The Objective-C 2.0 Programming Language

Cocoa > Objective-C Language

2008-07-08

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Introduction Introduction to The Objective-C 2.0 Programming Language 9

Who Should Read This Document 10
Organization of This Document 10
Conventions 11
See Also 11

Chapter 1 Objects and Classes 13

Objects 13
  id 13
  Dynamic Typing 14
Object Messaging 14
  Message Syntax 15
  Sending Messages to nil 16
  The Receiver's Instance Variables 17
  Polymorphism 17
  Dynamic Binding 18
  Dot Syntax 18
Classes 23
  Inheritance 23
  Class Types 26
  Class Objects 28
  Class Names in Source Code 32

Chapter 2 Defining a Class 35

Source Files 35
Class Interface 36
  Importing the Interface 37
  Referring to Other Classes 37
  The Role of the Interface 38
Class Implementation 38
  Referring to Instance Variables 39
  The Scope of Instance Variables 40

Chapter 3 Categories and Extensions 45

Adding Methods to Classes 45
How you Use Categories 46
Categories of the Root Class 47
Extensions 47

Chapter 4 Declared Properties 49

Overview 49
  Declared Properties 49
Using Properties 50
  Property Declaration 51
  Property Implementation Directives 51
  Property Declaration Attributes 53
  Property Re-declaration 54
  Copy 55
  Markup and Deprecation 56
  Core Foundation 56
  Example 56
Subclassing with Properties 58
Performance and Threading 59
Property Introspection 59
Runtime Differences 60

Chapter 5 Protocols 63

Declaring Interfaces for Others to Implement 63
Methods for Others to Implement 64
Declaring Interfaces for Anonymous Objects 65
Non-Hierarchical Similarities 65
Formal Protocols 66
  Declaring a Protocol 66
  Optional Protocol Methods 67
Informal Protocols 67
Protocol Objects 68
Adopting a Protocol 68
Conforming to a Protocol 69
Type Checking 70
Protocols Within Protocols 70
Referring to Other Protocols 71

Chapter 6 Fast Enumeration 73

The for…in Feature 73
  Using Fast Enumeration 74

Chapter 7 How Messaging Works 75

The objc_msgSend Function 75
Selectors 78
Methods and Selectors 78
Method Return and Argument Types 79
Varying the Message at Runtime 79
The Target-Action Design Pattern 79
Avoiding Messaging Errors 80
Dynamic Method Resolution 81
Using Hidden Arguments 81
Messages to self and super 82
An Example 83
Using super 84
Redefining self 85

Chapter 8 Enabling Static Behavior 87

Static Typing 87
Type Checking 88
Return and Argument Types 89
Static Typing to an Inherited Class 89
Getting a Method Address 90

Chapter 9 Exception Handling 91

Handling Exceptions 91
Throwing Exceptions 92
Processing Exceptions 92

Chapter 10 Threading 95

Synchronizing Thread Execution 95

Chapter 11 Using C++ With Objective-C 97

Overview 97
Mixing Objective-C and C++ Language Features 97
C++ Lexical Ambiguities and Conflicts 100

Chapter 12 The Runtime System 103

Interacting with the Runtime System 103
Allocating and Initializing Objects 104
The Returned Object 105
Arguments 106

Coordinating Classes 106
The Designated Initializer 108
Combining Allocation and Initialization 110
Memory Management 111
Forwarding 112
Forwarding and Multiple Inheritance 113
Surrogate Objects 114
Forwarding and Inheritance 115
Dynamic Method Resolution 116
Dynamic Loading 117
Remote Messaging 117
Distributed Objects 118
Language Support 119
Type Encodings 123

Appendix A Language Summary 127

Messages 127
Defined Types 127
Preprocessor Directives 128
Compiler Directives 128
Classes 129
Categories 130
Deprecation Syntax 130
Formal Protocols 131
Method Declarations 132
Method Implementations 132
Naming Conventions 132

Glossary 135

Document Revision History 139

Index 143

Figures, Tables, and Listings

Chapter 1 Objects and Classes 13

Figure 1-1 Some Drawing Program Classes 24
Figure 1-2 Rectangle Instance Variables 25
Figure 1-3 Inheritance hierarchy for NSCell 30
Listing 1-1 Accessing properties using the dot syntax 19
Listing 1-2 Accessing properties using accessor methods 19
Listing 1-3 Implementation of the initialize method 32

Chapter 2 Defining a Class 35

Figure 2-1 The scope of instance variables 41

Chapter 4 Declared Properties 49

Listing 4-1 Declaring properties in a class 50
Listing 4-2 Using @synthesize 51
Listing 4-3 Using @dynamic with direct method implementations 52
Listing 4-4 Declaring properties for a class 57

Chapter 7 How Messaging Works 75

Figure 7-1 Messaging Framework 77
Figure 7-2 High, Mid, Low 83

Chapter 9 Exception Handling 91

Listing 9-1 An exception handler 92

Chapter 10 Threading 95

Listing 10-1 Locking a method using self 95
Listing 10-2 Locking a method using _cmd 96
Listing 10-3 Locking a method using a custom semaphore 96

Chapter 11 Using C++ With Objective-C 97

Listing 11-1 Using C++ and Objective-C instances as instance variables 97

Chapter 12 The Runtime System 103

Figure 12-1 Incorporating an Inherited Initialization Method 107
Figure 12-2 Covering an Inherited Initialization Model 108
Figure 12-3 Covering the Designated Initializer 109
Figure 12-4 Initialization Chain 110
Figure 12-5 Forwarding 114
Figure 12-6 Remote Messages 118
Figure 12-7 Round-Trip Message 120
Table 12-1 Objective-C type encodings 123
Table 12-2 Objective-C method encodings 125

Introduction to The Objective-C 2.0 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:

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

This document is about the first 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 second component, the Mac OS X Objective-C application frameworks—collectively known as Cocoa. You can start to learn more about Cocoa by reading *Getting Started with Cocoa*. The two main development tools you use are Xcode and Interface Builder, described in *Xcode User Guide* and *Interface Builder* respectively.

Important: This document describes version 2.0 of the Objective-C language which is released with Mac OS X v10.5. Several new features are introduced in this version, including properties (see "Declared Properties" (page 49)), fast enumeration (see "Fast Enumeration" (page 73)), optional protocols, and (on modern platforms) non-fragile instance variables. These features are not available on versions of Mac OS X prior to 10.5. If you use these features, therefore, your application cannot run on versions of Mac OS X prior to 10.5. To learn about version 1.0 of the Objective-C language, read *Object Oriented Programming and the Objective-C Programming Language 1.0*.

The Objective-C language is a simple computer language designed to enable sophisticated object-oriented programming. Objective-C is defined as a 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 several chapters and two appendixes.

The following chapters describe the Objective-C language They cover all the features that the language adds to standard C and C++. These chapters present the language, but also touch on important elements of the runtime system.

- "Objects and Classes" (page 13)
- "Defining a Class" (page 35)
- "Declared Properties" (page 49)
- "Protocols" (page 63)
- "Fast Enumeration" (page 73)
- "How Messaging Works" (page 75)
- "Enabling Static Behavior" (page 87)
- "Exception Handling" (page 91)
- "Threading" (page 95)

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 "Using C++ With Objective-C" (page 97).

"The Runtime System" (page 103) 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 127) lists and briefly comments on all of the Objective-C extensions to the C language.

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

Object-Oriented Programming with Objective-C describes object-oriented programming and development from the perspective of an Objective-C developer.

Objective-C 2.0 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.

Objective-C supports two patterns for memory management: automatic garbage collection and reference counting:

- *Garbage Collection Programming Guide* describes the garbage collection system used by Cocoa.
- *Memory Management Programming Guide for Cocoa* describes the reference counting system used by Cocoa.

Objective-C Release Notes describes some of the changes in the Objective-C runtime in the latest release of Mac OS X.

Objects and Classes

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 40)). 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 23).

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 26) and "Enabling Static Behavior" (page 87).

Object Messaging

This section explains the syntax of sending messages, including how you can nest message expressions. It also discusses the "visibility" of an object's instance variables, and the concepts of polymorphism and dynamic binding.

Message Syntax

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 parameters, or "arguments." A message with a single argument affixes a colon (`:`) to the selector name and puts the argument right after the colon. This construct is called a keyword; a keyword ends with a colon, and an argument follows the colon, as shown in this example:

```
[myRect setWidth:20.0];
```

A selector name includes all keywords, including colons, but does not include anything else, such as return type or parameter types. The imaginary message below tells the `myRect` object to set its origin to the coordinates (30.0, 50.0):

```
[myRect setOrigin:30.0 :50.0]; // This is a bad example of multiple arguments
```

Since the colons are part of the method name, the method is named `setOrigin::`. It has two colons as it takes two arguments. This particular method does not interleave the method name with the arguments and, thus, the second argument is effectively unlabeled and it is difficult to determine the kind or purpose of the method's arguments.

Instead, method names should interleave the name with the arguments such that the method's name naturally describes the arguments expected by the method. For example, the Rectangle class could instead implement a `setOriginX:y:` method that makes the purpose of its two arguments clear:

```
[myRect setOriginX: 30.0 y: 50.0]; // This is a good example of multiple arguments
```

Important: The sub-parts of the method name—of the selector—are not optional, nor can their order be varied. "Named arguments" and "keyword arguments" often carry the implication that the arguments to a method can vary at runtime, can have default values, can be in a different order, can possibly have additional named arguments. This is not the case with Objective-C.

For all intents and purposes, an Objective-C method declaration is simply a C function that prepends two additional arguments (see "How Messaging Works" (page 75)). This is different from the named or keyword arguments available in a language like Python:

```
def func(a, b, NeatMode=SuperNeat, Thing=DefaultThing):
    pass
```

where Thing (and NeatMode) might be omitted or might have different values when called.

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, the color of one rectangle is set to the color of another:

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

Objective-C 2.0 also provides a dot (`.`) operator that offers a compact and convenient syntax for invoking an object's accessor methods. This is typically used in conjunction with the declared properties feature (see "Declared Properties" (page 49)), and is described in "Dot Syntax" (page 18).

Sending Messages to nil

In Objective-C, it is valid to send a message to nil—it simply has no effect at runtime. There are several patterns in Cocoa that take advantage of this fact. The value returned from a message to `nil` may also be valid:

- If the method returns an object, any pointer type, any integer scalar of size less than or equal to `sizeof(void*)`, a `float`, a `double`, a `long double`, or a `long long`, then a message sent to `nil` returns `0`.
- If the method returns a struct, as defined by the *Mac OS X ABI Function Call Guide* to be returned in registers, then a message sent to `nil` returns `0.0` for every field in the data structure. Other struct data types will not be filled with zeros.
- If the method returns anything other than the aforementioned value types the return value of a message sent to `nil` is undefined.

