at least three parts:

- A library of objects
- A set of development tools
- An object-oriented programming language and support library

This document is about the third component of the development environment—the programming language and its runtime environment. It fully describes the Objective-C language, and provides a foundation for learning about the Mac OS X Objective-C application development framework—Cocoa.

The Objective-C language is a simple computer language designed to enable sophisticated object-oriented programming. Objective-C is defined as small but powerful set of extensions to the standard ANSI C language. Its additions to C are mostly based on Smalltalk, one of the first object-oriented programming languages. Objective-C is designed to give C full object-oriented programming capabilities, and to do so in a simple and straightforward way.

For those who have never used object-oriented programming to create applications before, this document is also designed to help you become familiar with object-oriented development. It spells out some of the implications of object-oriented design and gives you a flavor of what writing an object-oriented program is really like.

Who Should Read This Document

The document is intended for readers who might be interested in:

- Learning about object-oriented programming
- Finding out about the basis for the Cocoa application framework
- Programming in Objective-C

This document both introduces the object-oriented model that Objective-C is based upon and fully documents the language. It concentrates on the Objective-C extensions to C, not on the C language itself.

Because this isn't a document about C, it assumes some prior acquaintance with that language. However, it doesn't have to be an extensive acquaintance. Object-oriented programming in Objective-C is sufficiently different from procedural programming in ANSI C that you won't be hampered if you're not an experienced C programmer.

Organization of This Document

This document is divided into four chapters and two appendixes. The chapters are:

- "Why Objective-C" (page 13) explains why Objective-C was chosen as the development language for the Cocoa frameworks.
- "Object-Oriented Programming" (page 15) discusses the rationale for object-oriented programming languages and introduces much of the terminology. It develops the ideas behind object-oriented programming techniques. Even if you're already familiar with object-oriented programming, you are encouraged to read this chapter to gain a sense of the Objective-C perspective on object-orientation and its use of terminology.
- "The Language" (page 39) describes the basic concepts and syntax of Objective-C. It covers many of the same topics as "Object-Oriented Programming" (page 15), but looks at them from the standpoint of the Objective-C language. It reintroduces the terminology of object-oriented programming, but in the context of Objective-C.
- "The Runtime System" (page 95) looks at the NSObject class and how Objective-C programs interact with the runtime system. In particular, it examines the paradigms for managing object allocations, dynamically loading new classes at runtime, and forwarding messages to other objects.

The appendixes contain reference material that might be useful for understanding the language. They are:

- "Language Summary" (page 121) lists and briefly comments on all of the Objective-C extensions to the C language.
- "Grammar" (page 129) presents, without comment, a formal grammar of the Objective-C extensions to the C language. This reference manual is meant to be read as a companion to the reference manual for C presented in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall.

Conventions

Where this document discusses functions, methods, and other programming elements, it makes special use of computer voice and italic fonts. Computer voice denotes words or characters that are to be taken literally (typed as they appear). Italic denotes words that represent something else or can be varied. For example, the syntax:

`@interface`*ClassName*`(`*CategoryName*`)`

means that `@interface` and the two parentheses are required, but that you can choose the class name and category name.

Where example code is shown, ellipsis points indicates the parts, often substantial parts, that have been omitted:

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    ...
}
```

The conventions used in the reference appendix are described in that appendix.

See Also

Objective-C Runtime Reference describes the data structures and functions of the Objective-C runtime support library. Your programs can use these interfaces to interact with the Objective-C runtime system. For example, you can add classes or methods, or obtain a list of all class definitions for loaded classes.

Why Objective-C?

The Objective-C language was chosen for the Cocoa frameworks for a variety of reasons. First and foremost, it's an object-oriented language. The kind of functionality that's packaged in the Cocoa frameworks can only be delivered through object-oriented techniques. This document explains the operation of the frameworks and how you can take advantage of them. Second, because Objective-C is an extension of standard ANSI C, existing C programs can be adapted to use the software frameworks without losing any of the work that went into their original development. Since Objective-C incorporates C, you get all the benefits of C when working within Objective-C. You can choose when to do something in an object-oriented way (define a new class, for example) and when to stick to procedural programming techniques (define a structure and some functions instead of a class).

Moreover, Objective-C is a simple language. Its syntax is small, unambiguous, and easy to learn. Object-oriented programming, with its self-conscious terminology and emphasis on abstract design, often presents a steep learning curve to new recruits. A well-organized language like Objective-C can make becoming a proficient object-oriented programmer that much less difficult. The length of the chapter describing the language itself is a testament to the simplicity of Objective-C. It's not a long chapter.

Compared to other object oriented languages based on C, Objective-C is very dynamic. The compiler preserves a great deal of information about the objects themselves for use at runtime. Decisions that otherwise might be made at compile time can be postponed until the program is running. This gives Objective-C programs unusual flexibility and power. For example, Objective-C's dynamism yields two big benefits that are hard to get with other nominally object-oriented languages:

- Objective-C supports an open style of dynamic binding, a style that can accommodate a simple architecture for interactive user interfaces. Messages are not necessarily constrained by either the class of the receiver or the method selector, so a software framework can allow for user choices at runtime and permit developers freedom of expression in their design. (Terminology like "dynamic binding," "message," "class," "receiver," and "selector" are explained in due course in this document.)
- Objective-C's dynamism enables the construction of sophisticated development tools. An interface to the runtime system provides access to information about running applications, so it's possible to develop tools that monitor, intervene, and reveal the underlying structure and activity of Objective-C applications.

Object-Oriented Programming

Programming languages have traditionally divided the world into two parts—data and operations on data. Data is static and immutable, except as the operations may change it. The procedures and functions that operate on data have no lasting state of their own; they're useful only in their ability to affect data.

This division is, of course, grounded in the way computers work, so it's not one that you can easily ignore or push aside. Like the equally pervasive distinctions between matter and energy and between nouns and verbs, it forms the background against which we work. At some point, all programmers—even object-oriented programmers—must lay out the data structures that their programs will use and define the functions that will act on the data.

With a procedural programming language like C, that's about all there is to it. The language may offer various kinds of support for organizing data and functions, but it won't divide the world any differently. Functions and data structures are the basic elements of design.

Object-oriented programming doesn't so much dispute this view of the world as restructure it at a higher level. It groups operations and data into modular units called **objects** and lets you combine objects into structured networks to form a complete program. In an object-oriented programming language, objects and object interactions are the basic elements of design.

Every object has both state (data) and behavior (operations on data). In that, they're not much different from ordinary physical objects. It's easy to see how a mechanical device, such as a pocket watch or a piano, embodies both state and behavior. But almost anything that's designed to do a job does, too. Even simple things with no moving parts such as an ordinary bottle combine state (how full the bottle is, whether or not it's open, how warm its contents are) with behavior (the ability to dispense its contents at various flow rates, to be opened or closed, to withstand high or low temperatures).

It's this resemblance to real things that gives objects much of their power and appeal. They can not only model components of real systems, but equally as well fulfill assigned roles as components in software systems.

Interface and Implementation

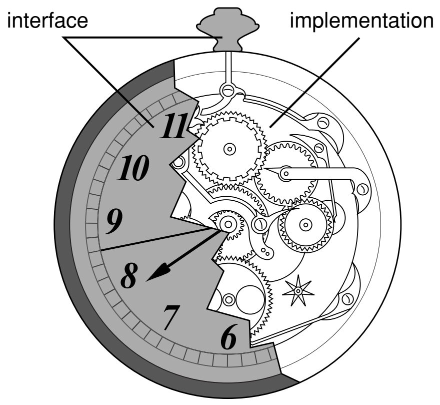
As humans, we're constantly faced with myriad facts and impressions that we must make sense of. To do so, we have to abstract underlying structure away from surface details and discover the fundamental relations at work. Abstractions reveal causes and effects, expose patterns and frameworks, and separate what's important from what's not. They're at the root of understanding.

To invent programs, you need to be able to capture the same kinds of abstractions and express them in the program design.

It's the job of a programming language to help you do this. The language should facilitate the process of invention and design by letting you encode abstractions that reveal the way things work. It should let you make your ideas concrete in the code you write. Surface details shouldn't obscure the architecture of your program.

All programming languages provide devices that help express abstractions. In essence, these devices are ways of grouping implementation details, hiding them, and giving them, at least to some extent, a common interface—much as a mechanical object separates its interface from its implementation, as illustrated in Figure 2-1.

Figure 2-1 Interface and Implementation



Looking at such a unit from the inside, as the implementor, you'd be concerned with what it's composed of and how it works. Looking at it from the outside, as the user, you're concerned only with what it is and what it does. You can look past the details and think solely in terms of the role that the unit plays at a higher level.

The principal units of abstraction in the C language are structures and functions. Both, in different ways, hide elements of the implementation:

- On the data side of the world, C structures group data elements into larger units which can then be handled as single entities. While some code must delve inside the structure and manipulate the fields separately, much of the program can regard it as a single thing—not as a collection of elements, but as what those elements taken together represent. One structure can include others, so a complex arrangement of information can be built from simpler layers.

  In modern C, the fields of a structure live in their own name space—that is, their names won't conflict with identically named data elements outside the structure. Partitioning the program name space is essential for keeping implementation details out of the interface. Imagine, for example, the enormous task of assigning a different name to every piece of data in a large program and of making sure new names don't conflict with old ones.

- On the procedural side of the world, functions encapsulate behaviors that can be used repeatedly without being reimplemented. Data elements local to a function, like the fields within a structure, are protected within their own name space. Functions can reference (call) other functions, so quite complex behaviors can be built from smaller pieces.

Functions are reusable. Once defined, they can be called any number of times without again considering the implementation. The most generally useful functions can be collected in libraries and reused in many different applications. All the user needs is the function interface, not the source code.

However, unlike data elements, functions aren't partitioned into separate name spaces. Each function must have a unique name. Although the function may be reusable, its name is not.

C structures and functions are able to express significant abstractions, but they maintain the distinction between data and operations on data. In a procedural programming language, the highest units of abstraction still live on one side or the other of the data-versus-operations divide. The programs you design must always reflect, at the highest level, the way the computer works.

Object-oriented programming languages don't lose any of the virtues of structures and functions—they go a step further and add a unit capable of abstraction at a higher level, a unit that hides the interaction between a function and its data.

Suppose, for example, that you have a group of functions that act on a particular data structure. You want to make those functions easier to use by, as far as possible, taking the structure out of the interface. So you supply a few additional functions to manage the data. All the work of manipulating the data structure—allocating it, initializing it, getting information from it, modifying values within it, keeping it up to date, and freeing it—is done through the functions. All the user does is call the functions and pass the structure to them.

With these changes, the structure has become an opaque token that other programmers never need to look inside. They can concentrate on what the functions do, not how the data is organized. You've taken the first step toward creating an object.

The next step is to give this idea support in the programming language and completely hide the data structure so that it doesn't even have to be passed between the functions. The data becomes an internal implementation detail; all that's exported to users is a functional interface. Because objects completely encapsulate their data (hide it), users can think of them solely in terms of their behavior.

With this step, the interface to the functions has become much simpler. Callers don't need to know how they're implemented (what data they use). It's fair now to call this an "object."

The hidden data structure unites all the functions that share access to it. So, an object is more than a collection of random functions; it's a bundle of related behaviors that are supported by shared data. To use a function that belongs to an object, you first create the object (thus giving it its internal data structure), then tell the object which function it should perform. You begin to think in terms of what the object does, rather than in terms of the individual functions.

This progression from thinking about functions and data structures to thinking about object behaviors is the essence of learning object-oriented programming. It may seem unfamiliar at first, but as you gain experience with object-oriented programming, you find it's a more natural way to think about things. Everyday programming terminology is replete with analogies to real-world objects of various kinds—lists, containers, tables, controllers, even managers. Implementing such things as programming objects merely extends the analogy in a natural way.

A programming language can be judged by the kinds of abstractions that it enables you to encode. You shouldn't be distracted by extraneous matters or forced to express yourself using a vocabulary that doesn't match the reality you're trying to capture.

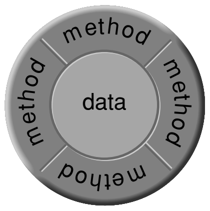
If, for example, you must always tend to the business of keeping the right data matched with the right procedure, you're forced at all times to be aware of the entire program at a low level of implementation. While you might still invent programs at a high level of abstraction, the path from imagination to implementation can become quite tenuous—and more and more difficult as programs become bigger and more complicated.

By providing another, higher level of abstraction, object-oriented programming languages give you a larger vocabulary and a richer model to program in.

The Object Model

The insight of object-oriented programming is to combine state and behavior—data and operations on data—in a high-level unit, an **object**, and to give it language support. An object is a group of related functions and a data structure that serves those functions. The functions are known as the object's **methods**, and the fields of its data structure are its **instance variables**. The methods wrap around the instance variables and hide them from the rest of the program, as Figure 2-2 illustrates:

Figure 2-2 An Object



Likely, if you've ever tackled any kind of difficult programming problem, your design has included groups of functions that work on a particular kind of data—implicit "objects" without the language support. Object-oriented programming makes these function groups explicit and permits you to think in terms of the group, rather than its components. The only way to an object's data, the only interface, is through its methods.

By combining both state and behavior in a single unit, an object becomes more than either alone; the whole really is greater than the sum of its parts. An object is a kind of self-sufficient "subprogram" with jurisdiction over a specific functional area. It can play a full-fledged modular role within a larger program design.

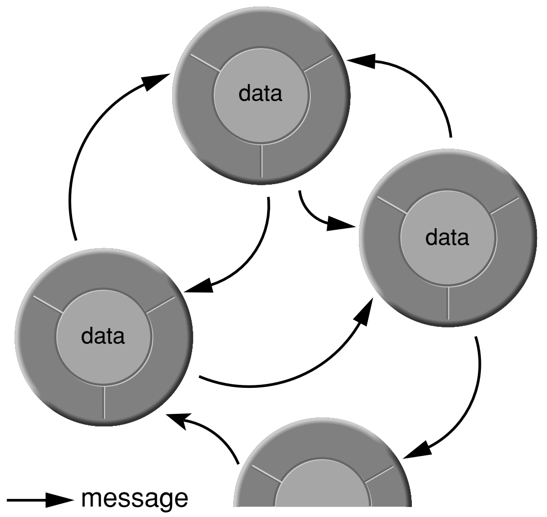
Terminology: Object-oriented terminology varies from language to language. For example, in C++, methods are called "member functions" and instance variables are known as "data members." This document uses the terminology of Objective-C, which has its basis in Smalltalk.

For example, if you were to write a program that modeled home water usage, you might invent objects to represent the various components of the water-delivery system. One might be a Faucet object that would have methods to start and stop the flow of water, set the rate of flow, return the amount of water consumed in a given period, and so on. To do this work, a Faucet object would need instance variables to keep track of whether the tap is open or shut, how much water is being used, and where the water is coming from.

Clearly, a programmatic Faucet can be smarter than a real one (it's analogous to a mechanical faucet with lots of gauges and instruments attached). But even a real faucet, like any system component, exhibits both state and behavior. To effectively model a system, you need programming units, like objects, that also combine state and behavior.

A program consists of a network of interconnected objects that call upon each other to solve a part of the puzzle. Each object has a specific role to play in the overall design of the program and is able to communicate with other objects. Objects communicate through **messages**, requests to perform methods.

Figure 2-3 Object Network



The objects in the network won't all be the same. For example, in addition to Faucets, the program that models water usage might also have WaterPipe objects that can deliver water to the Faucets and Valve objects to regulate the flow among WaterPipes. There could be a Building object to coordinate a set of WaterPipes, Valves, and Faucets, some Appliance objects—corresponding to dishwashers, toilets, and washing machines—that can turn Valves on and off, and maybe some Users to work the Appliances and Faucets. When a Building object is asked how much water is being used, it might call upon each Faucet and Valve to report its current state. When a User starts up an Appliance, the Appliance will need to turn on a Valve to get the water it requires.

The Messaging Metaphor

Every programming paradigm comes with its own terminology and metaphors. None more so than object-oriented programming. Its jargon invites you to think about what goes on in a program from a particular perspective.

There's a tendency, for example, to think of objects as "actors" and to endow them with human-like intentions and abilities. It's tempting sometimes to talk about an object "deciding" what to do about a situation, "asking" other objects for information, "introspecting" about itself to get requested information, "delegating" responsibility to another object, or "managing" a process.

Rather than think in terms of functions or methods doing the work, as you would in a procedural programming language, this metaphor asks you to think of objects as "performing" their methods. Objects are not passive containers for state and behavior, but are said to be the agents of the program's activity.

This is actually a useful metaphor. An object is like an actor in a couple of respects: It has a particular role to play within the overall design of the program, and within that role it can act fairly independently of the other parts of the program. It interacts with other objects as they play their own roles, but is self-contained and to a certain extent can act on its own. Like an actor on stage, it can't stray from the script, but the role it plays can be multi-faceted and quite complex.

The idea of objects as actors fits nicely with the principal metaphor of object-oriented programming—the idea that objects communicate through "messages." Instead of calling a method as you would a function, you send a message to an object requesting it to perform one of its methods.

Although it can take some getting used to, this metaphor leads to a useful way of looking at methods and objects. It abstracts methods away from the particular data they act on and concentrates on behavior instead. For example, in an object-oriented programming interface, a `start` method might initiate an operation, an `archive` method might archive information, and a `draw` method might produce an image. Exactly which operation is initiated, which information is archived, and which image is drawn isn't revealed by the method name. Different objects might perform these methods in different ways.

Thus, methods are a vocabulary of abstract behaviors. To invoke one of those behaviors, you have to make it concrete by associating the method with an object. This is done by naming the object as the "receiver" of a message. The object you choose as receiver will determine the exact operation that's initiated, the data that's archived, or the image that's drawn.

Since methods belong to objects, they can be invoked only through a particular receiver (the owner of the method and of the data structure the method will act on). Different receivers can have different implementations of the same method, so different receivers can do different things in response to the same message. The result of a message can't be calculated from the message or method name alone; it also depends on the object that receives the message.

By separating the message (the requested behavior) from the receiver (the owner of a method that can respond to the request), the messaging metaphor perfectly captures the idea that behaviors can be abstracted away from their particular implementations.

Classes

A program can have more than one object of the same kind. The program that models water usage, for example, might have several Faucets and WaterPipes and perhaps a handful of Appliances and Users. Objects of the same kind are said to be members of the same **class**. All members of a class are able to perform the same methods and have matching sets of instance variables. They also share a common definition; each kind of object is defined just once.

In this, objects are similar to C structures. Declaring a structure defines a type. For example, this declaration

```
struct key {
    char *word;
    int count;
};
```

defines the `struct key` type. Once defined, the structure name can be used to produce any number of instances of the type:

```
struct key  a, b, c, d;
struct key *p = malloc(sizeof(struct key) * MAXITEMS);
```

The declaration is a template for a kind of structure, but it doesn't create a structure that the program can use. It takes another step to allocate memory for an actual structure of that type, a step that can be repeated any number of times.

Similarly, defining an object creates a template for a kind of object. It defines a class of objects. The template can be used to produce any number of similar objects—**instances** of the class. For example, there would be a single definition of the Faucet class. Using this definition, a program could allocate as many Faucet instances as it needed.

A class definition is like a structure definition in that it lays out an arrangement of data elements (instance variables) that become part of every instance. Each instance has memory allocated for its own set of instance variables, which store values peculiar to the instance.

However, a class definition differs from a structure declaration in that it also defines methods that specify the behavior of class members. Every instance is characterized by its access to the methods defined for the class. Two objects with equivalent data structures but different methods would not belong to the same class.

Modularity

To a C programmer, a "module" is nothing more than a file containing source code. Breaking a large (or even not-so-large) program into different files is a convenient way of splitting it into manageable pieces. Each piece can be worked on independently and compiled alone, then integrated with other pieces when the program is linked. Using the `static` storage class designator to limit the scope of names to just the files where they're declared enhances the independence of source modules.

This kind of module is a unit defined by the file system. It's a container for source code, not a logical unit of the language. What goes into the container is up to each programmer. You can use them to group logically related parts of the code, but you don't have to. Files are like the drawers of a dresser; you can put your socks in one drawer, underwear in another, and so on, or you can use another organizing scheme or simply choose to mix everything up.

Access to Methods: It's convenient to think of methods as being part of an object, just as instance variables are. As in Figure 2-2 (page 18), methods can be diagrammed as surrounding the object's instance variables.But, of course, methods aren't grouped with instance variables in memory. Memory is allocated for the instance variables of each new object, but there's no need to allocate memory for methods. All an instance needs is access to its methods, and all instances of the same class share access to the same set of methods. There's only one copy of the methods in memory, no matter how many instances of the class are created.

Object-oriented programming languages support the use of file containers for source code, but they also add a logical module to the language—class definitions. As you'd expect, it's often the case that each class is defined in its own source file—logical modules are matched to container modules.

In Objective-C, for example, it would be possible to define the part of the Valve class that interacts with WaterPipes in the same file that defines the WaterPipe class, thus creating a container module for WaterPipe-related code and splitting the Valve class into more than one file. The Valve class definition would still act as a modular unit within the construction of the program—it would still be a logical module—no matter how many files the source code was located in.

The mechanisms that make class definitions logical units of the language are discussed in some detail under "Mechanisms Of Abstraction" (page 23).

Reusability

A principal goal of object-oriented programming is to make the code you write as reusable as possible—to have it serve many different situations and applications—so that you can avoid reimplementing, even if only slightly differently, something that's already been done.

Reusability is influenced by a variety of different factors, including:

- How reliable and bug-free the code is
- How clear the documentation is
- How simple and straightforward the programming interface is
- How efficiently the code performs its tasks
- How full the feature set is

Clearly, these factors don't apply just to the object model. They can be used to judge the reusability of any code—standard C functions as well as class definitions. Efficient and well-documented functions, for example, would be more reusable than undocumented and unreliable ones.

Nevertheless, a general comparison would show that class definitions lend themselves to reusable code in ways that functions do not. There are various things you can do to make functions more reusable—passing data as arguments rather than assuming specifically named global variables, for example. Even so, it turns out that only a small subset of functions can be generalized beyond the applications they were originally designed for. Their reusability is inherently limited in at least three ways:

- Function names are global; each function must have a unique name (except for those declared `static`). This makes it difficult to rely heavily on library code when building a complex system. The programming interface would be hard to learn and so extensive that it couldn't easily capture significant generalizations.

  Classes, on the other hand, can share programming interfaces. When the same naming conventions are used over and over, a great deal of functionality can be packaged with a relatively small and easy-to-understand interface.

- Functions are selected from a library one at a time. It's up to programmers to pick and choose the individual functions they need.

  In contrast, objects come as packages of functionality, not as individual methods and instance variables. They provide integrated services, so users of an object-oriented library won't get bogged down piecing together their own solutions to a problem.

- Functions are typically tied to particular kinds of data structures devised for a specific program. The interaction between data and function is an unavoidable part of the interface. A function is useful only to those who agree to use the same kind of data structures it accepts as arguments.

> Because it hides its data, an object doesn't have this problem. This is one of the principal reasons why classes can be reused more easily than functions.

An object's data is protected and won't be touched by any other part of the program. Methods can therefore trust its integrity. They can be sure that external access hasn't put it in an illogical or untenable state. This makes an object data structure more reliable than one passed to a function, so methods can depend on it more. Reusable methods are consequently easier to write.

Moreover, because an object's data is hidden, a class can be reimplemented to use a different data structure without affecting its interface. All programs that use the class can pick up the new version without changing any source code; no reprogramming is required.

Mechanisms Of Abstraction

To this point, objects have been introduced as units that embody higher-level abstractions and as coherent role-players within an application. However, they couldn't be used this way without the support of various language mechanisms. Two of the most important mechanisms are: **encapsulation** and **polymorphism**.

Encapsulation keeps the implementation of an object out of its interface, and polymorphism results from giving each class its own name space. The following sections discuss each of these mechanisms in turn.

Encapsulation

To design effectively at any level of abstraction, you need to be able to leave details of implementation behind and think in terms of units that group those details under a common interface. For a programming unit to be truly effective, the barrier between interface and implementation must be absolute. The interface must encapsulate the implementation—hide it from other parts of the program. Encapsulation protects an implementation from unintended actions and inadvertent access.

In C, a function is clearly encapsulated; its implementation is inaccessible to other parts of the program and protected from whatever actions might be taken outside the body of the function. Method implementations are similarly encapsulated, but, more importantly, so are an object's instance variables. They're hidden inside the object and invisible outside it. The encapsulation of instance variables is sometimes also called *information hiding*.

It might seem, at first, that hiding the information in instance variables would constrain your freedom as a programmer. Actually, it gives you more room to act and frees you from constraints that might otherwise be imposed. If any part of an object's implementation could leak out and become accessible or a concern to other parts of the program, it would tie the hands both of the object's implementor and of those who would use the object. Neither could make modifications without first checking with the other.

Suppose, for example, that you're interested in the Faucet object being developed for the program that models water use and you want to incorporate it in another program you're writing. Once the interface to the object is decided, you don't have to be concerned as others work on it, fix bugs, and find better ways to implement it. You get the benefit of these improvements, but none of them affects what you do in your program. Because you depend solely on the interface, nothing they do can break your code. Your program is insulated from the object's implementation.

Moreover, although those implementing the Faucet object would be interested in how you use the class and might try to make sure it meets your needs, they don't have to be concerned with the way you write your code. Nothing you do can touch the implementation of the object or limit their freedom to make implementation changes in future releases. The implementation is insulated from anything that you or other users of the object might do.

Polymorphism

The ability of different objects to respond, each in its own way, to identical messages is called polymorphism.

Polymorphism results from the fact that every class lives in its own name space. The names assigned within a class definition don't conflict with names assigned anywhere outside it. This is true both of the instance variables in an object's data structure and of the object's methods:

- Just as the fields of a C structure are in a protected namespace, so are an object's instance variables.
- Method names are also protected. Unlike the names of C functions, method names aren't global symbols. The name of a method in one class can't conflict with method names in other classes; two very different classes could implement identically named methods.

Method names are part of an object's interface. When a message is sent requesting an object to do something, the message names the method the object should perform. Because different objects can have methods with the same name, the meaning of a message must be understood relative to the particular object that receives the message. The same message sent to two different objects could invoke two distinct methods.

The main benefit of polymorphism is that it simplifies the programming interface. It permits conventions to be established that can be reused in class after class. Instead of inventing a new name for each new function you add to a program, the same names can be reused. The programming interface can be described as a set of abstract behaviors, quite apart from the classes that implement them.

Overloading: The terms "polymorphism" and "argument overloading" refer basically to the same thing, but from slightly different points of view. Polymorphism takes a pluralistic point of view and notes that several classes can each have a method with the same name. Argument overloading takes the point of the view of the method name and notes that it can have different effects depending on the arguments passed to it.Operator overloading is similar. It refers to the ability to turn operators of the language (such as == and + in C) into methods that can be assigned particular meanings for particular kinds of objects. Objective-C implements polymorphism of method names, but not argument or operator overloading.

For example, suppose you want to report the amount of water used by an Appliance object over a given period of time. Instead of defining an `amountConsumed` method for the Appliance class, an `amountDispensedAtFaucet` method for a Faucet class, and a `cumulativeUsage` method for a Building class, you can simply define a `waterUsed` method for each class. This consolidation reduces the number of methods used for what is conceptually the same operation.

Polymorphism also permits code to be isolated in the methods of different objects rather than be gathered in a single function that enumerates all the possible cases. This makes the code you write more extensible and reusable. When a new case comes along, you don't have to reimplement existing code; you need only to add a new class with a new method, leaving the code that's already written alone.

For example, suppose you have code that sends a `draw` message to an object. Depending on the receiver, the message might produce one of two possible images. When you want to add a third case, you don't have to change the message or alter existing code, but merely allow another object to be assigned as the message receiver.

Inheritance

The easiest way to explain something new is to start with something understood. If you want to describe what a "schooner" is, it helps if your listeners already know what a "sailboat" is. If you want to explain how a harpsichord works, it's best if you can assume your audience has already looked inside a piano, or has seen a guitar played, or at least is familiar with the idea of a "musical instrument."

The same is true if you want to define a new kind of object; the description is simpler if it can start from the definition of an existing object.

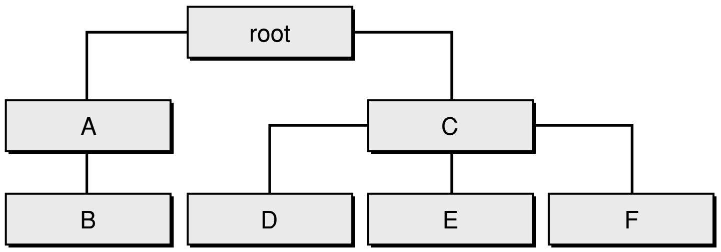
With this in mind, object-oriented programming languages permit you to base a new class definition on a class already defined. The base class is called a **superclass**; the new class is its **subclass**. The subclass definition specifies only how it differs from the superclass; everything else is taken to be the same.

Nothing is copied from superclass to subclass. Instead, the two classes are connected so that the subclass **inherits** all the methods and instance variables of its superclass, much as you want your listener's understanding of "schooner" to inherit what they already know about sailboats. If the subclass definition were empty (if it didn't define any instance variables or methods of its own), the two classes would be identical (except for their names) and share the same definition. It would be like explaining what a "fiddle" is by saying that it's exactly the same as a "violin." However, the reason for declaring a subclass isn't to generate synonyms, but to create something at least a little different from its superclass. You'd want to let the fiddle play bluegrass in addition to classical music.

Class Hierarchies

Any class can be used as a superclass for a new class definition. A class can simultaneously be a subclass of another class and a superclass for its own subclasses. Any number of classes can thus be linked in a hierarchy of inheritance, such as the one depicted in Figure 2-4.

Figure 2-4 Inheritance Hierarchy



Every inheritance hierarchy begins with a root class that has no superclass. From the root class, the hierarchy branches downward. Each class inherits from its superclass, and through its superclass, from all the classes above it in the hierarchy. Every class inherits from the root class.

Each class is the accumulation of all the class definitions in its inheritance chain. In the example above, class D inherits both from C, its superclass, and the root class. Members of the D class have methods and instance variables defined in all three classes—D, C, and root.

Typically, every class has just one superclass and can have an unlimited number of subclasses. However, in some object-oriented programming languages (though not in Objective-C), a class can have more than one superclass; it can inherit through multiple sources. Instead of a single hierarchy that branches downward as shown in Figure 2-4, multiple inheritance lets some branches of the hierarchy (or of different hierarchies) merge.

Subclass Definitions

A subclass can make three kinds of changes to the definition it inherits through its superclass:

- It can expand the class definition it inherits by adding new methods and instance variables. This is the most common reason for defining a subclass. Subclasses always add new methods, and new instance variables if the methods require it.
- It can modify the behavior it inherits by replacing an existing method with a new version. This is done by simply implementing a new method with the same name as one that's inherited. The new version *overrides* the inherited version. (The inherited method doesn't disappear; it's still valid for the class that defined it and other classes that inherit it.)
- It can refine or extend the behavior it inherits by replacing an existing method with a new version but still retain the old version by incorporating it in the new method. This is done by sending a message to perform the old version in the body of the new method. Each class in an inheritance chain can contribute part of a method's behavior. In Figure 2-4 (page 25), for example, class D might override a method defined in class C and incorporate C's version, while C's version incorporates a version defined in the root class.

Subclasses thus tend to fill out a superclass definition, making it more specific and specialized. They add, and sometimes replace, code rather than subtract it. Note that methods generally can't be disinherited and instance variables can't be removed or overridden.

Uses of Inheritance

The classic examples of an inheritance hierarchy are borrowed from animal and plant taxonomies. For example, there could be a class corresponding to the Pinaceae (pine) family of trees. Its subclasses could be Fir, Spruce, Pine, Hemlock, Tamarack, DouglasFir, and TrueCedar, corresponding to the various genera that make up the family. The Pine class might have SoftPine and HardPine subclasses, with WhitePine, SugarPine, and BristleconePine as subclasses of SoftPine, and PonderosaPine, JackPine, MontereyPine, and RedPine as subclasses of HardPine.

There's rarely a reason to program a taxonomy like this, but the analogy is a good one. Subclasses tend to specialize a superclass or adapt it to a special purpose, much as a species specializes a genus.

Here are some typical uses of inheritance:

- Reusing code. If two or more classes have some things in common but also differ in some ways, the common elements can be put in a single class definition that the other classes inherit. The common code is shared and need only be implemented once.

  For example, Faucet, Valve, and WaterPipe objects, defined for the program that models water use, all need a connection to a water source and should be able to record the rate of flow. These commonalities can be encoded once, in a class that the Faucet, Valve, and WaterPipe classes inherit from. A Faucet can be said to be a kind of Valve, so perhaps the Faucet class would inherit most of what it is from Valve, and add very little of its own.

- Setting up a protocol. A class can declare a number of methods that its subclasses are expected to implement. The class might have empty versions of the methods, or it might implement partial versions that are to be incorporated into the subclass methods. In either case, its declarations establish a **protocol** that all its subclasses must follow.

  When different classes implement similarly named methods, a program is better able to make use of polymorphism in its design. Setting up a protocol that subclasses must implement helps enforce these conventions.

- Delivering generic functionality. One implementor can define a class that contains a lot of basic, general code to solve a problem, but doesn't fill in all the details. Other implementors can then create subclasses to adapt the generic class to their specific needs. For example, the Appliance class in the program that models water use might define a generic water-using device that subclasses would turn into specific kinds of appliances.

  Inheritance is thus both a way to make someone else's programming task easier and a way to separate levels of implementation.

- Making slight modifications. When inheritance is used to deliver generic functionality, set up a protocol, or reuse code, a class is devised that other classes are expected to inherit from. But you can also use inheritance to modify classes that aren't intended as superclasses. Suppose, for example, that there's an object that would work well in your program, but you'd like to change one or two things that it does. You can make the changes in a subclass.

- Previewing possibilities. Subclasses can also be used to factor out alternatives for testing purposes. For example, if a class is to be encoded with a particular user interface, alternative interfaces can be factored into subclasses during the design phase of the project. Each alternative can then be demonstrated to potential users to see which they prefer. When the choice is made, the selected subclass can be reintegrated into its superclass.

Dynamism

At one time in programming history, the question of how much memory a program would use was settled when the source code was compiled and linked. All the memory the program would ever need was set aside for it as it was launched. This memory was fixed; it could neither grow nor shrink.

In hindsight, it's evident what a serious constraint this was. It limited not only how programs were constructed, but what you could imagine a program doing. It constrained design, not just programming technique. Functions (like `malloc`) that dynamically allocate memory as a program runs opened possibilities that didn't exist before.

Compile-time and link-time constraints are limiting because they force issues to be decided from information found in the programmer's source code, rather than from information obtained from the user as the program runs.

Although dynamic allocation removes one such constraint, many others, equally as limiting as static memory allocation, remain. For example, the elements that make up an application must be matched to data types at compile time. And the boundaries of an application are typically set at link time. Every part of the application must be united in a single executable file. New modules and new types can't be introduced as the program runs.

Objective-C seeks to overcome these limitations and to make programs as dynamic and fluid as possible. It shifts much of the burden of decision making from compile time and link time to runtime. The goal is to let program users decide what will happen, rather than constrain their actions artificially by the demands of the language and the needs of the compiler and linker.

Three kinds of dynamism are especially important for object-oriented design:

- Dynamic typing, waiting until runtime to determine the class of an object
- Dynamic binding, determining at runtime what method to invoke
- Dynamic loading, adding new components to a program as it runs

Dynamic Typing

The compiler typically complains if the code you write assigns a value to a type that can't accommodate it. You might see warnings like these:

