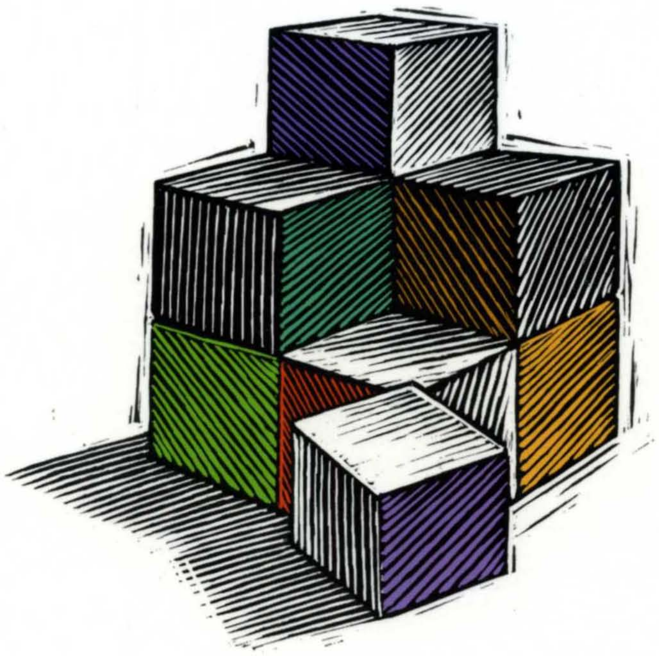
OBJECT-ORIENTED PROGRAMMING AND THE OBJECTIVE C LANGUAGE



NEXTSTEP™

Object-Oriented Software

NeXTSTEP™ OBJECT-ORIENTED PROGRAMMING AND THE OBJECTIVE C LANGUAGE

NeXTSTEP Developer's Library *Release 3*
NeXT Computer, Inc.

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California • New York • Don Mills, Ontario
Wokingham, England • Amsterdam • Bonn • Sydney • Singapore • Tokyo • Madrid
San Juan • Paris • Seoul • Milan • Mexico City • Taipei

This manual describes NeXTSTEP Release 3.

Written by NeXT Publications.

This manual was designed, written, and produced on NeXT computers. Proofs were printed on a NeXT 400 dpi Laser Printer and NeXT Color Printer. Final pages were transferred directly from a NeXT floppy disk to film using NeXT computers and an electronic imagesetter.

2 3 4 5 6–AL–96959493
Second printing, July 1993

ISBN 0-201-63251-9

Library of Congress Cataloging-in-Publication Data

NeXTSTEP object-oriented programming and the objective C Language, release 3 / NeXT Computer, Inc.
p. cm. -- (NeXTSTEP developer's library)
Includes index.
ISBN 0-201-63251-9
1. Object-oriented programming. 2. NeXTstep. 3. C (Computer program language) 4. NeXT (Computer)--Programming. I. NeXT Computer, Inc. II. Series.
QA76.64.N49 1993
005.4'3--dc20 93-7629
CIP

Contents

Introduction

Object-oriented programming, like most interesting new developments, builds on some old ideas, extends them, and puts them together in novel ways. The result is many faceted and a clear step forward for the art of programming. An object-oriented approach makes programs more intuitive to design, faster to develop, more amenable to modifications, and easier to understand. It leads not only to new ways of constructing programs, but also to new ways of conceiving the programming task.

Nevertheless, object-oriented programming presents some formidable obstacles to those who would like to understand what it's all about or begin trying it out. It introduces a new way of doing things that may seem strange at first, and it comes with an extensive terminology that can take some getting used to. The terminology will help in the end, but it's not always easy to learn. Moreover, there are as yet few full-fledged object-oriented development environments available to try out. It can be difficult to get started.

That's where this book comes in. It's designed to help you become familiar with object-oriented programming and get over the hurdle its terminology presents. It spells out some of the implications of object-oriented design and tries to give you a flavor of what writing an object-oriented program is really like. It fully documents the Objective C language, an object-oriented programming language based on standard C, and introduces the most extensive object-oriented development environment currently available—NeXTSTEP™.

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

- Learning about object-oriented programming,
- Finding out about the NeXTSTEP development environment, or
- Programming in Objective C.

NeXT supplies its own compiler for the Objective C language (a modification of the GNU C compiler) and a run-time system to carry out the dynamic functions of the language. It has tested and made steady improvements to both over the years; this book describes the latest release (Release 3), which includes provisions for declaring and adopting protocols and setting the scope of instance variables.

Throughout this manual and in other NeXT documentation, the term "Objective C" refers to the language as implemented for the NeXTSTEP development environment and presented here.

The Development Environment

Every object-oriented development environment worthy of the name consists of at least three parts:

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

NeXTSTEP comes with an extensive library. It includes several software kits containing definitions for objects that you can use "off the shelf" or adapt to your program's needs. The kits include the Application Kit™ for building a graphical user interface, the Database Kit™ for interacting with a database server, the 3D Graphics Kit™ for constructing and manipulating three-dimensional images, the Sound Kit™ for recording, editing, and playing sounds, and others. All these kits, and more, are documented in the *NeXTSTEP General Reference.*

NeXTSTEP also includes some exceptional development tools for putting together applications. There's Interface Builder™, a program that lets you design an application graphically and assemble its user interface on-screen, and Project Builder, a project-management program that provides graphical access to the compiler, the debugger, documentation, a program editor, and other tools. These programs are documented in *NeXTSTEP Development Tools and Techniques*.

This book is about the third component of the development environment—the programming language. All NeXTSTEP software kits are written in the Objective C language. To get the benefit of the kits, applications must also use Objective C.

Objective C is implemented as set of extensions to the C language. It's designed to give C a full capability for object-oriented programming, and to do so in a simple and straightforward way. Its additions to C are few and are mostly based on Smalltalk, one of the first object-oriented programming languages.

This book 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. There are many good books available on C; this manual doesn't attempt to duplicate them.

Because this isn't a book 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 standard C that you won't be hampered if you're not an experienced C programmer.

Why Objective C

The Objective C language was chosen for the NeXTSTEP development environment for a variety of reasons. First and foremost, it's an object-oriented language. The kind of functionality that's packaged in the NeXTSTEP software kits can only be delivered through object-oriented techniques. This manual will explain how the kits work and why this is the case.

Second, because Objective C is an extension of standard ANSI C, existing C programs can be adapted to use the software kits without losing any of the work that went into their original development. Since Objective C incorporates C, you get all the benefits of C when working within Objective C. You can choose when to do something in an object-oriented way (define a new class, for example) and when to stick to procedural programming techniques (define a structure and some functions instead of a class).