The following code fragment illustrates valid use of sending a message to `nil`.

```
id anObjectMaybeNil = nil;

// this is valid
if ([anObjectMaybeNil methodThatReturnsADouble] == 0.0)
{
    // implementation continues...
}
```

Note: The behavior of sending messages to `nil` changed with Mac OS X v10.5; for the behavior of previous versions, see prior documentation.

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 35) 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 75).)

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 75).

Dot Syntax

Objective-C provides a dot (`.`) operator that offers a compact and convenient syntax you can use as an alternative to square bracket notation (`[]`s) to invoke accessor methods. It is particularly useful when you want to access or modify a property that is a property of another object (that is a property of another object, and so on).

Using the Dot Syntax

Overview

You can use the **dot syntax** to invoke accessor methods using the same pattern as accessing structure elements as illustrated in the following example:

```
myInstance.value = @"New value";
NSLog(@"myInstance value: %@", myInstance.value);
```

The dot syntax is purely "syntactic sugar"—it is transformed by the compiler into invocation of accessor methods (so you are not actually accessing an instance variable directly). The code example above is exactly equivalent to the following:

```
[myInstance setValue:@"New value"];
NSLog(@"myInstance value: %@", [myInstance value]);
```

General Use

You can read and write properties using the dot (`.`) operator, as illustrated in the following example.

Listing 1-1 Accessing properties using the dot syntax

```
Graphic *graphic = [[Graphic alloc] init];

NSColor *color = graphic.color;
CGFloat xLoc = graphic.xLoc;
BOOL hidden = graphic.hidden;
int textCharacterLength = graphic.text.length;

if (graphic.textHidden != YES) {
    graphic.text = @"Hello";
}
graphic.bounds = NSMakeRect(10.0, 10.0, 20.0, 120.0);
```

Accessing a property *property* calls the get method associated with the property (by default, *property*) and setting it calls the set method associated with the property (by default, `set`*Property*`:`). You can change the methods that are invoked by using the Declared Properties feature (see "Declared Properties" (page 49)). Despite appearances to the contrary, the dot syntax therefore preserves encapsulation—you are not accessing an instance variable directly.

The following statements compile to exactly the same code as the statements shown in "Accessing properties using the dot syntax" (page $@), but use square bracket syntax:

Listing 1-2 Accessing properties using accessor methods

```
Graphic *graphic = [[Graphic alloc] init];

NSColor *color = [graphic color];
CGFloat xLoc = [graphic xLoc];
BOOL hidden = [graphic hidden];
int textCharacterLength = [[graphic text] length];

if ([graphic isTextHidden] != YES) {
    [graphic setText:@"Hello"];
```

```
}
[graphic setBounds:NSMakeRect(10.0, 10.0, 20.0, 120.0)];
```

An advantage of the dot syntax is that the compiler can signal an error when it detects a write to a read-only property, whereas at best it can only generate an undeclared method warning that you invoked a non-existent `set`*Property*`:` method, which will fail at runtime.

For properties of the appropriate C language type, the meaning of compound assignments is well-defined. For example, you could update the length property of an instance of `NSMutableData` using compound assignments:

```
NSMutableData *data = [NSMutableData dataWithLength:1024];
data.length += 1024;
data.length *= 2;
data.length /= 4;
```

which is equivalent to:

```
[data setLength:[data length] + 1024];
[data setLength:[data length] * 2];
[data setLength:[data length] / 4];
```

There is one case where properties cannot be used. Consider the following code fragment:

```
id y;
x = y.z;  // z is an undeclared property
```

Note that `y` is untyped and the `z` property is undeclared. There are several ways in which this could be interpreted. Since this is ambiguous, the statement is treated as an undeclared property error. If `z` is declared, then it is not ambiguous if there's only one declaration of a `z` property in the current compilation unit. If there are multiple declarations of a `z` property, as long as they all have the same type (such as `BOOL`) then it is legal. One source of ambiguity would also arise from one of them being declared `readonly`.

nil Values

If a `nil` value is encountered during property traversal, the result is the same as sending the equivalent message to `nil`. For example, the following pairs are all equivalent:

```
// each member of the path is an object
x = person.address.street.name;
x = [[[person address] street] name];

// the path contains a C struct
// will crash if window is nil or -contentView returns nil
y = window.contentView.bounds.origin.y;
y = [[window contentView] bounds].origin.y;

// an example of using a setter....
person.address.street.name = @"Oxford Road";
[[[person address] street] setName: @"Oxford Road"];
```

self

If you want to access a property of `self` using accessor methods, you must explicitly call out `self` as illustrated in this example:

```
self.age = 10;
```

If you do not use `self.`, you access the instance variable directly. In the following example, the set accessor method for the `age` property is *not* invoked:

```
age = 10;
```

Performance and Threading

The dot syntax generates code equivalent to the standard method invocation syntax. As a result, code using the dot syntax performs exactly the same as code written directly using the accessor methods. Since the dot syntax simply invokes methods, no additional thread dependencies are introduced as a result of its use.

Dot Syntax and Key-Value Coding

Key-value coding (KVC) defines generic property accessor methods—`valueForKey:` and `setValue:forKey:`—which identify properties with string-based keys. KVC is not meant as a general alternative to using accessor methods—it is for use by code that has no other option, because the code cannot know the names of the relevant properties at compile time.

Key-value coding and the dot syntax are orthogonal technologies. You can use KVC whether or not you use the dot syntax, and you can use the dot syntax whether or not you use KVC. Both, though, make use of a "dot syntax". In the case of KVC, the syntax is used to delimit elements in a key path. It is important to remember that when you access a property using the dot syntax, you invoke the receiver's standard accessor methods (as a corollary, to emphasize, the dot syntax does *not* result in invocation of KVC methods `valueForKey:` or `setValue:forKey:`).

You can use KVC methods to access a property, for example, given a class defined as follows:

```
@interface MyClass
@property NSString *stringProperty;
@property NSInteger integerProperty;
@property MyClass *linkedInstance;
@end
```

you could access the properties in an instance using KVC:

```
MyClass *myInstance = [[MyClass alloc] init];
NSString *string = [myInstance valueForKey:@"stringProperty"];
[myInstance setValue:[NSNumber numberWithInt:2] forKey:@"integerProperty"];
```

To illustrate the difference between the properties dot syntax and KVC key paths, consider the following.

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
myInstance.linkedInstance.integerProperty = 2;
```

This has the same result as:

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
[myInstance setValue:[NSNumber numberWithInt:2]
            forKeyPath:@"linkedInstance.integerProperty"];
```

Usage Summary

```
aVariable = anObject.aProperty;
```

Invokes the `aProperty` method and assigns the return value to `aVariable`. The type of the property `aProperty` and the type of `aVariable` must be compatible, otherwise you get a compiler warning.

```
anObject.name = @"New Name";
```

Invokes the `setName:` method on `anObject`, passing `@"New Name"` as the argument.

You get a compiler warning if `setName:` does not exist, if the property `name` does not exist, or if `setName:` returns anything but `void`.

```
xOrigin = aView.bounds.origin.x;
```

Invokes the `bounds` method and assigns `xOrigin` to be the value of the `origin.x` structure element of the `NSRect` returned by `bounds`.

```
NSInteger i = 10;
anObject.integerProperty = anotherObject.floatProperty = ++i;
```

Assigns `11` to both `anObject.integerProperty` and `anotherObject.floatProperty`. That is, the right hand side of the assignment is pre-evaluated and the result is passed to `setIntegerProperty:` and `setFloatProperty:`. The pre-evaluated result is coerced as required at each point of assignment.

Incorrect Use

The following patterns are strongly discouraged.

```
anObject.retain;
```

Generates a compiler warning (`warning: value returned from property not used.`).

```
/* method declaration */
- (BOOL) setFooIfYouCan: (MyClass *)newFoo;

/* code fragment */
anObject.fooIfYouCan = myInstance;
```

Generates a compiler warning that `setFooIfYouCan:` does not appear to be a setter method because it does not return `(void)`.

```
flag = aView.lockFocusIfCanDraw;
```

Invokes `lockFocusIfCanDraw` and assigns the return value to `flag`. This does not generate a compiler warning unless `flag`'s type mismatches the method's return type.

```
/* property declaration */
@property(readonly) NSInteger readonlyProperty;
/* method declaration */
- (void) setReadonlyProperty: (NSInteger)newValue;

/* code fragment */
self.readonlyProperty = 5;
```

Since the property is declared `readonly`, this code generates a compiler warning (`warning: assignment to readonly property 'readonlyProperty'`). It will work at runtime, but simply adding a setter for a property does not imply `readwrite`.

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.

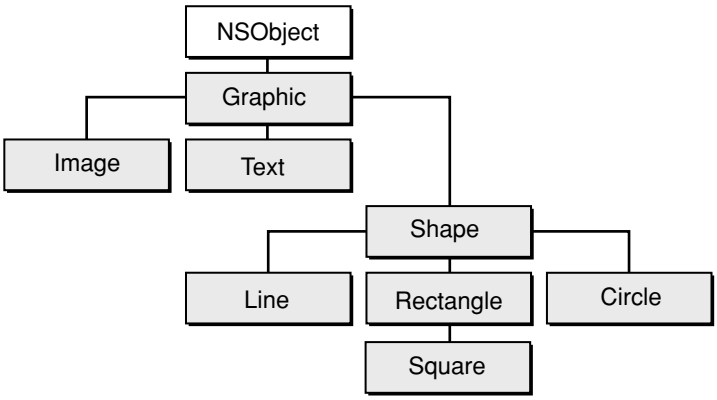
By convention, class names begin with an uppercase letter (such as "Rectangle"); the names of instances typically begin with a lowercase letter (such as "myRect").

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 1-1 illustrates the hierarchy for a few of the classes used in the drawing program.

Figure 1-1 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` is a root class, and so doesn't have a superclass. 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.

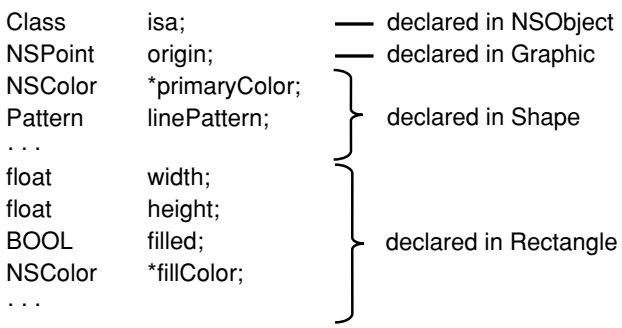
Note: Implementing a new root class is a delicate task and one with many hidden hazards. The class must duplicate much of what the `NSObject` class does, such as allocate instances, connect them to their class, and identify them to the runtime system. For this reason, you should generally use the `NSObject` 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 1-2 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 1-2 Rectangle Instance Variables



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 82) 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 or primarily 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 typically incomplete by itself, but contains useful code that reduces the implementation burden of its subclasses. (Because abstract classes must have subclasses to be useful, they're sometimes also called **abstract superclasses**.)