```
incompatible types in assignment
assignment of integer from pointer lacks a cast
```

Type checking is useful, but there are times when it can interfere with the benefits you get from polymorphism, especially if the type of every object must be known to the compiler.

Suppose, for example, that you want to send an object a message to perform the `start` method. Like other data elements, the object is represented by a variable. If the variable's type (its class) must be known at compile time, it would be impossible to let runtime factors influence the decision about what kind of object should be assigned to the variable. If the class of the variable is fixed in source code, so is the version of `start` that the message invokes.

If, on the other hand, it's possible to wait until runtime to discover the class of the variable, any kind of object could be assigned to it. Depending on the class of the receiver, the `start` message might invoke different versions of the method and produce very different results.

Dynamic typing thus gives substance to dynamic binding (discussed next). But it does more than that. It permits associations between objects to be determined at runtime, rather than forcing them to be encoded in a static design. For example, a message could pass an object as an argument without declaring exactly what kind of object it is—that is, without declaring its class. The message receiver might then send its own messages to the object, again without ever caring about what kind of object it is. Because the receiver uses the object it's passed to do some of its work, it is in a sense customized by an object of indeterminate type (indeterminate in source code, that is, not at runtime).

Dynamic Binding

In standard C, you can declare a set of alternative functions, like the standard string-comparison functions,

```
int strcmp(const char *, const char *);   /* case sensitive */
int strcasecmp(const char *, const char *); /*case insensitive*/
```

and declare a pointer to a function that has the same return and argument types:

```
int (* compare)(const char *, const char *);
```

You can then wait until runtime to determine which function to assign to the pointer,

```
if ( **argv == 'i' )
    compare = strcasecmp;
else
    compare = strcmp;
```

and call the function through the pointer:

```
if ( compare(s1, s2) )
    ...
```

This is akin to what in object-oriented programming is called **dynamic binding**, delaying the decision of exactly which method to perform until the program is running.

Although not all object-oriented languages support it, dynamic binding can be routinely and transparently accomplished through messaging. You don't have to go through the indirection of declaring a pointer and assigning values to it as shown in the example above. You also don't have to assign each alternative procedure a different name.

Messages invoke methods indirectly. Every message expression must find a method implementation to "call." To find that method, the messaging machinery must check the class of the receiver and locate its implementation of the method named in the message. When this is done at runtime, the method is dynamically bound to the message. When it's done by the compiler, the method is statically bound.

Late Binding: Some object-oriented programming languages (notably C++) require a message receiver to be statically typed in source code, but don't require the type to be exact. An object can be typed to its own class or to any class that it inherits from.The compiler therefore can't tell whether the message receiver is an instance of the class specified in the type declaration, an instance of a subclass, or an instance of some more distantly derived class. Since it doesn't know the exact class of the receiver, it can't know which version of the method named in the message to invoke.In this circumstance, the choice is between treating the receiver as if it were an instance of the specified class and simply bind the method defined for that class to the message, or waiting until some later time to resolve the situation. In C++, the decision is postponed to link time for methods (member functions) that are declared **virtual**.This is sometimes referred to as "late binding" rather than "dynamic binding." While "dynamic" in the sense that it happens at runtime, it carries with it strict compile-time type constraints. As discussed here (and implemented in Objective-C), "dynamic binding" is unconstrained.

Dynamic binding is possible even in the absence of dynamic typing, but it's not very interesting. There's little benefit in waiting until runtime to match a method to a message when the class of the receiver is fixed and known to the compiler. The compiler could just as well find the method itself; the runtime result won't be any different.

However, if the class of the receiver is dynamically typed, there's no way for the compiler to determine which method to invoke. The method can be found only after the class of the receiver is resolved at runtime. Dynamic typing thus entails dynamic binding.

Dynamic typing also makes dynamic binding interesting, for it opens the possibility that a message might have very different results depending on the class of the receiver. Runtime factors can influence the choice of receiver and the outcome of the message.

Dynamic typing and binding also open the possibility that the code you write can send messages to objects not yet invented. If object types don't have to be decided until runtime, you can give others the freedom to design their own classes and name their own data types, and still have your code send messages to their objects. All you need to agree on are the messages, not the data types.

Note: Dynamic binding is routine in Objective-C. You don't need to arrange for it specially, so your design never needs to bother with what's being done when.

Dynamic Loading

The usual rule has been that, before a program can run, all its parts must be linked together in one file. When it's launched, the entire program is loaded into memory at once.

Some object-oriented programming environments overcome this constraint and allow different parts of an executable program to be kept in different files. The program can be launched in bits and pieces as they're needed. Each piece is dynamically loaded and linked with the rest of program as it's launched. User actions can determine which parts of the program are in memory and which aren't.

Only the core of a large program needs to be loaded at the start. Other modules can be added as the user requests their services. Modules the user doesn't request make no memory demands on the system.

Dynamic loading raises interesting possibilities. For example, an entire program wouldn't have to be developed at once. You could deliver your software in pieces and update one part of it at a time. You could devise a program that groups several tools under a single interface, and load just the tools the user wants. The program could even offer sets of alternative tools to do the same job. The user would select one tool from the set and only that tool would be loaded. It's not hard to imagine the possibilities. But because dynamic loading is relatively new, it's harder to predict its eventual benefits.

Perhaps the most important current benefit of dynamic loading is that it makes applications extensible. You can allow others to add to and customize a program you've designed. All your program needs to do is provide a framework that others can fill in, and, at runtime, find the pieces that they've implemented and load them dynamically.

For example, Interface Builder dynamically loads custom palettes and inspectors, and the Desktop dynamically loads inspectors for particular file formats. Anyone can design their own custom palettes and inspectors that these applications load and incorporate into themselves.

The main challenge that dynamic loading faces is getting a newly loaded part of a program to work with parts already running, especially when the different parts were written by different people. However, much of this problem disappears in an object-oriented environment because code is organized into logical modules with a clear division between implementation and interface. When classes are dynamically loaded, nothing in the newly loaded code can clash with the code already in place. Each class encapsulates its implementation and has an independent namespace.

In addition, dynamic typing and dynamic binding let classes designed by others fit effortlessly into the program you've designed. Once a class is dynamically loaded, it's treated no differently than any other class. Your code can send messages to their objects and theirs to yours. Neither of you has to know what classes the other has implemented. You need only agree on a communications protocol.

Loading and Linking: Although it's the term commonly used, "dynamic loading" could just as well be called "dynamic linking." Programs are linked when their various parts are joined so that they can work together; they're loaded when they're read into volatile memory at launch time. Linking usually precedes loading. Dynamic loading refers to the process of separately loading new or additional parts of a program and linking them dynamically to the parts already running.

Structuring Programs

Object-oriented programs have two kinds of structure. One can be seen in the inheritance hierarchy of class definitions. The other is evident in the pattern of message passing as the program runs. These messages reveal a network of object connections.

- The inheritance hierarchy explains how objects are related by type. For example, in the program that models water use, it might turn out that Faucets and WaterPipes are the same kind of object, except that Faucets can be turned on and off and WaterPipes can have multiple connections to other WaterPipes. This similarity would be captured in the program design if the Faucet and WaterPipe classes inherit from a common superclass.
- The network of object connections explains how the program works. For example, Appliance objects might send messages requesting water to Valves, and Valves to WaterPipes. WaterPipes might communicate with the Building object, and the Building object with all the Valves, Faucets, and WaterPipes, but not directly with Appliances. To communicate with each other in this way, objects must know about each other. An Appliance would need a connection to a Valve, and a Valve to a WaterPipe, and so on. These connections define a program structure.

Object-oriented programs are designed by laying out the network of objects with their behaviors and patterns of interaction and by arranging the hierarchy of classes. There's structure both in the program's activity and in its definition.

Outlet Connections

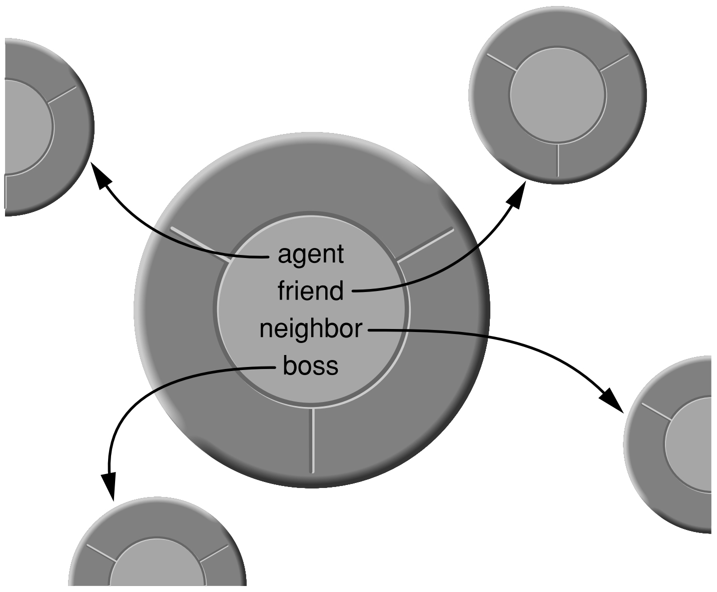
Part of the task of designing an object-oriented program is to arrange the object network. The network doesn't have to be static; it can change dynamically as the program runs. Relationships between objects can be improvised as needed, and the cast of objects that play assigned roles can change from time to time. But there has to be a script.

Some connections can be entirely transitory. A message might contain an argument identifying an object, perhaps the sender of the message, that the receiver can communicate with. As it responds to the message, the receiver can send messages to that object, perhaps identifying itself or still another object that the object can in turn communicate with. Such connections are fleeting; they last only as long as the chain of messages.

But not all connections between objects can be handled on the fly. Some need to be recorded in program data structures. There are various ways to do this. A table might be kept of object connections, or there might be a service that identifies objects by name. However, the simplest way is for each object to have instance variables that keep track of the other objects it must communicate with. These instance variables—termed **outlets** because they record the outlets for messages—define the principal connections between objects in the program network.

Although the names of outlet instance variables are arbitrary, they generally reflect the roles that outlet objects play. Figure 2-5 illustrates an object with four outlets—an “agent,” a “friend,” a “neighbor,” and a “boss.” The objects that play these parts may change every now and then, but the roles remain the same.

Figure 2-5 Outlets



Some outlets are set when the object is first initialized and may never change. Others might be set automatically as the consequence of other actions. Still others can be set freely, using methods provided just for that purpose.

However they’re set, outlet instance variables reveal the structure of the application. They link objects into a communicating network, much as the components of a water system are linked by their physical connections or as individuals are linked by their patterns of social relations.

Extrinsic and Intrinsic Connections

Outlet connections can capture many different kinds of relationships between objects. Sometimes the connection is between objects that communicate more or less as equal partners in an application, each with its own role to play and neither dominating the other. For example, an Appliance object might have an outlet instance variable to keep track of the Valve it’s connected to.

Sometimes one object should be seen as being part of another. For example, a Faucet might use a Meter object to measure the amount of water being released. The Meter would serve no other object and would act only under orders from the Faucet. It would be an intrinsic part of the Faucet, in contrast to an Appliance’s extrinsic connection to a Valve.

Similarly, an object that oversees other objects might keep a list of its charges. A Building object, for example, might have a list of all the WaterPipes in the program. The WaterPipes would be considered an intrinsic part of the Building and belong to it. WaterPipes, on the other hand, would maintain extrinsic connections to each other.

Intrinsic outlets behave differently from extrinsic ones. When an object is freed or archived in a file on disk, the objects that its intrinsic outlets point to must be freed or archived with it. For example, when a Faucet is freed, its Meter is rendered useless and therefore should be freed as well. A Faucet archived without its Meter would be of little use when it's unarchived (unless it could create a new Meter for itself).

Extrinsic outlets, on the other hand, capture the organization of the program at a higher level. They record connections between relatively independent program subcomponents. When an Appliance is freed, the Valve it was connected to still is of use and remains in place. When an Appliance is unarchived, it can be connected to another Valve and resume playing the same sort of role it played before.

Activating the Object Network

The object network is set into motion by an external stimulus. If you're writing an interactive application with a user interface, it will respond to user actions on the keyboard and mouse. A program that tries to factor very large numbers might start when you pass it a target number on the command line. Other programs might respond to data received over a phone line, information obtained from a database, or information about the state of a mechanical process the program monitors.

Programs often are activated by a flow of **events**, reports of external activity of some sort. Applications that display a user interface are driven by events from the keyboard and mouse. Every press of a key or click of the mouse generates events that the application receives and responds to. An object-oriented program structure (a network of objects that's prepared to respond to an external stimulus) is ideally suited for this kind of user-driven application.

Aggregation and Decomposition

Another part of the design task is deciding the arrangement of classes—when to add functionality to an existing class by defining a subclass and when to define an independent class. The problem can be clarified by imagining what would happen in the extreme case:

- It's possible to conceive of a program consisting of just one object. Since it's the only object, it can send messages only to itself. It therefore can't take advantage of polymorphism, or the modularity of a variety of classes, or a program design conceived as a network of interconnected objects. The true structure of the program would be hidden inside the class definition. Despite being written in an object-oriented language, there would be very little that was object-oriented about it.
- On the other hand, it's also possible to imagine a program that consists of hundreds of different kinds of objects, each with very few methods and limited functionality. Here, too, the structure of the program would be lost, this time in a maze of object connections.

Obviously, it's best to avoid either of these extremes, to keep objects large enough to take on a substantial role in the program but small enough to keep that role well-defined. The structure of the program should be easy to grasp in the pattern of object connections.

Nevertheless, the question often arises of whether to add more functionality to a class or to factor out the additional functionality and put it in an separate class definition. For example, a Faucet needs to keep track of how much water is being used over time. To do that, you could either implement the necessary methods in the Faucet class, or you could devise a generic Meter object to do the job, as suggested earlier. Each Faucet would have an outlet connecting it to a Meter, and the Meter would not interact with any object but the Faucet.

The choice often depends on your design goals. If the Meter object could be used in more than one situation, perhaps in another project entirely, it would increase the reusability of your code to factor the metering task into a separate class. If you have reason to make Faucet objects as self-contained as possible, the metering functionality could be added to the Faucet class.

It's generally better to try for reusable code and avoid having large classes that do so many things that they can't be adapted to other situations. When objects are designed as components, they become that much more reusable. What works in one system or configuration might well work in another.

Dividing functionality between different classes doesn't necessarily complicate the programming interface. If the Faucet class keeps the Meter object private, the Meter interface wouldn't have to be published for users of the Faucet class; the object would be as hidden as any other Faucet instance variable.

Models and Frameworks

Objects combine state and behavior, and so resemble things in the real world. Because they resemble real things, designing an object-oriented program is very much like thinking about real things—what they do, how they work, and how one thing is connected to another.

When you design an object-oriented program, you are, in effect, putting together a computer simulation of how something works. Object networks look and behave like models of real systems. An object-oriented program can be thought of as a model, even if there's no actual counterpart to it in the real world.

Each component of the model—each kind of object—is described in terms of its behavior, responsibilities, and interactions with other components. Because an object's interface lies in its methods, not its data, you can begin the design process by thinking about what a system component must do, not how it's represented in data. Once the behavior of an object is decided, the appropriate data structure can be chosen, but this is a matter of implementation, not the initial design.

For example, in the water-use program, you wouldn't begin by deciding what the Faucet data structure looked like, but what you wanted a Faucet to do—make a connection to a WaterPipe, be turned on and off, adjust the rate of flow, and so on. The design is therefore not bound from the outset by data choices. You can decide on the behavior first, and implement the data afterwards. Your choice of data structures can change over time without affecting the design.

Designing an object-oriented program doesn't necessarily entail writing great amounts of code. The reusability of class definitions means that the opportunity is great for building a program largely out of classes devised by others. It might even be possible to construct interesting programs entirely out of classes someone else defined. As the suite of class definitions grows, you have more and more reusable parts to choose from.

Reusable classes come from many sources. Development projects often yield reusable class definitions, and some enterprising developers market them. Object-oriented programming environments typically come with class libraries. There are well over two hundred classes in the Cocoa libraries. Some of these classes offer basic services (hashing, data storage, remote messaging). Others are more specific (user interface devices, video displays, sound).

Typically, a group of library classes work together to define a partial program structure. These classes constitute a software framework (or kit) that can be used to build a variety of different kinds of applications. When you use a framework, you accept the program model it provides and adapt your design to it. You use the framework by:

- Initializing and arranging instances of framework classes
- Defining subclasses of framework classes
- Defining new classes of your own to work with classes defined in the framework

In each of these ways, you not only adapt your program to the framework, but you also adapt the generic framework structure to the specialized purposes of your application.

The framework, in essence, sets up part of a object network for your program and provides part of its class hierarchy. Your own code completes the program model started by the framework.

Structuring the Programming Task

Object-oriented programming not only structures programs in a better way, it also helps structure the programming task.

As software tries to do more and more, and programs become bigger and more complicated, the problem of managing the task also grows. There are more pieces to fit together and more people working together to build them. The object-oriented approach offers ways of dealing with this complexity, not just in design, but also in the organization of the work.

Collaboration

Complex software requires an extraordinary collaborative effort among people who must be individually creative, yet still make what they do fit exactly with what others are doing.

The sheer size of the effort and the number of people working on the same project at the same time in the same place can get in the way of the group's ability to work cooperatively towards a common goal. In addition, collaboration is often impeded by barriers of time, space, and organization:

- Code must be maintained, improved, and used long after it's written. Programmers who collaborate on a project may not be working on it at the same time, so they may not be in a position to talk things over and keep each other informed about details of the implementation.
- Even if programmers work on the same project at the same time, they may not be located in the same place. This also inhibits how closely they can work together.
- Programmers working in different groups with different priorities and different schedules often must collaborate on projects. Communication across organizational barriers isn't always easy to achieve.

The answer to these difficulties must grow out of the way programs are designed and written. It can't be imposed from the outside in the form of hierarchical management structures and strict levels of authority. These often get in the way of people's creativity, and become burdens in and of themselves. Rather, collaboration must be built into the work itself.

That's where object-oriented programming techniques can help. For example, the reusability of object-oriented code means that programmers can collaborate effectively, even when they work on different projects at different times or are in different organizations, just by sharing their code in libraries. This kind of collaboration holds a great deal of promise, for it can conceivably lighten difficult tasks and make seemingly impossible projects achievable.

Organizing Object-Oriented Projects

Object-oriented programming helps restructure the programming task in ways that benefit collaboration. It helps eliminate the need to collaborate on low-level implementation details, while providing structures that facilitate collaboration at a higher level. Almost every feature of the object model, from the possibility of large-scale design to the increased reusability of code, has consequences for the way people work together.

Designing on a Large Scale

When programs are designed at a high level of abstraction, the division of labor is more easily conceived. It can match the division of the program on logical lines; the way a project is organized can grow out of its design.

With an object-oriented design, it's easier to keep common goals in sight, instead of losing them in the implementation, and easier for everyone to see how the piece they're working on fits into the whole. Their collaborative efforts are therefore more likely to be on target.

Separating the Interface from the Implementation

The connections between the various components of an object-oriented program are worked out early in the design process. They can be well-defined, at least for the initial phase of development, before implementation begins.

During implementation, only this interface needs to be coordinated, and most of that falls naturally out of the design. Since each class encapsulates its implementation and has its own name space, there's no need to coordinate implementation details. Collaboration is simpler when there are fewer coordination requirements.

Modularizing the Work

The modularity of object-oriented programming means that the logical components of a large program can each be implemented separately. Different people can work on different classes. Each implementation task is isolated from the others.

This has benefits, not just for organizing the implementation, but for fixing problems later. Since implementations are contained within class boundaries, problems that come up are also likely to be isolated. It's easier to track down bugs when they're located in a well-defined part of the program.

Separating responsibilities by class also means that each part can be worked on by specialists. Classes can be updated periodically to optimize their performance and make the best use of new technologies. These updates don't have to be coordinated with other parts of the program. As long as the interface to an object doesn't change, improvements to its implementation can be scheduled at any time.

Keeping the Interface Simple

The polymorphism of object-oriented programs yields simpler programming interfaces, since the same names and conventions can be reused in any number of classes. The result is less to learn, a greater shared understanding of how the whole system works, and a simpler path to cooperation and collaboration.

Making Decisions Dynamically

Because object-oriented programs make decisions dynamically at runtime, less information needs to be supplied at compile time (in source code) to make two pieces of code work together. Consequently, there's less to coordinate and less to go wrong.

Inheriting Generic Code

Inheritance is a way of reusing code. If you can define your classes as specializations of more generic classes, your programming task is simplified. The design is simplified as well, since the inheritance hierarchy lays out the relationships between the different levels of implementation and makes them easier to understand.

Inheritance also increases the reusability and reliability of code. The code placed in a superclass is tested by its subclasses. The generic class you find in a library will have been tested by other subclasses written by other developers for other applications.

Reusing Tested Code

The more software you can borrow from others and incorporate in your own programs, the less you have to do yourself. There's more software to borrow in an object-oriented programming environment because the code is more reusable. Collaboration between programmers working in different places for different organizations is enhanced, while the burden of each project is eased.

Classes and frameworks from an object-oriented library can make substantial contributions to your program. When you program with the software frameworks provided by Apple, for example, you're effectively collaborating with the programmers at Apple; you're contracting a part of your program, often a substantial part, to them. You can concentrate on what you do best and leave other tasks to the library developer. Your projects can be prototyped faster, completed faster, with less of a collaborative challenge at your own site.

The increased reusability of object-oriented code also increases its reliability. A class taken from a library is likely to have found its way into a variety of applications and situations. The more the code has been used, the more likely it is that problems will have been encountered and fixed. Bugs that would have seemed strange and hard to find in your program might already have been tracked down and eliminated.

The Language

This chapter describes the Objective-C language and discusses the principles of object-oriented programming as they're implemented in Objective-C. It covers all the features that the language adds to standard C and C++.

Because object-oriented programs postpone many decisions from compile time to runtime, object-oriented languages depend on a runtime system for executing the compiled code. The runtime system for the Objective-C language is discussed in "The Runtime System" (page 95). This chapter presents the language, but touches on important elements of the runtime system.

The Apple compilers are based on the compilers of the GNU Compiler Collection. Objective-C syntax is a superset of GNU C/C++ syntax, and the Objective-C compiler works for C, C++ and Objective-C source code. The compiler recognizes Objective-C source files by the filename extension `.m`, just as it recognizes files containing only standard C syntax by filename extension `.c`. Similarly, the compiler recognizes C++ files that use Objective-C by the extension `.mm`. Other issues when using Objective-C with C++ are covered in the section "Using C++ With Objective-C" (page 89).

Objects

As the name implies, object-oriented programs are built around **objects**. An object associates data with the particular operations that can use or affect that data. In Objective-C, these operations are known as the object's **methods**; the data they affect are its **instance variables**. In essence, an object bundles a data structure (instance variables) and a group of procedures (methods) into a self-contained programming unit.

For example, if you are writing a drawing program that allows a user to create images composed of lines, circles, rectangles, text, bit-mapped images, and so forth, you might create classes for many of the basic shapes that a user can manipulate. A Rectangle object, for instance, might have instance variables that identify the position of the rectangle within the drawing along with its width and its height. Other instance variables could define the rectangle's color, whether or not it is to be filled, and a line pattern that should be used to display the rectangle. A Rectangle class would have methods to set an instance's position, size, color, fill status, and line pattern, along with a method that causes the instance to display itself.

In Objective-C, an object's instance variables are internal to the object; generally, you get access to an object's state only through the object's methods (you can specify whether subclasses or other objects can access instance variables directly by using scope directives, see "The Scope of Instance

Variables" (page 59)). For others to find out something about an object, there has to be a method to supply the information. For example, a Rectangle would have methods that reveal its size and its position.

Moreover, an object sees only the methods that were designed for it; it can't mistakenly perform methods intended for other types of objects. Just as a C function protects its local variables, hiding them from the rest of the program, an object hides both its instance variables and its method implementations.

id

In Objective-C, object identifiers are a distinct data type: `id`. This type is defined as a pointer to an object—in reality, a pointer to the instance variables of the object, the object's unique data. Like a C function or an array, an object is identified by its address. All objects, regardless of their instance variables or methods, are of type `id`.