Moreover, Objective C is a simple language. Its syntax is small, unambiguous, and easy to learn. Object-oriented programming, with its self-conscious terminology and emphasis on abstract design, often presents a steep learning curve to new recruits. A well-organized language like Objective C can make becoming a proficient object-oriented programmer that much less difficult. The size of this manual is a testament to the simplicity of Objective C. It's not a big book—and Objective C is fully documented in just two of its chapters.

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

- Objective C supports an open style of dynamic binding, a style than can accommodate a simple architecture for interactive user interfaces. Messages are not necessarily constrained by either the class of the receiver or the method selector, so a software kit can allow for user choices at run time and permit developers freedom of expression in their design. (Terminology like "dynamic binding," "message," "class," "receiver," and "selector" will be explained in due course in this manual.)

- Objective C's dynamism enables the construction of sophisticated development tools. An interface to the run-time system provides access to information about running applications, so it's possible to develop tools that monitor, intervene, and reveal the underlying structure and activity of Objective C applications. Interface Builder could not have been developed with a less dynamic language. (The full interface to the run-time system is documented in the *NeXTSTEP General Reference* manual.)

How the Manual is Organized

This manual is divided into five chapters and three appendices. The chapters are:

- Chapter 1, "Object-Oriented Programming," discusses the rationale for object-oriented programming languages and introduces much of the terminology. It develops the ideas behind object-oriented programming techniques. If you're already familiar with object-oriented programming and are interested only in Objective C, you may want to skip this chapter and go directly to Chapter 2.

- Chapter 2, "The Objective C Language," describes the basic concepts and syntax of Objective C. It covers many of the same topics as Chapter 1, but looks at them from the standpoint of the Objective C language. It reintroduces the terminology of object-oriented programming, but in the context of Objective C.

- Chapter 3, "Objective C Extensions," concentrates on two of the principal innovations introduced into the language as part of NeXTSTEP Objective C—categories and protocols. It also takes up static typing and lesser used aspects of the language.

- Chapter 4, “The Run-Time System,” looks at the Object class and how Objective C programs interact with the run-time system. In particular, it examines the paradigms for allocating and initializing new objects, dynamically loading new classes at run time, archiving objects, and forwarding messages to other objects.
- Chapter 5, “Programming in Objective C,” tries to give a flavor of what programming with the NeXTSTEP software kits is like. As an example, it describes how you would go about programming with the Application Kit, the software kit that’s used to build and run a graphical user interface.

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

- Appendix A, “Objective C Language Summary,” lists and briefly comments on all the Objective C extensions to the C language.
- Appendix B, “Reference Manual for the Objective C Language,” presents, uncommented, a formal grammar of the Objective C extensions to the C language. This reference manual is meant to be read as a companion to the reference manual for C presented in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall.
- Appendix C, “The Object Class,” is a full specification of the Object class, the root class that all other Objective C classes inherit from. This specification is equivalent to the one found in the *NeXTSTEP General Reference* manual and is presented here for convenience.

Conventions

Where this manual discusses functions, methods, and other programming elements, it makes special use of bold and italic fonts. **Bold** 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 method syntax is shown, mainly in Appendix C, the method name is bold, parameters are italic, and other elements (mainly data types) are in regular font. For example:

– **write:**(NXTypedStream *)*stream*

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

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    . . .
    return self;
}
```

The conventions used in the reference manual are described there (Appendix B).

1 Object-Oriented Programming

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

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

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

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

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

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

Interface and Implementation

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

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

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

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

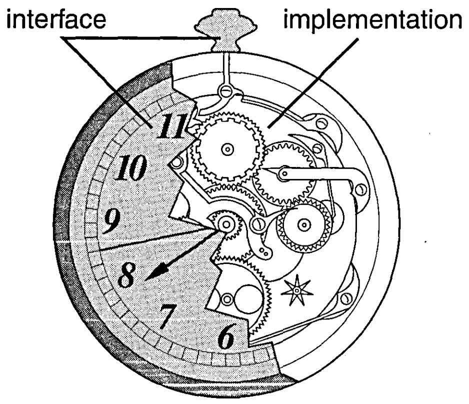


Figure 1. Interface and Implementation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Object Model

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

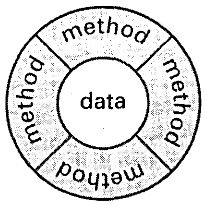


Figure 2. An Object

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

Terminology

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

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

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

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

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

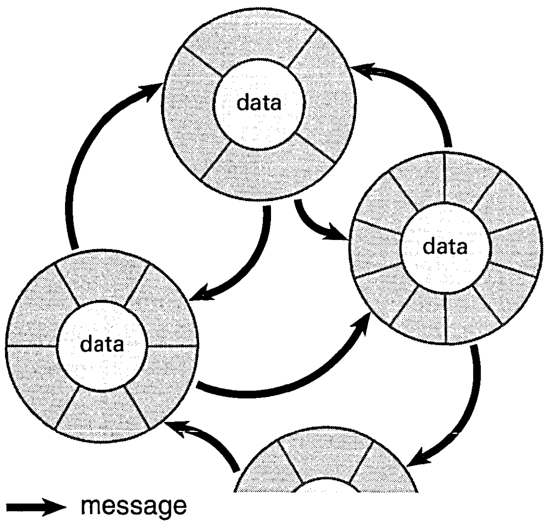


Figure 3. Object Network

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

The Messaging Metaphor

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

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

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

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

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

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

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

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

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

Classes

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

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

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

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

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

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

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

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

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

Access to Methods

It's convenient to think of methods as being part of an object, just as instance variables are. As in Figure 2 above, methods can be diagrammed as surrounding the object's instance variables.

But, of course, methods aren't grouped with instance variables in memory. Memory is allocated for the instance variables of each new object, but there's no need to allocate memory for methods. All an instance needs is *access* to its methods, and all instances of the same class share access to the same set of methods. There's only one copy of the methods in memory, no matter how many instances of the class are created.

Modularity

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

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

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

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

The mechanisms that make class definitions logical units of the language are discussed in some detail under "Mechanisms of Abstraction" below.

Reusability

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

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

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

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

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

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

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

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

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

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

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

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

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

Mechanisms of Abstraction

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

- Encapsulation, and
- Polymorphism.

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

Encapsulation

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

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

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

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

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

Polymorphism

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

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

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

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

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

For example, instead of defining an **amountConsumed** method for an Appliance object to report the amount of water it uses over a given period of time, an **amountDispensedAtFaucet** method for a Faucet to report virtually the same thing, and a **cumulativeUsage** method for the Building object to report the cumulative total for the whole building—requiring programmers to learn three different names for what is conceptually the same operation—each class can simply have a **waterUsed** method.

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

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

Overloading

The terms "polymorphism" and "argument overloading" refer basically to the same thing, but from slightly different points of view. Polymorphism takes a pluralistic point of view and notes that several classes can each have a method with the same name. Argument overloading takes the point of the view of the method name and notes that it can have different effects depending on what kind of object it applies to.

Operator overloading is similar. It refers to the ability to turn operators of the language (such as '==' and '+' in C) into methods that can be assigned particular meanings for particular kinds of objects. Objective C implements polymorphism of method names, but not operator overloading.

Inheritance

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

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

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

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

Class Hierarchies

Any class can be used as a superclass for a new class definition. A class can simultaneously be a subclass of another class and a superclass for its own subclasses. Any number of classes can thus be linked in a hierarchy of inheritance.

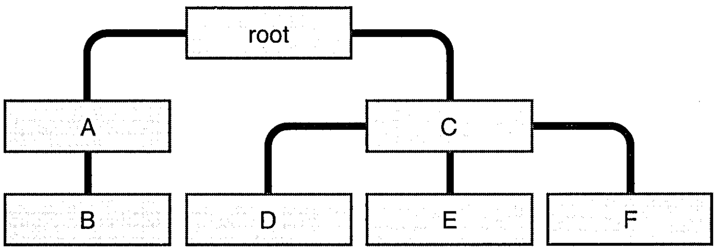


Figure 4. Inheritance Hierarchy

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

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

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

Subclass Definitions

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

- It can expand the class definition it inherits by adding new methods and instance variables. This is the most common reason for defining a subclass. Subclasses always add new methods, and new instance variables if the methods require it.

- It can modify the behavior it inherits by replacing an existing method with a new version. This is done by simply implementing a new method with the same name as one that's inherited. The new version *overrides* the inherited version. (The inherited method doesn't disappear; it's still valid for the class that defined it and other classes that inherit it.)

- It can refine or extend the behavior it inherits by replacing an existing method with a new version, but still retain the old version by incorporating it in the new method. This is done by sending a message to perform the old version in the body of the new method. Each class in an inheritance chain can contribute part of a method's behavior. In Figure 4, for example, class D might override a method defined in class C and incorporate C's version, while C's version incorporates a version defined in the root class.

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

Uses of Inheritance

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

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

Here are some typical uses of inheritance:

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

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

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

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

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

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

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

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

Dynamism

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

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

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

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

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

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

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

Dynamic Typing

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

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

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

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

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

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

Dynamic Binding

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

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

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

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

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

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

and call the function through the pointer:

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

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

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

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

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

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

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

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

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

Late Binding

Some object-oriented programming languages (notably C++) require a message receiver to be statically typed in source code, but don't require the type to be exact. An object can be typed to its own class or to any class that it inherits from.