Unlike some other languages, Objective-C does not have syntax to mark classes as abstract, nor does it prevent you from creating an instance of an abstract class.

The `NSObject` class is the canonical example of an abstract class in Cocoa. You never use instances of the `NSObject` class in an application—it wouldn't be good for anything; it would be a generic object with the ability to do nothing in particular.

The `NSView` class, on the other hand, provides an example of an abstract class instances of which you might occasionally use directly.

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.

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 Behavior" (page 87) 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".

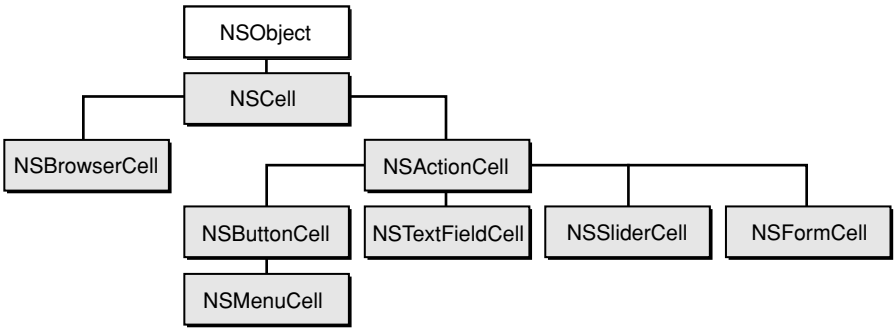
Customization With Class Objects

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` object.

An `NSMatrix` object can take responsibility for creating the individual objects that represent its cells. It can do this when the matrix 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 matrix 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 matrix displays just one kind of `NSCell`, but there are many different kinds. The inheritance hierarchy in Figure 1-3 shows some of those provided by the Application Kit. All inherit from the generic `NSCell` class:

Figure 1-3 Inheritance hierarchy for NSCell



When a matrix 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` object 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` object 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 you want to use a class object for anything besides allocating instances, you may need to initialize it just as you would an instance. 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 1-3 when implementing the `initialize` method.

Listing 1-3 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 Behavior" (page 87) 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, you can use `NSClassFromString` to 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)

These are typically split between two files, sometimes however a class definition may span several files through the use of a feature called a "category." Categories can compartmentalize a class definition or extend an existing one. Categories are described in "Categories and Extensions" (page 45).

Source Files

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.

Class 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 23). 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.

Class 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.

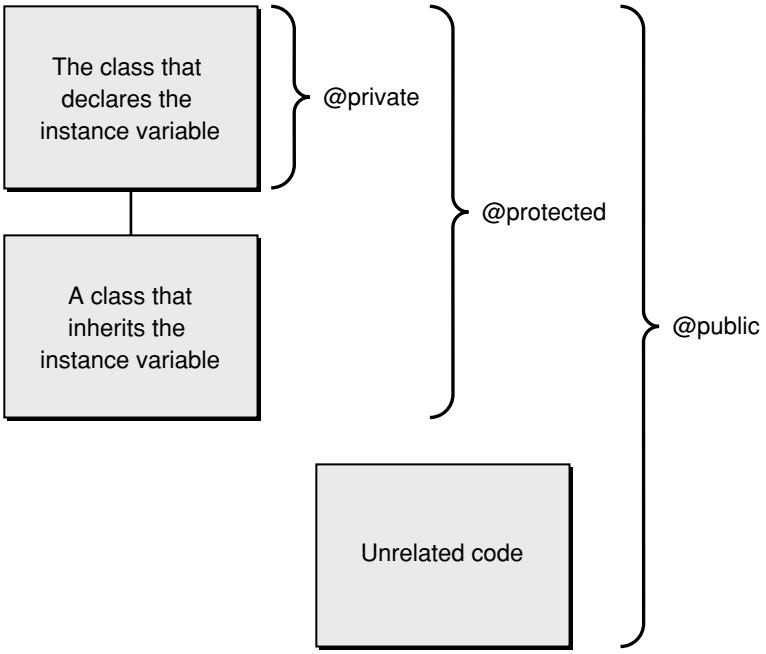
As a class is revised from time to time, the choice of instance variables may change, even though the methods it declares remain the same. As long as messages are the vehicle for interacting with instances of the class, these changes won't really affect its interface.

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.
`@package`	On 64-bit, an `@package` instance variable acts like `@public` inside the image that implements the class, but `@private` outside. This is analogous to `private_extern` for variables and functions. Any code outside the class implementation's image that tries to use the instance variable will get a link error. This is most useful for instance variables in framework classes, where `@private` may be too restrictive but `@protected` or `@public` too permissive.

This is illustrated in Figure 2-1.

Figure 2-1 The scope of instance variables



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.

Categories and Extensions

A category allows you to add methods to an existing class—even to one to which you do not have the source. This is a powerful feature that allows you to extend the functionality of existing classes without subclassing. Using categories, you can also split the implementation of your own classes between several files. Class extensions are similar, but allow additional *required* API to be declared for a class in locations other than within the primary class `@interface` block

Adding Methods to 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.

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.

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. A common naming convention is that the base file name of the category is the name of the class the category extends followed by "+" followed by the name of the category. A category implementation (in a file named `ClassName+CategoryName.m`) might therefore look like this:

```
#import "ClassName+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.

How you Use Categories

You can use categories 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.

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 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.

You can also use categories 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 (see "Informal Protocols " (page 67)), as discussed under "Declaring Interfaces for Others to Implement" (page 63).

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.

Extensions

Class extensions are like "anonymous" categories, except that the methods they declare must be implemented in the the main `@implementation` block for the corresponding class.

It is common for a class to have a publicly declared API and to then have additional API declared privately for use solely by the class or the framework within which the class resides. You can declare such API in a category (or in more than one category) in a private header file or implementation file as described above. This works, but the compiler cannot verify that all declared methods are implemented.

For example, the compiler will compile without error the following declarations and implementation:

```
@interface MyObject : NSObject
{
    NSNumber *number;
}
- (NSNumber *)number;
@end

@interface MyObject (Setter)
```

```
- (void)setNumber:(NSNumber *)newNumber;
@end


@implementation MyObject

- (NSNumber *)number
{
    return number;
}
@end
```

Note that there is no implementation of the `setNumber` method. If it is invoked at runtime, this will generate an error.

Class extensions allow you to declare additional *required* API for a class in locations other than within the primary class `@interface` block, as illustrated in the following example:

```
@interface MyObject : NSObject
{
    NSNumber *number;
}
- (NSNumber *)number;
@end

@interface MyObject ()
- (void)setNumber:(NSNumber *)newNumber;
@end


@implementation MyObject

- (NSNumber *)number
{
    return number;
}
- (void)setNumber(NSNumber *)newNumber
{
    number = newNumber;
}
@end
```

Notice that in this case:

- No name is given in the parentheses in the second `@interface` block;
- The implementation of the `setNumber` method appears within the main `@implementation` block for the class.

The implementation of the `setNumber` method *must* appear within the main `@implementation` block for the class (you cannot implement it in a category). If this is not the case, the compiler will emit a warning that it cannot find a method definition for `setNumber`.

Declared Properties

The Objective-C "declared properties" feature provides a simple way to declare and implement an object's properties.

Overview

There are two aspects to this language feature: the syntactic elements you use to specify and optionally synthesize *declared properties*, and a related syntactic element that is described in "Dot Syntax" (page $@).

Declared Properties

You typically access an object's properties (in the sense of its attributes and relationships) through a pair of accessor (getter/setter) methods. By using accessor methods, you adhere to the principle of encapsulation (see "Mechanisms Of Abstraction" in *Object-Oriented Programming with Objective-C* > The Object Model). You can exercise tight control of the behavior of the getter/setter pair and the underlying state management while clients of the API remain insulated from the implementation changes.

Although using accessor methods has significant advantages, writing accessor methods is nevertheless a tedious process—particularly if you have to write code to support both garbage collected and managed memory environments. Moreover, aspects of the property that may be important to consumers of the API are left obscured—such as whether the accessor methods are thread-safe or whether new values are copied when set.

A **declared property** is effectively a shorthand for declaring accessor methods. It also, though, provides a specification for the methods' behavior. It addresses the problems with standard accessor methods by providing the following features:

- The property declaration provides a clear, explicit specification of how the accessor methods behave.
- The compiler can synthesize accessor methods for you, according to the specification you provide in the declaration. This means you have less code to write and maintain.
- Properties are represented syntactically as identifiers and are scoped, so the compiler can detect use of undeclared properties.

- Runtime introspection of the properties declared by a class.

In addition to the declaration itself, there are implementation directives that instruct the compiler to synthesize the accessors and to inform the compiler that you will provide the methods yourself at runtime.

- You use the `@property` compiler directive to declare a property. It can appear anywhere in the method declaration section of a class, category, or protocol declaration.
- You use the `@synthesize` and `@dynamic` directives in `@implementation` blocks to trigger specific compiler actions: `@synthesize` instructs the compiler to synthesize the relevant accessors; `@dynamic` informs the compiler that you will provide the methods yourself at runtime.

Listing 4-1 illustrates the declaration of three properties and use of the `@synthesize` directive to instruct the compiler to create accessor methods to match the specifications given in the declaration.

Listing 4-1 Declaring properties in a class

```
// MyClass.h
@interface MyClass : NSObject
{
    NSString *value;
    NSTextField *textField;
@private
    NSDate *lastModifiedDate;
}
@property(copy, readwrite) NSString *value;
@property(retain) IBOutlet NSTextField *textField;
@end

// MyClass.m
// Class extension to declare private property
@interface MyClass ()
@property(retain) NSDate *lastModifiedDate;
@end

@implementation MyClass
@synthesize value;
@synthesize textField;
@synthesize lastModifiedDate;
// implementation continues
@end
```

Using Properties

There are two parts to a declared property, its declaration and its implementation. Typically, a class cannot redefine a property defined by its superclass—the exception is that a property declared as read-only may be re-declared as read-write.

Properties also support

Property Declaration

A property declaration begins with the keyword `@property`. `@property` can appear anywhere in the method declaration list found in the `@interface` of a class. `@property` can also appear in the declaration of a protocol or category.

`@property(attributes) type name;`

`@property` declares a property. An optional parenthesized set of attributes provides additional details about the storage semantics and other behaviors of the property—see "Property Declaration Attributes" (page 53) for possible values. Like any other Objective-C type, each property has a type specification and a name.

You can think of a property declaration as being equivalent to declaring two accessor methods. Thus, for example,

```
@property NSString *name;
```

is equivalent to:

```
- (NSString *)name;
- (void)setName:(NSString *)newName;
```

A property declaration, however, provides additional information about how the accessor methods are implemented (as described in "Property Declaration Attributes" (page 53)).

Property Implementation Directives

You can use the `@synthesize` and `@dynamic` directives in `@implementation` blocks to trigger specific compiler actions. Note that neither is *required* for any given `@property` declaration.

Important: The default value is `@dynamic`. If, therefore, you do not specify either `@synthesize` or `@dynamic` for a particular property, you must provide a getter and setter (or just a getter in the case of a `readonly` property) method implementation for that property.

`@synthesize`

You use the `@synthesize` keyword to tell the compiler that it should synthesize the setter and/or getter methods for the property if you do not supply them within the `@implementation` block.

Listing 4-2 Using @synthesize

```
@interface MyClass : NSObject
{
    NSString *value;
}
@property(copy, readwrite) NSString *value;
@end

@implementation MyClass
@synthesize value;
```

```
@end
```

You can use the form `property=ivar` to indicate that a particular instance variable should be used for the property, for example:

```
@synthesize firstName, lastName, age = yearsOld;
```

This specifies that the accessor methods for `firstName`, `lastName`, and `age` should be synthesized and that the property `age` is represented by the instance variable `yearsOld`. Other aspects of the synthesized methods are determined by the optional attributes (see "Property Declaration Attributes" (page 53)).

There are differences in the behavior that depend on the runtime (see also "Runtime Differences" (page 60)):

- For the legacy runtimes, instance variables must already be declared in the `@interface` block. If an instance variable of the same name and compatible type as the property exists, it is used—otherwise, you get a compiler error.
- For the modern runtimes, instance variables are synthesized as needed. If an instance variable of the same name already exists, it is used.

`@dynamic`

You use the `@dynamic` keyword to tell the compiler that you will fulfill the API contract implied by a property either by providing method implementations directly or at runtime using other mechanisms such as dynamic loading of code or dynamic method resolution. The example shown in Listing 4-3 illustrates using direct method implementations—it is equivalent to the example given in Listing 4-2 (page 51).

Listing 4-3 Using @dynamic with direct method implementations

```
@interface MyClass : NSObject
{
    NSString *value;
}
@property(copy, readwrite) NSString *value;
@end

// assume using garbage collection
@implementation MyClass
@dynamic value;

- (NSString *)value {
    return value;
}

- (void)setValue:(NSString *)newValue {
    if (newValue != value) {
        value = [newValue copy];
    }
}
@end
```

Property Declaration Attributes

You can decorate a property with attributes by using the form `@property(attribute [, attribute2, ...])`. Like methods, properties are scoped to their enclosing interface declaration. For property declarations that use a comma delimited list of variable names, the property attributes apply to all of the named properties. If you use garbage collection, you can use the storage modifiers `__weak` and `__strong` in a property's declaration, but they are not a formal part of the list of attributes.

If you use the `@synthesize` directive to tell the compiler to create the accessor method(s), the code it generates matches the specification given by the keywords. If you implement the accessor method(s) yourself, you should ensure that it they match the specification (for example, if you specify `copy` you must make sure that you do copy the input value in the setter method).

`getter=getterName`, `setter=setterName`

`getter=` and `setter=` specify respectively the names of get and set accessors for the property. The getter must return a type matching the property's type and take no arguments. The setter method must take a single argument of a type matching the property's type and must return `void`.

The default names are *propertyName* and `set`*PropertyName*`:` respectively—for example, given a property "foo", the accessors would be `foo` and `setFoo:`. Typically you should specify accessor method names that are key-value coding compliant (see *Key-Value Coding Programming Guide*)—a common reason for using this decorator is to adhere to the `is`*PropertyName* convention for Boolean values.

`readonly`

Indicates that the property is read-only. The default is read/write.

If you specify `readonly`, only a getter method is required in the `@implementation`. If you use `@synthesize` in the implementation block, only the getter method is synthesized. Moreover, if you attempt to assign a value using the dot syntax, you get a compiler error.

`readwrite`

Indicates that the property should be treated as read/write. This is the default.

Both a getter and setter method will be required in the `@implementation`. If you use `@synthesize` in the implementation block, the getter and setter methods are synthesized.

`assign`

Specifies that the setter uses simple assignment. This is the default.

If your application uses garbage collection, if you want to use `assign` for a property whose class adopts the `NSCopying` protocol you should specify the attribute explicitly rather than simply relying on the default—otherwise you will get a compiler warning. (This is to reassure the compiler that you really do want to assign the value, even though it's copyable.)

`retain`

Specifies that `retain` should be invoked on the object upon assignment. (The default is `assign`.)

This attribute is valid only for Objective-C object types. (You cannot specify `retain` for Core Foundation objects—see "Core Foundation" (page 56).)

`copy`

Specifies that a copy of the object should be used for assignment. (The default is `assign`.)

The copy is made by invoking the `copy` method. This attribute is valid only for object types, which must implement the `NSCopying` protocol. For further discussion, see "Copy" (page 55).

`nonatomic`

Specifies that accessors are non-atomic. *By default, accessors are atomic.*(There is no keyword to denote atomic.)

Properties are atomic by default so that synthesized accessors provide robust access to properties in a multi-threaded environment—that is, the value returned from the getter or set via the setter is always fully retrieved or set regardless of what other threads are executing concurrently. For more details, see "Performance and Threading" (page 59).

If you do not specify `nonatomic`, then in a managed memory environment a synthesized get accessor for an object property retains and autoreleases the returned value; if you specify `nonatomic`, then a synthesized accessor for an object property simply returns the value directly.

You can use the interface attributes `getter=`, `setter=`, `readonly`, `readwrite` in a class, category or protocol declaration. You can use only one of `readonly` and `readwrite` in the attribute list. `setter=/getter=` are both optional and may appear with any other attribute save for `readonly`. If you specify that a property is `readonly` then also specify a setter with `setter=`, you will get a compiler warning.

`assign`, `retain`, and `copy` are mutually exclusive. Different constraints apply depending on whether or not you use garbage collection:

- If you do not use garbage collection, for object properties you must explicitly specify one of `assign`, `retain` or `copy`—otherwise you will get a compiler warning. (This encourages you to think about what memory management behavior you want and type it explicitly.)
- If you use garbage collection, you don't get a warning if you use the default (that is, if you don't specify any of `assign`, `retain` or `copy`) unless the property's type is a class that conforms to `NSCopying`. The default is usually what you want; if the property type can be copied, however, to preserve encapsulation you often want to make a private copy of the object.

Property Re-declaration

You can re-declare a property in a subclass, but (with the exception of `readonly` vs. `readwrite`) you must repeat its attributes in whole in the subclasses. The same holds true for a property declared in a category or protocol—while the property may be redeclared in a category or protocol, the property's attributes must be repeated in whole.

If you declare a property in one class as `readonly`, you can redeclare it as `readwrite` in a class extension (see "Extensions" (page 47)), a protocol, or a subclass—see "Subclassing with Properties" (page 58). In the case of a class extension redeclaration, the fact that the property was redeclared prior to any `@synthesize` statement will cause the setter to be synthesized. The ability to redeclare a read-only property as read/write enables two common implementation patterns: a mutable subclass of an immutable class (`NSString`, `NSArray`, and `NSDictionary` are all examples) and a property that has public API that is `readonly` but a private `readwrite` implementation internal to the class. The following example shows using a class extension to provide a property that is declared as read-only in the public header but is redeclared privately as read/write.

```
// public header file
@interface MyObject : NSObject
{
    NSString *language;
}
@property (readonly, copy) NSString *language;
@end
```

```
// private implementation file
@interface MyObject ()
@property (readwrite, copy) NSString *language;
@end

@implementation MyObject
@synthesize language;
@end
```

Copy

If you use the `copy` declaration attribute, you specify that a value is copied during assignment. If you synthesize the corresponding accessor, the synthesized method uses the `copy` method. This is useful for attributes such as string objects where there is a possibility that the new value passed in a setter may be mutable (for example, an instance of `NSMutableString`) and you want to ensure that your object has its own private immutable copy. For example, if you declare a property as follows:

```
@property (nonatomic, copy) NSString *string;
```

then the synthesized setter method is similar to the following:

```
-(void)setString:(NSString *)newString
{
    if (string != newString) {
        [string release];
        string = [newString copy];
    }
}
```

Although this works well for strings, it may present a problem if the attribute is a collection such as an array or a set. Typically you want such collections to be mutable, but the `copy` method returns an *immutable* version of the collection. In this situation, you have to provide your own implementation of the setter method, as illustrated in the following example.

```
@interface MyClass : NSObject
{
    NSMutableArray *myArray;
}
@property (nonatomic, copy) NSMutableArray *myArray;
@end

@implementation MyClass

@synthesize myArray;

- (void)setMyArray:(NSMutableArray *)newArray
{
    if (myArray != newArray) {
        [myArray release];
        myArray = [newArray mutableCopy];
    }
}

@end
```

Markup and Deprecation

Properties support the full range of C style decorators. Properties can be deprecated and support `__attribute__` style markup, as illustrated in the following example.

```
@property CGFloat x
AVAILABLE_MAC_OS_X_VERSION_10_1_AND_LATER_BUT_DEPRECATED_IN_MAC_OS_X_VERSION_10_4;
@property CGFloat y __attribute__((...));
```

Core Foundation

As noted in "Property Implementation Directives" (page 51), you cannot specify the `retain` attribute for non-object types. If, therefore, you declare a property whose type is a CFType and synthesize the accessors as illustrated in the following example:

```
@interface MyClass : NSObject
{
    CGImageRef myImage;
}
@property(readwrite) CGImageRef myImage;
@end

@implementation MyClass
@synthesize myImage;
@end
```

then in a managed memory environment the generated set accessor will simply assign the new value to the instance variable (the new value is not retained and the old value is not released). This is typically incorrect, so you should not synthesize the methods, you should implement them yourself.

In a garbage collected environment, if the variable is declared `__strong`:

```
...
__strong CGImageRef myImage;
...
@property CGImageRef myImage;
```

then the accessors are synthesized appropriately—the image will not be CFRetain'd, but the setter will trigger a write barrier.

Example

The following example illustrates the use of properties in several different ways:

- The Link protocol declares a property, `next`.
- MyClass adopts the Link protocol so implicitly also declares the property `next`. MyClass also declares several other properties.
- `creationTimestamp` and `next` are synthesized but use existing instance variables with different names;

- `name` is synthesized, and uses instance variable synthesis (recall that instance variable synthesis is not supported using the 32-bit runtime—see "Property Implementation Directives" (page 51) and "Runtime Differences" (page 60));
- `gratuitousFloat` has a `dynamic` directive—it is supported using direct method implementations;
- `nameAndAge` does not have a `dynamic` directive, but this is the default value; it is supported using a direct method implementation (since it is read-only, it only requires a getter) with a specified name (`nameAndAgeAsString`).

Listing 4-4 Declaring properties for a class

```
@protocol Link
@property id <Link> next;
@end


@interface MyClass : NSObject <Link>
{
    NSTimeInterval intervalSinceReferenceDate;
    CGFloat gratuitousFloat;
    id <Link> nextLink;
}
@property(readonly) NSTimeInterval creationTimestamp;
@property(copy) __strong NSString *name;
@property CGFloat gratuitousFloat;
@property(readonly, getter=nameAndAgeAsString) NSString *nameAndAge;