```
id anObject;
```

For the object-oriented constructs of Objective-C, such as method return values, `id` replaces `int` as the default data type. (For strictly C constructs, such as function return values, `int` remains the default type.)

The keyword `nil` is defined as a null object, an `id` with a value of `0`. `id`, `nil`, and the other basic types of Objective-C are defined in the header file `objc/objc.h`.

Dynamic Typing

The `id` type is completely nonrestrictive. By itself, it yields no information about an object, except that it is an object.

But objects aren't all the same. A Rectangle won't have the same methods or instance variables as an object that represents a bit-mapped image. At some point, a program needs to find more specific information about the objects it contains—what the object's instance variables are, what methods it can perform, and so on. Since the `id` type designator can't supply this information to the compiler, each object has to be able to supply it at runtime.

This is possible because every object carries with it an `isa` instance variable that identifies the object's **class**—what kind of object it is. Every Rectangle object would be able to tell the runtime system that it is a Rectangle. Every Circle can say that it is a Circle. Objects with the same behavior (methods) and the same kinds of data (instance variables) are members of the same class.

Objects are thus **dynamically typed** at runtime. Whenever it needs to, the runtime system can find the exact class that an object belongs to, just by asking the object. Dynamic typing in Objective-C serves as the foundation for dynamic binding, discussed later.

The `isa` pointer also enables objects to perform **introspection**—to find out about themselves (or other objects). The compiler records information about class definitions in data structures for the runtime system to use. The functions of the runtime system use `isa`, to find this information at runtime. Using the runtime system, you can, for example, determine whether an object implements a particular method, or discover the name of its superclass.

Object classes are discussed in more detail under "Classes" (page 43).

It’s also possible to give the compiler information about the class of an object by statically typing it in source code using the class name. Classes are particular kinds of objects, and the class name can serve as a type name. See "Class Types" (page 47) and "Enabling Static Behaviors" (page 81).

Object Messaging

To get an object to do something, you send it a **message** telling it to apply a method. In Objective-C, **message expressions** are enclosed in brackets:

```
[receiver message]
```

The receiver is an object, and the message tells it what to do. In source code, the message is simply the name of a method and any arguments that are passed to it. When a message is sent, the runtime system selects the appropriate method from the receiver’s repertoire and invokes it.

For example, this message tells the `myRect` object to perform its `display` method, which causes the rectangle to display itself:

```
[myRect display];
```

Methods can also take arguments. The imaginary message below tells `myRect` to set its location within the window to coordinates (30.0, 50.0):

```
[myRect setOrigin:30.0 :50.0];
```

Here the method name, `setOrigin::`, has two colons, one for each of its arguments. The arguments are inserted after the colons. This method name uses unlabeled arguments. Unlabeled arguments make it difficult to determine the kind and purpose of a method’s arguments. Instead, method names should include labels describing each of their arguments. Argument labels precede each colon in the method name. The `setWidth:height:` method, for example, makes the purpose of its two arguments clear:

```
[myRect setWidth:10.0 height:15.0];
```

Methods that take a variable number of arguments are also possible, though they’re somewhat rare. Extra arguments are separated by commas after the end of the method name. (Unlike colons, the commas aren’t considered part of the name.) In the following example, the imaginary `makeGroup:` method is passed one required argument (**group**) and three that are optional:

```
[receiver makeGroup:group, memberOne, memberTwo, memberThree];
```

Like standard C functions, methods can return values. The following example sets the variable `isFilled` to `YES` if `myRect` is drawn as a solid rectangle, or `NO` if it’s drawn in outline form only.

```
BOOL isFilled;
isFilled = [myRect isFilled];
```

Note that a variable and a method can have the same name.

One message expression can be nested inside another. Here one rectangle is set to the color of another:

```
[myRect setPrimaryColor:[otherRect primaryColor]];
```

A message to `nil` also is valid, as long as the message returns an object, any pointer type, or any integer scalar of size less than or equal to `sizeof(void*)`; if it does, a message sent to `nil` returns `nil`. If the message sent to `nil` returns anything other than the forementioned value types (for example, if it returns any struct type, any floating-point type, or any vector type) the return value is undefined. In general, however, it is considered bad style to rely on this behavior for return types other than an object.

The Receiver's Instance Variables

A method has automatic access to the receiving object's instance variables. You don't need to pass them to the method as arguments. For example, the `primaryColor` method illustrated above takes no arguments, yet it can find the primary color for `otherRect` and return it. Every method assumes the receiver and its instance variables, without having to declare them as arguments.

This convention simplifies Objective-C source code. It also supports the way object-oriented programmers think about objects and messages. Messages are sent to receivers much as letters are delivered to your home. Message arguments bring information from the outside to the receiver; they don't need to bring the receiver to itself.

A method has automatic access only to the receiver's instance variables. If it requires information about a variable stored in another object, it must send a message to the object asking it to reveal the contents of the variable. The `primaryColor` and `isFilled` methods shown above are used for just this purpose.

See "Defining a Class" (page 54) for more information on referring to instance variables.

Polymorphism

As the examples above illustrate, messages in Objective-C appear in the same syntactic positions as function calls in standard C. But, because methods "belong to" an object, messages behave differently than function calls.

In particular, an object can be operated on by only those methods that were defined for it. It can't confuse them with methods defined for other kinds of objects, even if another object has a method with the same name. This means that two objects can respond differently to the same message. For example, each kind of object sent a `display` message could display itself in a unique way. A Circle and a Rectangle would respond differently to identical instructions to track the cursor.

This feature, referred to as **polymorphism**, plays a significant role in the design of object-oriented programs. Together with dynamic binding, it permits you to write code that might apply to any number of different kinds of objects, without you having to choose at the time you write the code what kinds of objects they might be. They might even be objects that will be developed later, by other programmers working on other projects. If you write code that sends a `display` message to an `id` variable, any object that has a `display` method is a potential receiver.

Dynamic Binding

A crucial difference between function calls and messages is that a function and its arguments are joined together in the compiled code, but a message and a receiving object aren't united until the program is running and the message is sent. Therefore, the exact method that's invoked to respond to a message can only be determined at runtime, not when the code is compiled.

The precise method that a message invokes depends on the receiver. Different receivers may have different method implementations for the same method name (polymorphism). For the compiler to find the right method implementation for a message, it would have to know what kind of object the receiver is—what class it belongs to. This is information the receiver is able to reveal at runtime when it receives a message (dynamic typing), but it's not available from the type declarations found in source code.

The selection of a method implementation happens at runtime. When a message is sent, a runtime messaging routine looks at the receiver and at the method named in the message. It locates the receiver's implementation of a method matching the name, "calls" the method, and passes it a pointer to the receiver's instance variables. (For more on this routine, see "How Messaging Works" (page 61).)

The method name in a message thus serves to "select" a method implementation. For this reason, method names in messages are often referred to as **selectors**.

This **dynamic binding** of methods to messages works hand-in-hand with polymorphism to give object-oriented programming much of its flexibility and power. Since each object can have its own version of a method, a program can achieve a variety of results, not by varying the message itself, but by varying just the object that receives the message. This can be done as the program runs; receivers can be decided "on the fly" and can be made dependent on external factors such as user actions.

When executing code based upon the Application Kit, for example, users determine which objects receive messages from menu commands like Cut, Copy, and Paste. The message goes to whatever object controls the current selection. An object that displays text would react to a `copy` message differently from an object that displays scanned images. An object that represents a set of shapes would respond differently from a Rectangle. Since messages don't select methods (methods aren't bound to messages) until runtime, these differences are isolated in the methods that respond to the message. The code that sends the message doesn't have to be concerned with them; it doesn't even have to enumerate the possibilities. Each application can invent its own objects that respond in their own way to `copy` messages.

Objective-C takes dynamic binding one step further and allows even the message that's sent (the method selector) to be a variable that's determined at runtime. This is discussed in the section "How Messaging Works" (page 61).

Classes

An object-oriented program is typically built from a variety of objects. A program based on the Cocoa frameworks might use NSMatrix objects, NSWindow objects, NSDictionary objects, NSFont objects, NSText objects, and many others. Programs often use more than one object of the same kind or class—several NSArray objects or NSWindow objects, for example.

In Objective-C, you define objects by defining their class. The class definition is a prototype for a kind of object; it declares the instance variables that become part of every member of the class, and it defines a set of methods that all objects in the class can use.

The compiler creates just one accessible object for each class, a **class object** that knows how to build new objects belonging to the class. (For this reason it's traditionally called a "factory object.") The class object is the compiled version of the class; the objects it builds are **instances** of the class. The objects that do the main work of your program are instances created by the class object at runtime.

All instances of a class have the same set of methods, and they all have a set of instance variables cut from the same mold. Each object gets its own instance variables, but the methods are shared.

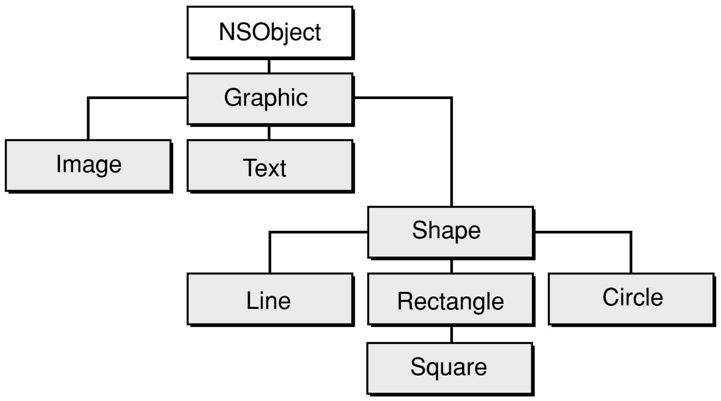
By convention, class names begin with an uppercase letter (such as "Rectangle"); the names of instances typically begin with a lowercase letter (such as "myRect").

Inheritance

Class definitions are additive; each new class that you define is based on another class from which it **inherits** methods and instance variables. The new class simply adds to or modifies what it inherits. It doesn't need to duplicate inherited code.

Inheritance links all classes together in a hierarchical tree with a single class at its root. When writing code that is based upon the Foundation framework, that root class is typically NSObject. Every class (except a root class) has a **superclass** one step nearer the root, and any class (including a root class) can be the superclass for any number of **subclasses** one step farther from the root. Figure 3-1 illustrates the hierarchy for a few of the classes used in the drawing program.

Figure 3-1 Some Drawing Program Classes



This figure shows that the Square class is a subclass of the Rectangle class, the Rectangle class is a subclass of Shape, Shape is a subclass of Graphic, and Graphic is a subclass of NSObject. Inheritance is cumulative. So a Square object has the methods and instance variables defined for Rectangle, Shape, Graphic, and NSObject, as well as those defined specifically for Square. This is simply to say that a Square object isn't only a Square, it's also a Rectangle, a Shape, a Graphic, and an NSObject.

Every class but NSObject can thus be seen as a specialization or an adaptation of another class. Each successive subclass further modifies the cumulative total of what's inherited. The Square class defines only the minimum needed to turn a Rectangle into a Square.

When you define a class, you link it to the hierarchy by declaring its superclass; every class you create must be the subclass of another class (unless you define a new root class). Plenty of potential superclasses are available. Cocoa includes the NSObject class and several frameworks containing definitions for more than 250 additional classes. Some are classes that you can use "off the shelf"—incorporate into your program as is. Others you might want to adapt to your own needs by defining a subclass.

Some framework classes define almost everything you need, but leave some specifics to be implemented in a subclass. You can thus create very sophisticated objects by writing only a small amount of code, and reusing work done by the programmers of the framework.

The NSObject Class

NSObject, being a root class, doesn't have a superclass. In Cocoa, it's in the inheritance path for all other classes. That's because it defines the basic framework for Objective-C objects and object interactions. It imparts to the classes and instances of classes that inherit from it the ability to behave as objects and cooperate with the runtime system.

A class that doesn't need to inherit any special behavior from another class should nevertheless be made a subclass of the NSObject class. Instances of the class must at least have the ability to behave like Objective-C objects at runtime. Inheriting this ability from the NSObject class is much simpler and much more reliable than reinventing it in a new class definition.

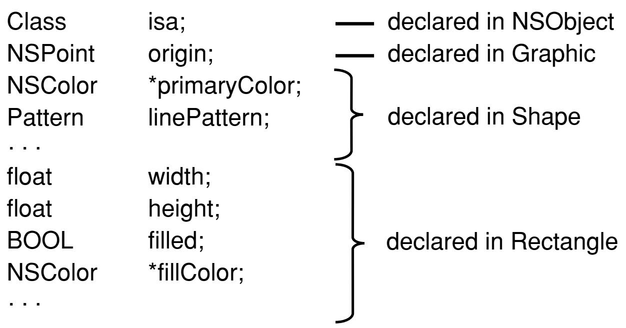
Note: Implementing a new root class is a delicate task and one with many hidden hazards. The class must duplicate much of what the NSObject class does, such as allocate instances, connect them to their class, and identify them to the runtime system. For this reason, you should generally use the NSObject class provided with Cocoa as the root class. For more information, see the Foundation framework documentation for the NSObject class and the NSObject protocol.

Inheriting Instance Variables

When a class object creates a new instance, the new object contains not only the instance variables that were defined for its class but also the instance variables defined for its superclass and for its superclass's superclass, all the way back to the root class. Thus, the `isa` instance variable defined in the NSObject class becomes part of every object. `isa` connects each object to its class.

Figure 3-2 shows some of the instance variables that could be defined for a particular implementation of Rectangle, and where they may come from. Note that the variables that make the object a Rectangle are added to the ones that make it a Shape, and the ones that make it a Shape are added to the ones that make it a Graphic, and so on.

Figure 3-2 Rectangle Instance Variables



```
Class      isa;             — declared in NSObject
NSPoint    origin;          — declared in Graphic
NSColor    *primaryColor;
Pattern    linePattern;     } declared in Shape
. . .
float      width;
float      height;
BOOL       filled;          } declared in Rectangle
NSColor    *fillColor;
. . .
```

A class doesn't have to declare instance variables. It can simply define new methods and rely on the instance variables it inherits, if it needs any instance variables at all. For example, Square might not declare any new instance variables of its own.

Inheriting Methods

An object has access not only to the methods defined for its class, but also to methods defined for its superclass, and for its superclass's superclass, all the way back to the root of the hierarchy. For instance, a Square object can use methods defined in the Rectangle, Shape, Graphic, and NSObject classes as well as methods defined in its own class.

Any new class you define in your program can therefore make use of the code written for all the classes above it in the hierarchy. This type of inheritance is a major benefit of object-oriented programming. When you use one of the object-oriented frameworks provided by Cocoa, your programs can take advantage of the basic functionality coded into the framework classes. You have to add only the code that customizes the standard functionality to your application.

Class objects also inherit from the classes above them in the hierarchy. But because they don't have instance variables (only instances do), they inherit only methods.

Overriding One Method With Another

There's one useful exception to inheritance: When you define a new class, you can implement a new method with the same name as one defined in a class farther up the hierarchy. The new method overrides the original; instances of the new class perform it rather than the original, and subclasses of the new class inherit it rather than the original.

For example, Graphic defines a `display` method that Rectangle overrides by defining its own version of `display`. The Graphic method is available to all kinds of objects that inherit from the Graphic class—but not to Rectangle objects, which instead perform the Rectangle version of `display`.

Although overriding a method blocks the original version from being inherited, other methods defined in the new class can skip over the redefined method and find the original (see "Messages to self and super" (page 67) to learn how).

A redefined method can also incorporate the very method it overrides. When it does, the new method serves only to refine or modify the method it overrides, rather than replace it outright. When several classes in the hierarchy define the same method, but each new version incorporates the version it overrides, the implementation of the method is effectively spread over all the classes.

Although a subclass can override inherited methods, it can't override inherited instance variables. Since an object has memory allocated for every instance variable it inherits, you can't override an inherited variable by declaring a new one with the same name. If you try, the compiler will complain.

Abstract Classes

Some classes are designed only so that other classes can inherit from them. These **abstract classes** group methods and instance variables that can be used by a number of different subclasses into a common definition. The abstract class is incomplete by itself, but contains useful code that reduces the implementation burden of its subclasses.

The NSObject class is the prime example of an abstract class. Although programs often define NSObject subclasses and use instances belonging to the subclasses, they never use instances belonging directly to the NSObject class. An NSObject instance wouldn't be good for anything; it would be a generic object with the ability to do nothing in particular.

Abstract classes often contain code that helps define the structure of an application. When you create subclasses of these classes, instances of your new classes fit effortlessly into the application structure and work automatically with other objects.

(Because abstract classes must have subclasses to be useful, they're sometimes also called **abstract superclasses**.)

Class Types

A class definition is a specification for a kind of object. The class, in effect, defines a data type. The type is based not just on the data structure the class defines (instance variables), but also on the behavior included in the definition (methods).

A class name can appear in source code wherever a type specifier is permitted in C—for example, as an argument to the `sizeof` operator:

```
int i = sizeof(Rectangle);
```

Static Typing

You can use a class name in place of `id` to designate an object's type:

```
Rectangle *myRect;
```

Because this way of declaring an object type gives the compiler information about the kind of object it is, it's known as **static typing**. Just as `id` is defined as a pointer to an object, objects are statically typed as pointers to a class. Objects are always typed by a pointer. Static typing makes the pointer explicit; `id` hides it.

Static typing permits the compiler to do some type checking—for example, to warn if an object could receive a message that it appears not to be able to respond to—and to loosen some restrictions that apply to objects generically typed `id`. In addition, it can make your intentions clearer to others who read your source code. However, it doesn't defeat dynamic binding or alter the dynamic determination of a receiver's class at runtime.

An object can be statically typed to its own class or to any class that it inherits from. For example, since inheritance makes a Rectangle a kind of Graphic, a Rectangle instance could be statically typed to the Graphic class:

```
Graphic *myRect;
```

This is possible because a Rectangle is a Graphic. It's more than a Graphic since it also has the instance variables and method capabilities of a Shape and a Rectangle, but it's a Graphic nonetheless. For purposes of type checking, the compiler considers `myRect` to be a Graphic, but at runtime it's treated as a Rectangle.

See "Enabling Static Behaviors" (page 81) in the next chapter for more on static typing and its benefits.

Type Introspection

Instances can reveal their types at runtime. The `isMemberOfClass:` method, defined in the NSObject class, checks whether the receiver is an instance of a particular class:

```
if ( [anObject isMemberOfClass:someClass] )
    ...
```

The `isKindOfClass:` method, also defined in the NSObject class, checks more generally whether the receiver inherits from or is a member of a particular class (whether it has the class in its inheritance path):

```
if ( [anObject isKindOfClass:someClass] )
    ...
```

The set of classes for which `isKindOfClass:` returns `YES` is the same set to which the receiver can be statically typed.

Introspection isn't limited to type information. Later sections of this chapter discuss methods that return the class object, report whether an object can respond to a message, and reveal other information.

See the NSObject class specification in the Foundation framework reference for more on `isKindOfClass:`, `isMemberOfClass:`, and related methods.

Class Objects

A class definition contains various kinds of information, much of it about instances of the class:

- The name of the class and its superclass
- A template describing a set of instance variables
- The declarations of method names and their return and argument types
- The method implementations

This information is compiled and recorded in data structures made available to the runtime system. The compiler creates just one object, a **class object**, to represent the class. The class object has access to all the information about the class, which means mainly information about what instances of the class are like. It's able to produce new instances according to the plan put forward in the class definition.

Although a class object keeps the prototype of a class instance, it's not an instance itself. It has no instance variables of its own and it can't perform methods intended for instances of the class. However, a class definition can include methods intended specifically for the class object—**class methods** as opposed to **instance methods**. A class object inherits class methods from the classes above it in the hierarchy, just as instances inherit instance methods.

In source code, the class object is represented by the class name. In the following example, the Rectangle class returns the class version number using a method inherited from the NSObject class:

```
int versionNumber = [Rectangle version];
```

However, the class name stands for the class object only as the receiver in a message expression. Elsewhere, you need to ask an instance or the class to return the class `id`. Both respond to a `class` message:

```
id aClass = [anObject class];
id rectClass = [Rectangle class];
```

As these examples show, class objects can, like all other objects, be typed `id`. But class objects can also be more specifically typed to the Class data type:

```
Class aClass = [anObject class];
Class rectClass = [Rectangle class];
```

All class objects are of type Class. Using this type name for a class is equivalent to using the class name to statically type an instance.

Class objects are thus full-fledged objects that can be dynamically typed, receive messages, and inherit methods from other classes. They're special only in that they're created by the compiler, lack data structures (instance variables) of their own other than those built from the class definition, and are the agents for producing instances at runtime.

Note: The compiler also builds a "metaclass object" for each class. It describes the class object just as the class object describes instances of the class. But while you can send messages to instances and to the class object, the metaclass object is used only internally by the runtime system.

Creating Instances

A principal function of a class object is to create new instances. This code tells the Rectangle class to create a new Rectangle instance and assign it to the `myRect` variable:

```
id  myRect;
myRect = [Rectangle alloc];
```

The `alloc` method dynamically allocates memory for the new object's instance variables and initializes them all to `0`—all, that is, except the `isa` variable that connects the new instance to its class. For an object to be useful, it generally needs to be more completely initialized. That's the function of an `init` method. Initialization typically follows immediately after allocation:

```
myRect = [[Rectangle alloc] init];
```

This line of code, or one like it, would be necessary before `myRect` could receive any of the messages that were illustrated in previous examples in this chapter. The `alloc` method returns a new instance and that instance performs an `init` method to set its initial state. Every class object has at least one method (like `alloc`) that enables it to produce new objects, and every instance has at least one method (like `init`) that prepares it for use. Initialization methods often take arguments to allow particular values to be passed and have keywords to label the arguments (`initWithPosition:size:`, for example, is a method that might initialize a new Rectangle instance), but they all begin with "init".

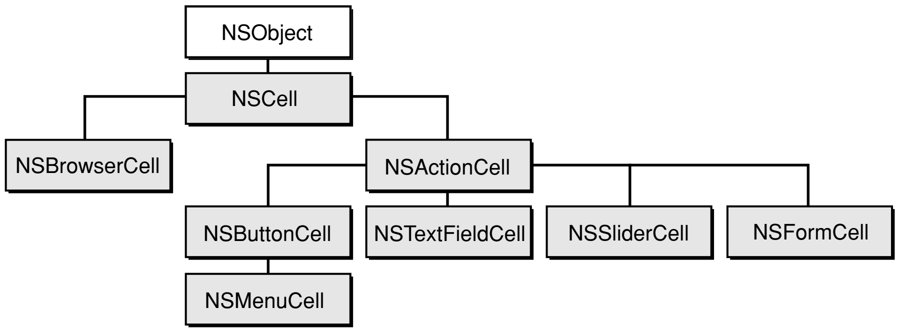
Customization With Class Objects

It's not just a whim of the Objective-C language that classes are treated as objects. It's a choice that has intended, and sometimes surprising, benefits for design. It's possible, for example, to customize an object with a class, where the class belongs to an open-ended set. In the Application Kit, for example, an NSMatrix object can be customized with a particular kind of NSCell.

An NSMatrix can take responsibility for creating the individual objects that represent its cells. It can do this when the NSMatrix is first initialized and later when new cells are needed. The visible matrix that an NSMatrix object draws on the screen can grow and shrink at runtime, perhaps in response to user actions. When it grows, the NSMatrix needs to be able to produce new objects to fill the new slots that are added.

But what kind of objects should they be? Each NSMatrix displays just one kind of NSCell, but there are many different kinds. The inheritance hierarchy in Figure 3-3 shows some of those provided by the Application Kit. All inherit from the generic NSCell class:

Figure 3-3 Inheritance hierarchy for NSCells



When an NSMatrix creates NSCell objects, should they be NSButtonCell objects to display a bank of buttons or switches, NSTextFieldCell objects to display fields where the user can enter and edit text, or some other kind of NSCell? The NSMatrix must allow for any kind of cell, even types that haven't been invented yet.

One solution to this problem is to define the NSMatrix class as an abstract class and require everyone who uses it to declare a subclass and implement the methods that produce new cells. Because they would be implementing the methods, users of the class could be sure that the objects they created were of the right type.

But this requires others to do work that ought to be done in the NSMatrix class, and it unnecessarily proliferates the number of classes. Since an application might need more than one kind of NSMatrix, each with a different kind of NSCell, it could become cluttered with NSMatrix subclasses. Every time you invented a new kind of NSCell, you'd also have to define a new kind of NSMatrix. Moreover, programmers on different projects would be writing virtually identical code to do the same job, all to make up for NSMatrix's failure to do it.

A better solution, the solution the NSMatrix class actually adopts, is to allow NSMatrix instances to be initialized with a kind of NSCell—with a class object. It defines a `setCellClass:` method that passes the class object for the kind of NSCell object an NSMatrix should use to fill empty slots:

```
[myMatrix setCellClass:[NSButtonCell class]];
```

The NSMatrix uses the class object to produce new cells when it's first initialized and whenever it's resized to contain more cells. This kind of customization would be difficult if classes weren't objects that could be passed in messages and assigned to variables.

Variables and Class Objects

When you define a new class, you can specify instance variables. Every instance of the class can maintain its own copy of the variables you declare—each object controls its own data. There is, however, no "class variable" counterpart to an instance variable. Only internal data structures, initialized from the class definition, are provided for the class. Moreover, a class object has no access to the instance variables of any instances; it can't initialize, read, or alter them.

For all the instances of a class to share data, you must define an external variable of some sort. The simplest way to do this is to declare a variable in the class implementation file as illustrated in the following code fragment.

```
int MCLSGlobalVariable;

@implementation MyClass
// implementation continues
```

In a more sophisticated implementation, you can declare a variable to be `static`, and provide class methods to manage it. Declaring a variable `static` limits its scope to just the class—and to just the part of the class that's implemented in the file. (Thus unlike instance variables, static variables cannot be inherited by, or directly manipulated by, subclasses.) This pattern is commonly used to define shared instances of a class (such as singletons, see Creating a Singleton Instance).

```
static MyClass *MCLSSharedInstance;

@implementation MyClass

+ (MyClass *)sharedInstance
{
    // check for existence of shared instance
    // create if necessary
    return MCLSSharedInstance;
}
// implementation continues
```

Static variables help give the class object more functionality than just that of a "factory" producing instances; it can approach being a complete and versatile object in its own right. A class object can be used to coordinate the instances it creates, dispense instances from lists of objects already created, or manage other processes essential to the application. In the case when you need only one object of a particular class, you can put all the object's state into static variables and use only class methods. This saves the step of allocating and initializing an instance.

Note: It is also possible to use external variables that are not declared `static`, but the limited scope of static variables better serves the purpose of encapsulating data into separate objects.

Initializing a Class Object

If a class object is to be used for anything besides allocating instances, it may need to be initialized just as an instance is. Although programs don't allocate class objects, Objective-C does provide a way for programs to initialize them.

If a class makes use of static or global variables, the `initialize` method is a good place to set their initial values. For example, if a class maintains an array of instances, the `initialize` method could set up the array and even allocate one or two default instances to have them ready.

The runtime system sends an `initialize` message to every class object before the class receives any other messages and after its superclass has received the `initialize` message. This gives the class a chance to set up its runtime environment before it's used. If no initialization is required, you don't need to write an `initialize` method to respond to the message.

Because of inheritance, an `initialize` message sent to a class that doesn't implement the `initialize` method is forwarded to the superclass, even though the superclass has already received the `initialize` message. For example, assume class A implements the `initialize` method, and class B inherits from class A but does not implement the `initialize` method. Just before class B is to receive its first message, the runtime system sends `initialize` to it. But, because class B doesn't implement `initialize`, class A's `initialize` is executed instead. Therefore, class A should ensure that its initialization logic is performed only once.

To avoid performing initialization logic more than once, use the template in Listing 3-1 when implementing the `initialize` method.

Listing 3-1 Implementation of the initialize method

```
+ (void)initialize
{
    static BOOL initialized = NO;
    if (!initialized) {
        // Perform initialization here.
        ...
        initialized = YES;
    }
}
```

Note: Remember that the runtime system sends `initialize` to each class individually. Therefore, in a class's implementation of the `initialize` method, you must not send the `initialize` message to its superclass.

Methods of the Root Class

All objects, classes and instances alike, need an interface to the runtime system. Both class objects and instances should be able to introspect about their abilities and to report their place in the inheritance hierarchy. It's the province of the NSObject class to provide this interface.

So that NSObject's methods don't have to be implemented twice—once to provide a runtime interface for instances and again to duplicate that interface for class objects—class objects are given special dispensation to perform instance methods defined in the root class. When a class object receives a message that it can't respond to with a class method, the runtime system determines whether there's a root instance method that can respond. The only instance methods that a class object can perform are those defined in the root class, and only if there's no class method that can do the job.

For more on this peculiar ability of class objects to perform root instance methods, see the NSObject class specification in the Foundation framework reference.

Class Names in Source Code

In source code, class names can be used in only two very different contexts. These contexts reflect the dual role of a class as a data type and as an object:

- The class name can be used as a type name for a kind of object. For example:

  ```
  Rectangle * anObject;
  ```

  Here `anObject` is statically typed to be a pointer to a Rectangle. The compiler expects it to have the data structure of a Rectangle instance and the instance methods defined and inherited by the Rectangle class. Static typing enables the compiler to do better type checking and makes source code more self-documenting. See "Enabling Static Behaviors" (page 81) for details.

  Only instances can be statically typed; class objects can't be, since they aren't members of a class, but rather belong to the Class data type.

- As the receiver in a message expression, the class name refers to the class object. This usage was illustrated in several of the earlier examples. The class name can stand for the class object only as a message receiver. In any other context, you must ask the class object to reveal its `id` (by sending it a class message). The example below passes the Rectangle class as an argument in an `isKindOfClass:` message.

  ```
  if ( [anObject isKindOfClass:[Rectangle class]] )
      ...
  ```

  It would have been illegal to simply use the name "Rectangle" as the argument. The class name can only be a receiver.

  If you don't know the class name at compile time but have it as a string at runtime, `NSClassFromString` will return the class object:

  ```
  NSString *className;
      ...
  if ( [anObject isKindOfClass:NSClassFromString(className)] )
      ...
  ```

  This function returns `nil` if the string it's passed is not a valid class name.

Classnames exist in the same namespace as global variables and function names. A class and a global variable can't have the same name. Classnames are about the only names with global visibility in Objective-C.

Defining a Class

Much of object-oriented programming consists of writing the code for new objects—defining new classes. In Objective-C, classes are defined in two parts:

- An **interface** that declares the methods and instance variables of the class and names its superclass
- An **implementation** that actually defines the class (contains the code that implements its methods)

Although the compiler doesn't require it, the interface and implementation are usually separated into two different files. The interface file must be made available to anyone who uses the class.

A single file can declare or implement more than one class. Nevertheless, it's customary to have a separate interface file for each class, if not also a separate implementation file. Keeping class interfaces separate better reflects their status as independent entities.

Interface and implementation files typically are named after the class. The name of the implementation file has the `.m` extension, indicating that it contains Objective-C source code. The interface file can be assigned any other extension. Because it's included in other source files, the name of the interface file usually has the `.h` extension typical of header files. For example, the Rectangle class would be declared in `Rectangle.h` and defined in `Rectangle.m`.

Separating an object's interface from its implementation fits well with the design of object-oriented programs. An object is a self-contained entity that can be viewed from the outside almost as a "black box." Once you've determined how an object interacts with other elements in your program—that is, once you've declared its interface—you can freely alter its implementation without affecting any other part of the application.

The Interface

The declaration of a class interface begins with the compiler directive `@interface` and ends with the directive `@end`. (All Objective-C directives to the compiler begin with "@".)