The compiler therefore can't tell whether the message receiver is an instance of the class specified in the type declaration, an instance of a subclass, or an instance of some more distantly derived class. Since it doesn't know the exact class of the receiver, it can't know which version of the method named in the message to invoke.

In this circumstance, the choice is between treating the receiver as if it were an instance of the specified class and simply bind the method defined for that class to the message, or waiting until run time to resolve the situation. In C++, the decision is postponed to run time for methods (member functions) that are declared **virtual**.

This is sometimes referred to as "late binding" rather than "dynamic binding." While "dynamic" in the sense that it happens at run time, it carries with it strict compile-time type constraints. As discussed here (and implemented in Objective C), "dynamic binding" is unconstrained.

Dynamic Loading

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

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

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

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

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

For example, in the NeXTSTEP environment, Interface Builder dynamically loads custom palettes and inspectors, and the Workspace Manager™ dynamically loads inspectors for particular file formats. Anyone can design their own custom palettes and inspectors that these applications will load and incorporate into themselves.

Loading and Linking

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

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

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

Structuring Programs

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

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

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

Outlet Connections

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

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

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

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

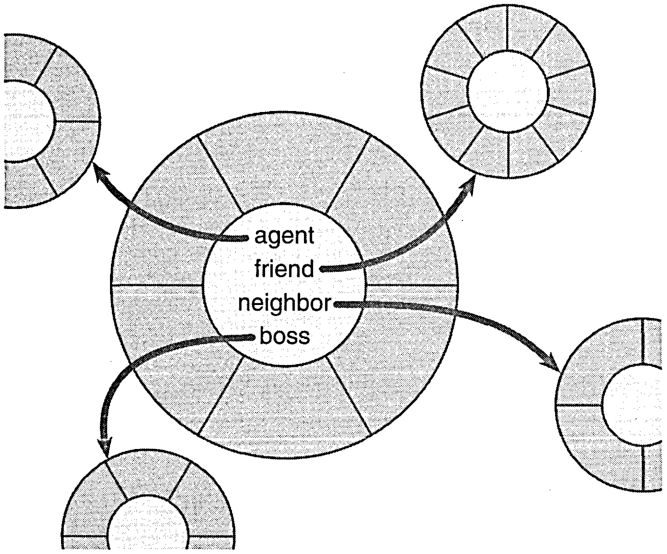


Figure 5. Outlets

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

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

Extrinsic and Intrinsic Connections

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

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

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

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

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

Activating the Object Network

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

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

Aggregation and Decomposition

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

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

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

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

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

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

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

Models and Kits

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

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

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

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

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

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

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

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

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

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

Chapter 5, "Programming in Objective C," has more on the NeXTSTEP software kits and how to work with them.

Structuring the Programming Task

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

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

Collaboration

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

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

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

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

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

Organizing Object-Oriented Projects

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

Designing on a Large Scale

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

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

Separating the Interface from the Implementation

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

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

Modularizing the Work

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

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

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

Keeping the Interface Simple

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

Making Decisions Dynamically

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

Inheriting Generic Code

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

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

Reusing Tested Code

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

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

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

2 The Objective C Language

This chapter describes the Objective C language as it's implemented for the NeXTSTEP development environment and discusses the principles of object-oriented programming as they're implemented in Objective C. It covers all the basic features that the language adds to standard C. The next chapter continues the discussion by taking up more advanced and less commonly used language features.

Objective C syntax is a superset of standard C syntax, and its compiler works for both C and Objective C source code. The compiler recognizes Objective C source files by a ".m" extension, just as it recognizes files containing only standard C syntax by a ".c" extension. As implemented for NeXTSTEP, the Objective C language is fully compatible with ANSI standard C.

Objective C can also be used as an extension to C++. At first glance, this may seem superfluous since C++ is itself an object-oriented extension of C. But C++ was designed primarily as "a better C," and not necessarily as a full-featured object-oriented language. It lacks some of the possibilities for object-oriented design that dynamic typing and dynamic binding bring to Objective C. At the same time, it has useful language features not found in Objective C. When you use the two languages in combination, you can assign appropriate roles to the features found in each and take advantage of what's best in both. Chapter 5, "Programming in Objective C," has more on combining C++ with Objective C.

Because object-oriented programs postpone many decisions from compile time to run time, object-oriented languages depend on a run-time system for executing the compiled code. The run-time system for the Objective C language is discussed in Chapter 4. This chapter and the next present the language, but touch on important elements of the run-time system as they're important for understanding language features. NeXT has modified the GNU C compiler to also compile Objective C and provides its own run-time system.

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, through the NeXTSTEP Application Kit, you can produce an object that displays a matrix of cells to users of your application. The cells might be text fields where the user can enter data, a series of mutually exclusive switches, a list of buttons or menu commands, or a bank of sliders. The figure below illustrates some of the different kinds of cells a matrix can contain:

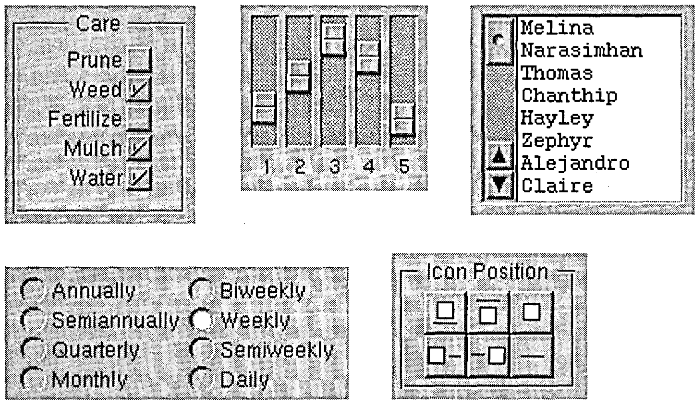


Figure 6. Some Matrices

A Matrix object has instance variables that define the matrix, including its dimensions and coordinates, the font used to display character strings in the cells, the arrangement of cells into rows and columns, and what to do when a cell is selected. A Matrix also has methods that do such things as alter its size, change its position on-screen, add and remove cells, highlight a particular cell, and set the color that's displayed between cells.

Each cell in a Matrix is also an object. Cells have instance variables that record their contents and what action to take when the cell is selected. They have methods to determine what the cell looks like and to track the cursor as it moves from cell to cell.

In Objective C, an object's instance variables are internal to the object; you get access to an object's state only through the object's methods. For others to find out something about an object, there has to be a method to supply the information. For example, a Matrix has methods that reveal its size, the currently selected cell, and the current number of columns and rows.

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, objects are identified by a distinct data type, **id**. This type is defined as a pointer to an object—in reality, a pointer to the object's data (its instance variables). 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.h**, which is located in the **objc** subdirectory of **/NextDeveloper/Headers**.

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 Matrix won't have the same methods or instance variables as an object that represents one of its cells. Cells that display buttons (ButtonCells) won't be exactly like those that display text (TextFieldCells). 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 run time.

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 Matrix object would be able to tell the run-time system that it is a Matrix. Every ButtonCell can say that it is a ButtonCell. 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 run time. Whenever it needs to, the run-time 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 introspect about themselves as objects. The compiler doesn't discard much of the information it finds in source code; it arranges most of it in data structures for the run-time system to use. Through **isa**, objects can find this information and reveal it at run time. An object can, for example, say whether it has a particular method in its repertoire and what the name of its superclass is.

Object classes are discussed in more detail under "Classes" below.

Note: 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" later in this chapter and "Static Options" in Chapter 3.

Messages

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 square 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 run-time system selects the appropriate method from the receiver's repertoire and invokes it.

For example, this message tells the **myMatrix** object to perform its **display** method, which draws the matrix and its cells in a window:

```
[myMatrix display];
```

Methods can also take arguments. The message below tells **myMatrix** to change its location within the window to coordinates (30.0, 50.0):

```
[myMatrix moveTo:30.0 :50.0];
```

Here the method name, **moveTo::**, has two colons, one for each of its arguments. The arguments are inserted after the colons, breaking the name apart. Colons don't have to be grouped at the end of a method name, as they are here. Usually a keyword describing the argument precedes each colon. The **getRow:andColumn:ofCell:** method, for example, takes three arguments:

```
int  row, column;
[myMatrix getRow:&row andColumn:&column ofCell:someCell];
```

This method finds **someCell** in the matrix and puts the row and column where it's located in the two variables provided.

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 assigns the identifying integer returned by the **tag** method to a variable also named **tag**.

```
int  tag;
tag = [myMatrix tag];
```

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

One message can be nested inside another. Here the **selectedCell** method returns an object that then receives a **tag** message:

```
int tag = [[myMatrix selectedCell] tag];
```

A message to **nil** also is valid,

```
[nil moveTo:100.0 :22.5];
```

but it has no effect and makes little sense. Messages to **nil** simply return **nil**.

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 **tag** method illustrated above takes no arguments, yet it can find the tag for **myMatrix** 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 **selectedCell** and **tag** methods shown above are used for just this purpose.

See "Defining a Class" 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 has access only to the 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 ButtonCell and a TextFieldCell 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 your 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 will be invoked to respond to a message can only be determined at run time, 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 run time 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 run time. When a message is sent, a run-time 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" below.)

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.

In 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 editable text would react to a **copy:** message differently than an object that displays scanned images. A Matrix would respond differently than a Cell. Since messages don't select methods (methods aren't bound to messages) until run time, 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 run time. This is discussed in the section on "How Messaging Works."

Classes

An object-oriented program is typically built from a variety of objects. A program based on the NeXTSTEP software kits might use Matrix objects, Window objects, List objects, SoundView objects, Text objects, and many others. Programs often use more than one object of the same kind or *class*—several Lists or Windows, 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 sometimes also 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 will do the main work of your program are instances created by the class object at run time.

All instances of a class have access to 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 "Matrix"); the names of instances typically begin with a lowercase letter (such as "myMatrix").

Inheritance

Class definitions are additive; each new class that you define is based on another class through 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, the Object class, at its root. Every class (but Object) has a *superclass* one step nearer the root, and any class (including Object) can be the superclass for any number of *subclasses* one step farther from the root. Figure 7 below illustrates the hierarchy for a few of the classes in the NeXTSTEP Application Kit.

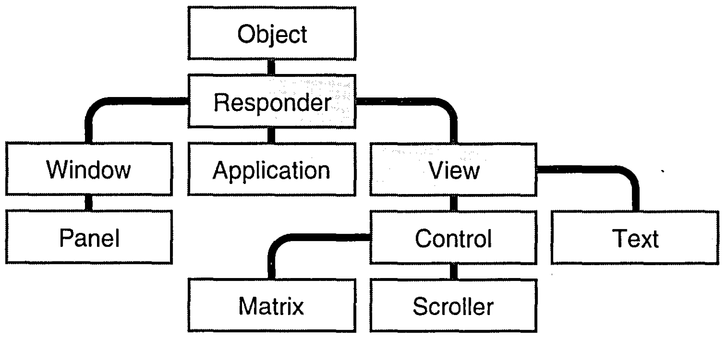


Figure 7. Some Application Kit Classes

This figure shows that the Matrix class is a subclass of the Control class, the Control class is a subclass of View, View is a subclass of Responder, and Responder is a subclass of Object. Inheritance is cumulative. So a Matrix object has the methods and instance variables defined for Control, View, Responder, and Object, as well as those defined specifically for Matrix. This is simply to say that a Matrix object isn't only a Matrix, it's also a Control, a View, a Responder, and an Object.

Every class (but Object) 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 Matrix class defines only the minimum needed to turn a Control into a Matrix.

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. The NeXTSTEP development environment includes the Object class and several software kits containing definitions for more than 125 different 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 kit 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 at NeXT.

The Object Class

Object is the only class without a superclass, and the only one that's in the inheritance path for every other class. That's because it defines the basic framework for Objective C objects and object interactions. It imparts to the classes and instances that inherit from it the ability to behave as objects and cooperate with the run-time system.

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

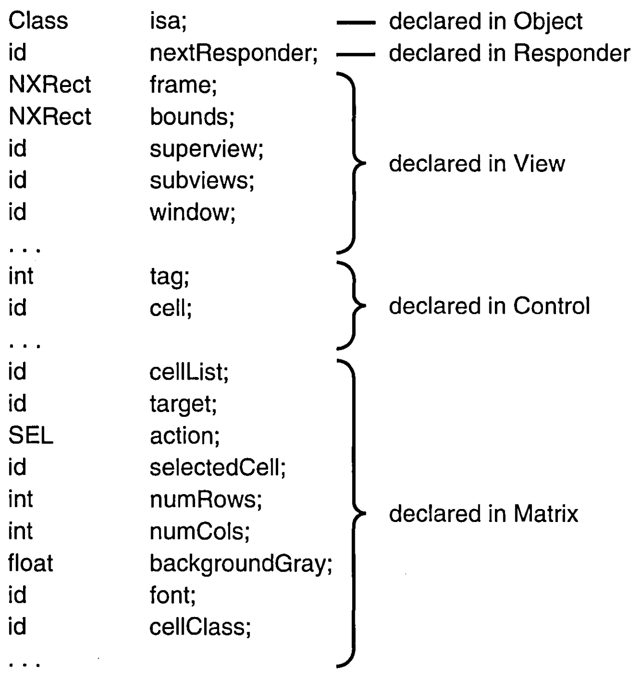
Appendix C, "The Object Class," has a full specification of the root class and describes its methods in detail.

Note: Implementing a new root class is a delicate task and one with many hidden hazards. The class must duplicate much of what the Object class does, such as allocate instances, connect them to their class, and identify them to the run-time system. It's strongly recommended that you use the Object class provided with NeXTSTEP as the root class. This manual doesn't explain all the ins and outs that you would need to know to replace it.

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 Object class. The **isa** instance variable defined in the Object class becomes part of every object. **isa** connects each object to its class.

Figure 8 below shows some of a Matrix object's instance variables and where they come from. Note that the variables that make the object a Matrix are added to the ones that make it a Control, and the ones that make it a Control are added to the ones that make it a View, and so on.



```
Class     isa;             — declared in Object
id        nextResponder;   — declared in Responder
NXRect    frame;           ⎫
NXRect    bounds;          ⎪
id        superview;       ⎬ declared in View
id        subviews;        ⎪
id        window;          ⎪
. . .                      ⎭
int       tag;             ⎫
id        cell;            ⎬ declared in Control
. . .                      ⎭
id        cellList;        ⎫
id        target;          ⎪
SEL       action;          ⎪
id        selectedCell;    ⎪
int       numRows;         ⎬ declared in Matrix
int       numCols;         ⎪
float     backgroundGray;  ⎪
id        font;            ⎪
id        cellClass;       ⎪
. . .                      ⎭
```

Figure 8. Matrix 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.

Inheriting Methods

An object has access not only to the methods that were 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. A Matrix object can use methods defined in the Control, View, Responder, and Object 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 kits provided by NeXTSTEP, your programs can take advantage of all the basic functionality coded into the kit classes. You have to add only the code that customizes the kit 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 will perform it rather than the original, and subclasses of the new class will inherit it rather than the original.

For example, the View class defines a **display** method that Matrix overrides by defining its own version of **display**. The View method is available to all kinds of objects that inherit from the View class—but not to Matrix objects, which instead perform the Matrix 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**," below, to learn how).

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

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

Abstract Classes

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

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

In the NeXTSTEP software kits, 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 kit objects.

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

Class Types

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

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

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

Static Typing

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

```
Matrix *myMatrix;
```

Since this way of declaring an object type gives the compiler information about what 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 receives 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 run time.

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

```
View *myMatrix;
```

This is possible because a Matrix is a View. It's more than a View since it also has the instance variables and method capabilities of a Control and a Matrix, but it's a View nonetheless. For purposes of type checking, the compiler will consider **myMatrix** to be a View, but at run time it will be treated as a Matrix.

See "Static Options" in the next chapter for more on static typing and its benefits.

Type Introspection

Instances can reveal their types at run time. The **isMemberOf:** method, defined in the Object class, checks whether the receiver is an instance of a particular class:

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

The **isKindOf:** method, also defined in the Object 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 isKindOf:someClass] )
    . . .
```

The set of classes for which **isKindOf:** 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 Appendix C, "The Object Class," for more on **isKindOf:**, **isMemberOf:**, and kindred 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 declaration 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 run-time 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 Matrix class returns the class version number using a method inherited from the Object class:

```
int versionNumber = [Matrix 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 matrixClass = [Matrix 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 matrixClass = [Matrix 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 run time.

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 run-time system.

Creating Instances

A principal function of a class object is to create new instances. This code tells the Matrix class to create a new Matrix instance and assign it to the **myMatrix** variable:

```
id  myMatrix;
myMatrix = [Matrix 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:

```
myMatrix = [[Matrix alloc] init];
```

This line of code, or one like it, would be necessary before **myMatrix** 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 (**initFrame:mode:cellClass:numRows:numColumns:**, for example, is the method that would most often initialize a new Matrix 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, a Matrix object can be customized with a particular kind of Cell.