@end


@implementation MyClass

@synthesize creationTimestamp = intervalSinceReferenceDate, name;
// synthesizing 'name' is an error in legacy runtimes
// in modern runtimes, the instance variable is synthesized

@synthesize next = nextLink;
// uses instance variable "nextLink" for storage

@dynamic gratuitousFloat;
// will warn unless -gratuitousFloat and -setGratuitousFloat: occur in
@implementation


- (CGFloat)gratuitousFloat {
    return gratuitousFloat;
}
- (void)setGratuitousFloat:(CGFloat)aValue {
    gratuitousFloat = aValue;
}


- (NSString *)nameAndAgeAsString {
    return [NSString stringWithFormat:@"%@ (%fs)", [self name],
                [NSDate timeIntervalSinceReferenceDate] -
intervalSinceReferenceDate];
}
```

```
 - (id)init {
    if (self = [super init]) {
        intervalSinceReferenceDate = [NSDate timeIntervalSinceReferenceDate];
    }
    return self;
}

- (void)dealloc {
    [nextLink release];
    [name release];
    [super dealloc];
}

@end
```

Subclassing with Properties

You can override a `readonly` property to make it writable. For example, you could define a class `MyInteger` with a `readonly` property, `value`:

```
@interface MyInteger : NSObject
{
    NSInteger value;
}
@property(readonly) NSInteger value;
@end

@implementation MyInteger
@synthesize value;
@end
```

You could then implement a subclass, `MyMutableInteger`, which redefines the property to make it writable:

```
@interface MyMutableInteger : MyInteger
@property(readwrite) NSInteger value;
@end

@implementation MyMutableInteger
@dynamic value;

- (void)setValue:(NSInteger)newX {
    value = newX;
}
@end
```

Performance and Threading

If you supply your own method implementation, the fact that you declared a property has no effect on its efficiency or thread safety.

If you use synthesized properties, the method implementations generated by the compiler depend on the specification you supply. The declaration attributes that affect performance and threading are `retain`, `assign`, `copy`, and `nonatomic`. The first three of these affect only the implementation of the assignment part of the set method, as *illustrated* below (note that the implementation may not be exactly as shown):

```
// assign
property = newValue;

// retain
if (property != newValue)
{
    [property release];
    property = [newValue retain];
}

// copy
if (property != newValue)
{
    [property release];
    property = [newValue copy];
}
```

The effect of the `nonatomic` attribute depends on the environment. By default, the synthesized accessors are atomic. In a managed memory environment, guaranteeing atomic behavior requires the use of a lock; moreover a returned object is retained and autoreleased. If such accessors are invoked frequently, this may have a significant impact on performance. In a garbage collected environment, most synthesized methods are atomic without incurring this overhead.

It is important to understand that the goal of the atomic implementation is to provide *robust* accessors—it does not guarantee *correctness* of your code. Although "atomic" means that access to the *property* is thread-safe, simply making all the properties in your class atomic does not mean that your *class* or more generally your object graph is "thread safe"—thread safety cannot be expressed at the level of individual accessor methods. For more about multi-threading, see *Threading Programming Guide*.

Property Introspection

When the compiler encounters property declarations, it generates descriptive metadata that is associated with the enclosing class, category or protocol. You can access this metadata using functions that support looking up a property by name on a class or protocol, obtaining the type of a property as an `@encode` string, and copying a list of a property's attributes as an array of C strings. A list of declared properties is available for each class and protocol.

The `Property` structure defines an opaque handle to a property descriptor.

```
typedef struct objc_property *Property;
```

The functions `class_getProperty` and `protocol_getProperty` look up a named property in a class and protocol respectively.

```
Property class_getProperty(Class cls, const char *name);
Property protocol_getProperty(Protocol *proto, const char *name);
```

The `class_copyPropertyList` and `protocol_copyPropertyList` functions return a malloc'd array of pointers to all of the properties declared in a class (including loaded categories) and protocol respectively.

```
Property* class_copyPropertyList(Class cls, uint32_t *count);
Property* protocol_copyPropertyList(Protocol *protocol, uint32_t *count);
```

The `property_getInfo` function returns the name and `@encode` type string of a property.

```
void property_getInfo(Property *property, const char **name, const char **type);
```

The `property_copyAttributeList` function returns a malloc'd array of pointers to C strings which represent a property's compile-time attribute list.

```
const char **property_copyAttributeList(Property *property, unint32_t *count);
```

Runtime Differences

In general the behavior of properties is identical on all runtimes. There is one key difference, modern (64-bit) runtimes supports non-fragile instance variables whereas the legacy runtime does not.

Using the legacy (32-bit) runtime, you must fully declare instance variables in the `@interface` declaration block of a class since all clients of the class including—of course—subclasses must know the full storage details of the class. Since the storage definition of a class cannot change without incurring binary compatibility issues, synthesis of instance variables in the legacy runtime is not supported. For `@synthesize` to work in the legacy runtime, you must either provide an instance variable with the same name and compatible type of the property or specify another existing instance variable in the `@synthesize` statement.

For example:

```
@interface My32BitClass : NSObject
{
    CGFloat gratuitousFloat;
}
@property CGFloat gratuitousFloat;
@end

@implementation My32BitClass
@synthesize gratuitousFloat; // uses the instance variable "gratuitousFloat"
for storage
@end


@interface My64BitClass : NSObject
{
```

```
}
@property CGFloat gratuitousFloat;
@end

@implementation My64BitClass
@synthesize gratuitousFloat; // synthesizes the instance variable
"gratuitousFloat" for storage
@end
```

Protocols

Protocols declare methods that can be implemented by any class. 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

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 that are independent of any specific 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 protocols 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.

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 117), 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, you might want to add support for creating XML representations of objects in your application and for initializing objects from an XML representation:

```
- (NSXMLElement *)XMLRepresentation;
- initFromXMLRepresentation:(NSXMLElement *)xmlString;
```

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` instance must communicate with the objects that represent its cells. The matrix 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` object 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` object wouldn't care what class a cell object belonged to, just that it implemented the methods.

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.

Declaring a Protocol

You declare formal protocols with the `@protocol` directive:

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

For example, you could declare the XML representation protocol like this:

```
@protocol MyXMLSupport
- (NSXMLElement *)XMLRepresentation;
- initFromXMLRepresentation:(NSXMLElement *)XMLElement;
@end
```

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

Optional Protocol Methods

Protocol methods can be marked as optional using the `@optional` keyword. Corresponding to the `@optional` modal keyword, there is a `@required` keyword to formally denote the semantics of the default behavior. You can use `@optional` and `@required` to partition your protocol into sections as you see fit. If you do not specify any keyword, the default is `@required`.

```
@protocol MyProtocol

- (void)requiredMethod;

@optional
- (void)anOptionalMethod;
- (void)anotherOptionalMethod;

@required
- (void)anotherRequiredMethod;

@end
```

Informal Protocols

In addition to formal protocols, you can also define an **informal** protocol by grouping the methods in a category declaration:

```
@interface NSObject ( MyXMLSupport )
- (NSXMLElement *) XMLRepresentation;
- initFromXMLRepresentation:(NSXMLElement *)XMLElement;
@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 may be useful when implementing all the methods is optional, such as for a delegate, but (on Mac OS X v10.5 and later) it is typically better to use a formal protocol with optional methods.

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 *myXMLSupportProtocol = @protocol(MyXMLSupport);
```

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.

Adopting a Protocol

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. A class is said to **adopt** a formal protocol if in its declaration it lists the protocol 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 >
```

A class can adopt more than one protocol; names in the protocol list are separated by commas.

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

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

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
```

Conforming to a Protocol

A class is said to **conform** to a formal protocol if it adopts the protocol or inherits from another class that adopts it. 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 required methods declared in the protocols it adopts, saying that a class or an instance conforms to a protocol is equivalent 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(MyXMLSupport)]  ) {
    // Object does not conform to MyXMLSupport protocol
    // If you are expecting receiver to implement methods declared in the
    //  MyXMLSupport protocol, this is probably an error
}
```

(Note that there is also a class method with the same name—`conformsToProtocol:`.)

The `conformsToProtocol:` test is like the `respondsToSelector:` 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 all the methods in the protocol, `conformsToProtocol:` can be more efficient than `respondsToSelector:`.

The `conformsToProtocol:` test is also 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 <MyXMLSupport> 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 required 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.

Fast Enumeration

Objective-C 2.0 provides a language feature that allows you to efficiently and safely enumerate over the contents of a collection using a concise syntax.

The for…in Feature

Objective-C 2.0 provides a language feature that allows you to enumerate over the contents of a collection. The syntax is defined as follows:

```
for ( Type newVariable in expression ) { stmts }
```

or

```
Type existingItem;
for ( existingItem in expression ) { stmts }
```

In both cases, *expression* yields an object that conforms to the `NSFastEnumeration` protocol, typically an array or enumerator (the Cocoa collection classes—`NSArray`, `NSDictionary`, and `NSSet`—adopt this protocol, as does `NSEnumerator`). Any class whose instances provide access to a collection of other objects can adopt the `NSFastEnumeration` protocol. It should be obvious that in the cases of `NSArray` and `NSSet` the enumeration is over their contents. Other classes should make clear what property is iterated over—for example, `NSDictionary` and the Core Data class `NSManagedObjectModel` provide support for fast enumeration; `NSDictionary` enumerates its keys, and `NSManagedObjectModel` enumerates its entities.

There are several advantages to using fast enumeration:

- The enumeration is considerably more efficient than, for example, using `NSEnumerator` directly.
- The syntax is concise.
- Enumeration is "safe"—the enumerator has a mutation guard so that if you attempt to modify the collection during enumeration, an exception is raised.

Since mutation of the object during iteration is forbidden, you can perform multiple enumerations concurrently.

Using Fast Enumeration

The following code example illustrates using fast enumeration with `NSArray` and `NSDictionary` objects.

```
NSArray *array = [NSArray arrayWithObjects:
        @"One", @"Two", @"Three", @"Four", nil];

for (NSString *element in array) {
    NSLog(@"element: %@", element);
}

NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
    @"quattuor", @"four", @"quinque", @"five", @"sex", @"six", nil];

NSString *key;
for (key in dictionary) {
    NSLog(@"English: %@, Latin: %@", key, [dictionary valueForKey:key]);
}
```

You can also use `NSEnumerator` objects with fast enumeration, as illustrated in the following example:

```
NSArray *array = [NSArray arrayWithObjects:
        @"One", @"Two", @"Three", @"Four", nil];

NSEnumerator *enumerator = [array reverseObjectEnumerator];
for (NSString *element in enumerator) {
    if ([element isEqualToString:@"Three"]) {
        break;
    }
}

NSString *next = [enumerator nextObject];
// next = "Four"
```

For collections or enumerators that have a well-defined order—such as `NSArray` or `NSEnumerator` instance derived from an array—the enumeration proceeds in that order, so simply counting iterations will give you the proper index into the collection if you need it.

```
NSArray *array = /* assume this exists */;
NSInteger index = 0;
BOOL ok = NO;

for (id element in array) {
    if (/* some test for element */) {
        ok = YES;
        break;
    }
    index++;
}

if (ok) {
    NSLog(@"Test passed by element at index %d", index");
}
```

How Messaging Works

This chapter describes how the message expressions are converted into `objc_msgSend` function calls, and how you can refer to methods by name. It then explains how you can take advantage of `objc_msgSend`, and the roles of `self` and `super`.