```
@interface ClassName : ItsSuperclass
{
    instance variable declarations
}
method declarations
@end
```

The first line of the declaration presents the new class name and links it to its superclass. The superclass defines the position of the new class in the inheritance hierarchy, as discussed under "Inheritance" (page 44). If the colon and superclass name are omitted, the new class is declared as a root class, a rival to the NSObject class.

Following the first part of the class declaration, braces enclose declarations of **instance variables**, the data structures that are part of each instance of the class. Here's a partial list of instance variables that might be declared in the Rectangle class:

```
float width;
float height;
BOOL filled;
```

```
NSColor *fillColor;
```

Methods for the class are declared next, after the braces enclosing instance variables and before the end of the class declaration. The names of methods that can be used by class objects, **class methods**, are preceded by a plus sign:

```
+ alloc;
```

The methods that instances of a class can use, **instance methods**, are marked with a minus sign:

```
- (void)display;
```

Although it's not a common practice, you can define a class method and an instance method with the same name. A method can also have the same name as an instance variable. This is more common, especially if the method returns the value in the variable. For example, Circle has a `radius` method that could match a `radius` instance variable.

Method return types are declared using the standard C syntax for casting one type to another:

```
- (float)radius;
```

Argument types are declared in the same way:

```
- (void)setRadius:(float)aRadius;
```

If a return or argument type isn't explicitly declared, it's assumed to be the default type for methods and messages—an `id`. The `alloc` method illustrated earlier returns `id`.

When there's more than one argument, the arguments are declared within the method name after the colons. Arguments break the name apart in the declaration, just as in a message. For example:

```
- (void)setWidth:(float)width height:(float)height;
```

Methods that take a variable number of arguments declare them using a comma and ellipsis points, just as a function would:

```
- makeGroup:group, ...;
```

Importing the Interface

The interface file must be included in any source module that depends on the class interface—that includes any module that creates an instance of the class, sends a message to invoke a method declared for the class, or mentions an instance variable declared in the class. The interface is usually included with the `#import` directive:

```
#import "Rectangle.h"
```

This directive is identical to `#include`, except that it makes sure that the same file is never included more than once. It's therefore preferred and is used in place of `#include` in code examples throughout Objective-C–based documentation.

To reflect the fact that a class definition builds on the definitions of inherited classes, an interface file begins by importing the interface for its superclass:

```
#import "ItsSuperclass.h"
```

```
@interface ClassName : ItsSuperclass
{
    instance variable declarations
}
method declarations
@end
```

This convention means that every interface file includes, indirectly, the interface files for all inherited classes. When a source module imports a class interface, it gets interfaces for the entire inheritance hierarchy that the class is built upon.

Note that if there is a *precomp*—a precompiled header—that supports the superclass, you may prefer to import the precomp instead.

Referring to Other Classes

An interface file declares a class and, by importing its superclass, implicitly contains declarations for all inherited classes, from NSObject on down through its superclass. If the interface mentions classes not in this hierarchy, it must import them explicitly or declare them with the `@class` directive:

```
@class Rectangle, Circle;
```

This directive simply informs the compiler that "Rectangle" and "Circle" are class names. It doesn't import their interface files.

An interface file mentions class names when it statically types instance variables, return values, and arguments. For example, this declaration

```
- (void)setPrimaryColor:(NSColor *)aColor;
```

mentions the NSColor class.

Since declarations like this simply use the class name as a type and don't depend on any details of the class interface (its methods and instance variables), the `@class` directive gives the compiler sufficient forewarning of what to expect. However, where the interface to a class is actually used (instances created, messages sent), the class interface must be imported. Typically, an interface file uses `@class` to declare classes, and the corresponding implementation file imports their interfaces (since it will need to create instances of those classes or send them messages).

The `@class` directive minimizes the amount of code seen by the compiler and linker, and is therefore the simplest way to give a forward declaration of a class name. Being simple, it avoids potential problems that may come with importing files that import still other files. For example, if one class declares a statically typed instance variable of another class, and their two interface files import each other, neither class may compile correctly.

The Role of the Interface

The purpose of the interface file is to declare the new class to other source modules (and to other programmers). It contains all the information they need to work with the class (programmers might also appreciate a little documentation).

- The interface file tells users how the class is connected into the inheritance hierarchy and what other classes—inherited or simply referred to somewhere in the class—are needed.

- The interface file also lets the compiler know what instance variables an object contains, and tells programmers what variables subclasses inherit. Although instance variables are most naturally viewed as a matter of the implementation of a class rather than its interface, they must nevertheless be declared in the interface file. This is because the compiler must be aware of the structure of an object where it's used, not just where it's defined. As a programmer, however, you can generally ignore the instance variables of the classes you use, except when defining a subclass.
- Finally, through its list of method declarations, the interface file lets other modules know what messages can be sent to the class object and instances of the class. Every method that can be used outside the class definition is declared in the interface file; methods that are internal to the class implementation can be omitted.

The Implementation

The definition of a class is structured very much like its declaration. It begins with the `@implementation` directive and ends with the `@end` directive:

```
@implementation ClassName : ItsSuperclass
{
    instance variable declarations
}
method definitions
@end
```

However, every implementation file must import its own interface. For example, `Rectangle.m` imports `Rectangle.h`. Because the implementation doesn't need to repeat any of the declarations it imports, it can safely omit:

- The name of the superclass
- The declarations of instance variables

This simplifies the implementation and makes it mainly devoted to method definitions:

```
#import "ClassName.h"

@implementation ClassName
method definitions
@end
```

Methods for a class are defined, like C functions, within a pair of braces. Before the braces, they're declared in the same manner as in the interface file, but without the semicolon. For example:

```
+ alloc
{
    ...
}

- (BOOL)isfilled
{
    ...
}

- (void)setFilled:(BOOL)flag
{
    ...
}
```

Methods that take a variable number of arguments handle them just as a function would:

```
#import <stdarg.h>

 ...

- getGroup:group, ...
{
    va_list ap;
    va_start(ap, group);
    ...
}
```

Referring to Instance Variables

By default, the definition of an instance method has all the instance variables of the object within its scope. It can refer to them simply by name. Although the compiler creates the equivalent of C structures to store instance variables, the exact nature of the structure is hidden. You don't need either of the structure operators (`.` or `->`) to refer to an object's data. For example, the following method definition refers to the receiver's `filled` instance variable:

```
- (void)setFilled:(BOOL)flag
{
    filled = flag;
    ...
}
```

Neither the receiving object nor its `filled` instance variable is declared as an argument to this method, yet the instance variable falls within its scope. This simplification of method syntax is a significant shorthand in the writing of Objective-C code.

When the instance variable belongs to an object that's not the receiver, the object's type must be made explicit to the compiler through static typing. In referring to the instance variable of a statically typed object, the structure pointer operator (`->`) is used.

Suppose, for example, that the Sibling class declares a statically typed object, `twin`, as an instance variable:

```
@interface Sibling : NSObject
{
    Sibling *twin;
    int gender;
    struct features *appearance;
}
```

As long as the instance variables of the statically typed object are within the scope of the class (as they are here because `twin` is typed to the same class), a Sibling method can set them directly:

```
- makeIdenticalTwin
{
    if ( !twin ) {
        twin = [[Sibling alloc] init];
        twin->gender = gender;
        twin->appearance = appearance;
    }
    return twin;
}
```

The Scope of Instance Variables

Although they're declared in the class interface, instance variables are more a matter of the way a class is implemented than of the way it's used. An object's interface lies in its methods, not in its internal data structures.

Often there's a one-to-one correspondence between a method and an instance variable, as in the following example:

```
- (BOOL)isFilled
{
    return filled;
}
```

But this need not be the case. Some methods might return information not stored in instance variables, and some instance variables might store information that an object is unwilling to reveal.

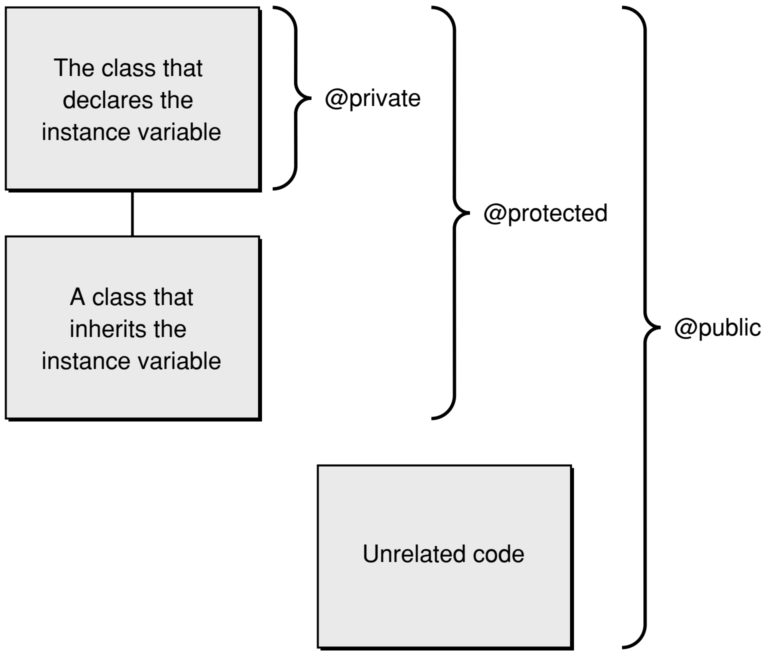
As a class is revised from time to time, the choice of instance variables may change, even though the methods it declares remain the same. As long as messages are the vehicle for interacting with instances of the class, these changes won't really affect its interface.

To enforce the ability of an object to hide its data, the compiler limits the scope of instance variables—that is, limits their visibility within the program. But to provide flexibility, it also lets you explicitly set the scope at three different levels. Each level is marked by a compiler directive:

Directive	Meaning
`@private`	The instance variable is accessible only within the class that declares it.
`@protected`	The instance variable is accessible within the class that declares it and within classes that inherit it.
`@public`	The instance variable is accessible everywhere.

This is illustrated in Figure 3-4.

Figure 3-4 The scope of instance variables


The class that declares the instance variable
A class that inherits the instance variable
Unrelated code
@private
@protected
@public

A directive applies to all the instance variables listed after it, up to the next directive or the end of the list. In the following example, the `age` and `evaluation` instance variables are private, `name`, `job`, and `wage` are protected, and `boss` is public.

```
@interface Worker : NSObject
{
    char *name;
@private
    int age;
    char *evaluation;
@protected
    id job;
    float wage;
@public
    id boss;
}
```

By default, all unmarked instance variables (like `name` above) are `@protected`.

All instance variables that a class declares, no matter how they're marked, are within the scope of the class definition. For example, a class that declares a `job` instance variable, such as the Worker class shown above, can refer to it in a method definition:

```
- promoteTo:newPosition
{
    id old = job;
    job = newPosition;
    return old;
}
```

Obviously, if a class couldn't access its own instance variables, the instance variables would be of no use whatsoever.

Normally, a class also has access to the instance variables it inherits. The ability to refer to an instance variable is usually inherited along with the variable. It makes sense for classes to have their entire data structures within their scope, especially if you think of a class definition as merely an elaboration of the classes it inherits from. The `promoteTo:` method illustrated earlier could just as well have been defined in any class that inherits the `job` instance variable from the Worker class.

However, there are reasons why you might want to restrict inheriting classes from directly accessing an instance variable:

- Once a subclass accesses an inherited instance variable, the class that declares the variable is tied to that part of its implementation. In later versions, it can't eliminate the variable or alter the role it plays without inadvertently breaking the subclass.
- Moreover, if a subclass accesses an inherited instance variable and alters its value, it may inadvertently introduce bugs in the class that declares the variable, especially if the variable is involved in class-internal dependencies.

To limit an instance variable's scope to just the class that declares it, you must mark it `@private`. Instance variables marked `@private` are only available to subclasses by calling public accessor methods, if they exist.

At the other extreme, marking a variable `@public` makes it generally available, even outside of class definitions that inherit or declare the variable. Normally, to get information stored in an instance variable, other objects must send a message requesting it. However, a public instance variable can be accessed anywhere as if it were a field in a C structure. For example:

```
Worker *ceo = [[Worker alloc] init];
ceo->boss = nil;
```

Note that the object must be statically typed.

Marking instance variables `@public` defeats the ability of an object to hide its data. It runs counter to a fundamental principle of object-oriented programming—the encapsulation of data within objects where it's protected from view and inadvertent error. Public instance variables should therefore be avoided except in extraordinary cases.

How Messaging Works

In Objective-C, messages aren't bound to method implementations until runtime. The compiler converts a message expression,

```
[receiver message]
```

into a call on a messaging function, `objc_msgSend`. This function takes the receiver and the name of the method mentioned in the message—that is, the method selector—as its two principal parameters:

```
objc_msgSend(receiver, selector)
```

Any arguments passed in the message are also handed to `objc_msgSend`:

```
objc_msgSend(receiver, selector, arg1, arg2, ...)
```

The messaging function does everything necessary for dynamic binding:

- It first finds the procedure (method implementation) that the selector refers to. Since the same method can be implemented differently by separate classes, the precise procedure that it finds depends on the class of the receiver.
- It then calls the procedure, passing it the receiving object (a pointer to its data), along with any arguments that were specified for the method.
- Finally, it passes on the return value of the procedure as its own return value.

Note: The compiler generates calls to the messaging function. You should never call it directly in the code you write.

The key to messaging lies in the structures that the compiler builds for each class and object. Every class structure includes these two essential elements:

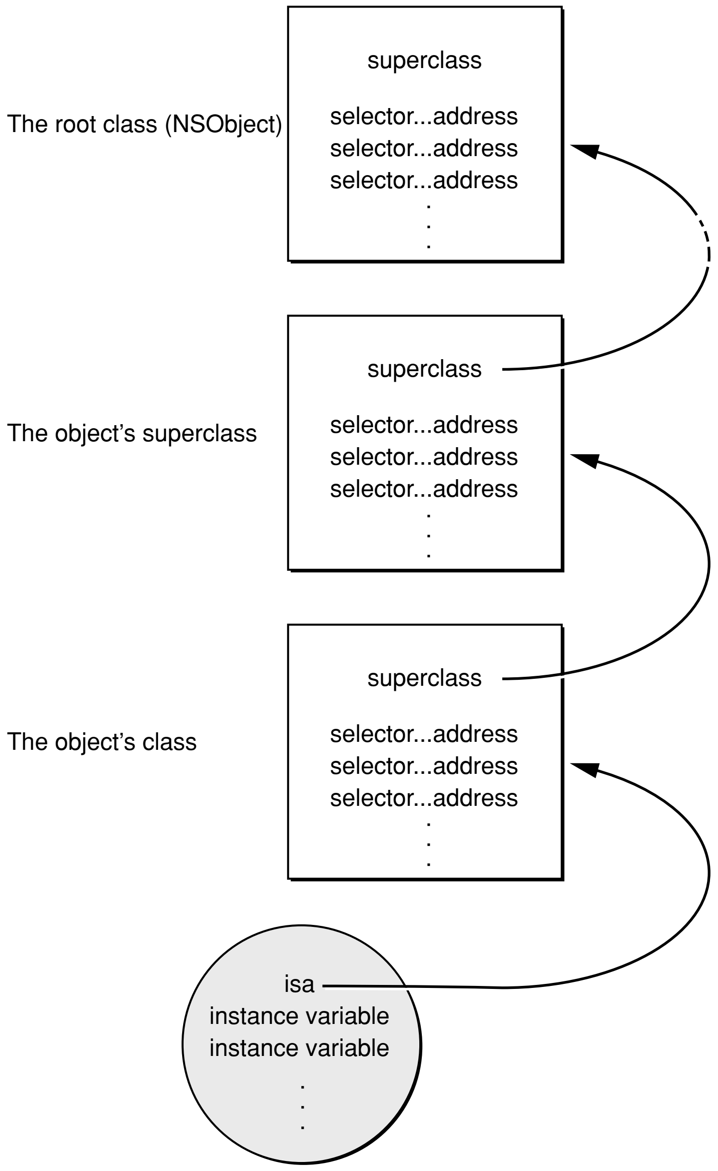
- A pointer to the superclass.
- A class **dispatch table**. This table has entries that associate method selectors with the class-specific addresses of the methods they identify. The selector for the `setOrigin::` method is associated with the address of (the procedure that implements) `setOrigin::`, the selector for the `display` method is associated with `display`'s address, and so on.

When a new object is created, memory for it is allocated, and its instance variables are initialized. First among the object's variables is a pointer to its class structure. This pointer, called `isa`, gives the object access to its class and, through the class, to all the classes it inherits from.

Note: While not strictly a part of the language, the `isa` pointer is required for an object to work with the Objective-C runtime system. An object needs to be "equivalent" to a `struct objc_object` (defined in `objc/objc.h`) in whatever fields the structure defines. However, you rarely, if ever, need to create your own root object, and objects that inherit from NSObject or NSProxy automatically have the `isa` variable.

These elements of class and object structure are illustrated in Figure 3-5.

Figure 3-5 Messaging Framework



When a message is sent to an object, the messaging function follows the object's `isa` pointer to the class structure where it looks up the method selector in the dispatch table. If it can't find the selector there, `objc_msgSend` follows the pointer to the superclass and tries to find the selector in its dispatch table. Successive failures cause `objc_msgSend` to climb the class hierarchy until it reaches the NSObject class. Once it locates the selector, the function calls the method entered in the table and passes it the receiving object's data structure.

This is the way that method implementations are chosen at runtime—or, in the jargon of object-oriented programming, that methods are dynamically bound to messages.

To speed the messaging process, the runtime system caches the selectors and addresses of methods as they are used. There's a separate cache for each class, and it can contain selectors for inherited methods as well as for methods defined in the class. Before searching the dispatch tables, the messaging routine first checks the cache of the receiving object's class (on the theory that a method that was used once may likely be used again). If the method selector is in the cache, messaging is only slightly slower than a function call. Once a program has been running long enough to "warm up" its caches, almost all the messages it sends find a cached method. Caches grow dynamically to accommodate new messages as the program runs.

Selectors

For efficiency, full ASCII names are not used as method selectors in compiled code. Instead, the compiler writes each method name into a table, then pairs the name with a unique identifier that represents the method at runtime. The runtime system makes sure each identifier is unique: No two selectors are the same, and all methods with the same name have the same selector. Compiled selectors are assigned to a special type, `SEL`, to distinguish them from other data. Valid selectors are never 0. You must let the system assign `SEL` identifiers to methods; it's futile to assign them arbitrarily.

The `@selector()` directive lets Objective-C source code refer to the compiled selector, rather than to the full method name. Here, the selector for `setWidth:height:` is assigned to the `setWidthHeight` variable:

```
SEL  setWidthHeight;
setWidthHeight = @selector(setWidth:height:);
```

It's most efficient to assign values to `SEL` variables at compile time with the `@selector()` directive. However, in some cases, a program may need to convert a character string to a selector at runtime. This can be done with the `NSSelectorFromString` function:

```
setWidthHeight = NSSelectorFromString(aBuffer);
```

Conversion in the opposite direction is also possible. The `NSStringFromSelector` function returns a method name for a selector:

```
NSString *method;
method = NSStringFromSelector(setWidthHeight);
```

These and other runtime functions are described in the Cocoa framework reference documentation.

Methods and Selectors

Compiled selectors identify method names, not method implementations. Rectangle's `display` method, for example, has the same selector as `display` methods defined in other classes. This is essential for polymorphism and dynamic binding; it lets you send the same message to receivers belonging to different classes. If there were one selector per method implementation, a message would be no different than a function call.

A class method and an instance method with the same name are assigned the same selector. However, because of their separate domains, there's no confusion between the two. A class could define a `display` class method in addition to a `display` instance method.

Method Return and Argument Types

The messaging routine has access to method implementations only through selectors, so it treats all methods with the same selector alike. It discovers the return type of a method, and the data types of its arguments, from the selector. Therefore, except for messages sent to statically typed receivers, dynamic binding requires all implementations of identically named methods to have the same return type and the same argument types. (Statically typed receivers are an exception to this rule, since the compiler can learn about the method implementation from the class type.)

Although identically named class methods and instance methods are represented by the same selector, they can have different argument and return types.

Varying the Message at Runtime

The `performSelector:`, `performSelector:withObject:`, and `performSelector:withObject:withObject:` methods, defined in the NSObject protocol, take `SEL` identifiers as their initial arguments. All three methods map directly into the messaging function. For example,

```
[friend performSelector:@selector(gossipAbout:)
    withObject:aNeighbor];
```

is equivalent to:

```
[friend gossipAbout:aNeighbor];
```

These methods make it possible to vary a message at runtime, just as it's possible to vary the object that receives the message. Variable names can be used in both halves of a message expression:

```
id   helper = getTheReceiver();
SEL  request = getTheSelector();
[helper performSelector:request];
```

In this example, the receiver (`helper`) is chosen at runtime (by the fictitious `getTheReceiver` function), and the method the receiver is asked to perform (`request`) is also determined at runtime (by the equally fictitious `getTheSelector` function).

Note: `performSelector:` and its companion methods return an `id`. If the method that's performed returns a different type, it should be cast to the proper type. (However, casting doesn't work for all types; the method should return a pointer or a type compatible with a pointer.)

The Target-Action Paradigm

In its treatment of user-interface controls, the Application Kit makes good use of the ability to vary both the receiver and the message.

NSControl objects are graphical devices that can be used to give instructions to an application. Most resemble real-world control devices such as buttons, switches, knobs, text fields, dials, menu items, and the like. In software, these devices stand between the application and the user. They interpret events coming from hardware devices like the keyboard and mouse and translate them into application-specific instructions. For example, a button labeled "Find" would translate a mouse click into an instruction for the application to start searching for something.

The Application Kit defines a template for creating control devices and defines a few "off-the-shelf" devices of its own. For example, the NSButtonCell class defines an object that you can assign to an NSMatrix and initialize with a size, a label, a picture, a font, and a keyboard alternative. When the user clicks the button (or uses the keyboard alternative), the NSButtonCell sends a message instructing the application to do something. To do this, an NSButtonCell must be initialized not just with an image, a size, and a label, but with directions on what message to send and who to send it to. Accordingly, an NSButtonCell can be initialized for an action message, the method selector it should use in the message it sends, and a target, the object that should receive the message.

```
[myButtonCell setAction:@selector(reapTheWind:)];
[myButtonCell setTarget:anObject];
```

The NSButtonCell sends the message using NSObject's `performSelector:withObject:` method. All action messages take a single argument, the `id` of the control device sending the message.

If Objective-C didn't allow the message to be varied, all NSButtonCell objects would have to send the same message; the name of the method would be frozen in the NSButtonCell source code. Instead of simply implementing a mechanism for translating user actions into action messages, button cells and other controls would have to constrain the content of the message. This would make it difficult for any object to respond to more than one button cell. There would either have to be one target for each button, or the target object would have to discover which button the message came from and act accordingly. Each time you rearranged the user interface, you would also have to re-implement the method that responds to the action message. This would be an unnecessary complication that Objective-C happily avoids.

Avoiding Messaging Errors

If an object receives a message to perform a method that isn't in its repertoire, an error results. It's the same sort of error as calling a nonexistent function. But because messaging occurs at runtime, the error often isn't evident until the program executes.

It's relatively easy to avoid this error when the message selector is constant and the class of the receiving object is known. As you write your programs, you can make sure that the receiver is able to respond. If the receiver is statically typed, the compiler performs this test for you.

However, if the message selector or the class of the receiver varies, it may be necessary to postpone this test until runtime. The `respondsToSelector:` method, defined in the NSObject class, determines whether a receiver can respond to a message. It takes the method selector as an argument and returns whether the receiver has access to a method matching the selector:

```
if ( [anObject respondsToSelector:@selector(setOrigin::)] )
    [anObject setOrigin:0.0 :0.0];
else
    fprintf(stderr, "%s can't be placed\n",
        [NSStringFromClass([anObject class]) UTF8String]);
```

The `respondsToSelector:` test is especially important when sending messages to objects that you don't have control over at compile time. For example, if you write code that sends a message to an object represented by a variable that others can set, you should make sure the receiver implements a method that can respond to the message.

Note: An object can also arrange to have the messages it receives forwarded to other objects if it can't respond to them directly itself. In that case, it appears that the object can handle the message, even though the object responds to the message indirectly by assigning it to another object. See "Inheriting Instance Variables" (page 45) for more information.

Hidden Arguments

When the messaging function finds the procedure that implements a method, it calls the procedure and passes it all the arguments in the message. It also passes the procedure two hidden arguments:

- The receiving object
- The selector for the method

These arguments give every method implementation explicit information about the two halves of the message expression that invoked it. They're said to be "hidden" because they aren't declared in the source code that defines the method. They're inserted into the implementation when the code is compiled.

Although these arguments aren't explicitly declared, source code can still refer to them (just as it can refer to the receiving object's instance variables). A method refers to the receiving object as `self`, and to its own selector as `_cmd`. In the example below, `_cmd` refers to the selector for the `strange` method and `self` to the object that receives a `strange` message.

```
- strange
{
    id  target = getTheReceiver();
    SEL method = getTheMethod();

    if ( target == self || method == _cmd )
        return nil;
    return [target performSelector:method];
}
```

`self` is the more useful of the two arguments. It is, in fact, the way the receiving object's instance variables are made available to the method definition.

Messages to self and super

Objective-C provides two terms that can be used within a method definition to refer to the object that performs the method—`self` and `super`.

Suppose, for example, that you define a `reposition` method that needs to change the coordinates of whatever object it acts on. It can invoke the `setOrigin::` method to make the change. All it needs to do is send a `setOrigin::` message to the same object that the `reposition` message itself was sent to. When you're writing the reposition code, you can refer to that object as either `self` or `super`. The `reposition` method could read either:

```
- reposition
{
    ...
    [self setOrigin:someX :someY];
    ...
}
```

or:

```
- reposition
{
    ...
    [super setOrigin:someX :someY];
    ...
}
```

Here, `self` and `super` both refer to the object receiving a `reposition` message, whatever object that may happen to be. The two terms are quite different, however. `self` is one of the hidden arguments that the messaging routine passes to every method; it's a local variable that can be used freely within a method implementation, just as the names of instance variables can be. `super` is a term that substitutes for `self` only as the receiver in a message expression. As receivers, the two terms differ principally in how they affect the messaging process:

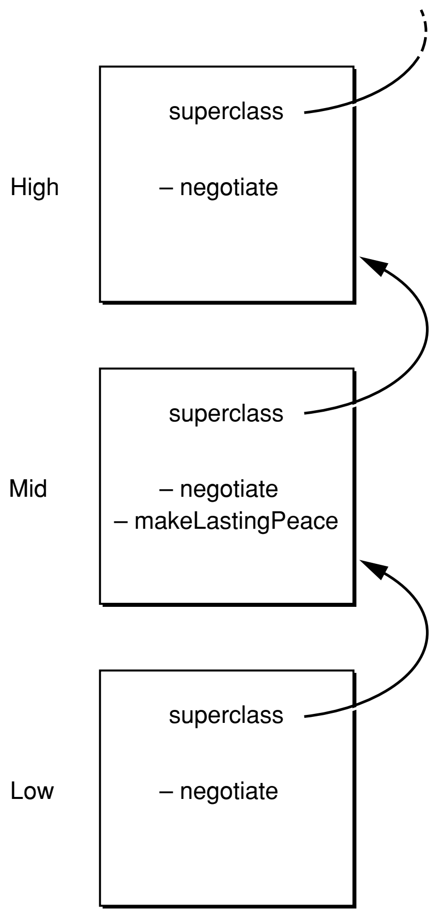
- `self` searches for the method implementation in the usual manner, starting in the dispatch table of the receiving object's class. In the example above, it would begin with the class of the object receiving the reposition message.
- `super` starts the search for the method implementation in a very different place. It begins in the superclass of the class that defines the method where `super` appears. In the example above, it would begin with the superclass of the class where reposition is defined.

Wherever `super` receives a message, the compiler substitutes another messaging routine for the `objc_msgSend` function. The substitute routine looks directly to the superclass of the defining class—that is, to the superclass of the class sending the message to `super`—rather than to the class of the object receiving the message.

An Example

The difference between `self` and `super` becomes clear in a hierarchy of three classes. Suppose, for example, that we create an object belonging to a class called Low. Low's superclass is Mid; Mid's superclass is High. All three classes define a method called `negotiate`, which they use for a variety of purposes. In addition, Mid defines an ambitious method called `makeLastingPeace`, which also has need of the `negotiate` method. This is illustrated in Figure 3-6:

Figure 3-6 High, Mid, Low



We now send a message to our Low object to perform the `makeLastingPeace` method, and `makeLastingPeace`, in turn, sends a `negotiate` message to the same Low object. If source code calls this object `self`,

```
- makeLastingPeace
{
    [self negotiate];
    ...
}
```

the messaging routine finds the version of `negotiate` defined in Low, `self`'s class. However, if Mid's source code calls this object `super`,

```
- makeLastingPeace
{
    [super negotiate];
    ...
}
```

the messaging routine will find the version of `negotiate` defined in High. It ignores the receiving object's class (Low) and skips to the superclass of Mid, since Mid is where `makeLastingPeace` is defined. Neither message finds Mid's version of `negotiate`.

As this example illustrates, `super` provides a way to bypass a method that overrides another method. Here it enabled `makeLastingPeace` to avoid the Mid version of `negotiate` that redefined the original High version.

Not being able to reach Mid's version of `negotiate` may seem like a flaw, but, under the circumstances, it's right to avoid it:

- The author of the Low class intentionally overrode Mid's version of `negotiate` so that instances of the Low class (and its subclasses) would invoke the redefined version of the method instead. The designer of Low didn't want Low objects to perform the inherited method.
- In sending the message to `super`, the author of Mid's `makeLastingPeace` method intentionally skipped over Mid's version of `negotiate` (and over any versions that might be defined in classes like Low that inherit from Mid) to perform the version defined in the High class. Mid's designer wanted to use the High version of `negotiate` and no other.

Mid's version of `negotiate` could still be used, but it would take a direct message to a Mid instance to do it.

Using super

Messages to `super` allow method implementations to be distributed over more than one class. You can override an existing method to modify or add to it, and still incorporate the original method in the modification:

```
- negotiate
{
    ...
    return [super negotiate];
}
```

For some tasks, each class in the inheritance hierarchy can implement a method that does part of the job and passes the message on to `super` for the rest. The `init` method, which initializes a newly allocated instance, is designed to work like this. Each `init` method has responsibility for initializing the instance variables defined in its class. But before doing so, it sends an `init` message to `super` to have the classes it inherits from initialize their instance variables. Each version of `init` follows this procedure, so classes initialize their instance variables in the order of inheritance:

```
- (id)init
{
    [super init];
    ...
}
```

It's also possible to concentrate core functionality in one method defined in a superclass, and have subclasses incorporate the method through messages to `super`. For example, every class method that creates an instance must allocate storage for the new object and initialize its `isa` pointer to the class structure. This is typically left to the `alloc` and `allocWithZone:` methods defined in the NSObject class. If another class overrides these methods (a rare case), it can still get the basic functionality by sending a message to `super`.

Redefining self

super is simply a flag to the compiler telling it where to begin searching for the method to perform; it's used only as the receiver of a message. But self is a variable name that can be used in any number of ways, even assigned a new value.

There's a tendency to do just that in definitions of class methods. Class methods are often concerned not with the class object, but with instances of the class. For example, many class methods combine allocation and initialization of an instance, often setting up instance variable values at the same time. In such a method, it might be tempting to send messages to the newly allocated instance and to call the instance self, just as in an instance method. But that would be an error. self and super both refer to the receiving object—the object that gets a message telling it to perform the method. Inside an instance method, self refers to the instance; but inside a class method, self refers to the class object. This is an example of what not to do:

```
+ (Rectangle *)rectangleOfColor:(NSColor *) color
{
    self = [[self alloc] init]; // BAD
    [self setColor:color];
    return [self autorelease];
}
```

To avoid confusion, it's usually better to use a variable other than self to refer to an instance inside a class method:

```
+ (id)rectangleOfColor:(NSColor *)color
{
    id newInstance = [[self alloc] init]; // GOOD
    [newInstance setColor:color];
    return [newInstance autorelease];
}
```

In fact, rather than sending the alloc message to self in a class method, it's often better to send alloc to [self class]. This way, if the class is subclassed, and the rectangleOfColor: message is received by a subclass, the instance returned will be the same type as the subclass.