A Matrix 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 a Matrix object draws on-screen can grow and shrink at run time, 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 Cell, but there are many different kinds. The inheritance hierarchy in Figure 9 below shows some of those provided by the Application Kit. All inherit from the generic Cell class:

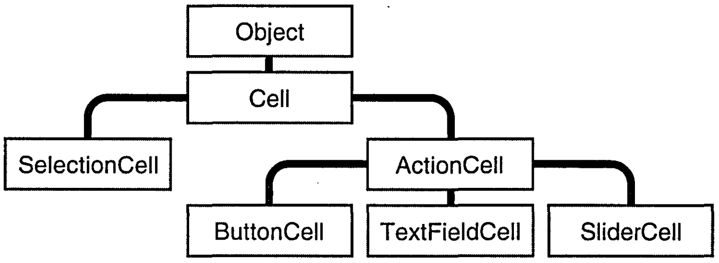


Figure 9. Inheritance Hierarchy for Cells

When a Matrix creates new Cell objects, should they be ButtonCells to display a bank of buttons or switches, TextFieldCells to display a field where the user can enter and edit text, or some other kind of Cell? The Matrix must allow for any kind of Cell, even types that haven't been invented yet.

One solution to this problem would be to define the Matrix 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 Matrix class, and it unnecessarily proliferates the number of classes. Since an application might need more than one kind of Matrix, each with a different kind of Cell, it could become cluttered with Matrix subclasses. Every time you invented a new kind of Cell, you'd also have to define a new kind of Matrix. Moreover, programmers on different projects would be writing virtually identical code to do the same job, all to make up for Matrix's failure to do it.

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

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

The Matrix 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 impossible 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 of objects, you can decide what instance variables they should have. Every instance of the class will have its own copy of all the variables you declare; each object controls its own data.

However, you can't prescribe variables for the class object; there are no "class variable" counterparts to instance variables. Only internal data structures, initialized from the class definition, are provided for the class. The class object also has no access to the instance variables of any instances; it can't initialize, read, or alter them.

Therefore, for all the instances of a class to share data, an external variable of some sort is required. Some classes declare static variables and provide class methods to manage them. (Declaring a variable **static** in the same file as the class definition limits its scope to just the class—and to just the part of the class that's implemented in the file. Unlike instance variables, static variables can't be inherited by subclasses, unless the subclasses are defined in the same file.)

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 limiting 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 would also be possible to use external variables that weren't declared **static**, but the limited scope of static variables better serves the purpose of encapsulating data into separate objects.

Initializing a Class Object

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

The run-time system sends an **initialize** message to every class object before the class receives any other messages. This gives the class a chance to set up its run-time environment before it's used. If no initialization is required, you don't need to write an **initialize** method to respond to the message; the Object class defines an empty version that your class can inherit and perform.

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.

Methods of the Root Class

All objects, classes and instances alike, need an interface to the run-time 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 Object class to provide this interface.

So that Object's methods won't all have to be implemented twice—once to provide a run-time 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 run-time system will see if 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 "Class Description" section in Appendix C, "The Object Class."

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:

  ```
  Matrix *anObject;
  anObject = [[Matrix alloc] init];
  ```

  Here **anObject** is statically typed to be a Matrix. The compiler will expect it to have the data structure of a Matrix instance and the instance methods defined and inherited by the Matrix class. Static typing enables the compiler to do better type checking and makes source code more self-documenting. See "Static Options" in the next chapter 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 examples above. 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 Matrix class as an argument in an **isKindOf:** message.

  ```
  if ( [anObject isKindOf:[Matrix class]] )
      . . .
  ```

  It would have been illegal to simply use the name "Matrix" 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 run time, **objc_lookUpClass()** will return the class object:

  ```
  if ( [anObject isKindOf:objc_lookUpClass(aBuffer)] )
      . . .
  ```

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

Class names compete in the same name space as variables and functions. A class and a global variable can't have the same name. Class names are about the only names with global visibility in Objective C.

Defining a Class

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

- An *interface* that declares the methods and instance variables of the class and names its superclass

- An *implementation* that actually defines the class (contains the code that implements its methods)

Although the compiler doesn't require it, the interface and implementation are usually separated into two different files. The interface file must be made available to anyone who uses the class. You generally wouldn't want to distribute the implementation file that widely; users don't need source code for the implementation.

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 implementation file has a ".m" suffix, 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 interface file usually has the ".h" suffix typical of header files. For example, the Matrix class would be declared in **Matrix.h** and defined in **Matrix.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 will interact with other elements in your program—that is, once you've declared its interface—you can freely alter its implementation without affecting any other part of the application.

The Interface

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

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

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

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

```
id      selectedCell;
int     numRows;
int     numCols;
float   backgroundGray;
id      cellClass;
```

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:

```
- 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, Matrix has a **selectedCell** method to match its **selectedCell** instance variable.

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

```
- (int)tag;
```

Argument types are declared in the same way:

```
- setTag:(int)anInt;
```

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**, **display**, and **setTag:** methods illustrated above all return **id**s.

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

```
- moveTo:(NXCoord)x :(NXCoord)y;
- getRow:(int *)aRow andColumn:(int *)aColumn ofCell:aCell;
```

(NXCoord is a defined type for floating-point values that specify coordinate measurements.)

Methods that take a variable number of arguments declare them using a comma and an ellipsis, 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 "Matrix.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 NeXTSTEP 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.

Referring to Other Classes

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

```
@class Matrix, List;
```

This directive simply informs the compiler that "Matrix" and "List" 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

```
   - getCells:(List *)theCells;
```

mentions the List 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).

- 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.

- It also lets the compiler know what instance variables an object contains and programmers know what variables their subclasses will 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, the interface file also 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 Implementation

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

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

However, every implementation file must import its own interface. For example, **Matrix.m** imports **Matrix.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
{
    . . .
}

- (int)tag
{
    . . .
}

- moveTo:(NXCoord)x :(NXCoord)y
{
    . . .
}
```

Methods that take a variable number of arguments handle them just as a functions 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 a potential receiving 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 **tag** instance variable:

```
- setTag:(int)anInt
{
    tag = anInt;
    . . .
}
```

Neither the receiving object nor its **tag** 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.