The objc_msgSend Function

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.

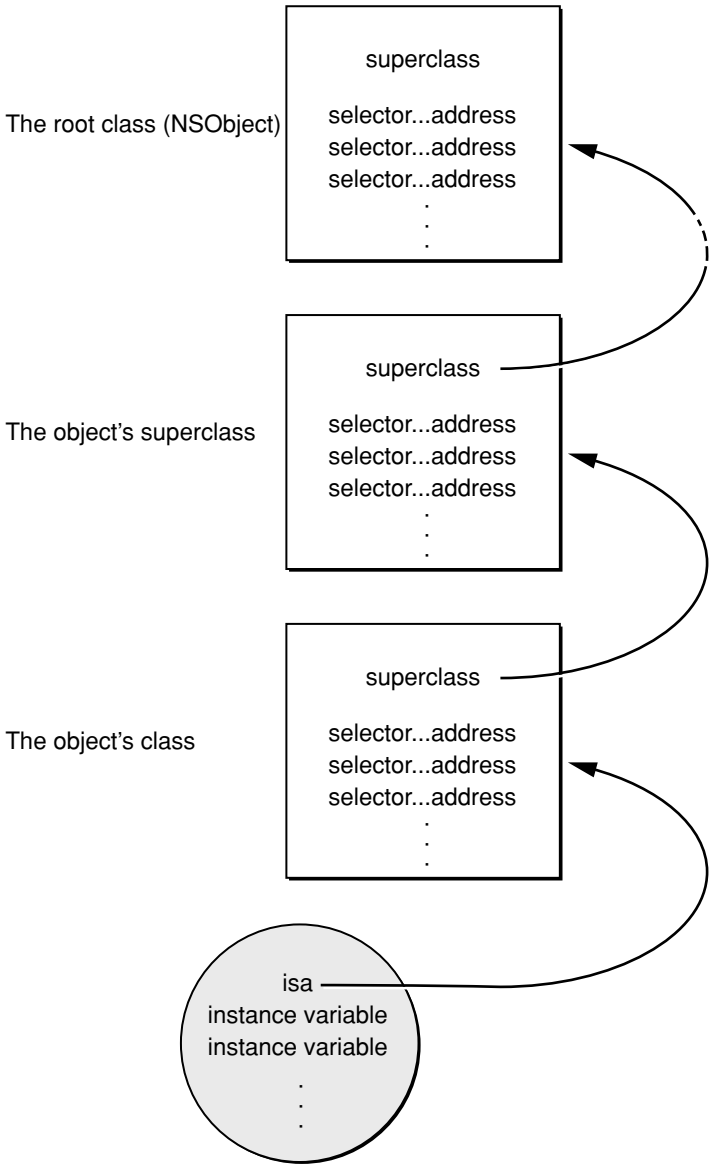
- A class **dispatch table**. This table has entries that associate method selectors with the class-specific addresses of the methods they identify. The selector for the `setOrigin::` method is associated with the address of (the procedure that implements) `setOrigin::`, the selector for the `display` 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 7-1.

Figure 7-1 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 Design Pattern

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` instance 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` object sends a message instructing the application to do something. To do this, an `NSButtonCell` object 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` instance 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 button cell 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 "Forwarding" (page 112) for more information.

Dynamic Method Resolution

You can provide implementations of class and instance methods at runtime using dynamic method resolution. See "Dynamic Method Resolution" (page 116) in "The Runtime System" (page 103) for more details.

Using Hidden Arguments

When the `objc_msgSend` 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:

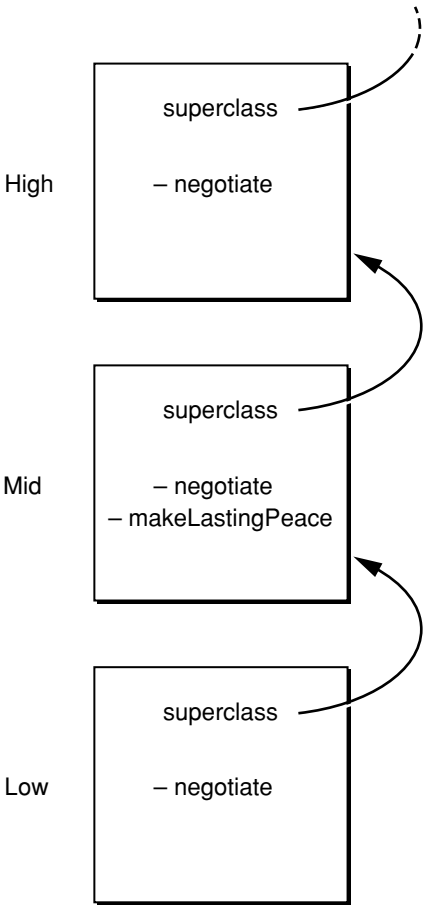
- `self` searches for the method implementation in the usual manner, starting in the dispatch table of the receiving object's class. In the example above, it would begin with the class of the object receiving the reposition message.
- `super` starts the search for the method implementation in a very 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 7-2:

Figure 7-2 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 = [[Rectangle 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 = [[Rectangle alloc] init]; // GOOD
    [newInstance setColor:color];
    return [newInstance autorelease];
}
```

In fact, rather than sending the `alloc` message to the class in a class method, it's often better to send `alloc` to `self`. 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 (for example, the `array` method of `NSArray` is inherited by `NSMutableArray`).

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

See "Allocating and Initializing Objects" (page 104) for more information about object allocation.

Enabling Static Behavior

This chapter 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 75). 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 35).

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 1-2 (page 25), 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.

Exception Handling

Objective-C provides support for exception handling and thread synchronization, which are explained in this article and "Threading" (page 95). 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 9-1.

Listing 9-1 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 re-throw it to the next exception handler.

4. Performs any clean-up processing that must always be performed, whether exceptions were thrown or not.

Threading

Objective-C provides support for thread synchronization and exception handling, which are explained in this article and "Exception Handling" (page 91). 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.

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 10-1 shows an example of code that uses `self` as the mutex to synchronize access to the instance methods of the current object. You can take a similar approach to synchronize the class methods of the associated class, using the Class object instead of `self`. In the latter case, of course, only one thread at a time is allowed to execute a class method because there is only one class object that is shared by all callers.

Listing 10-1 Locking a method using self

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

```
        ...
    }
}
```

Listing 10-2 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 10-2 Locking a method using _cmd

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

Listing 10-3 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 10-3 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

Overview

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" (page 91).

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 11-1 illustrates this.

Listing 11-1 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* initial_greeting_text) {
            greeting_text = [[NSString alloc]
initWithUTF8String:initial_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's 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 pre-declares 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 "Defining a Class" (page 35) and "Protocols" (page 63), 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 75). 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 112) 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 2.0 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.

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` object 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`, then 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:

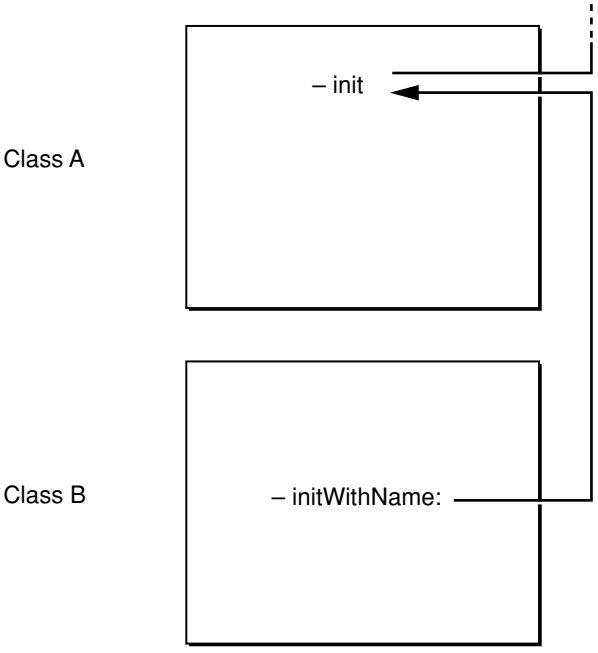
```
- initWithName:(char *)string
{
    if ( self = [super init] ) {
        name = (char *)NSZoneMalloc([self zone],
            strlen(string) + 1);
        strcpy(name, string);
    }
    return self;
```