```
+ (id)rectangleOfColor:(NSColor *)color
{
    id newInstance = [[[self class] alloc] init]; // EXCELLENT
    [newInstance setColor:color];
    return [newInstance autorelease];
}
```

See "Memory Management" (page 96) for more information about object allocation.

Extending Classes

Class definitions are at the heart of Objective-C programming, but they're not the only mechanism for structuring object definitions in Objective-C. This section discusses two other ways of declaring methods and associating them with a class:

- Categories can compartmentalize a class definition or extend an existing one.
- Protocols declare methods that can be implemented by any class.

Categories—Adding Methods to Existing Classes

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 a new class. You cannot, however, use a category to add additional instance variables to a class.

A category can be an alternative to a subclass. Rather than define a subclass to extend an existing class, through a category you can add methods to the class directly. For example, you could add categories to NSArray and other Cocoa classes. As in the case of a subclass, you don't need source code for the class you're extending.

The methods the category adds become part of the class type. For example, methods added to the NSArray class in a category are among the methods the compiler expects an NSArray instance to have in its repertoire. Methods added to the NSArray class in a subclass are not included in the NSArray type. (This matters only for statically typed objects, since static typing is the only way the compiler can know an object's class.)

Category methods can do anything that methods defined in the class proper can do. At runtime, there's no difference. The methods the category adds to the class are inherited by all the class's subclasses, just like other methods.

Adding to a Class

The declaration of a category interface looks very much like a class interface declaration—except the category name is listed within parentheses after the class name and the superclass isn't mentioned. Unless its methods don't access any instance variables of the class, the category must import the interface file for the class it extends:

```
#import "ClassName.h"

@interface ClassName ( CategoryName )
// method declarations
@end
```

The implementation, as usual, imports its own interface. Assuming that the interface file is named after the category, a category implementation looks like this:

```
#import "CategoryName.h"

@implementation ClassName ( CategoryName )
// method definitions
@end
```

Note that a category can't declare additional instance variables for the class; it includes only methods. However, all instance variables within the scope of the class are also within the scope of the category. That includes all instance variables declared by the class, even ones declared `@private`.

There's no limit to the number of categories that you can add to a class, but each category name must be different, and each should declare and define a different set of methods.

The methods added in a category can be used to extend the functionality of the class or override methods the class inherits. A category can also override methods declared in the class interface. However, it cannot reliably override methods declared in another category of the same class. A category is not a substitute for a subclass. It's best if categories don't attempt to redefine methods that are explicitly declared in the class's `@interface` section. Also note that a class can't define the same method more than once.

When a category overrides an inherited method, the new version can, as usual, incorporate the inherited version through a message to `super`. But there's no way for a category method to incorporate a method with the same name defined for the same class.

How Categories Are Used

Categories can be used to extend classes defined by other implementors—for example, you can add methods to the classes defined in the Cocoa frameworks. The added methods are inherited by subclasses and are indistinguishable at runtime from the original methods of the class.

Categories can also be used to distribute the implementation of a new class into separate source files—for example, you could group the methods of a large class into several categories and put each category in a different file. When used like this, categories can benefit the development process in a number of ways:

- They provide a simple way of grouping related methods. Similar methods defined in different classes can be kept together in the same source file.
- They simplify the management of a large class when several developers contribute to the class definition.
- They let you achieve some of the benefits of incremental compilation for a very large class.
- They can help improve locality of reference for commonly used methods.
- They enable you to configure a class differently for separate applications, without having to maintain different versions of the same source code.

Categories are also used to declare informal protocols, as discussed under "Protocols—Declaring Interfaces for Others to Implement" (page 74).

Categories of the Root Class

A category can add methods to any class, including the root class. Methods added to NSObject become available to all classes that are linked to your code. While this can be useful at times, it can also be quite dangerous. Although it may seem that the modifications the category makes are well understood and of limited impact, inheritance gives them a wide scope. You may be making unintended changes to unseen classes; you may not know all the consequences of what you're doing. Moreover, others who are unaware of your changes won't understand what they're doing.

In addition, there are two other considerations to keep in mind when implementing methods for the root class:

- Messages to `super` are invalid (there is no superclass).
- Class objects can perform instance methods defined in the root class.

Normally, class objects can perform only class methods. But instance methods defined in the root class are a special case. They define an interface to the runtime system that all objects inherit. Class objects are full-fledged objects and need to share the same interface.

This feature means that you need to take into account the possibility that an instance method you define in a category of the NSObject class might be performed not only by instances but by class objects as well. For example, within the body of the method, `self` might mean a class object as well as an instance. See the NSObject class specification in the Foundation framework reference for more information on class access to root instance methods.

Protocols—Declaring Interfaces for Others to Implement

Class and category interfaces declare methods that are associated with a particular class—mainly methods that the class implements. Informal and formal **protocols**, on the other hand, declare methods not associated with a class, but which any class, and perhaps many classes, might implement.

A protocol is simply a list of method declarations, unattached to a class definition. For example, these methods that report user actions on the mouse could be gathered into a protocol:

```
- (void)mouseDown:(NSEvent *)theEvent;
- (void)mouseDragged:(NSEvent *)theEvent;
- (void)mouseUp:(NSEvent *)theEvent;
```

Any class that wanted to respond to mouse events could adopt the protocol and implement its methods.

Protocols free method declarations from dependency on the class hierarchy, so they can be used in ways that classes and categories cannot. Protocols list methods that are (or may be) implemented somewhere, but the identity of the class that implements them is not of interest. What is of interest is whether or not a particular class **conforms** to the protocol—whether it has implementations of the methods the protocol declares. Thus objects can be grouped into types not just on the basis of similarities due to the fact that they inherit from the same class, but also on the basis of their similarity in conforming to the same protocol. Classes in unrelated branches of the inheritance hierarchy might be typed alike because they conform to the same protocol.

Protocols can play a significant role in object-oriented design, especially where a project is divided among many implementors or it incorporates objects developed in other projects. Cocoa software uses them heavily to support interprocess communication through Objective-C messages.

However, an Objective-C program doesn't need to use protocols. Unlike class definitions and message expressions, they're optional. Some Cocoa frameworks use them; some don't. It all depends on the task at hand.

When to Use Protocols

Protocols are useful in at least three situations:

- To declare methods that others are expected to implement
- To declare the interface to an object while concealing its class
- To capture similarities among classes that are not hierarchically related

The following sections discuss these situations and the roles protocols can play.

Methods for Others to Implement

If you know the class of an object, you can look at its interface declaration (and the interface declarations of the classes it inherits from) to find what messages it responds to. These declarations advertise the messages it can receive. Protocols provide a way for it to also advertise the messages it sends.

Communication works both ways; objects send messages as well as receive them. For example, an object might delegate responsibility for a certain operation to another object, or it may on occasion simply need to ask another object for information. In some cases, an object might be willing to notify other objects of its actions so that they can take whatever collateral measures might be required.

If you develop the class of the sender and the class of the receiver as part of the same project (or if someone else has supplied you with the receiver and its interface file), this communication is easily coordinated. The sender simply imports the interface file of the receiver. The imported file declares the method selectors the sender uses in the messages it sends.

However, if you develop an object that sends messages to objects that aren't yet defined—objects that you're leaving for others to implement—you won't have the receiver's interface file. You need another way to declare the methods you use in messages but don't implement. A protocol serves this purpose. It informs the compiler about methods the class uses and also informs other implementors of the methods they need to define to have their objects work with yours.

Suppose, for example, that you develop an object that asks for the assistance of another object by sending it `helpOut:` and other messages. You provide an `assistant` instance variable to record the outlet for these messages and define a companion method to set the instance variable. This method lets other objects register themselves as potential recipients of your object's messages:

```
- setAssistant:anObject
{
    assistant = anObject;
}
```

Then, whenever a message is to be sent to the `assistant`, a check is made to be sure that the receiver implements a method that can respond:

```
- (BOOL)doWork
{
    ...
    if ( [assistant respondsTo:@selector(helpOut:)] ) {
        [assistant helpOut:self];
        return YES;
    }
    return NO;
}
```

Since, at the time you write this code, you can't know what kind of object might register itself as the `assistant`, you can only declare a protocol for the `helpOut:` method; you can't import the interface file of the class that implements it.

Declaring Interfaces for Anonymous Objects

A protocol can be used to declare the methods of an **anonymous object**, an object of unknown class. An anonymous object may represent a service or handle a limited set of functions, especially where only one object of its kind is needed. (Objects that play a fundamental role in defining an application's architecture and objects that you must initialize before using are not good candidates for anonymity.)

Objects are not anonymous to their developers, of course, but they are anonymous when the developer supplies them to someone else. For example, consider the following situations:

- Someone who supplies a framework or a suite of objects for others to use can include objects that are not identified by a class name or an interface file. Lacking the name and class interface, users have no way of creating instances of the class. Instead, the supplier must provide a ready-made instance. Typically, a method in another class returns a usable object:

  ```
  id formatter = [receiver formattingService];
  ```

  The object returned by the method is an object without a class identity, at least not one the supplier is willing to reveal. For it to be of any use at all, the supplier must be willing to identify at least some of the messages that it can respond to. This is done by associating the object with a list of methods declared in a protocol.

- You can send Objective-C messages to **remote objects**—objects in other applications. ("Remote Messaging" (page 113), discusses this possibility in more detail.)

  Each application has its own structure, classes, and internal logic. But you don't need to know how another application works or what its components are to communicate with it. As an outsider, all you need to know is what messages you can send (the protocol) and where to send them (the receiver).

  An application that publishes one of its objects as a potential receiver of remote messages must also publish a protocol declaring the methods the object will use to respond to those messages. It doesn't have to disclose anything else about the object. The sending application doesn't need to know the class of the object or use the class in its own design. All it needs is the protocol.

Protocols make anonymous objects possible. Without a protocol, there would be no way to declare an interface to an object without identifying its class.

Note: Even though the supplier of an anonymous object doesn't reveal its class, the object itself reveals it at runtime. A class message returns the anonymous object's class. However, there's usually little point in discovering this extra information; the information in the protocol is sufficient.

Non-Hierarchical Similarities

If more than one class implements a set of methods, those classes are often grouped under an abstract class that declares the methods they have in common. Each subclass may re-implement the methods in its own way, but the inheritance hierarchy and the common declaration in the abstract class captures the essential similarity between the subclasses.

However, sometimes it's not possible to group common methods in an abstract class. Classes that are unrelated in most respects might nevertheless need to implement some similar methods. This limited similarity may not justify a hierarchical relationship. For example, many different kinds of classes might implement methods to facilitate reference counting (this is just an example, since the Foundation Framework already implements reference counting for you):

```
- setRefCount:(int)count;
- (int)refCount;
- incrementCount;
- decrementCount;
```

These methods could be grouped into a protocol and the similarity between implementing classes accounted for by noting that they all conform to the same protocol.

Objects can be typed by this similarity (the protocols they conform to), rather than by their class. For example, an NSMatrix must communicate with the objects that represent its cells. The NSMatrix could require each of these objects to be a kind of NSCell (a type based on class) and rely on the fact that all objects that inherit from the NSCell class have the methods needed to respond to NSMatrix messages. Alternatively, the NSMatrix could require objects representing cells to have methods that can respond to a particular set of messages (a type based on protocol). In this case, the NSMatrix wouldn't care what class a cell object belonged to, just that it implemented the methods.

Informal Protocols

The simplest way of declaring a protocol is to group the methods in a category declaration:

```
@interface NSObject ( RefCounting )
- (int)refCount;
- incrementCount;
- decrementCount;
@end
```

Informal protocols are typically declared as categories of the NSObject class, since that broadly associates the method names with any class that inherits from NSObject. Because all classes inherit from the root class, the methods aren't restricted to any part of the inheritance hierarchy. (It would also be possible to declare an informal protocol as a category of another class to limit it to a certain branch of the inheritance hierarchy, but there is little reason to do so.)

When used to declare a protocol, a category interface doesn't have a corresponding implementation. Instead, classes that implement the protocol declare the methods again in their own interface files and define them along with other methods in their implementation files.

An informal protocol bends the rules of category declarations to list a group of methods but not associate them with any particular class or implementation.

Being informal, protocols declared in categories don't receive much language support. There's no type checking at compile time nor a check at runtime to see whether an object conforms to the protocol. To get these benefits, you must use a formal protocol. An informal protocol is good for times when implementing all the methods is optional, such as for a delegate.

Formal Protocols

The Objective-C language provides a way to formally declare a list of methods as a protocol. Formal protocols are supported by the language and the runtime system. For example, the compiler can check for types based on protocols, and objects can introspect at runtime to report whether or not they conform to a protocol.

Formal protocols are declared with the `@protocol` directive:

```
@protocol ProtocolName
method declarations
@end
```

For example, the reference-counting protocol could be declared like this:

```
@protocol ReferenceCounting
- (int)refCount;
- incrementCount;
- decrementCount;
```

```
@end
```

Unlike class names, protocol names don't have global visibility. They live in their own namespace.

A class is said to **adopt** a formal protocol if it agrees to implement the methods the protocol declares. Class declarations list the names of adopted protocols within angle brackets after the superclass name:

```
@interface ClassName : ItsSuperclass < protocol list >
```

Categories adopt protocols in much the same way:

```
@interface ClassName ( CategoryName ) < protocol list >
```

Names in the protocol list are separated by commas.

A class or category that adopts a protocol must import the header file where the protocol is declared. The methods declared in the adopted protocol are not declared elsewhere in the class or category interface.

It's possible for a class to simply adopt protocols and declare no other methods. For example, the following class declaration adopts the Formatting and Prettifying protocols, but declares no instance variables or methods of its own:

```
@interface Formatter : NSObject < Formatting, Prettifying >
@end
```

A class or category that adopts a protocol is obligated to implement all the methods the protocol declares. Otherwise, the compiler issues a warning. The Formatter class above would define all the methods declared in the two protocols it adopts, in addition to any it might have declared itself.

Adopting a protocol is similar in some ways to declaring a superclass. Both assign methods to the class. The superclass declaration assigns it inherited methods; the protocol assigns it methods declared in the protocol list.

Protocol Objects

Just as classes are represented at runtime by class objects and methods by selector codes, formal protocols are represented by a special data type—instances of the Protocol class. Source code that deals with a protocol (other than to use it in a type specification) must refer to the Protocol object.

In many ways, protocols are similar to class definitions. They both declare methods, and at runtime they're both represented by objects—classes by class objects and protocols by Protocol objects. Like class objects, Protocol objects are created automatically from the definitions and declarations found in source code and are used by the runtime system. They're not allocated and initialized in program source code.

Source code can refer to a Protocol object using the `@protocol()` directive—the same directive that declares a protocol, except that here it has a set of trailing parentheses. The parentheses enclose the protocol name:

```
Protocol *counter = @protocol(ReferenceCounting);
```

This is the only way that source code can conjure up a Protocol object. Unlike a class name, a protocol name doesn't designate the object—except inside `@protocol()`.

The compiler creates a Protocol object for each protocol declaration it encounters, but only if the protocol is also:

- Adopted by a class, or
- Referred to somewhere in source code (using `@protocol()`)

Protocols that are declared but not used (except for type checking as described below) aren't represented by Protocol objects at runtime.

Conforming to a Protocol

A class is said to **conform** to a formal protocol if it (or a superclass) implements the methods declared in the protocol. An instance of a class is said to conform to the same set of protocols its class conforms to.

Since a class must implement all the methods declared in the protocols it adopts, and those methods are inherited by its subclasses, saying that a class or an instance conforms to a protocol is tantamount to saying that it has in its repertoire all the methods the protocol declares.

It's possible to check whether an object conforms to a protocol by sending it a `conformsToProtocol:` message.

```
if ( [receiver conformsToProtocol:@protocol(ReferenceCounting)]  )
    [receiver incrementCount];
```

The `conformsToProtocol:` test is very much like the `respondsTo:` test for a single method, except that it tests whether a protocol has been adopted (and presumably all the methods it declares implemented) rather than just whether one particular method has been implemented. Because it checks for a whole list of methods, `conformsToProtocol:` can be more efficient than `respondsTo:`.

The `conformsToProtocol:` test is also very much like the `isKindOfClass:` test, except that it tests for a type based on a protocol rather than a type based on the inheritance hierarchy.

Type Checking

Type declarations for objects can be extended to include formal protocols. Protocols thus offer the possibility of another level of type checking by the compiler, one that's more abstract since it's not tied to particular implementations.

In a type declaration, protocol names are listed between angle brackets after the type name:

```
- (id <Formatting>)formattingService;
id <ReferenceCounting, AutoFreeing> anObject;
```

Just as static typing permits the compiler to test for a type based on the class hierarchy, this syntax permits the compiler to test for a type based on conformance to a protocol.

For example, if Formatter is an abstract class, this declaration

```
Formatter *anObject;
```

groups all objects that inherit from Formatter into a type and permits the compiler to check assignments against that type.

Similarly, this declaration,

```
id <Formatting> anObject;
```

groups all objects that conform to the Formatting protocol into a type, regardless of their positions in the class hierarchy. The compiler can make sure only objects that conform to the protocol are assigned to the type.

In each case, the type groups similar objects—either because they share a common inheritance, or because they converge on a common set of methods.

The two types can be combined in a single declaration:

```
Formatter <Formatting> *anObject;
```

Protocols can't be used to type class objects. Only instances can be statically typed to a protocol, just as only instances can be statically typed to a class. (However, at runtime, both classes and instances will respond to a `conformsToProtocol:` message.)

Protocols Within Protocols

One protocol can incorporate other protocols using the same syntax that classes use to adopt a protocol:

```
@protocol ProtocolName < protocol list >
```

All the protocols listed between angle brackets are considered part of the *ProtocolName* protocol. For example, if the Paging protocol incorporates the Formatting protocol,

```
@protocol Paging < Formatting >
```

any object that conforms to the Paging protocol also conforms to Formatting. Type declarations

```
id <Paging> someObject;
```

and `conformsToProtocol:` messages

```
if ( [anotherObject conformsToProtocol:@protocol(Paging)] )
    ...
```

need to mention only the Paging protocol to test for conformance to Formatting as well.

When a class adopts a protocol, it must implement the methods the protocol declares, as mentioned earlier. In addition, it must conform to any protocols the adopted protocol incorporates. If an incorporated protocol incorporates still other protocols, the class must also conform to them. A class can conform to an incorporated protocol by either:

- Implementing the methods the protocol declares, or
- Inheriting from a class that adopts the protocol and implements the methods.

Suppose, for example, that the Pager class adopts the Paging protocol. If Pager is a subclass of NSObject,

```
@interface Pager : NSObject < Paging >
```

it must implement all the Paging methods, including those declared in the incorporated Formatting protocol. It adopts the Formatting protocol along with Paging.

On the other hand, if Pager is a subclass of Formatter (a class that independently adopts the Formatting protocol),

```
@interface Pager : Formatter < Paging >
```

it must implement all the methods declared in the Paging protocol proper, but not those declared in Formatting. Pager inherits conformance to the Formatting protocol from Formatter.

Note that a class can conform to a protocol without formally adopting it simply by implementing the methods declared in the protocol.

Referring to Other Protocols

When working on complex applications, you occasionally find yourself writing code that looks like this:

```
#import "B.h"

@protocol A
- foo:(id <B>)anObject;
@end
```

where protocol B is declared like this:

```
#import "A.h"

@protocol B
- bar:(id <A>)anObject;
@end
```

In such a situation, circularity results and neither file will compile correctly. To break this recursive cycle, you must use the `@protocol` directive to make a forward reference to the needed protocol instead of importing the interface file where the protocol is defined. The following code excerpt illustrates how you would do this:

```
@protocol B;

@protocol A
- foo:(id <B>)anObject;
@end
```

Note that using the `@protocol` directive in this manner simply informs the compiler that "B" is a protocol to be defined later. It doesn't import the interface file where protocol B is defined.

Enabling Static Behaviors

This section explains how static typing works and discusses some other features of Objective-C, including ways to temporarily overcome its inherent dynamism.

Objective-C objects are dynamic entities. As many decisions about them as possible are pushed from compile time to runtime:

- The memory for objects is **dynamically allocated** at runtime by class methods that create new instances.
- Objects are **dynamically typed**. In source code (at compile time), any object pointer can be of type `id` no matter what the object's class is. The exact class of an `id` variable (and therefore its particular methods and data structure) isn't determined until the program runs.
- Messages and methods are **dynamically bound**, as described under "How Messaging Works" (page 61). A runtime procedure matches the method selector in the message to a method implementation that "belongs to" the receiver.

These features give object-oriented programs a great deal of flexibility and power, but there's a price to pay. Messages are somewhat slower than function calls, for example, (though not much slower due to the efficiency of the runtime system) and the compiler can't check the exact types (classes) of `id` variables.

To permit better compile-time type checking, and to make code more self-documenting, Objective-C allows objects to be statically typed with a class name rather than generically typed as `id`. It also lets you turn some of its object-oriented features off in order to shift operations from runtime back to compile time.

Static Typing

If a pointer to a class name is used in place of `id` in an object declaration,

```
Rectangle *thisObject;
```

the compiler restricts the value of the declared variable to be either an instance of the class named in the declaration or an instance of a class that inherits from the named class. In the example above, `thisObject` can only be a Rectangle of some kind.

Statically typed objects have the same internal data structures as objects declared to be `id`s. The type doesn't affect the object; it affects only the amount of information given to the compiler about the object and the amount of information available to those reading the source code.

Static typing also doesn't affect how the object is treated at runtime. Statically typed objects are dynamically allocated by the same class methods that create instances of type `id`. If Square is a subclass of Rectangle, the following code would still produce an object with all the instance variables of a Square, not just those of a Rectangle:

```
Rectangle *thisObject = [[Square alloc] init];
```

Messages sent to statically typed objects are dynamically bound, just as objects typed `id` are. The exact type of a statically typed receiver is still determined at runtime as part of the messaging process. A `display` message sent to `thisObject`

```
[thisObject display];
```

performs the version of the method defined in the Square class, not the one in its Rectangle superclass.

By giving the compiler more information about an object, static typing opens up possibilities that are absent for objects typed `id`:

- In certain situations, it allows for compile-time type checking.

- It can free objects from the restriction that identically named methods must have identical return and argument types.
- It permits you to use the structure pointer operator to directly access an object's instance variables.

The first two topics are discussed in the sections that follow. The third is covered in "Defining a Class" (page 54).

Type Checking

With the additional information provided by static typing, the compiler can deliver better type-checking services in two situations:

- When a message is sent to a statically typed receiver, the compiler can make sure the receiver can respond. A warning is issued if the receiver doesn't have access to the method named in the message.
- When a statically typed object is assigned to a statically typed variable, the compiler makes sure the types are compatible. A warning is issued if they're not.

An assignment can be made without warning, provided the class of the object being assigned is identical to, or inherits from, the class of the variable receiving the assignment. The following example illustrates this:

```
Shape     *aShape;
Rectangle *aRect;

aRect = [[Rectangle alloc] init];
aShape = aRect;
```

Here `aRect` can be assigned to `aShape` because a Rectangle is a kind of Shape—the Rectangle class inherits from Shape. However, if the roles of the two variables are reversed and `aShape` is assigned to `aRect`, the compiler generates a warning; not every Shape is a Rectangle. (For reference, see Figure 3-2 (page 46), which shows the class hierarchy including Shape and Rectangle.)

There's no check when the expression on either side of the assignment operator is an `id`. A statically typed object can be freely assigned to an `id`, or an `id` to a statically typed object. Because methods like `alloc` and `init` return `id`s, the compiler doesn't ensure that a compatible object is returned to a statically typed variable. The following code is error-prone, but is allowed nonetheless:

```
Rectangle *aRect;
aRect = [[Shape alloc] init];
```

Return and Argument Types

In general, methods in different classes that have the same selector (the same name) must also share the same return and argument types. This constraint is imposed by the compiler to allow dynamic binding. Because the class of a message receiver (and therefore class-specific details about the method it's asked to perform), can't be known at compile time, the compiler must treat all methods with the same name alike. When it prepares information on method return and argument types for the runtime system, it creates just one method description for each method selector.

However, when a message is sent to a statically typed object, the class of the receiver is known by the compiler. The compiler has access to class-specific information about the methods. Therefore, the message is freed from the restrictions on its return and argument types.

Static Typing to an Inherited Class

An instance can be statically typed to its own class or to any class that it inherits from. All instances, for example, can be statically typed as NSObject.

However, the compiler understands the class of a statically typed object only from the class name in the type designation, and it does its type checking accordingly. Typing an instance to an inherited class can therefore result in discrepancies between what the compiler thinks would happen at runtime and what actually happens.

For example, if you statically type a Rectangle instance as a Shape,

```
Shape *myRect = [[Rectangle alloc] init];
```

the compiler will treat it as a Shape. If you send the object a message to perform a Rectangle method,

```
BOOL solid = [myRect isFilled];
```

the compiler will complain. The `isFilled` method is defined in the Rectangle class, not in Shape.

However, if you send it a message to perform a method that the Shape class knows about,

```
[myRect display];
```

the compiler won't complain, even though Rectangle overrides the method. At runtime, Rectangle's version of the method is performed.

Similarly, suppose that the Upper class declares a `worry` method that returns a `double`,

```
- (double)worry;
```

and the Middle subclass of Upper overrides the method and declares a new return type:

```
- (int)worry;
```

If an instance is statically typed to the Upper class, the compiler will think that its `worry` method returns a `double`, and if an instance is typed to the Middle class, it will think that `worry` returns an `int`. Errors will obviously result if a Middle instance is typed to the Upper class. The compiler will inform the runtime system that a `worry` message sent to the object returns a `double`, but at runtime it actually returns an `int` and generates an error.

Static typing can free identically named methods from the restriction that they must have identical return and argument types, but it can do so reliably only if the methods are declared in different branches of the class hierarchy.

Getting a Method Address

The only way to circumvent dynamic binding is to get the address of a method and call it directly as if it were a function. This might be appropriate on the rare occasions when a particular method will be performed many times in succession and you want to avoid the overhead of messaging each time the method is performed.

With a method defined in the NSObject class, `methodForSelector:`, you can ask for a pointer to the procedure that implements a method, then use the pointer to call the procedure. The pointer that `methodForSelector:` returns must be carefully cast to the proper function type. Both return and argument types should be included in the cast.

The example below shows how the procedure that implements the `setFilled:` method might be called:

```
void (*setter)(id, SEL, BOOL);
int i;

setter = (void (*)(id, SEL, BOOL))[target
    methodForSelector:@selector(setFilled:)];
for ( i = 0; i < 1000, i++ )
    setter(targetList[i], @selector(setFilled:), YES);
```

The first two arguments passed to the procedure are the receiving object (`self`) and the method selector (`_cmd`). These arguments are hidden in method syntax but must be made explicit when the method is called as a function.

Using `methodForSelector:` to circumvent dynamic binding saves most of the time required by messaging. However, the savings will be significant only where a particular message is repeated many times, as in the `for` loop shown above.

Note that `methodForSelector:` is provided by the Cocoa runtime system; it's not a feature of the Objective-C language itself.

Getting an Object Data Structure

A fundamental tenet of object-oriented programming is that the data structure of an object is private to the object. Information stored there can be accessed only through messages sent to the object. Although it is generally considered a poor programming practice, there is a way to strip an object data structure of its "objectness" and treat it like any other C structure. This makes all the object's instance variables publicly available.

When given a class name as an argument, the `@defs()` directive produces the declaration list for an instance of the class. This list is useful only in declaring structures, so `@defs()` can appear only in the body of a structure declaration. This code, for example, declares a structure that's identical to the template for an instance of the Worker class:

```
struct workerDef {
    @defs(Worker)
} *public;
```

Here `public` is declared as a pointer to a structure that's essentially indistinguishable from a Worker instance. With a little help from a type cast, a Worker `id` can be assigned to the pointer. The object's instance variables can then be accessed publicly through the pointer:

```
id aWorker;
aWorker = [[Worker alloc] init];

public = (struct workerDef *)aWorker;
public->boss = nil;
```

This technique of turning an object into a structure makes all its instance variables public, whether they are declared `@private`, `@protected`, or `@public`.

Objects generally aren't designed with the expectation that they be turned into C structures. You may want to use `@defs()` for classes you define entirely yourself, but it should not be applied to classes found in a framework or to classes you define that inherit from framework classes.

Exception Handling and Thread Synchronization

Objective-C provides support for exception handling and thread synchronization, which are explained in "Handling Exceptions" (page 86) and "Synchronizing Thread Execution" (page 88). To turn on support for these features, use the `-fobjc-exceptions` switch of the GNU Compiler Collection (GCC) version 3.3 and later.

Note: Using either of these features in a program, renders the application runnable only in Mac OS X v10.3 and later because runtime support for exception handling and synchronization is not present in earlier versions of the software.

Handling Exceptions

The Objective-C language has an exception-handling syntax similar to that of Java and C++. Coupled with the use of the NSException, NSError, or custom classes, you can add robust error-handling to your programs.

The exception support revolves around four compiler directives: `@try`, `@catch`, `@throw`, and `@finally`. Code that can potentially throw an exception is enclosed in a `@try` block. `@catch()`blocks contain the exception-handling logic for exceptions thrown in a `@try`block. A `@finally`block contains code that must be executed whether an exception is thrown or not. You use the `@throw`directive to throw an exception, which is essentially a pointer to an Objective-C object. You can use NSException objects but are not limited to them.

The example below depicts a simple exception-handling algorithm:

```
Cup *cup = [[Cup alloc] init];

@try {
    [cup fill];
}
@catch (NSException *exception) {
    NSLog(@"main: Caught %@: %@", [exception name], [exception  reason]);
}
@finally {
    [cup release];
}
```

Throwing Exceptions

To throw an exception you must instantiate an object with the appropriate information, such as the exception name and the reason it was thrown.