The instance variables of the receiving object are not the only ones that you can refer to within the implementation of a class. You can refer to any instance variable of any object as long as two conditions are met:

- The instance variable must be within the scope of the class definition. Normally that means the instance variable must be one that the class declares or inherits. (Scope is discussed in more detail in the next section.)
- The compiler must know what kind of object the instance variable belongs to.

When the instance variable belongs to the receiver (as it does in the **setTag:** example above), this second condition is met automatically. The receiver's type is implicit but clear—it's the very type that the class defines.

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 : Object
{
    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:

```
- (int)tag
{
    return tag;
}
```

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

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

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

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

This is illustrated in Figure 10.

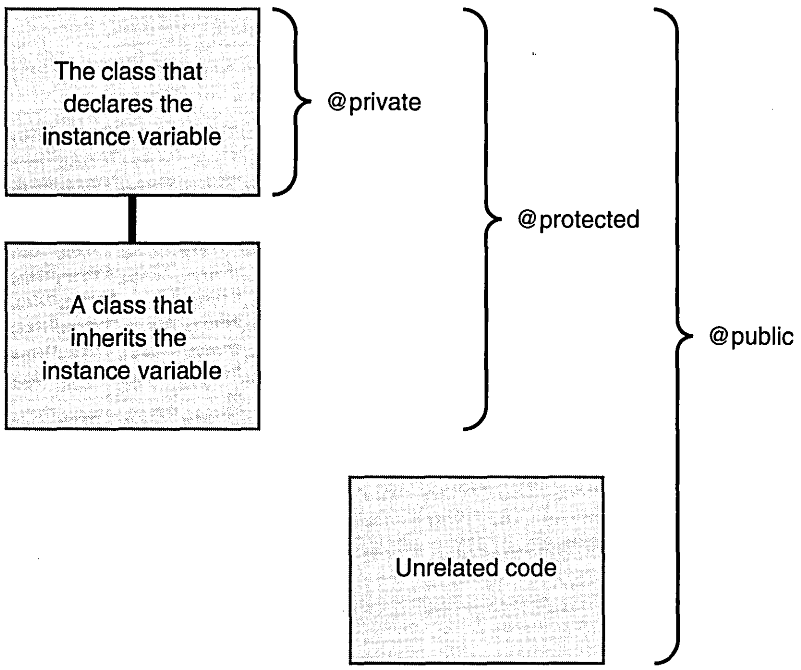

The class that declares the instance variable
@private
@protected
A class that inherits the instance variable
@public
Unrelated code

Figure 10. 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 : Object
{
    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 variable 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 above 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 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**.

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 modules 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.

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

Note that the object must be statically typed.

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

How Messaging Works

In Objective C, messages aren't bound to method implementations until run time. 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 arguments:

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 different 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 **moveTo::** method is associated with the address of (the procedure that implements) **moveTo::**, 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.

These elements of class and object structure are illustrated in Figure 11.

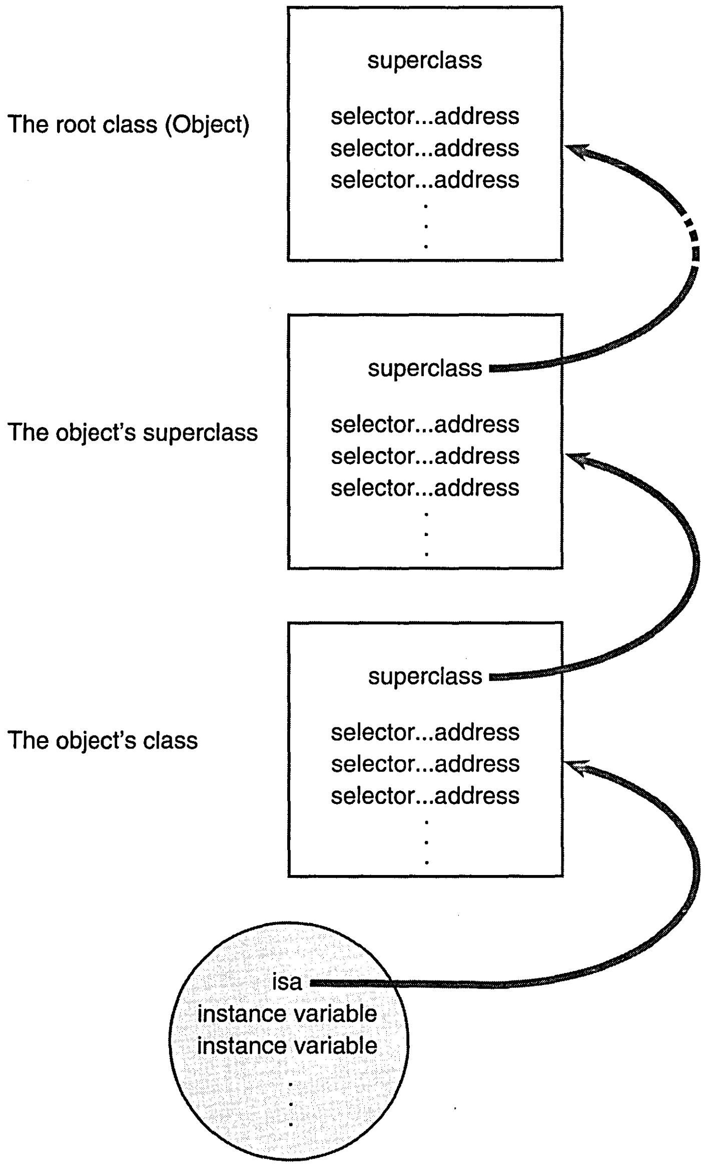


Figure 11. Messaging Framework

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 Object class. Once it locates the selector, it 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 run time—or, in the jargon of object-oriented programming, that methods are dynamically bound to messages.

To speed the messaging process, the run-time 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 will 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 will represent the method at run time. The run-time 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.

A compiled selector contains fields of coded information that aid run-time messaging. You should therefore let the system assign SEL identifiers to methods; it won't work to assign them arbitrarily yourself.