```
}
```

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 12-1:

Figure 12-1 Incorporating an Inherited Initialization Method

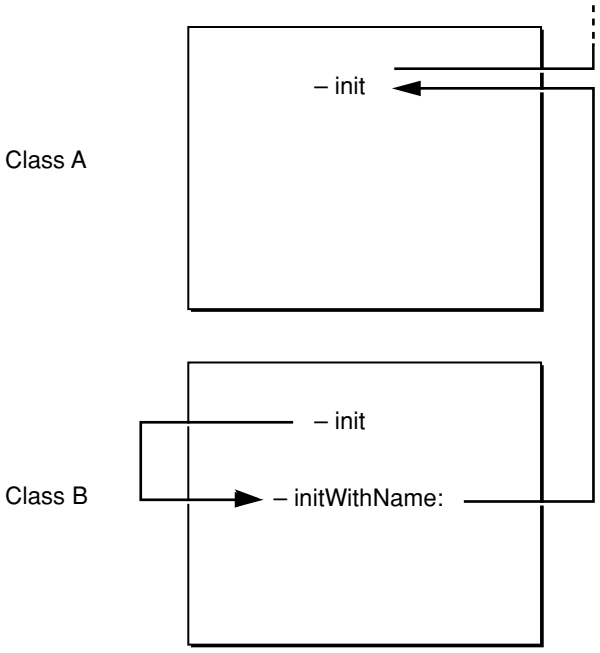


A class must also make sure that all inherited initialization methods work. For example, if class A defines an `init` method and its subclass B defines an `initWithName:` method, as shown in Figure 12-1, B must also make sure that an `init` message successfully initializes B instances. The easiest way 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 12-2 includes B's version of `init`:

Figure 12-2 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

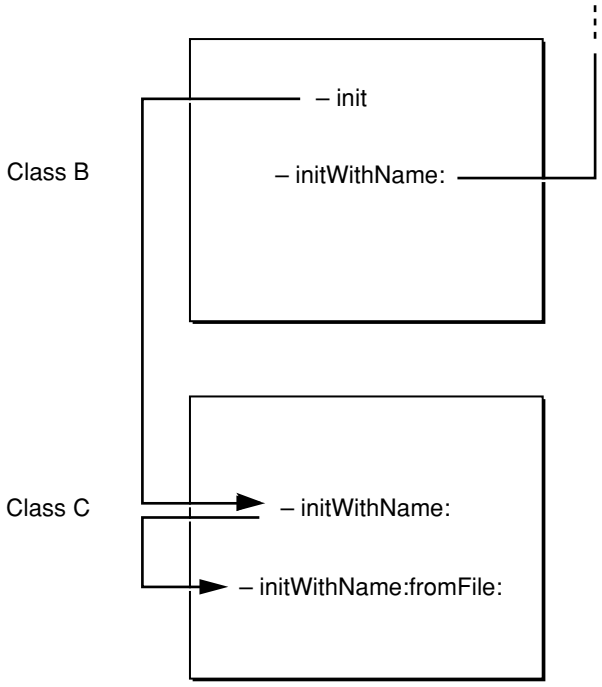
In the example above, `initWithName:` would be the **designated initializer** for its class (class B). The designated initializer is the method in each class that guarantees inherited instance variables are initialized (by sending a message to `super` to perform an inherited method). It's also the method 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 12-3:

Figure 12-3 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:`:

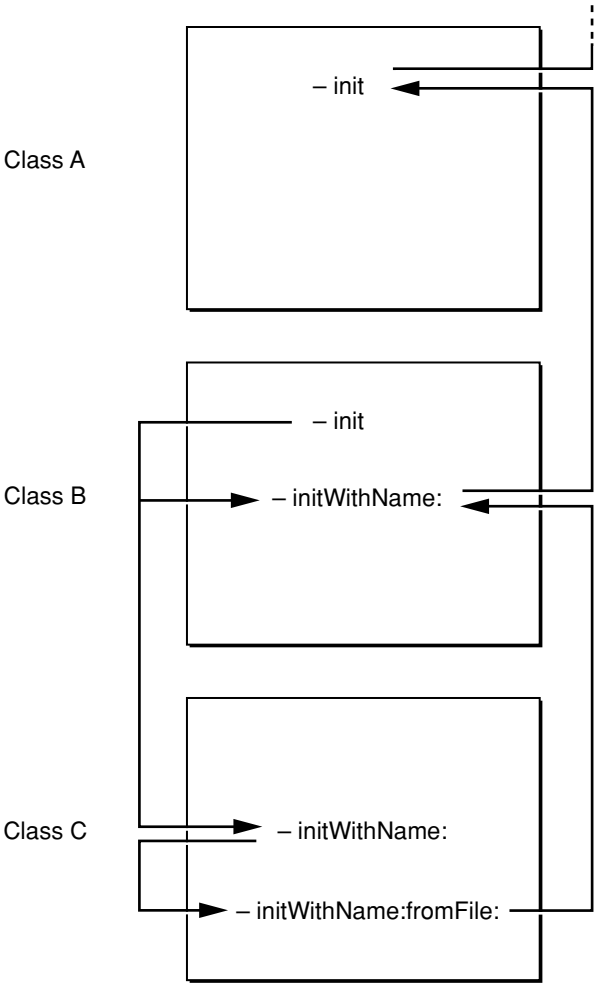
```
- initWithName:(char *)string fromFile:(char *)pathname
{
    if ( self = [super initWithName:string] )
        ...
}
```

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 12-4 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 12-4 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 are often referred to as **convenience constructors** and typically take the form + *className...* where *className* is the name of the class. For example, `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, ...;
```

Important: It is important to understand the memory management implications of using these methods if you do not use garbage collection (see "Memory Management" (page 111)). You must read *Memory Management Programming Guide for Cocoa* to understand the policy that applies to these convenience constructors.

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 105), 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;
}
```

Memory Management

In an Objective-C program, it is important to ensure that objects are deallocated when they are no longer needed—otherwise your application's memory footprint is larger than necessary. It is also important to ensure that you do not deallocate objects while they're still being used.

Objective-C 2.0 offers two paradigms for memory management that allow you to meet these goals:

- **Managed memory**, where you are ultimately responsible for determining the lifetime of objects using a technique called **reference counting**.

Managed memory is described in *Memory Management Programming Guide for Cocoa*.

- **Garbage collection**, where you pass responsibility for determining the lifetime of objects to an automatic "collector."

  Garbage collection is described in *Garbage Collection Programming Guide*.

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.

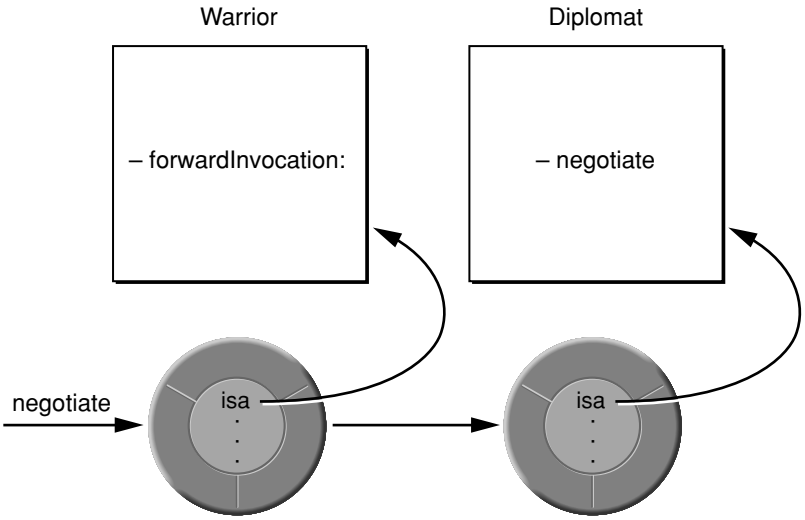
Note: The `forwardInvocation:` method gets to handle messages only if they don't invoke an existing method in the nominal receiver. If, for example, you want your object to forward `negotiate` messages to another object, it can't have a `negotiate` method of its own. If it does, the message will never 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 12-5 (page 114), an object that responds to a message by forwarding it appears to borrow or "inherit" a method implementation defined in another class.

Figure 12-5 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 provides most of the features that you typically want from 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 117) 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 12-5 (page 114).)

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 Method Resolution

There are situations where you might want to provide an implementation of a method dynamically. For example, the Objective-C declared properties feature (see "Declared Properties" (page 49)) includes the `@dynamic` directive:

```
@dynamic propertyName;
```

which tells the compiler that the methods associated with the property will be provided dynamically.

You can implement the methods `resolveInstanceMethod:` and `resolveClassMethod:` to dynamically provide an implementation for a given selector for an instance and class method respectively.

An Objective-C method is simply a C function that take at least two arguments—`self` and `_cmd`. You can add a function to a class as a method using the function `class_addMethod`. Therefore, given the following function:

```
void dynamicMethodIMP(id self, SEL _cmd) {
    // implementation ....
}
```

you can dynamically add it to a class as a method (called `resolveThisMethodDynamically`) using `resolveInstanceMethod:` like this:

```
@implementation MyClass
+ (BOOL)resolveInstanceMethod:(SEL)aSEL
{
    if (aSEL == @selector(resolveThisMethodDynamically)) {
          class_addMethod([self class], aSEL, (IMP) dynamicMethodIMP, "v@:");
          return YES;
    }
    return [super resolveInstanceMethod:aSEL];
}
@end
```

Forwarding methods (as described in "Forwarding" (page 112)) and dynamic method resolution are, largely, orthogonal. A class has the opportunity to dynamically resolve a method before the forwarding mechanism kicks in. If `respondsToSelector:` or `instancesRespondToSelector:` is invoked, the dynamic method resolver is given the opportunity to provide an `IMP` for the selector first. If you implement `resolveInstanceMethod:` but want particular selectors to actually be forwarded via the forwarding mechanism, you return `NO` for those selectors.

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 *Mac OS X ABI 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

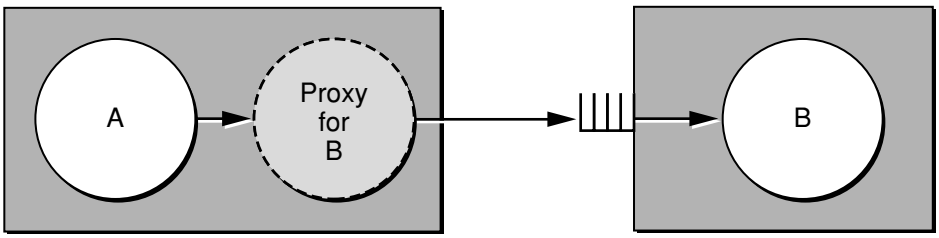
Remote messaging in Objective-C requires a runtime system that can establish connections between objects in different address spaces, recognize when a message is intended for an object in a remote address space, and transfer data from one address space to another. It must also mediate between the separate 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 12-6 (page 118), 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 12-6 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 and *Distributed Objects Programming Topics*.

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.

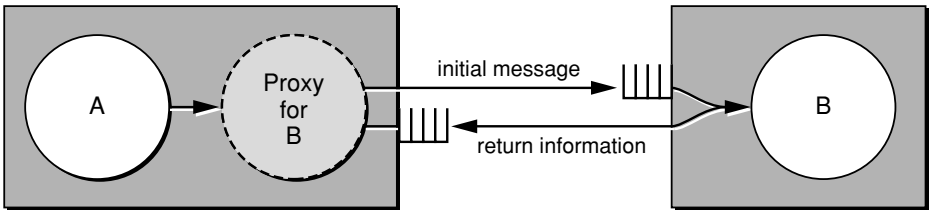
Synchronous and Asynchronous Messages

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 12-7 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 (non-pointers) 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—any type, 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 12-1 Objective-C type encodings

Code	Meaning
`c`	A `char`
`i`	An `int`
`s`	A `short`
`l`	A `long` `l` is treated as a 32-bit quantity on 64-bit programs.
`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 *`)

Code	Meaning
`@`	An object (whether statically typed or typed `id`)
`#`	A class object (`Class`)
`:`	A method selector (`SEL`)
[*array type*]	An array
{*name=type...*}	A structure
(*name=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)

Important: Objective-C does not support the `long double` type. `@encode(long double)` returns `d`, which is the same encoding as for `double`.

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 12-2 for type qualifiers when they're used to declare methods in a protocol.

Table 12-2 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 other chapters in this document. .

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`.
`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.

Directive	Definition
`@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*.
`@"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
```

Deprecation Syntax

Syntax is provided to mark methods as deprecated:

```
@interface SomeClass
-method __attribute__((deprecated));
@end
```

or:

```
#include <AvailabilityMacros.h>
@interface SomeClass
-method DEPRECATED_ATTRIBUTE;  // or some other deployment-target-specific macro
@end
```

This syntax is available only in Objective-C 2.0 and later.