```
NSException *exception = [NSException exceptionWithName:@"HotTeaException"
                            reason:@"The tea is too hot"  userInfo:nil];
@throw exception;
```

Inside a `@catch()` block, you can re-throw the caught exception using the `@throw` directive without an argument. This can help make your code more readable.

You can subclass NSException to implement specialized types of exceptions, such as file-system exceptions or communications exceptions.

Note: You are not limited to throwing NSException objects. You can throw any Objective-C object as an exception object. The NSException class provides methods that help in exception processing, but you can implement your own if you so desire.

Processing Exceptions

To catch an exception thrown in a `@try` block, use one or more `@catch()`blocks following the `@try` block. The `@catch()` blocks should be ordered from most-specific to the least-specific. That way you can tailor the processing of exceptions as groups, as shown in Listing 3-2.

Listing 3-2 An exception handler

```
@try {
    ...
}
@catch (CustomException *ce) {                                        // 1
    ...
}
@catch (NSException *ne) {                                            // 2
    // Perform processing necessary at this level.
    ...

    // Rethrow the exception so that it's handled at a higher level.
    @throw;                                                           // 3
}
@catch (id ue) {
    ...
}
@finally {                                                            // 4
    // Perform processing necessary whether an exception occurred  or not.
    ...
}
```

The following list describes the numbered code-lines:

1. Catches the most specific exception type.
2. Catches a more general exception type.
3. Re-throws the exception caught.

   To compartmentalize exception processing, you can nest exception handlers in a program. That way if a method or function catches an exception that it cannot process, it can rethrow it to the next exception handler.
4. Performs any clean-up processing that must always be performed, whether exceptions were thrown or not.

Synchronizing Thread Execution

Objective-C supports multithreading in applications. This means that two threads can try to modify the same object at the same time, a situation that can cause serious problems in a program. To protect sections of code from being executed by more than one thread at a time, Objective-C provides the `@synchronized()` directive.

The `@synchronized()`directive locks a section of code for use by a single thread. Other threads are blocked until the thread exits the protected code; that is, when execution continues past the last statement in the `@synchronized()` block.

The `@synchronized()` directive takes as its only argument any Objective-C object, including `self`. This object is known as a *mutual exclusion* semaphore or *mutex*. It allows a thread to lock a section of code to prevent its use by other threads. You should use separate semaphores to protect different critical sections of a program. It's safest to create all the mutual exclusion objects before the application becomes multithreaded to avoid race conditions.

Listing 3-3 is an example of code that uses `self` as the mutex to synchronize an instance method. This kind of thread synchronization works best with singletons unless you want to synchronize a method call at the instance level. For example, if two threads instantiate separate instances of the same class, they can both call the synchronized method at the same time. You can take a similar approach to synchronize a class method, where `self` refers to the class itself. In that case, only one thread at a time is allowed to execute the method because of classes' global scope.

Listing 3-3 Locking a method using self

```
- (void)criticalMethod
{
    @synchronized(self) {
        // Critical code.
        ...
    }
}
```

Listing 3-4 uses the current selector, `_cmd`, as the mutex. This kind of synchronization is beneficial only when the method being synchronized has a unique name. This is because no other object or class would be allowed to execute a different method with the same name until the current method ends.

Listing 3-4 Locking a method using _cmd

```
- (void)criticalMethod
{
    @synchronized(NSStringFromSelector(_cmd)) {
        // Critical code.
        ...
    }
}
```

Listing 3-5 shows a general approach. Before executing a critical process, the code obtains a semaphore from the Account class and uses it to lock the critical section. The Account class could create the semaphore in its `initialize` method.

Listing 3-5 Locking a method using a custom semaphore

```
Account *account = [Account accountFromString:[accountField stringValue]];

// Get the semaphore.
id accountSemaphore = [Account semaphore];

@synchronized(accountSemaphore) {
    // Critical code.
    ...
}
```

The Objective-C synchronization feature supports recursive and reentrant code. A thread can use a single semaphore several times in a recursive manner; other threads are blocked from using it until the thread releases all the locks obtained with it; that is, every `@synchronized()` block is exited normally or through an exception.

When code in an `@synchronized()` block throws an exception, the Objective-C runtime catches the exception, releases the semaphore (so that the protected code can be executed by other threads), and re-throws the exception to the next exception handler.

Using C++ With Objective-C

Apple's Objective-C compiler allows you to freely mix C++ and Objective-C code in the same source file. This Objective-C/C++ language hybrid is called Objective-C++. With it you can make use of existing C++ libraries from your Objective-C applications. Note that XCode requires that file names have a ".mm" extension for the Objective-C++ extensions to be enabled by the compiler.

Objective-C++ does not add C++ features to Objective-C classes, nor does it add Objective-C features to C++ classes. For example, you cannot use Objective-C syntax to call a C++ object, you cannot add constructors or destructors to an Objective-C object, and you cannot use the keywords `this` and `self` interchangeably. The class hierarchies are separate; a C++ class cannot inherit from an Objective-C class, and an Objective-C class cannot inherit from a C++ class. In addition, multi-language exception handling is not supported. That is, an exception thrown in Objective-C code cannot be caught in C++ code and, conversely, an exception thrown in C++ code cannot be caught in Objective-C code. For more information on exceptions in Objective-C, see "Exception Handling and Thread Synchronization" (page 86).

The next section discusses what you *can* do with Objective-C++.

Mixing Objective-C and C++ Language Features

In Objective-C++, you can call methods from either language in C++ code and in Objective-C methods. Pointers to objects in either language are just pointers, and as such can be used anywhere. For example, you can include pointers to Objective-C objects as data members of C++ classes, and you can include pointers to C++ objects as instance variables of Objective-C classes. Listing 3-6 (page 90) illustrates this.

Listing 3-6 Using C++ and Objective-C instances as instance variables

```
/* Hello.mm
 * Compile with: g++ -x objective-c++ -framework Foundation Hello.mm  -o hello
 */

#import <Foundation/Foundation.h>
class Hello {
    private:
        id _greeting_text;  // holds an NSString
    public:
        Hello() {
            _greeting_text = @"Hello, world!";
        }
        Hello(const char* greeting_text) {
            _greeting_text = [NSString stringWithUTF8String:greeting_text];
        }
        void say_hello() {
            printf("%s\n", [_greeting_text UTF8String]);
        }
};

@interface Greeting : NSObject {
    @private
        Hello *_hello;
}
- (id)init;
- (void)dealloc;
- (void)sayGreeting;
- (void)sayGreeting:(Hello*)greeting;
@end

@implementation Greeting
- (id)init {
    if (self = [super init]) {
        _hello = new Hello();
    }
    return self;
}
- (void)dealloc {
    delete _hello;
    [super dealloc];
}
- (void)sayGreeting {
    _hello->say_hello();
}
- (void)sayGreeting:(Hello*)greeting {
    greeting->say_hello();
}
```

```
@end

int main() {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    Greeting *greeting = [[Greeting alloc] init];
    [greeting sayGreeting];                        // > Hello, world!

    Hello *hello = new Hello("Bonjour, monde!");
    [greeting sayGreeting:hello];                  // > Bonjour, monde!

    delete hello;
    [greeting release];
    [pool release];
    return 0;
}
```

As you can declare C structs in Objective-C interfaces, you can also declare C++ classes in Objective-C interfaces. As with C structs, C++ classes defined within an Objective-C interface are globally-scoped, not nested within the Objective-C class. (This is consistent with the way in which standard C—though not C++—promotes nested struct definitions to file scope.)

To allow you to conditionalize your code based on the language variant, the Objective-C++ compiler defines both the `__cplusplus` and the `__OBJC__` preprocessor constants, as specified by (respectively) the C++ and Objective-C language standards.

As previously noted, Objective-C++ does not allow you to inherit C++ classes from Objective-C objects, nor does it allow you to inherit Objective-C classes from C++ objects.

```
class Base { /* ... */ };
@interface ObjCClass: Base ... @end // ERROR!
class Derived: public ObjCClass ... // ERROR!
```

Unlike Objective-C, objects in C++ are statically typed, with runtime polymorphism available as an exceptional case. The object models of the two languages are thus not directly compatible. More fundamentally, the layout of Objective-C and C++ objects in memory is mutually incompatible, meaning that it is generally impossible to create an object instance that would be valid from the perspective of both languages. Hence, the two type hierarchies cannot be intermixed.

You can declare a C++ class within an Objective-C class declaration. The compiler treats such classes as having been declared in the global namespace, as follows:

```
@interface Foo {
 class Bar { ... } // OK
}
@end

Bar *barPtr; // OK
```

Objective-C allows C structures (whether declared inside of an Objective-C declaration or not) to be used as instance variables.

```
@interface Foo {
   struct CStruct { ... };
   struct CStruct bigIvar; // OK
} ... @end
```

Objective-C++ similarly strives to allow C++ class instances to serve as instance variables. This is possible as long as the C++ class in question (along with all of its superclasses) does not have any virtual member functions defined. If any virtual member functions are present, the C++ class may not serve as an Objective-C instance variable.

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

struct Class0 { void foo(); };
struct Class1 { virtual void foo(); };
struct Class2 { Class2(int i, int j); };

@interface Foo : NSObject {
    Class0 class0;      // OK
    Class1 class1;      // ERROR!
    Class1 *ptr;        // OK—call 'ptr = new Class1()' from Foo'  init,
                        // 'delete ptr' from Foo's dealloc
    Class2 class2;      // WARNING - constructor not called!
...
@end
```

C++ requires each instance of a class containing virtual functions to contain a suitable virtual function table pointer. However, the Objective-C runtime cannot initialize the virtual function table pointer, because it is not familiar with the C++ object model. Similarly, the Objective-C runtime cannot dispatch calls to C++ constructors or destructors for those objects. If a C++ class has any user-defined constructors or destructors, they are not called. The compiler emits a warning in such cases.

Objective-C does not have a notion of nested namespaces. You cannot declare Objective-C classes within C++ namespaces, nor can you declare namespaces within Objective-C classes.

Objective-C classes, protocols, and categories cannot be declared inside a C++ template, nor can a C++ template be declared inside the scope of an Objective-C interface, protocol, or category.

However, Objective-C classes may serve as C++ template parameters. C++ template parameters can also be used as receivers or parameters (though not as selectors) in Objective-C message expressions.

C++ Lexical Ambiguities and Conflicts

There are a few identifiers that are defined in the Objective-C header files that every Objective-C program must include. These identifiers are `id`, `Class`, `SEL`, `IMP`, and `BOOL`.

Inside an Objective-C method, the compiler predeclares the identifiers `self` and `super`, similarly to the keyword `this` in C++. However, unlike the C++ `this` keyword, `self` and `super` are context-sensitive; they may be used as ordinary identifiers outside of Objective-C methods.

In the parameter list of methods within a protocol, there are five more context-sensitive keywords (`oneway`, `in`, `out`, `inout`, and `bycopy`). These are not keywords in any other contexts.

From an Objective-C programmer's point of view, C++ adds quite a few new keywords. You can still use C++ keywords as a part of an Objective-C selector, so the impact isn't too severe, but you cannot use them for naming Objective-C classes or instance variables. For example, even though `class` is a C++ keyword, you can still use the NSObject method `class`:

```
[foo class]; // OK
```

However, because it is a keyword, you cannot use `class` as the name of a variable:

```
NSObject *class; // Error
```

In Objective-C, the names for classes and categories live in separate namespaces. That is, both `@interface foo` and `@interface(foo)` can exist in the same source code. In Objective-C++, you can also have a category whose name matches that of a C++ class or structure.

Protocol and template specifiers use the same syntax for different purposes:

```
id<someProtocolName> foo;
TemplateType<SomeTypeName> bar;
```

To avoid this ambiguity, the compiler doesn't permit `id` to be used as a template name.

Finally, there is a lexical ambiguity in C++ when a label is followed by an expression that mentions a global name, as in:

```
label: ::global_name = 3;
```

The space after the first colon is required. Objective-C++ adds a similar case, which also requires a space:

```
receiver selector: ::global_c++_name;
```

The Runtime System

The Objective-C language defers as many decisions as it can from compile time and link time to runtime. Whenever possible, it does things dynamically. This means that the language requires not just a compiler, but also a runtime system to execute the compiled code. The runtime system acts as a kind of operating system for the Objective-C language; it's what makes the language work.

The following sections look in particular at three areas where the NSObject class provides a framework and defines conventions:

- Allocating and initializing new instances of a class, and deallocating instances when they're no longer needed
- Forwarding messages to another object
- Dynamically loading new modules into a running program

Additional conventions of the NSObject class are described in the NSObject class specification in the Foundation framework reference.

Other sections look at how you interact with the runtime at an abstract level; how you can use the Distributed Objects system for sending messages between objects in different address spaces; and how the compiler encodes the return and argument types for each method.

Interacting with the Runtime System

Objective-C programs interact with the runtime system at three distinct levels:

1. Through Objective-C source code.

   For the most part, the runtime system works automatically and behind the scenes. You use it just by writing and compiling Objective-C source code.

   When you compile code containing Objective-C classes and methods, the compiler creates the data structures and function calls that implement the dynamic characteristics of the language. The data structures capture information found in class and category definitions and in protocol declarations; they include the class and protocol objects discussed in "Extending Classes" (page 71), as well as method selectors, instance variable templates, and other information distilled from source code. The principal runtime function is the one that sends messages, as described in "How Messaging Works" (page 61). It's invoked by source-code message expressions.

2. Through the methods defined in the NSObject class of the Foundation framework.

   Most objects in Cocoa are subclasses of the NSObject class, so most objects inherit the methods it defines. (The notable exception is the NSProxy class; see "Forwarding" (page 108) for more information.)

   Some of the NSObject methods simply query the runtime system for information. These methods allow objects to perform introspection. Examples of such methods are the `class` method, which asks an object to identify its class; `isKindOfClass:` and `isMemberOfClass:`, which test an object's position in the inheritance hierarchy; `respondsToSelector:`, which indicates whether an object can accept a particular message; `conformsToProtocol:`, which indicates whether an object claims to implement the methods defined in a specific protocol; and `methodForSelector:`, which provides the address of a method's implementation. Methods like these give an object the ability to introspect about itself.

   All these methods were mentioned in previous chapters and are described in detail in the NSObject class specification in the Foundation framework reference.

3. Through direct calls to runtime functions.

   The runtime system is a dynamic shared library with a public interface consisting of a set of functions and data structures in the header files located within the directory `/usr/include/objc`. Many of these functions allow you to use plain C to replicate what the compiler does when you write Objective-C code. Others form the basis for functionality exported through the methods of the NSObject class. These functions make it possible to develop other interfaces to the runtime system and produce tools that augment the development environment; they're not needed when programming in Objective-C. However, a few of the runtime functions might on occasion be useful when writing an Objective-C program. All of these functions are documented in Objective-C Runtime Reference.

Because the NSObject class is at the root of the inheritance hierarchy of the Foundation framework, the methods it defines are usually inherited by all classes. Its methods therefore establish behaviors that are inherent to every instance and every class object. However, in a few cases, the NSObject class merely defines a template for how something should be done; it doesn't provide all the necessary code itself.

For example, the NSObject class defines a `description` instance method that returns a string describing the contents of the class. This is primarily used for debugging—the GDB `print-object` command prints the string returned from this method. NSObject's implementation of this method doesn't know what the class contains, so it returns a string with the name and address of the object. Subclasses of NSObject can implement this method to return more details. For example, the Foundation class NSArray returns a list of descriptions of the objects it contains.

Memory Management

This section describes memory management using the Cocoa frameworks; in particular, it discusses how to allocate and initialize objects, and when you should (and when you should not) release an object.

Cocoa uses a memory-management technique called **reference counting** (also known as *refcounting*), in which each entity that claims ownership of an object increments the object's reference count and later decrements the reference count when finished with the object; when the reference count reaches zero, the object is deallocated. This technique allows one instance of an object to be safely shared among several other objects. The next few sections describe Cocoa's implementation of reference counting.

Allocating and Initializing Objects

It takes two steps to create an object using Objective-C. You must:

- Dynamically allocate memory for the new object
- Initialize the newly allocated memory to appropriate values

An object isn't fully functional until both steps have been completed. Each step is accomplished by a separate method but typically in a single line of code:

```
id anObject = [[Rectangle alloc] init];
```

Separating allocation from initialization gives you individual control over each step so that each can be modified independently of the other. The following sections look first at allocation and then at initialization, and discuss how they are controlled and modified.

In Objective-C, memory for new objects is allocated using class methods defined in the NSObject class. NSObject defines two principal methods for this purpose, `alloc` and `allocWithZone:`.

```
+ (id)alloc;
+ (id)allocWithZone:(NSZone *)zone;
```

These methods allocate enough memory to hold all the instance variables for an object belonging to the receiving class. They don't need to be overridden and modified in subclasses.

The `alloc` and `allocWithZone:` methods initialize a newly allocated object's `isa` instance variable so that it points to the object's class (the class object). All other instance variables are set to `0`. Usually, an object needs to be more specifically initialized before it can be safely used.

This initialization is the responsibility of class-specific instance methods that, by convention, begin with the abbreviation "init". If the method takes no arguments, the method name is just those four letters, `init`. If it takes arguments, labels for the arguments follow the "init" prefix. For example, an NSView can be initialized with an `initWithFrame:` method.

Every class that declares instance variables must provide an `init...` method to initialize them. The NSObject class declares the `isa` variable and defines an `init` method. However, since `isa` is initialized when memory for an object is allocated, all NSObject's `init` method does is return `self`. NSObject declares the method mainly to establish the naming convention described earlier.

The Returned Object

An `init...` method normally initializes the instance variables of the receiver, then returns it. It's the responsibility of the method to return an object that can be used without error.

However, in some cases, this responsibility can mean returning a different object than the receiver. For example, if a class keeps a list of named objects, it might provide an `initWithName:` method to initialize new instances. If there can be no more than one object per name, `initWithName:` might refuse to assign the same name to two objects. When asked to assign a new instance a name that's already being used by another object, it might free the newly allocated instance and return the other object—thus ensuring the uniqueness of the name while at the same time providing what was asked for, an instance with the requested name.

In a few cases, it might be impossible for an `init...` method to do what it's asked to do. For example, an `initFromFile:` method might get the data it needs from a file passed as an argument. If the file name it's passed doesn't correspond to an actual file, it won't be able to complete the initialization. In such a case, the `init...` method could free the receiver and return `nil`, indicating that the requested object can't be created.

Because an `init...` method might return an object other than the newly allocated receiver, or even return `nil`, it's important that programs use the value returned by the initialization method, not just that returned by `alloc` or `allocWithZone:`. The following code is very dangerous, since it ignores the return of `init`.

```
id anObject = [SomeClass alloc];
[anObject init];
[anObject someOtherMessage];
```

Instead, to safely initialize an object, you should combine allocation and initialization messages in one line of code.

```
id anObject = [[SomeClass alloc] init];
[anObject someOtherMessage];
```

If there's a chance that the `init...` method might return `nil`, the you should check the return value before proceeding:

```
id anObject = [[SomeClass alloc] init];
if ( anObject )
    [anObject someOtherMessage];
else
    ...
```

Arguments

An `init...` method must ensure that all of an object's instance variables have reasonable values. This doesn't mean that it needs to provide an argument for each variable. It can set some to default values or depend on the fact that (except for `isa`) all bits of memory allocated for a new object are set to `0`. For example, if a class requires its instances to have a name and a data source, it might provide an `initWithName:fromFile:` method, but set nonessential instance variables to arbitrary values or allow them to have the null values set by default. It could then rely on methods like `setEnabled:`, `setFriend:`, and `setDimensions:` to modify default values after the initialization phase had been completed.

Any `init...` method that takes arguments must be prepared to handle cases where an inappropriate value is passed.

Coordinating Classes

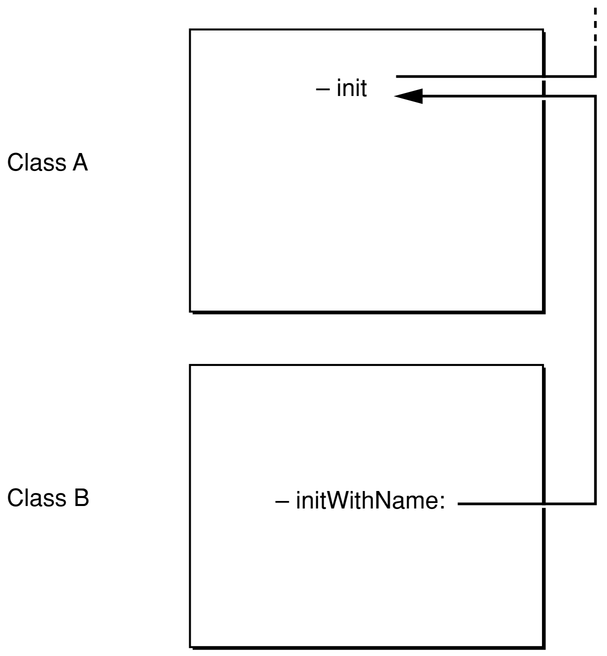
Every class that declares instance variables must provide an `init...` method to initialize them (unless the variables require no initialization). The `init...` methods the class defines initialize only those variables declared in the class. Inherited instance variables are initialized by sending a message to `super` to perform an initialization method defined somewhere farther up the inheritance hierarchy:

```
- initWithName:(char *)string
{
    if ( self = [super init] ) {
        name = (char *)NSZoneMalloc([self zone],
            strlen(string) + 1);
        strcpy(name, string);
    }
    return self;
}
```

The message to `super` chains together initialization methods in all inherited classes. Because it comes first, it ensures that superclass variables are initialized before those declared in subclasses. For example, a Rectangle object must be initialized as an NSObject, a Graphic, and a Shape before it's initialized as a Rectangle.

The connection between the `initWithName:` method illustrated above and the inherited `init` method it incorporates is illustrated in Figure 4-1:

Figure 4-1 Incorporating an Inherited Initialization Method

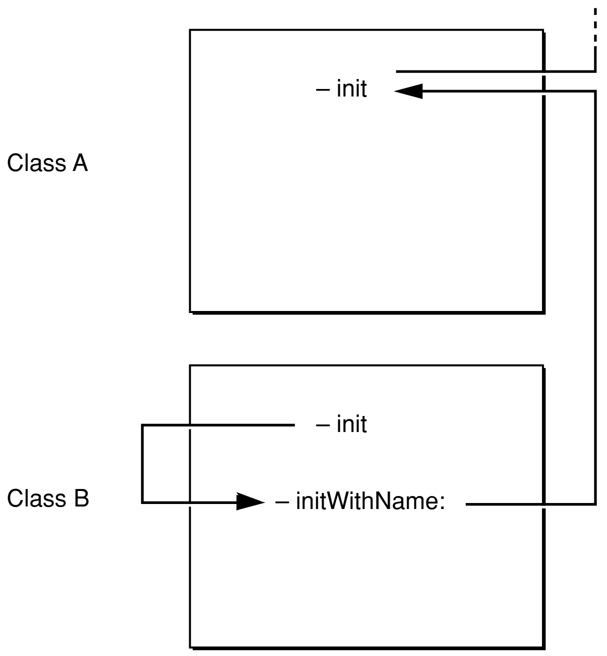


A class must also make sure that all inherited initialization methods work. For example, if class A defines an `init` method and its subclass B defines an `initWithName:` method, as shown in Figure 4-1, B must also make sure that an `init` message successfully initializes B instances. The easiest way to do that is to replace the inherited `init` method with a version that invokes `initWithName::`

```
- init
{
    return [self initWithName:"default"];
}
```

The `initWithName:` method would, in turn, invoke the inherited method, as shown earlier. Figure 4-2 includes B's version of `init`:

Figure 4-2 Covering an Inherited Initialization Model



Covering inherited initialization methods makes the class you define more portable to other applications. If you leave an inherited method uncovered, someone else may use it to produce incorrectly initialized instances of your class.

The Designated Initializer

In the example above, `initWithName:` would be the **designated initializer** for its class (class B). The designated initializer is the method in each class that guarantees inherited instance variables are initialized (by sending a message to `super` to perform an inherited method). It's also the method that does most of the work, and the one that other initialization methods in the same class invoke. It's a Cocoa convention that the designated initializer is always the method that allows the most freedom to determine the character of a new instance (usually this is the one with the most arguments, but not always).

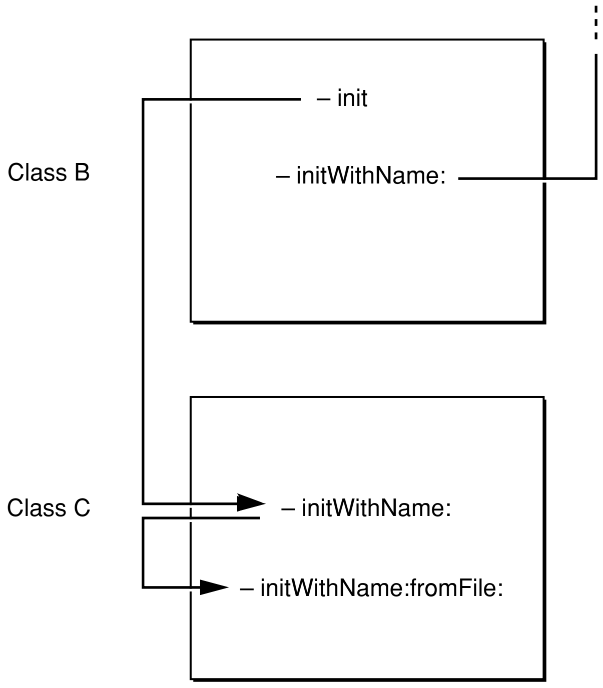
It's important to know the designated initializer when defining a subclass. For example, suppose we define class C, a subclass of B, and implement an `initWithName:fromFile:` method. In addition to this method, we have to make sure that the inherited `init` and `initWithName:` methods also work for instances of C. This can be done just by covering B's `initWithName:` with a version that invokes `initWithName:fromFile:`.

```
- initWithName:(char *)string
```

```
{
    return [self initWithName:string fromFile:NULL];
}
```

For an instance of the C class, the inherited `init` method invokes this new version of `initWithName:` which invokes `initWithName:fromFile:`. The relationship between these methods is shown in Figure 4-3:

Figure 4-3 Covering the Designated Initializer



This figure omits an important detail. The `initWithName:fromFile:` method, being the designated initializer for the C class, sends a message to `super` to invoke an inherited initialization method. But which of B's methods should it invoke, `init` or `initWithName:`? It can't invoke `init`, for two reasons:

- Circularity would result (`init` invokes C's `initWithName:`, which invokes `initWithName:fromFile:`, which invokes `init` again).
- It won't be able to take advantage of the initialization code in B's version of `initWithName:`.

Therefore, `initWithName:fromFile:` must invoke `initWithName:`:

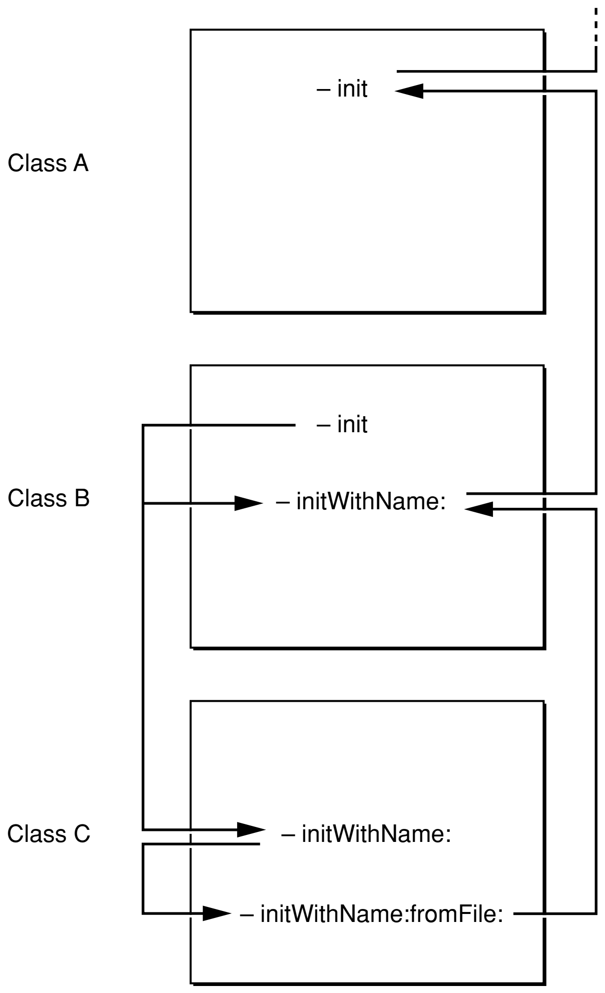
```
- initWithName:(char *)string fromFile:(char *)pathname
{
    if ( self = [super initWithName:string] )
        ...
}
```

General Principle: The designated initializer in a class must, through a message to `super`, invoke the designated initializer in a superclass.

Designated initializers are chained to each other through messages to `super`, while other initialization methods are chained to designated initializers through messages to `self`.

Figure 4-4 shows how all the initialization methods in classes A, B, and C are linked. Messages to `self` are shown on the left and messages to `super` are shown on the right.

Figure 4-4 Initialization Chain



Note that B's version of `init` sends a message to `self` to invoke the `initWithName:` method. Therefore, when the receiver is an instance of the B class, it invokes B's version of `initWithName:`, and when the receiver is an instance of the C class, it invokes C's version.

Combining Allocation and Initialization

In Cocoa, some classes define creation methods that combine the two steps of allocating and initializing to return new, initialized instances of the class. These methods typically take the form + *className...* where *className* is the name of the class. For instance, NSString has the following methods (among others):

```
+ (NSString *)stringWithCString:(const char *)bytes;
+ (NSString *)stringWithFormat:(NSString *)format, ...;
```

Similarly, NSArray defines the following class methods that combine allocation and initialization:

```
+ (id)array;
+ (id)arrayWithObject:(id)anObject;
+ (id)arrayWithObjects:(id)firstObj, ...;
```

Instances created with any of these methods are deallocated automatically (as described in "Marking Objects for Later Release" (page 108)), so you don't have to release them unless you have retained them (as described in "Retaining Objects" (page 104)). Usually there are equivalent `-init...` methods provided along with these conveniences.

Methods that combine allocation and initialization are particularly valuable if the allocation must somehow be informed by the initialization. For example, if the data for the initialization is taken from a file, and the file might contain enough data to initialize more than one object, it would be impossible to know how many objects to allocate until the file is opened. In this case, you might implement a `listFromFile:` method that takes the name of the file as an argument. It would open the file, see how many objects to allocate, and create a List object large enough to hold all the new objects. It would then allocate and initialize the objects from data in the file, put them in the List, and finally return the List.

It also makes sense to combine allocation and initialization in a single method if you want to avoid the step of blindly allocating memory for a new object that you might not use. As mentioned in "The Returned Object" (page 97), an `init...` method might sometimes substitute another object for the receiver. For example, when `initWithName:` is passed a name that's already taken, it might free the receiver and in its place return the object that was previously assigned the name. This means, of course, that an object is allocated and freed immediately without ever being used.

If the code that determines whether the receiver should be initialized is placed inside the method that does the allocation instead of inside `init...`, you can avoid the step of allocating a new instance when one isn't needed.

In the following example, the `soloist` method ensures that there's no more than one instance of the Soloist class. It allocates and initializes an instance only once:

```
+ soloist
{
    static Soloist *instance = nil;

    if ( instance == nil )
    {
        instance = [[self alloc] init];
    }
    return instance;
}
```

Object Ownership

In an Objective-C program, objects are constantly creating and disposing of other objects. Much of the time an object creates things for private use and can dispose of them as it needs. However, when an object passes something to another object through a method invocation, the lines of ownership—and responsibility for disposal—blur.

Suppose, for example, that you have a Thingamajig object that contains a number of Sprocket objects, which another object accesses with this method:

```
- (NSArray *) sprockets
```

This declaration says nothing about who should release the returned array. If the Thingamajig object returned an instance variable, it's responsible; if the Thingamajig created an array and returned it, the recipient is responsible. This problem applies both to objects returned by a method and objects passed in as arguments to a method.

Ideally, a body of code should never be concerned with releasing something it didn't create. Cocoa therefore sets this policy:

- If you:
  - ❏ Create an object (using `alloc` or `allocWithZone:`)
  - ❏ Copy an object (using `copy`, `copyWithZone:`, `mutableCopy` or `mutableCopyWithZone:`)
  - ❏ Retain an object (using `[retain]`)

  you must release it (using `[release]`).
- If you did not directly create or copy the object as described above, you do not own it and should not release it.

When you write a method that creates and returns an object, that method is responsible for releasing the object. However, it's clearly not fruitful to dispose of an object before the recipient of the object gets it. What is needed is a way to mark an object for release at a later time, so that it's properly disposed of after the recipient has had a chance to use it. Cocoa provides just such a mechanism.

Retaining Objects

There are times when you don't want a received object to be disposed of; for example, you may need to cache the object in an instance variable. In this case, only you know when the object is no longer needed, so you need the power to ensure that the object is not disposed of while you are still using it. You do this with the `retain` method, which stays the effect of a pending `autorelease` (or preempts a later `release` or `autorelease` message; see "Deallocation" (page 106) for details on the `autorelease` message). By retaining an object you ensure that it isn't deallocated until you're done with it. For example, if your object allows its main Sprocket to be set, you might want to retain that Sprocket like this:

```
- (void)setMainSprocket:(Sprocket *)newSprocket
{
    [mainSprocket autorelease];
    mainSprocket = [newSprocket retain]; /* Claim the new Sprocket.  */
    return;
}
```

Now, `setMainSprocket:` might get invoked with a Sprocket that the invoker intends to keep around, which means your object would be sharing the Sprocket with that other object. If that object changes the Sprocket, your object's main Sprocket changes. You might want that, but if your Thingamajig needs to have its own Sprocket, the method should make a private copy:

```
- (void)setMainSprocket:(Sprocket *)newSprocket
{
    [mainSprocket autorelease];
    mainSprocket = [newSprocket copy]; /* Get a private copy. */
    return;
}
```

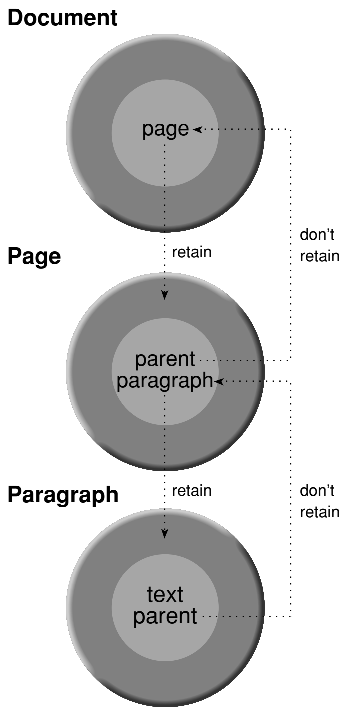
Note that both of these methods autorelease the original main sprocket, so they don't need to check that the original main sprocket and the new one are the same. If they simply released the original when it was the same as the new one, that sprocket would be released and possibly deallocated, causing an error as soon as it was retained or copied. Although they could store the old main sprocket and release it later, that kind of code tends to be slightly more complex. For example:

```
- (void)setMainSprocket:(Sprocket *)newSprocket
{
    Sprocket *oldSprocket = mainSprocket;
    mainSprocket = [newSprocket copy];
    [oldSprocket release];
    return;
}
```

Handling Cyclical References

In general, you retain all objects that you create. However, in some cases you may have two objects with cyclical references; that is, each object contains an instance variable that refers to the other object. For example, consider a text program with the object relationships shown in Figure 4-5. The Document object creates a Page object for each page in the document. Each Page object has an instance variable that keeps track of which document it's in. If the Document object retained the Page object and the Page object retained the Document object, neither object would ever be released. The Document's reference count can't become `0` until the Page object is released, and the Page object isn't released until the Document object is deallocated.

Figure 4-5 Retaining Objects



The solution to the problem of retain cycles is that the "parent" object should retain its "children," but the children should not retain their parents. So, in Figure 4-5 the document object retains its page objects but the page object does not retain the document object.

Deallocation

The NSObject class defines a `dealloc` method that relinquishes the memory originally allocated for an object. You must never invoke `dealloc` directly; you should instead invoke either the `release` method or the `autorelease` method to decrement the method's refcount. When the object's refcount reaches zero, the `release` method invokes `dealloc`. In some situations, you don't release an object at all. The rules for object deallocation are as follows:

- If you obtain an instance of an object via a method with `alloc` or `copy` in its name, or if you send an instance a `retain` message, you have a reference to the instance and you will need to release it when you're finished with it.
- If you obtain an instance of an object any other way (exceptions include multithreaded applications and some Distributed Objects situations), you must not release it and you must not hang onto it beyond the scope of the current method (if you need to hang onto it, you must invoke its `retain` method).
- `autorelease` just means "release this later" (for some definition of later).

Releasing Shared Objects

Cocoa's ownership policy limits itself to the question of when you have to dispose of an object; it doesn't specify that any object received in a method must remain valid throughout that method's scope. A received object nearly always becomes invalid when its owner is released, and usually becomes invalid when its owner reassigns the instance variable holding that object. Any method other than `release` that immediately disposes of an object should be documented as doing so.

For example, if you ask for an object's main sprocket and then release the object, you have to consider the main sprocket gone, because it belonged to the object. Similarly, if you ask for the main sprocket and then send `setMainSprocket:` you can't assume that the sprocket you received remains valid:

```
Sprocket *oldMainSprocket;
Sprocket *newMainSprocket;

oldMainSprocket = [myObject mainSprocket];

/* If this releases the original Sprocket... */
[myObject setMainSprocket:newMainSprocket];

/* ...then this causes the application to crash. */
[oldMainSprocket anyMessage];
```

`setMainSprocket:` may release the object's original main sprocket, possibly rendering it invalid. Sending any message to the invalid sprocket would then cause your application to crash. If you need to use an object after disposing of its owner or rendering it invalid by some other means, you can retain and autorelease it before sending the message that would invalidate it:

```
Sprocket *oldMainSprocket;
Sprocket *newMainSprocket;

oldMainSprocket = [[[myObject mainSprocket] retain] autorelease];
[myObject setMainSprocket:newMainSprocket];
[oldMainSprocket anyMessage];
```

Retaining and autoreleasing `oldMainSprocket` guarantees that it will remain valid throughout your scope, even though its owner may release it when you send `setMainSprocket:`.

Releasing Instance Variables

The purpose of a `dealloc` message is to deallocate all the memory occupied by the receiver. NSObject's version of the method deallocates the receiver's instance variables, but doesn't follow any variable that points to other memory. If the receiver allocated any additional memory—to store a character string or an array of structures, for example—that memory must also be deallocated (unless it's shared by other objects). Similarly, if the receiver is served by another object that would be rendered useless in its absence, that object must also be deallocated.

Therefore, it's necessary for subclasses to override NSObject's version of `dealloc` and implement a version that deallocates all of the other memory the object occupies. Every class that has its objects allocate additional memory must have its own `dealloc` method. Each version of `dealloc` ends with a message to `super` to perform an inherited version of the method, as illustrated in the following example:

```
- dealloc {
    [companion release];
    free(privateMemory);
    vm_deallocate(task_self(), sharedMemory, memorySize);
    [super dealloc];
}
```

By working its way up the inheritance hierarchy, every `dealloc` message eventually invokes NSObject's version of the of the method.

Marking Objects for Later Release

The `autorelease` method, defined by NSObject, marks the receiver for later release. By autoreleasing an object—that is, by sending it an `autorelease` message—you declare that you don't need the object to exist beyond the scope you sent `autorelease` in. When your code completely finishes executing and control returns to the application object (that is, at the end of the event loop), the application object releases the object. The `sprockets` method above could be implemented in this way:

```
- (NSArray *)sprockets
{
    NSArray *array;

    array = [[NSArray alloc] initWithObjects:mainSprocket,
                             auxiliarySprocket, nil];
    return [array autorelease];
}
```

When another method gets the array of Sprockets, that method can assume that the array is disposed of when it's no longer needed, but can still be safely used anywhere within its scope (with certain exceptions; see "Releasing Shared Objects" (page 107)). It can even return the array to its invoker, since the application object defines the bottom of the call stack for your code. The `autorelease` method thus allows every object to use other objects without worrying about disposing of them.

Note: Just as it's an error to release an object after it's already been deallocated, it's an error to send so many `autorelease` messages that the object would later be released after it had already been deallocated. You should send `release` or `autorelease` to an object only as many times as are allowed by its creation (one) plus the number of `retain` messages you have sent it (`retain` messages are described in "Object Ownership" (page 104)).

Forwarding

Sending a message to an object that does not handle that message is an error. However, before announcing the error, the runtime system gives the receiving object a second chance to handle the message. It sends the object a `forwardInvocation:` message with an NSInvocation object as its sole argument—the NSInvocation object encapsulates the original message and the arguments that were passed with it.

You can implement a `forwardInvocation:` method to give a default response to the message, or to avoid the error in some other way. As its name implies, `forwardInvocation:` is commonly used to forward the message to another object.

To see the scope and intent of forwarding, imagine the following scenarios: Suppose, first, that you're designing an object that can respond to a message called `negotiate`, and you want its response to include the response of another kind of object. You could accomplish this easily by passing a `negotiate` message to the other object somewhere in the body of the `negotiate` method you implement.

Take this a step further, and suppose that you want your object's response to a `negotiate` message to be exactly the response implemented in another class. One way to accomplish this would be to make your class inherit the method from the other class. However, it might not be possible to arrange things this way. There may be good reasons why your class and the class that implements `negotiate` are in different branches of the inheritance hierarchy.

Even if your class can't inherit the `negotiate` method, you can still "borrow" it by implementing a version of the method that simply passes the message on to an instance of the other class:

```
- negotiate
{
    if ( [someOtherObject respondsTo:@selector(negotiate)] )
        return [someOtherObject negotiate];
    return self;
}
```

This way of doing things could get a little cumbersome, especially if there were a number of messages you wanted your object to pass on to the other object. You'd have to implement one method to cover each method you wanted to borrow from the other class. Moreover, it would be impossible to handle cases where you didn't know, at the time you wrote the code, the full set of messages you might want to forward. That set might depend on events at runtime, and it might change as new methods and classes are implemented in the future.

The second chance offered by a `forwardInvocation:` message provides a less ad hoc solution to this problem, and one that's dynamic rather than static. It works like this: When an object can't respond to a message because it doesn't have a method matching the selector in the message, the runtime system informs the object by sending it a `forwardInvocation:` message. Every object inherits a `forwardInvocation:` method from the NSObject class. However, NSObject's version of the method simply invokes `doesNotRecognizeSelector:`. By overriding NSObject's version and implementing your own, you can take advantage of the opportunity that the `forwardInvocation:` message provides to forward messages to other objects.

To forward a message, all a `forwardInvocation:` method needs to do is:

- Determine where the message should go, and
- Send it there with its original arguments.

The message can be sent with the `invokeWithTarget:` method:

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    if ([someOtherObject respondsToSelector:
            [anInvocation selector]])
        [anInvocation invokeWithTarget:someOtherObject];
    else
        [super forwardInvocation:anInvocation];
}
```

The return value of the message that's forwarded is returned to the original sender. All types of return values can be delivered to the sender, including `id`s, structures, and double-precision floating-point numbers.

A `forwardInvocation:` method can act as a distribution center for unrecognized messages, parceling them out to different receivers. Or it can be a transfer station, sending all messages to the same destination. It can translate one message into another, or simply "swallow" some messages so there's no response and no error. A `forwardInvocation:` method can also consolidate several messages into a single response. What `forwardInvocation:` does is up to the implementor. However, the opportunity it provides for linking objects in a forwarding chain opens up possibilities for program design.

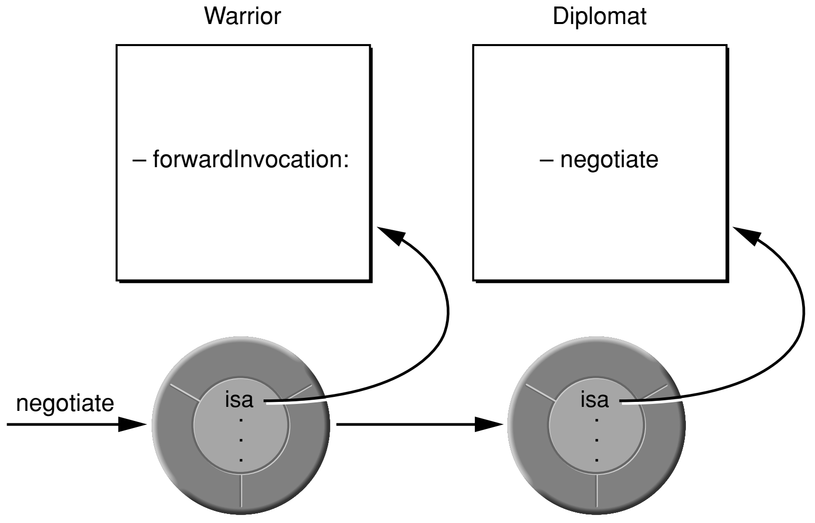
Note: The `forwardInvocation:` method gets to handle messages only if they don't invoke an existing method in the nominal receiver. If, for example, you want your object to forward `negotiate` messages to another object, it can't have a `negotiate` method of its own. If it does, the message will never reach `forwardInvocation:`.

For more information on forwarding and invocations, see the NSInvocation class specification in the Foundation framework reference.

Forwarding and Multiple Inheritance

Forwarding mimics inheritance, and can be used to lend some of the effects of multiple inheritance to Objective-C programs. As shown in Figure 4-6 (page 110), an object that responds to a message by forwarding it appears to borrow or "inherit" a method implementation defined in another class.

Figure 4-6 Forwarding



In this illustration, an instance of the Warrior class forwards a `negotiate` message to an instance of the Diplomat class. The Warrior will appear to negotiate like a Diplomat. It will seem to respond to the `negotiate` message, and for all practical purposes it does respond (although it's really a Diplomat that's doing the work).

The object that forwards a message thus "inherits" methods from two branches of the inheritance hierarchy—its own branch and that of the object that responds to the message. In the example above, it appears as if the Warrior class inherits from Diplomat as well as its own superclass.

Forwarding addresses most needs that lead programmers to value multiple inheritance. However, there's an important difference between the two: Multiple inheritance combines different capabilities in a single object. It tends toward large, multifaceted objects. Forwarding, on the other hand, assigns separate responsibilities to disparate objects. It decomposes problems into smaller objects, but associates those objects in a way that's transparent to the message sender.

Surrogate Objects

Forwarding not only mimics multiple inheritance, it also makes it possible to develop lightweight objects that represent or "cover" more substantial objects. The surrogate stands in for the other object and funnels messages to it.

The proxy discussed in "Remote Messaging" (page 113) is such a surrogate. A proxy takes care of the administrative details of forwarding messages to a remote receiver, making sure argument values are copied and retrieved across the connection, and so on. But it doesn't attempt to do much else; it doesn't duplicate the functionality of the remote object but simply gives the remote object a local address, a place where it can receive messages in another application.

Other kinds of surrogate objects are also possible. Suppose, for example, that you have an object that manipulates a lot of data—perhaps it creates a complicated image or reads the contents of a file on disk. Setting this object up could be time-consuming, so you prefer to do it lazily—when it's really needed or when system resources are temporarily idle. At the same time, you need at least a placeholder for this object in order for the other objects in the application to function properly.

In this circumstance, you could initially create, not the full-fledged object, but a lightweight surrogate for it. This object could do some things on its own, such as answer questions about the data, but mostly it would just hold a place for the larger object and, when the time came, forward messages to it. When the surrogate's `forwardInvocation:` method first receives a message destined for the other object, it would ensure that the object existed and would create it if it didn't. All messages for the larger object go through the surrogate, so, as far as the rest of the program is concerned, the surrogate and the larger object would be the same.

Forwarding and Inheritance

Although forwarding mimics inheritance, the NSObject class never confuses the two. Methods like `respondsToSelector:` and `isKindOfClass:` look only at the inheritance hierarchy, never at the forwarding chain. If, for example, a Warrior object is asked whether it responds to a `negotiate` message,

```
if ( [aWarrior respondsToSelector:@selector(negotiate)] )
    ...
```

the answer is `NO`, even though it can receive `negotiate` messages without error and respond to them, in a sense, by forwarding them to a Diplomat. (See Figure 4-6 (page 110).)

In many cases, `NO` is the right answer. But it may not be. If you use forwarding to set up a surrogate object or to extend the capabilities of a class, the forwarding mechanism should probably be as transparent as inheritance. If you want your objects to act as if they truly inherited the behavior of the objects they forward messages to, you'll need to re-implement the `respondsToSelector:` and `isKindOfClass:` methods to include your forwarding algorithm:

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
    if ( [super respondsToSelector:aSelector] )
        return YES;
    else {
        /* Here, test whether the aSelector message can     *
         * be forwarded to another object and whether that  *
         * object can respond to it. Return YES if it can.  */
    }
    return NO;
}
```

In addition to `respondsToSelector:` and `isKindOfClass:`, the `instancesRespondToSelector:` method should also mirror the forwarding algorithm. If protocols are used, the `conformsToProtocol:` method should likewise be added to the list. Similarly, if an object forwards any remote messages it receives, it should have a version of `methodSignatureForSelector:` that can return accurate descriptions of the methods that ultimately respond to the forwarded messages.

You might consider putting the forwarding algorithm somewhere in private code and have all these methods, `forwardInvocation:` included, call it.

Note: This is an advanced technique, suitable only for situations where no other solution is possible. It is not intended as a replacement for inheritance. If you must make use of this technique, make sure you fully understand the behavior of the class doing the forwarding and the class you're forwarding to.

The methods mentioned in this section are described in the NSObject class specification in the Foundation framework reference. For information on `invokeWithTarget:`, see the NSInvocation class specification in the Foundation framework reference.

Dynamic Loading

An Objective-C program can load and link new classes and categories while it's running. The new code is incorporated into the program and treated identically to classes and categories loaded at the start.

Dynamic loading can be used to do a lot of different things. For example, the various modules in the System Preferences application are dynamically loaded.

In the Cocoa environment, dynamic loading is commonly used to allow applications to be customized. Others can write modules that your program loads at runtime—much as Interface Builder loads custom palettes and the Mac OS X System Preferences application loads custom preference modules. The loadable modules extend what your application can do. They contribute to it in ways that you permit but could not have anticipated or defined yourself. You provide the framework, but others provide the code.

Although there is a runtime function that performs dynamic loading of Objective-C modules in Mach-O files (`objc_loadModules`, defined in `objc/objc-load.h`), Cocoa's NSBundle class provides a significantly more convenient interface for dynamic loading—one that's object-oriented and integrated with related services. See the NSBundle class specification in the Foundation framework reference for information on the NSBundle class and its use. See Mach-O File Format Reference for information on Mach-O files.

Remote Messaging

Like most other programming languages, Objective-C was initially designed for programs that are executed as a single process in a single address space.

Nevertheless, the object-oriented model, where communication takes place between relatively self-contained units through messages that are resolved at runtime, would seem well suited for interprocess communication as well. It's not hard to imagine Objective-C messages between objects that reside in different address spaces (that is, in different tasks) or in different threads of execution of the same task.

For example, in a typical server-client interaction, the client task might send its requests to a designated object in the server, and the server might target specific client objects for the notifications and other information it sends.

Or imagine an interactive application that needs to do a good deal of computation to carry out a user command. It could simply display a dialog telling the user to wait while it was busy, or it could isolate the processing work in a subordinate task, leaving the main part of the application free to accept user input. Objects in the two tasks would communicate through Objective-C messages.

Similarly, several separate processes could cooperate on the editing of a single document. There could be a different editing tool for each type of data in the document. One task might be in charge of presenting a unified onscreen user interface and of sorting out which user instructions are the responsibility of the various editing tools. Each cooperating task could be written in Objective-C, with Objective-C messages being the vehicle of communication between the user interface and the tools and between one tool and another.

Distributed Objects

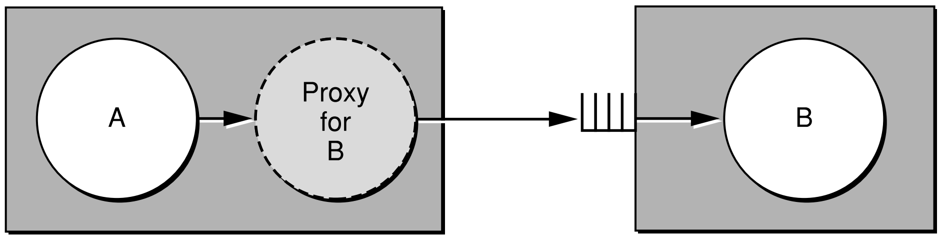
Remote messaging in Objective-C requires a runtime system that can establish connections between objects in different address spaces, recognize when a message is intended for an object in a remote address space, and transfer data from one address space to another. It must also mediate between the separate schedules of the two tasks; it has to hold messages until their remote receivers are free to respond to them.

Cocoa includes a **distributed objects** architecture that is essentially this kind of extension to the runtime system. Using distributed objects, you can send Objective-C messages to objects in other tasks or have messages executed in other threads of the same task. (When remote messages are sent between two threads of the same task, the threads are treated exactly like threads in different tasks.) Note that Cocoa's distributed objects system is built on top of the runtime system; it doesn't alter the fundamental behavior of your Cocoa objects.

To send a remote message, an application must first establish a connection with the remote receiver. Establishing the connection gives the application a proxy for the remote object in its own address space. It then communicates with the remote object through the proxy. The proxy assumes the identity of the remote object; it has no identity of its own. The application is able to regard the proxy as if it were the remote object; for most purposes, it is the remote object.

Remote messaging is illustrated in Figure 4-7 (page 114), where object A communicates with object B through a proxy, and messages for B wait in a queue until B is ready to respond to them:

Figure 4-7 Remote Messages



The sender and receiver are in different tasks and are scheduled independently of each other. So there's no guarantee that the receiver is free to accept a message when the sender is ready to send it. Therefore, arriving messages are placed in a queue and retrieved at the convenience of the receiving application.

A proxy doesn't act on behalf of the remote object or need access to its class. It isn't a copy of the object, but a lightweight substitute for it. In a sense, it's transparent; it simply passes the messages it receives on to the remote receiver and manages the interprocess communication. Its main function is to provide a local address for an object that wouldn't otherwise have one. A proxy isn't fully transparent, however. For instance, a proxy doesn't allow you to directly set and get an object's instance variables.

A remote receiver is typically anonymous. Its class is hidden inside the remote application. The sending application doesn't need to know how that application is designed or what classes it uses. It doesn't need to use the same classes itself. All it needs to know is what messages the remote object responds to.

Because of this, an object that's designated to receive remote messages advertises its interface in a formal protocol. Both the sending and the receiving application declare the protocol—they both import the same protocol declaration. The receiving application declares it because the remote object must conform to the protocol. The sending application declares it to inform the compiler about the messages it sends and because it may use the `conformsToProtocol:` method and the `@protocol()` directive to test the remote receiver. The sending application doesn't have to implement any of the methods in the protocol; it declares the protocol only because it initiates messages to the remote receiver.

The distributed objects architecture, including the NSProxy and NSConnection classes, is documented in the Foundation framework reference.

Language Support

Remote messaging raises not only a number of intriguing possibilities for program design, it also raises some interesting issues for the Objective-C language. Most of the issues are related to the efficiency of remote messaging and the degree of separation that the two tasks should maintain while they're communicating with each other.

So that programmers can give explicit instructions about the intent of a remote message, Objective-C defines six type qualifiers that can be used when declaring methods inside a formal protocol:

```
oneway
in
out
inout
bycopy
byref
```

These modifiers are restricted to formal protocols; they can't be used inside class and category declarations. However, if a class or category adopts a protocol, its implementation of the protocol methods can use the same modifiers that are used to declare the methods.

The following sections explain how these modifiers are used.

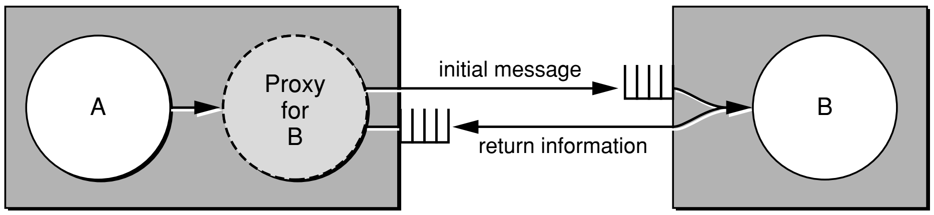
Synchronous and Asynchronous Messages

Consider first a method with just a simple return value:

```
- (BOOL)canDance;
```

When a `canDance` message is sent to a receiver in the same application, the method is invoked and the return value provided directly to the sender. But when the receiver is in a remote application, two underlying messages are required—one message to get the remote object to invoke the method, and the other message to send back the result of the remote calculation. This is illustrated in the figure below:

Figure 4-8 Round-Trip Message



Most remote messages are, at bottom, two-way (or "round trip") remote procedure calls (RPCs) like this one. The sending application waits for the receiving application to invoke the method, complete its processing, and send back an indication that it has finished, along with any return information requested. Waiting for the receiver to finish, even if no information is returned, has the advantage of coordinating the two communicating applications, of keeping them both "in sync." For this reason, round-trip messages are often called **synchronous**. Synchronous messages are the default.

However, it's not always necessary or a good idea to wait for a reply. Sometimes it's sufficient simply to dispatch the remote message and return, allowing the receiver to get to the task when it can. In the meantime, the sender can go on to other things. Objective-C provides a return type modifier, `oneway`, to indicate that a method is used only for **asynchronous** messages:

```
- (oneway void)waltzAtWill;
```

Although `oneway` is a type qualifier (like `const`) and can be used in combination with a specific type name, such as `oneway float` or `oneway id`, the only such combination that makes any sense is `oneway void`. An asynchronous message can't have a valid return value.

Pointer Arguments

Next, consider methods that take pointer arguments. A pointer can be used to pass information to the receiver by reference. When invoked, the method looks at what's stored in the address it's passed.

```
- setTune:(struct tune *)aSong
{
    tune = *aSong;
    ...
}
```

The same sort of argument can also be used to return information by reference. The method uses the pointer to find where it should place information requested in the message.

```
- getTune:(struct tune *)theSong
{
    ...
    *theSong = tune;
}
```

The way the pointer is used makes a difference in how the remote message is carried out. In neither case can the pointer simply be passed to the remote object unchanged; it points to a memory location in the sender's address space and would not be meaningful in the address space of the remote receiver. The runtime system for remote messaging must make some adjustments behind the scenes.

If the argument is used to pass information by reference, the runtime system must dereference the pointer, ship the value it points to over to the remote application, store the value in an address local to that application, and pass that address to the remote receiver.

If, on the other hand, the pointer is used to return information by reference, the value it points to doesn't have to be sent to the other application. Instead, a value from the other application must be sent back and written into the location indicated by the pointer.

In the first case, information is passed on the first leg of the round trip. In the second case, information is returned on the second leg of the round trip. Because these cases result in very different actions on the part of the runtime system for remote messaging, Objective-C provides type modifiers that can clarify the programmer's intention:

- The type modifier`in` indicates that information is being passed in a message:

  ```
  - setTune:(in struct tune *)aSong;
  ```

- The modifier `out` indicates that an argument is being used to return information by reference:

  ```
  - getTune:(out struct tune *)theSong;
  ```

- A third modifier, `inout`, indicates that an argument is used both to provide information and to get information back:

  ```
  - adjustTune:(inout struct tune *)aSong;
  ```

The Cocoa distributed objects system takes `inout` to be the default modifier for all pointer arguments except those declared `const`, for which `in` is the default. `inout` is the safest assumption but also the most time-consuming since it requires passing information in both directions. The only modifier that makes sense for arguments passed by value (nonpointers) is `in`. While `in` can be used with any kind of argument, `out` and `inout` make sense only for pointers.