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 129) and "Categories" (page 130))
- 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.
- Argument and return types are declared using the C syntax for type casting.
- Arguments are declared after colons (:), for example:

  ```
  - (void)setWidth:(int)newWidth height:(int)newHeight
  ```

  Typically, a label describing the argument precedes the colon—the following example is valid but is considered bad style:

  ```
  - (void)setWidthAndHeight:(int)newWidth :(int)newHeight
  ```

  Both labels and colons are considered part of the method name.
- 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.

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.

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.

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.

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 views 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.

factory method Same as class method.

factory object Same as class object.

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 and informal protocol.

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.

Document Revision History

This table describes the changes to *The Objective-C 2.0 Programming Language*.

Date	Notes
2008-07-08	Corrected typographical errors.
2008-06-09	Made several minor bug fixes and clarifications, particularly in the "Properties" chapter.
2008-02-08	Extended the discussion of properties to include mutable objects.
2007-12-11	Corrected minor errors.
2007-10-31	Provided an example of fast enumeration for dictionaries and enhanced the description of properties.
2007-07-22	Added references to documents describing new features in Objective-C 2.
2007-03-26	Corrected minor typographical errors.
2007-02-08	Clarified the discussion of sending messages to nil.
2006-12-05	Clarified the description of Code Listing 3-3.
2006-05-23	Moved the discussion of memory management to "Memory Management Programming Guide for Cocoa."
2006-04-04	Corrected minor typographical errors.
2006-02-07	Corrected minor typographical errors.
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”.

Date	Notes
	Clarified example in Listing 11-1 (page 97).
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 2.0 Runtime Reference*.
	Added examples of thread synchronization approaches to "Synchronizing Thread Execution".
	Clarified when the `initialize` method is called and provided a template for its implementation in "Initializing a Class Object".
	Added exception and synchronization grammar to "Grammar".
	Replaced `conformsTo:` with `conformsToProtocol:` throughout document.
2004-02-02	Corrected typos in "An exception handler".
2003-09-16	Corrected definition of `id`.
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".
	Documented the language support for concatenating constant strings in "Compiler Directives" (page 128).
	Moved "Memory Management" before "Retaining Objects".
	Corrected the descriptions for the `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.

Date	Notes
	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*.

Document Revision History

Index

Symbols

+ (plus sign) before method names 36
- (minus sign) before method names 36
// marker comment 128
@"" directive (string declaration) 129
_cmd 81, 90, 132
__cplusplus preprocessor constant 99
__OBJC__ preprocessor constant 99

A

abstract classes 26, 65
action messages 80
adaptation 24
adopting a protocol 68, 131
alloc method 29, 105
allocating memory 110
allocWithZone: method 105
anonymous objects 65
argument types
  and dynamic binding 89
  and selectors 89
  declaring 36
  encoding 123
  in declarations 132
arguments
  during initialization 106
  hidden 81, 132
  in remote messages 120
  type modifiers 121
  variable 16

B

behaviors
  inheriting 115
  of Cocoa objects 118
  of NSObject class 104
  overriding 122
BOOL data type 127
bycopy type qualifier 125, 131
byref type qualifier 125, 131

C

.c extension 10
C language support 9
C++ language support 97–101
@catch() directive 91, 92, 129
categories 45–47
  See also subclasses
  and informal protocols 67
  declaration of 45–46
  declaring 130
  defining 130
  implementation of 45–46
  loading dynamically 117
  naming conventions 133
  of root classes 47
  scope of variables 46
  uses of 46, 67
Class data type 28, 127
@class directive 37, 129
class method 104
class methods
  and selectors 78
  and static variables 31
  declaration of 36, 132
  defined 28
  of root class 32
  using self 85
class object
  defined 23
  initializing 31
class objects 28–32
  and root class 32
  and root instance methods 32, 47
  and static typing 32

as receivers of messages 33
variables and 30
classes 23–33
root. *See* root classes
abstract 26
and inheritance 23, 25
and instances 23
and namespaces 133
declaring 36–38, 129, 133
defining 35–43, 130
designated initializer of 108
examples 13
identifying 14
implementation of 35, 38
instance methods 28
interfaces 35
introspection 14, 27
loading dynamically 117
naming conventions 132
subclasses 23
superclass 23
uses of 32
comment marker (//) 128
compiler directives, summary of 128
conforming to protocols 63
conformsToProtocol: method 104, 119
const type qualifier 125
conventions of this book 11
customization with class objects 29–30

D

data members. *See* instance variables
data structures. *See* instance variables
data types defined by Objective-C 127
designated initializer 108–110
development environment 10
directives, summary of 128–129
dispatch tables 76
dispatching messages 75
distributed objects 118
doesNotRecognizeSelector: method 113
dynamic binding 18
dynamic loading 117
dynamic typing 14

E

@encode() directive 123, 129
encoding methods 123
@end directive 36, 38, 128
exceptions 91–93
catching 92
clean-up processing 93
compiler switch 91, 95
exception handler 92
nesting exception handlers 93
NSException 91, 92
synchronization 96
system requirements 91, 95
throwing 92, 93

F

@finally directive 91, 93, 129
formal protocols 66, 131
See also protocols
forwarding messages 112–116
forwardInvocation: method 112

G

GNU Compiler Collection 10

H

.h extension 35, 132
hidden arguments 81, 132

I

id data type 127
and method declarations 132
and static typing 27, 88
as default method return type 36
of class objects 28
overview 13
IMP data type 127
@implementation directive 38, 128
implementation files 35, 39
implementation
of classes 38–43, 130
of methods 39, 132
#import directive 37, 128
in type qualifier 125, 131
#include directive 37
#include directive *See* #import directive

informal protocols 67
  See also protocols
inheritance 23–26
  and forwarding 115
  of instance variables 106
  of interface files 37
`init` method 29, 105
`initialize` method 31
initializing objects 84, 110
`inout` type qualifier 125, 131
instance methods 28
  and selectors 78
  declaration of 132
  declaring 36
  naming conventions 133
  syntax 36
instance variables
  declaring 25, 36, 128
  defined 13
  encapsulation 41
  inheriting 25, 42
  initializing 106
  naming conventions 133
  of the receiver 17
  public access to 129
  referring to 39
  scope of 13, 40–43, 128
instances of a class
  allocating 104–105
  creating 29
  defined 23
  initializing 29, 104–111
instances of the class
  See also objects
`@interface` directive 36, 128, 129
interface files 37, 129
introspection 14, 27
`isa` instance variable 14, 29, 77
`isKindOfClass:` method 27, 33, 104
`isMemberOfClass:` method 27, 104

M

`.m` extension 10, 35, 132
memory
  allocating 105, 110
message expressions 15, 127
message receivers 127
messages 75–81
  See also methods
  and selectors 18
  and static typing 88
  asynchronous 120
  binding 88
  defined 15, 127
  encoding 123
  forwarding 112–116
  remote 117
  sending 15, 16
  synchronous 120
  syntax 127
  varying at runtime 18, 79
messaging 75–85
  avoiding errors 80
  to remote objects 117–122
metaclass object 29
method implementations 39, 132
`methodForSelector:` method 90, 104
methods 13
  See also behaviors
  See also messages
  adding with categories 45
  address of 90
  and selectors 18, 78
  and variable arguments 37
  argument types 79
  arguments 89, 106
  calling `super` 106
  class methods 28
  declaring 36, 132
  encoding 123
  hidden arguments 132
  implementing 39, 132
  inheriting 25
  instance methods 28
  naming conventions 133
  overriding 26
  return types 79, 89
  returning values 14, 16
  selecting 18
  specifying arguments 15
  using instance variables 39
minus sign (-) before method names 36
`.mm` extension 10
multiple inheritance 113

N

name spaces 133
naming conventions 132
`Nil` constant 128
`nil` constant 14, 128
`NO` constant 128
NSBundle 117

NSClassFromString function 33
NSException 91, 92
NSInvocation 112
NSObject 23, 24, 104
NSSelectorFromString function 78
NSStringFromSelector function 78

O

objc_loadModules function 117
objc_msgSend function 75, 77
objc_object structure 76
object 13
object identifiers 13
Objective-C 9
Objective-C++ 97
objects 13, 23
  See also class objects, Protocol objects
  allocating memory for 105
  anonymous 65
  creating 29
  customizing 29
  designated initializer 108
  dynamic typing 14, 87
  examples 13
  initializing 29, 84, 105, 110
  initializing a class object 31
  instance variables 17
  introspection 27
  method inheritance 25
  Protocol 68
  remote 65
  static typing 87
  surrogate 114–115
oneway type qualifier 125, 131
out type qualifier 125, 131
overriding methods 26

P

parameters. *See* arguments
performSelector: method 79
performSelector:withObject: method 79
performSelector:withObject:withObject: method 79
plus sign (+) before method names 36
polymorphism
  defined 17
precompiled headers 37
preprocessor directives, summary of 128
@private directive 41, 128
procedures. *See* methods
@protected directive 41, 129
@protocol directive 72, 119, 128, 129, 131
Protocol objects 68
protocols 63–72
  adopting 70, 131
  conforming to 63, 69, 70
  declaring 63, 131
  formal 66
  forward references to 72, 131
  incorporating other protocols 70–71, 131
  informal 67
  naming conventions 133
  type checking 70
  uses of 63–72
proxy objects 114, 118
@public directive 41, 129

R

receivers of messages
  and class names 33
  defined 127
  in messaging expression 15
  in messaging function 75
  instance variables of 17
remote messages 117
remote objects 65
remote procedure calls (RPC) 120
respondsToSelector: method 80, 104
return types
  and messaging 79
  and statically typed objects 89
  declaration of 36
  encoding 123
root classes
  See also NSObject
  and class interface declarations 36
  and inheritance 23
  categories of 47
  declaration of 130
runtime system
  functions 104
  overview 103

S

SEL data type 78, 127
@selector() directive 78, 129

selectors 78
  and hidden arguments 81
  and messaging errors 80
  defined 18
  in messaging function 75
self 81, 82, 85, 90, 132
Smalltalk 9
specialization 24
static type checking 88
static typing 87–90
  and instance variables 40
  in interface files 38
  introduced 27
  to inherited classes 89
strings, declaring 129
subclasses 23
super variable 82, 84, 132
superclasses
  See also abstract classes
  and inheritance 23
  importing 37
surrogate objects 114
synchronization 95–96
  compiler switch 91, 95
  exceptions 96
  mutexes 95
  system requirements 91, 95
@synchronized() directive 95, 129

T

target-action paradigm 79–80
targets 80
this keyword 100
@throw directive 91, 92, 93, 129
@try directive 91, 129
type checking
  class types 87, 88
  protocol types 70
type encoding 123–125
type introspection 27
types defined by Objective-C 127

U

unsigned int data type 132

V

variable arguments 37
void data type 132

Y

YES constant 128