In C, pointers are sometimes used to represent composite values. For example, a string is represented as a character pointer (`char *`). Although in notation and implementation there's a level of indirection here, in concept there's not. Conceptually, a string is an entity in and of itself, not a pointer to something else.

In cases like this, the distributed objects system automatically dereferences the pointer and passes whatever it points to as if by value. Therefore, the `out` and `inout` modifiers make no sense with simple character pointers. It takes an additional level of indirection in a remote message to pass or return a string by reference:

```
- getTuneTitle:(out char **)theTitle;
```

The same is true of objects:

```
- adjustRectangle:(inout Rectangle **)theRect;
```

These conventions are enforced at runtime, not by the compiler.

Proxies and Copies

Finally, consider a method that takes an object as an argument:

```
- danceWith:(id)aPartner;
```

A `danceWith:` message passes an object `id` to the receiver. If the sender and the receiver are in the same application, they would both be able to refer to the same *`aPartner`* object.

This is true even if the receiver is in a remote application, except that the receiver needs to refer to the object through a proxy (since the object isn't in its address space). The pointer that `danceWith:` delivers to a remote receiver is actually a pointer to the proxy. Messages sent to the proxy would be passed across the connection to the real object and any return information would be passed back to the remote application.

There are times when proxies may be unnecessarily inefficient, when it's better to send a copy of the object to the remote process so that it can interact with it directly in its own address space. To give programmers a way to indicate that this is intended, Objective-C provides a `bycopy` type modifier:

```
- danceWith:(bycopy id)aClone;
```

`bycopy` can also be used for return values:

```
- (bycopy)dancer;
```

It can similarly be used with `out` to indicate that an object returned by reference should be copied rather than delivered in the form of a proxy:

```
- getDancer:(bycopy out id *)theDancer;
```

Note: When a copy of an object is passed to another application, it cannot be anonymous. The application that receives the object must have the class of the object loaded in its address space.

`bycopy` makes so much sense for certain classes—classes that are intended to contain a collection of other objects, for instance—that often these classes are written so that a copy is sent to a remote receiver, instead of the usual reference. You can override this behavior with `byref`, however, thereby specifying that objects passed to a method or objects returned from a method should be passed or returned by reference. Since passing by reference is the default behavior for the vast majority of Objective-C objects, you will rarely, if ever, make use of the `byref` keyword.

The only type that it makes sense for `bycopy` or `byref` to modify is an object, whether dynamically typed `id` or statically typed by a class name.

Although `bycopy` and `byref` can't be used inside class and category declarations, they can be used within formal protocols. For instance, you could write a formal protocol `foo` as follows:

```
@Protocol foo
- (bycopy)array;
@end
```

A class or category can then adopt your protocol `foo`. This allows you to construct protocols so that they provide "hints" as to how objects should be passed and returned by the methods described by the protocol.

Type Encodings

To assist the runtime system, the compiler encodes the return and argument types for each method in a character string and associates the string with the method selector. The coding scheme it uses is also useful in other contexts and so is made publicly available with the `@encode()` compiler directive. When given a type specification, `@encode()` returns a string encoding that type. The type can be a basic type such as an `int`, a pointer, a tagged structure or union, or a class name—anything, in fact, that can be used as an argument to the C `sizeof()` operator.

```
char *buf1 = @encode(int **);
char *buf2 = @encode(struct key);
char *buf3 = @encode(Rectangle);
```

The table below lists the type codes. Note that many of them overlap with the codes you use when encoding an object for purposes of archiving or distribution. However, there are codes listed here that you can't use when writing a coder, and there are codes that you may want to use when writing a coder that aren't generated by `@encode()`. (See the NSCoder class specification in the Foundation Framework reference for more information on encoding objects for archiving or distribution.)

Table 4-1 Objective-C type encodings

Code	Meaning
`c`	A `char`
`i`	An `int`
`s`	A `short`
`l`	A `long`
`q`	A `long long`
`C`	An `unsigned char`
`I`	An `unsigned int`
`S`	An `unsigned short`
`L`	An `unsigned long`
`Q`	An `unsigned long long`
`f`	A `float`
`d`	A `double`
`B`	A C++ `bool` or a C99 `_Bool`
`v`	A `void`
`*`	A character string (`char *`)
`@`	An object (whether statically typed or typed `id`)
`#`	A class object (`Class`)
`:`	A method selector (`SEL`)
[*array type*]	An array
{*name=type...*}	A structure
(*type...*)	A union
`b`num	A bit field of *num* bits
^type	A pointer to *type*
`?`	An unknown type (among other things, this code is used for function pointers)

The type code for an array is enclosed within square brackets; the number of elements in the array is specified immediately after the open bracket, before the array type. For example, an array of 12 pointers to `float`s would be encoded as:

```
[12^f]
```

Structures are specified within braces, and unions within parentheses. The structure tag is listed first, followed by an equal sign and the codes for the fields of the structure listed in sequence. For example, the structure

```
typedef struct example {
    id   anObject;
    char *aString;
    int  anInt;
} Example;
```

would be encoded like this:

```
{example=@*i}
```

The same encoding results whether the defined type name (`Example`) or the structure tag (`example`) is passed to `@encode()`. The encoding for a structure pointer carries the same amount of information about the structure's fields:

```
^{example=@*i}
```

However, another level of indirection removes the internal type specification:

```
^^{example}
```

Objects are treated like structures. For example, passing the NSObject class name to `@encode()` yields this encoding:

```
{NSObject=#}
```

The NSObject class declares just one instance variable, `isa`, of type Class.

Note that although the `@encode()` directive doesn't return them, the runtime system uses the additional encodings listed in Table 4-2 for type qualifiers when they're used to declare methods in a protocol.

Table 4-2 Objective-C method encodings

Code	Meaning
`r`	`const`
`n`	`in`
`N`	`inout`
`o`	`out`
`O`	`bycopy`
`R`	`byref`
`V`	`oneway`

Language Summary

Objective-C adds a small number of constructs to the C language and defines a handful of conventions for effectively interacting with the runtime system. This appendix lists all the additions to the language but doesn't go into great detail. For more information, see "The Language" (page 39). For a more formal presentation of Objective-C syntax, see "Grammar" (page 129).

Messages

Message expressions are enclosed in square brackets:

```
[receiver message]
```

The receiver can be:

- A variable or expression that evaluates to an object (including the variable `self`)
- A class name (indicating the class object)
- `super` (indicating an alternative search for the method implementation)

The **message** is the name of a method plus any arguments passed to it.

Defined Types

The principal types used in Objective-C are defined in `objc/objc.h`. They are:

Type	Definition
`id`	An object (a pointer to its data structure).
`Class`	A class object (a pointer to the class 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`.

Type	Definition
BOOL	A Boolean value, either YES or NO.

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

The objc.h header file also defines these useful terms:

Type	Definition
nil	A null object pointer, (id)0.
Nil	A null class pointer, (Class)0.
NO	A boolean false value, (BOOL)0.
YES	A boolean true value, (BOOL)1.

Preprocessor Directives

The preprocessor understands these special notations:

Notation	Definition
#import	Imports a header file. This directive is identical to #include, except that it doesn't include the same file more than once.
//	Begins a comment that continues to the end of the line.

Compiler Directives

Directives to the compiler begin with "@". The following directives are used to declare and define classes, categories, and protocols:

Directive	Definition
@interface	Begins the declaration of a class or category interface.
@implementation	Begins the definition of a class or category.
@protocol	Begins the declaration of a formal protocol.
@end	Ends the declaration/definition of a class, category, or protocol.

The following mutually exclusive directives specify the visibility of instance variables:

Directive	Definition
`@private`	Limits the scope of an instance variable to the class that declares it.
`@protected`	Limits instance variable scope to declaring and inheriting classes.
`@public`	Removes restrictions on the scope of instance variables.

The default is `@protected`.

These directives support exception handling:

Directive	Definition
`@try`	Defines a block within which exceptions can be thrown.
`@throw`	Throws an exception object.
`@catch()`	Catches an exception thrown within the preceding `@try` block.
`@finally`	Defines a block of code that is executed whether exceptions were thrown or not in a preceding `@try` block.

In addition, there are directives for these particular purposes:

Directive	Definition
`@class`	Declares the names of classes defined elsewhere.
`@selector(method_name)`	Returns the compiled selector that identifies *method_name*.
`@protocol(protocol_name)`	Returns the *protocol_name* protocol (an instance of the Protocol class). (`@protocol` is also valid without (*protocol_name*) for forward declarations.)
`@encode(type_spec)`	Yields a character string that encodes the type structure of *type_spec*.
`@defs(class_name)`	Yields the internal data structure of *class_name* instances
`@"string"`	Defines a constant NSString object in the current module and initializes the object with the specified 7-bit ASCII-encoded string.
`@"string1" @"string2" ... @"stringN"`	Defines a constant NSString object in the current module. The string created is the result of concatenating the strings specified in the two directives.
`@synchronized()`	Defines a block of code that must be executed only by one thread at a time.

Classes

A new class is declared with the `@interface` directive. The interface file for its superclass must be imported:

```
#import "ItsSuperclass.h"

@interface ClassName : ItsSuperclass < protocol_list >
{
    instance variable declarations
}
method declarations
@end
```

Everything but the compiler directives and class name is optional. If the colon and superclass name are omitted, the class is declared to be a new root class. If any protocols are listed, the header files where they're declared must also be imported.

A file containing a class definition imports its own interface:

```
#import “ClassName.h”

@implementation ClassName
method definitions
@end
```

Categories

A category is declared in much the same way as a class. The interface file that declares the class must be imported:

```
#import "ClassName.h"

@interface ClassName ( CategoryName ) < protocol list >
method declarations
@end
```

The protocol list and method declarations are optional. If any protocols are listed, the header files where they're declared must also be imported.

Like a class definition, a file containing a category definition imports its own interface:

```
#import "CategoryName.h"

@implementation ClassName ( CategoryName )
method definitions
@end
```

Formal Protocols

Formal protocols are declared using the `@protocol` directive:

```
@protocol ProtocolName < protocol list >
method declarations
@end
```

The list of incorporated protocols and the method declarations are optional. The protocol must import the header files that declare any protocols it incorporates.

You can create a forward reference to a protocol using the `@protocol` directive in the following manner:

```
@protocol ProtocolName;
```

Within source code, protocols are referred to using the similar `@protocol()` directive, where the parentheses enclose the protocol name.

Protocol names listed within angle brackets (<...>) are used to do three different things:

- In a protocol declaration, to incorporate other protocols (as shown earlier)
- In a class or category declaration, to adopt the protocol (as shown in "Classes" (page 124) and "Categories" (page 124))
- In a type specification, to limit the type to objects that conform to the protocol

Within protocol declarations, these type qualifiers support remote messaging:

Type Qualifier	Definition
`oneway`	The method is for asynchronous messages and has no valid return type.
`in`	The argument passes information to the remote receiver.
`out`	The argument gets information returned by reference.
`inout`	The argument both passes information and gets information.
`bycopy`	A copy of the object, not a proxy, should be passed or returned.
`byref`	A reference to the object, not a copy, should be passed or returned.

Method Declarations

The following conventions are used in method declarations:

- A "+" precedes declarations of class methods.
- A "-" precedes declarations of instance methods.

- Arguments are declared after colons (:). Typically, a label describing the argument precedes the colon. Both labels and colons are considered part of the method name.
- Argument and return types are declared using the C syntax for type casting.
- The default return and argument type for methods is `id`, not `int` as it is for functions. (However, the modifier `unsigned` when used without a following type always means `unsigned int`.)

Method Implementations

Each method implementation is passed two hidden arguments:

- The receiving object (`self`).
- The selector for the method (`_cmd`).

Within the implementation, both `self` and `super` refer to the receiving object. `super` replaces `self` as the receiver of a message to indicate that only methods inherited by the implementation should be performed in response to the message.

Methods with no other valid return typically return `void`.

Naming Conventions

The names of files that contain Objective-C source code have the `.m` extension. Files that declare class and category interfaces or that declare protocols have the `.h` extension typical of header files.

Class, category, and protocol names generally begin with an uppercase letter; the names of methods and instance variables typically begin with a lowercase letter. The names of variables that hold instances usually also begin with lowercase letters.

In Objective-C, identical names that serve different purposes don't clash. Within a class, names can be freely assigned:

- A class can declare methods with the same names as methods in other classes.
- A class can declare instance variables with the same names as variables in other classes.
- An instance method can have the same name as a class method.
- A method can have the same name as an instance variable.
- Method names beginning with "_", a single underscore character, are reserved for use by Apple.

Likewise, protocols and categories of the same class have protected name spaces:

- A protocol can have the same name as a class, a category, or anything else.
- A category of one class can have the same name as a category of another class.

However, class names are in the same name space as global variables and defined types. A program can't have a global variable with the same name as a class.

Grammar

This appendix presents a formal grammar for the Objective-C extensions to the C language—as the Objective-C language is implemented for the Cocoa development environment. It adds to the grammar for ANSI standard C found in Appendix A of *The C Programming Language* (second edition, 1988) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall, and should be read in conjunction with that book.

The Objective-C extensions introduce some new symbols (such as *class interface*), but also make use of symbols (such as *function definition*) that are explained in the standard C grammar. The symbols mentioned but not explained here are listed below:

compound-statement
constant
declaration
declaration-list
enum-specifier
expression
function-definition
identifier
parameter-type-list
string
struct-declaration-list
struct-or-union
typedef-name
type-name

Of these, *identifier* and *string* are undefined terminal symbols. Objective-C adds no undefined terminal symbols of its own.

Two notational conventions used here differ from those used in *The C Programming Language*:

- Literal symbols are shown in `code font`.
- Brackets enclose optional elements.

Otherwise, this appendix attempts to follow the conventions of *The C Programming Language*. Each part of the grammar consists of a symbol and a colon in **bold** and a list of mutually-exclusive possibilities for expanding the symbol. For example:

receiver:
expression
class-name
`super`

However, there is an exception: Even though they're not mutually exclusive, the constituents of classes, categories, protocols, and blocks are listed on separate lines to clearly show the ordering of elements. For example:

protocol-declaration:
`@protocol`*protocol-name*
[*protocol-reference-list*]
[*interface-declaration-list*]
`@end`

This exception to the general rule is easily recognized since each list terminates with `@end` or the symbol name ends with "-block".

There are six entry points where the Objective-C language modifies the rules defined for standard C:

- External declarations
- Type specifiers
- Type qualifiers
- Primary expressions
- Exceptions and synchronization

This appendix is therefore divided into five sections corresponding to these points. Where a rule in the standard C grammar is modified by an Objective-C extension, the entire rule is repeated in its modified form.

External Declarations

external-declaration:
function-definition
declaration
class-interface
class-implementation
category-interface
category-implementation
protocol-declaration
protocol-declaration-list
class-declaration-list
class-interface:
`@interface`*class-name* [`:`*superclass-name*]
[*protocol-reference-list*]

[*instance-variables*]
[*interface-declaration-list*]
`@end`
class-implementation:
`@implementation`*class-name* [`:`*superclass-name*]
[*instance-variables*]
[*implementation-definition-list*]
`@end`
category-interface:
`@interface`*class-name*`(`*category-name*`)`
[*protocol-reference-list*]
[*interface-declaration-list*]
`@end`
category-implementation:
`@implementation`*class-name*`(`*category-name*`)`
[*implementation-definition-list*]
`@end`
protocol-declaration:
`@protocol`*protocol-name*
[*protocol-reference-list*]
[*interface-declaration-list*]
`@end`
protocol-declaration-list:
`@protocol` *protocol-list*`;`
class-declaration-list:
`@class`*class-list*`;`
class-list:
class-name
class-list`,`*class-name*
protocol-reference-list:
<*protocol-list*>
protocol-list:
protocol-name
protocol-list`,`*protocol-name*
class-name:
identifier
superclass-name:
identifier
category-name:
identifier
protocol-name:
identifier
instance-variables:
`{`*instance-variable-declaration*`}`
instance-variable-declaration:

visibility-specification
struct-declaration-listinstance-variables
instance-variable-declaration visibility-specification
instance-variable-declaration struct-declaration-listinstance-variables

visibility-specification:
`@private`
`@protected`
`@public`

interface-declaration-list:
declaration
method-declaration
interface-declaration-listdeclaration
interface-declaration-listmethod-declaration

method-declaration:
class-method-declaration
instance-method-declaration

class-method-declaration:
+ [*method-type*] *method-selector* ;

instance-method-declaration:
– [*method-type*] *method-selector* ;

implementation-definition-list:
function-definition
declaration
method-definition
implementation-definition-listfunction-definition
implementation-definition-listdeclaration
implementation-definition-listmethod-definition

method-definition:
class-method-definition
instance-method-definition

class-method-definition:
+ [*method-type*] *method-selector* [*declaration-list*] *compound-statement*

instance-method-definition:
– [*method-type*] *method-selector* [*declaration-list*] *compound-statement*

method-selector:
unary-selector
keyword-selector [, . . .]
keyword-selector [,*parameter-type-list*]

unary-selector:
selector

keyword-selector:
keyword-declarator
keyword-selectorkeyword-declarator

keyword-declarator:
: [*method-type*] *identifier*

selector`:` [*method-type*] *identifier*
selector:
identifier
method-type:
`(`*type-name*`)`

Type Specifiers

type-specifier:
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
`id` [*protocol-reference-list*]
class-name [*protocol-reference-list*]
struct-or-union-specifier
enum-specifier
typedef-name
struct-or-union-specifier:
struct-or-union [*identifier*] `{`*struct-declaration-list*`}`
struct-or-union [*identifier*] `{ @defs (`*class-name*`) }`
struct-or-unionidentifier

Type Qualifiers

type-qualifier:
`const`
`volatile`
protocol-qualifier
protocol-qualifier:
`in`
`out`
`inout`
`bycopy`
`byref`
`oneway`

Primary Expressions

primary-expression:
identifier
constant
string
(*expression*)
`self`
message-expression
selector-expression
protocol-expression
encode-expression
message-expression:
[*receivermessage-selector*]
receiver:
expression
class-name
`super`
message-selector:
selector
keyword-argument-list
keyword-argument-list:
keyword-argument
keyword-argument-listkeyword-argument
keyword-argument:
selector : *expression*
: *expression*
selector-expression:
`@selector (`*selector-name*`)`
selector-name:
selector
keyword-name-list
keyword-name-list:
keyword-name
keyword-name-listkeyword-name
keyword-name:
selector :
:
protocol-expression:
`@protocol (`*protocol-name*`)`
encode-expression:
`@encode (`*type-name*`)`

Exceptions and Synchronization

exception-declarator:
declarator
try-statement:
`@try`*statement*
catch-statement:
`@catch (`*exception-declarator*`)`*statement*
finally-statement:
`@finally`*statement*
throw-statement:
`@throw(`*identifier*`)`
try-block:
try-statement
catch-statement
[*finally-statement*]
synchronize-statement:
`@synchronize(`*identifier*`)`

Document Revision History

This table describes the changes to *The Objective-C Programming Language*.

Date	Notes
2006-01-10	Clarified use of the static specifier for global variables used by a class.
2005-10-04	Clarified effect of sending messages to nil; noted use of ".mm" extension to signal Objective-C++ to compiler.
2005-04-08	Corrected typo in language grammar specification and modified a code example.
	Corrected the grammar for the protocol-declaration-list declaration in "External Declarations" (page 130).
	Clarified example in Listing 3-6 (page 90).
2004-08-31	Removed function and data structure reference. Added exception and synchronization grammar. Made technical corrections and minor editorial changes.
	Moved function and data structure reference to Objective-C Runtime Reference.
	Added examples of thread synchronization approaches to "Synchronizing Thread Execution" (page 88).
	Clarified when the `initialize` method is called and provided a template for its implementation in "Initializing a Class Object" (page 52).
	Added exception and synchronization grammar to "Grammar" (page 129).
	Replaced `conformsTo:` with `conformsToProtocol:` throughout document.
2004-02-02	Corrected typos in Listing 3-2 (page 87).
	Added initialization code for `methodLists` in "Creating an Objective-C class definition" for compatibility with Mac OS X versions 10.2 and 10.3.
2003-09-16	Corrected definition of `id`.

Date	Notes
2003-08-14	Documented the Objective-C exception and synchronization support available in Mac OS X version 10.3 and later in "Exception Handling and Thread Synchronization" (page 86).
	Documented the language support for concatenating constant strings in "Compiler Directives" (page 122).
	Moved "Object Ownership" (page 104) before "Retaining Objects" (page 104).
	Corrected the descriptions for the `objc_ivar` structure and the `objc_ivar_list` structure.
	Changed the font of *function result* in `class_getInstanceMethod` and `class_getClassMethod`.
	Corrected definition of the term *conform* in the glossary.
	Corrected definition of `method_getArgumentInfo`.
	Renamed from *Inside Mac OS X: The Objective-C Programming Language* to *The Objective-C Programming Language*.
2003-01-01	Documented the language support for declaring constant strings. Fixed several typographical errors. Added an index.
2002-05-01	Mac OS X 10.1 introduces a compiler for Objective-C++, which allows C++ constructs to be called from Objective-C classes, and vice versa.
	Added runtime library reference material.
	Fixed a bug in the Objective-C language grammar's description of instance variable declarations.
	Updated grammar and section names throughout the book to reduce ambiguities, passive voice, and archaic tone. Restructured some sections to improve cohesiveness.
	Renamed from *Object Oriented Programming and the Objective-C Language* to *Inside Mac OS X: The Objective-C Programming Language*.

Glossary

abstract class
A class that's defined solely so that other classes can inherit from it. Programs don't use instances of an abstract class, only of its subclasses.

abstract superclass
Same as "abstract class" (page 139).

adopt
In the Objective-C language, a class is said to adopt a protocol if it declares that it implements all the methods in the protocol. Protocols are adopted by listing their names between angle brackets in a class or category declaration.

anonymous object
An object of unknown class. The interface to an anonymous object is published through a protocol declaration.

Application Kit
A Cocoa framework that implements an application's user interface. The Application Kit provides a basic program structure for applications that draw on the screen and respond to events.

archiving
The process of preserving a data structure, especially an object, for later use. An archived data structure is usually stored in a file, but it can also be written to memory, copied to the pasteboard, or sent to another application. In Cocoa, archiving involves writing data to an NSData object.

asynchronous message
A remote message that returns immediately, without waiting for the application that receives the message to respond. The sending application and the receiving application act independently, and are therefore not "in sync." See also "synchronous message" (page 143).

category
In the Objective-C language, a set of method definitions that is segregated from the rest of the class definition. Categories can be used to split a class definition into parts or to add methods to an existing class.

class
In the Objective-C language, a prototype for a particular kind of object. A class definition declares instance variables and defines methods for all members of the class. Objects that have the same types of instance variables and have access to the same methods belong to the same class. See also "class object" (page 139).

class method
In the Objective-C language, a method that can operate on class objects rather than instances of the class.

class object
In the Objective-C language, an object that represents a class and knows how to create new instances of the class. Class objects are created by the compiler, lack instance variables, and can't be statically typed, but otherwise behave like all other objects. As the receiver in a message expression, a class object is represented by the class name.

Cocoa
An advanced object-oriented development platform on Mac OS X. Cocoa is a set of frameworks with programming interfaces in both Java and Objective-C.

compile time
The time when source code is compiled. Decisions made at compile time are constrained by the amount and kind of information encoded in source files.

conform
In the Objective-C language, a class is said to conform to a protocol if it (or a superclass) implements the methods declared in the protocol. An instance conforms to a protocol if its class does. Thus, an instance that conforms to a protocol can perform any of the instance methods declared in the protocol.

content view
In the Application Kit, the NSView object that's associated with the content area of a window—all the area in the window excluding the title bar and border. All other NSViews in the window are arranged in a hierarchy beneath the content view.

delegate
An object that acts on behalf of another object.

designated initializer
The `init...` method that has primary responsibility for initializing new instances of a class. Each class defines or inherits its own designated initializer. Through messages to `self`, other `init...` methods in the same class directly or indirectly invoke the designated initializer, and the designated initializer, through a message to `super`, invokes the designated initializer of its superclass.

dispatch table
Objective-C runtime table that contains entries that associate method selectors with the class-specific addresses of the methods they identify.

distributed objects
Architecture that facilitates communication between objects in different address spaces.

dynamic allocation
Technique used in C-based languages where the operating system provides memory to a running application as it needs it, instead of when it launches.

dynamic binding
Binding a method to a message—that is, finding the method implementation to invoke in response to the message—at runtime, rather than at compile time.

dynamic typing
Discovering the class of an object at runtime rather than at compile time.

encapsulation
Programming technique that hides the implementation of an operation from its users behind an abstract interface. This allows the implementation to be updated or changed without impacting the users of the interface.

event
The direct or indirect report of external activity, especially user activity on the keyboard and mouse.

factory
Same as "class object" (page 139).

factory method
Same as "class method" (page 139).

factory object
Same as "class object" (page 139).

formal protocol
In the Objective-C language, a protocol that's declared with the `@protocol` directive. Classes can adopt formal protocols, objects can respond at runtime when asked if they conform to a formal protocol, and instances can be typed by the formal protocols they conform to.

framework
A way to package a logically-related set of classes, protocols and functions together with localized strings, on-line documentation, and other pertinent files. Cocoa provides the Foundation framework and the Application Kit framework, among others. Frameworks are sometimes referred to as "kits."

gdb
The standard Mac OS X debugging tool.

id
In the Objective-C language, the general type for any kind of object regardless of class. `id` is defined as a pointer to an object data structure. It can be used for both class objects and instances of a class.

implementation
Part of an Objective-C class specification that defines its implementation. This section defines both public methods as well as private methods—methods that are not declared in the class's interface.

informal protocol
In the Objective-C language, a protocol declared as a category, usually as a category of the NSObject class. The language gives explicit support to formal protocols, but not to informal ones.

inheritance
In object-oriented programming, the ability of a superclass to pass its characteristics (methods and instance variables) on to its subclasses.

inheritance hierarchy
In object-oriented programming, the hierarchy of classes that's defined by the arrangement of superclasses and subclasses. Every class (except root classes such as NSObject) has a superclass, and any class may have an unlimited number of subclasses. Through its superclass, each class inherits from those above it in the hierarchy.

instance
In the Objective-C language, an object that belongs to (is a member of) a particular class. Instances are created at runtime according to the specification in the class definition.

instance method
In the Objective-C language, any method that can be used by an instance of a class rather than by the class object.

instance variable
In the Objective-C language, any variable that's part of the internal data structure of an instance. Instance variables are declared in a class definition and become part of all objects that are members of or inherit from the class.

interface
Part of an Objective-C class specification that declares its public interface, which include its superclass name, instances variables, and public-method prototypes.

Interface Builder
A tool that lets you graphically specify your application's user interface. It sets up the corresponding objects for you and makes it easy for you to establish connections between these objects and your own code where needed.

introspection
The ability of an object to reveal information about itself as an object—such as its class and superclass, the messages it can respond to, and the protocols it conforms to.

key window
The window in the active application that receives keyboard events and is the focus of user activity.

link time
The time when files compiled from different source modules are linked into a single program. Decisions made by the linker are constrained by the compiled code and ultimately by the information contained in source code.

localize
To adapt an application to work under various local conditions—especially to have it use a language selected by the user. Localization entails freeing application code from language-specific and culture-specific references and making it able to import localized resources (such as character strings, images, and sounds). For example, an application localized in Spanish would display "Salir" in the application menu. In Italian, it would be "Esci," in German "Verlassen," and in English "Quit."

main event loop
The principal control loop for applications that are driven by events. From the time it's launched until the moment it's terminated, an application gets one keyboard or mouse event after another from the Window Manager and responds to them, waiting between events if the next event isn't ready. In the Application Kit, the NSApplication object runs the main event loop.

menu
A small window that displays a list of commands. Only menus for the active application are visible on-screen.

message
In object-oriented programming, the method selector (name) and accompanying arguments that tell the receiving object in a message expression what to do.

message expression
In object-oriented programming, an expression that sends a message to an object. In the Objective-C language, message expressions are enclosed within square brackets and consist of a receiver followed by a message (method selector and arguments).

method
In object-oriented programming, a procedure that can be executed by an object.

multiple inheritance
In object-oriented programming, the ability of a class to have more than one superclass—to inherit from different sources and thus combine separately-defined behaviors in a single class. Objective-C doesn't support multiple inheritance.

mutex
Also known as mutual exclusion semaphore. Used to synchronize thread execution.

name space
A logical subdivision of a program within which all names must be unique. Symbols in one name space won't conflict with identically named symbols in another name space. For example, in Objective-C, the instance methods of each class are in a separate name space, as are the class methods and instance variables.

nil
In the Objective-C language, an object `id` with a value of 0.

object
A programming unit that groups together a data structure (instance variables) and the operations (methods) that can use or affect that data. Objects are the principal building blocks of object-oriented programs.

outlet
An instance variable that points to another object. Outlet instance variables are a way for an object to keep track of the other objects to which it may need to send messages.

polymorphism
In object-oriented programming, the ability of different objects to respond, each in its own way, to the same message.

procedural programming language
A language, like C, that organizes a program as a set of procedures that have definite beginnings and ends.

protocol
In the Objective-C language, the declaration of a group of methods not associated with any particular class. See also "formal protocol" (page 140) and "informal protocol" (page 141).

receiver
In object-oriented programming, the object that is sent a message.

reference counting
Memory-management technique in which each entity that claims ownership of an object increments the object's reference count and later decrements it. When the object's reference count reaches zero, the object is deallocated. This technique allows one instance of an object to be safely shared among several other objects.

remote message
A message sent from one application to an object in another application.

remote object
An object in another application, one that's a potential receiver for a remote message.

runtime
The time after a program is launched and while it's running. Decisions made at runtime can be influenced by choices the user makes.

selector
In the Objective-C language, the name of a method when it's used in a source-code message to an object, or the unique identifier that replaces the name when the source code is compiled. Compiled selectors are of type `SEL`.

static typing
In the Objective-C language, giving the compiler information about what kind of object an instance is, by typing it as a pointer to a class.

subclass
In the Objective-C language, any class that's one step below another class in the inheritance hierarchy. Occasionally used more generally to mean any class that inherits from another class, and sometimes also used as a verb to mean the process of defining a subclass of another class.

superclass
In the Objective-C language, a class that's one step above another class in the inheritance hierarchy; the class through which a subclass inherits methods and instance variables.

surrogate
An object that stands in for and forwards messages to another object.

synchronous message
A remote message that doesn't return until the receiving application finishes responding to the message. Because the application that sends the message waits for an acknowledgment or return information from the receiving application, the two applications are kept "in sync." See also "asynchronous message" (page 139).

Index