The **@selector()** directive lets Objective C source code refer to the compiled selector, rather than to the full method name. Here the selector for **moveTo::** is assigned to the **mover** variable:

```
SEL  mover;
mover = @selector(moveTo::);
```

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 run time. This can be done with the **sel_getUid()** function:

```
mover = sel_getUid(aBuffer);
```

Conversion in the opposite direction is also possible. The **sel_getName()** function returns a method name for a selector:

```
char *method;
method = sel_getName(mover);
```

These and other run-time functions are described in the *NeXTSTEP General Reference* manual.

Methods and Selectors

Compiled selectors identify method names, not method implementations. Matrix's **display** method, for example, will have 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 different 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 Run Time

The **perform:**, **perform:with:**, and **perform:with:with:** methods, defined in the Object class, take SEL identifiers as their initial arguments. All three methods map directly into the messaging function. For example,

```
[friend perform:@selector(gossipAbout:) with:aNeighbor];
```

is equivalent to:

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

These methods make it possible to vary a message at run time, 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 perform:request];
```

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

Note: perform: 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 won't work for all types; the method should return a pointer or a type compatible with a pointer.)

The Target-Action Paradigm

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

Controls 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 framework for creating control devices and defines a few "off-the-shelf" devices of its own. For example, the ButtonCell class defines an object that you can assign to a Matrix 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 ButtonCell sends a message instructing the application to do something. To do this, a ButtonCell 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, a ButtonCell 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 ButtonCell sends the message using Object's **perform:with:** 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 ButtonCells would have to send the same message; the name of the method would be frozen in the ButtonCell source code. Instead of simply implementing a mechanism for translating user actions into action messages, ButtonCells 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 ButtonCell. 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'd also have to reimplement 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 run time, the error often won't be 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're programming, you can check to be sure that the receiver is able to respond. If the receiver is statically typed, the compiler will check for you.

However, if the message selector or the class of the receiver varies, it may be necessary to postpone this check until run time. The **respondsTo:** method, defined in the Object class, determines whether a potential receiver can respond to a potential message. It takes the method selector as an argument, and returns whether the receiver has access to a method matching the selector:

```
if ( [anObject respondsTo:@selector(moveTo::)] )
    [anObject moveTo:0.0 :0.0];
else
    fprintf(stderr, "%s can't be moved\n", [anObject name]);
```

The **respondsTo:** 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 check to be sure the receiver implements a method that can respond to the message.

Note: An object can also arrange to have messages it receives forwarded to other objects, if it can't respond to them directly itself. In that case, it will appear that the object can't handle the message, even though it responds to it indirectly by assigning it to another object. Forwarding is discussed in Chapter 4, "The Run-Time System."

Hidden Arguments

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

- The receiving object.
- The selector for the method.

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

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

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

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

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.

Methods that have no other meaningful return value typically return **self**, rather than **void**. This enables messages to be nested in source code. For example:

```
[[[myMatrix setMode:NX_RADIOMODE] setEnabled:YES] setTag:99];
```

self is discussed in more detail in the next section.

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 **moveTo::** method to make the change. All it needs to do is send a **moveTo::** message to the very 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 moveTo:someX :someY];
    . . .
}
```

or:

```
- reposition
{
    . . .
    [super moveTo: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 **objc_msgSend()**. 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 12 below:

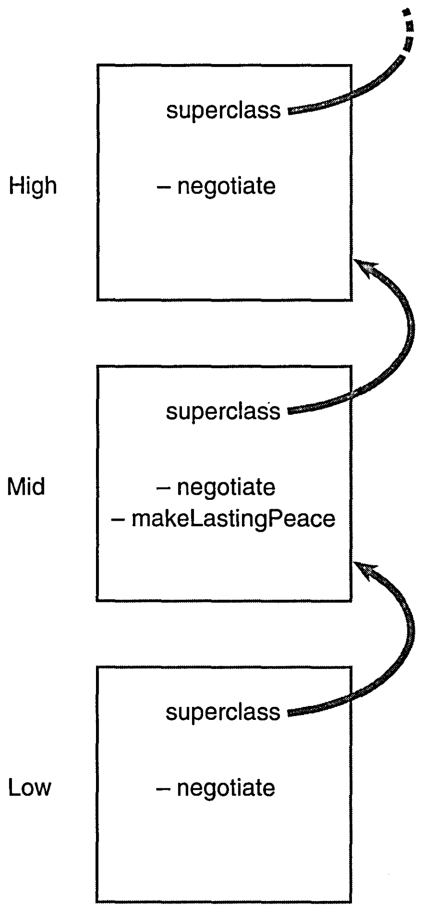


Figure 12. High, Mid, and Low

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 will find the version of **negotiate** defined in Low, **self**'s class. However, if 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 pass the message on to **super** for the rest. The **init** method, which initializes a newly allocated instance, and the **write:** method, which archives an object by writing it to a data stream, are designed to work like this. Each **write:** method has responsibility for writing the instance variables defined in its class. But before doing so, it sends a **write:** message to **super** to have the classes it inherits from archive their instance variables. Each version of **write:** follows this same procedure, so classes write their instance variables in the order of inheritance:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    . . .
    return self;
}
```

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 a new 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 **allocFromZone:** methods defined in the Object class. If another class overrides these methods for any reason (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, a method might combine allocation and initialization of an instance:

```
+ newTag:(int)anInt
{
    return [[self alloc] initTag:anInt];
}
```

In such a method, it's tempting to send messages to the 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.

Before a class method can send a message telling **self** to perform an instance method, it must redefine **self** to be the instance:

```
+ newTag:(int)anInt andColor:(NXColor)aColor
{
    self = [[self alloc] initTag:anInt];
    [self setColor:aColor];
    return self;
}
```

The method shown above is a class method, so, initially, **self** refers to the class object. It's as the class object that **self** receives the **alloc** message. **self** is then redefined to be the instance that **alloc** returns and **initTag:** initializes. It's as the new instance that it receives the **setColor:** message.

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

```
+ newTag:(int)anInt andColor:(NXColor)aColor
{
    id newInstance = [[self alloc] initTag:anInt];
    [newInstance setColor:aColor];
    return newInstance;
}
```

Note: In these examples, the class method sends messages (**initTag:** and **setColor:**) to initialize the instance. It doesn't assign a new value directly to an instance variable as an instance method might have done:

```
tag = anInt;
color = NX_REDCOLOR;
```

Only instance variables of the receiver can be directly set this way. Since the receiver for a class method (the class object) has no instance variables, this syntax can't be used. However, if **newInstance** had been statically typed, something similar would have been possible:

```
newInstance->tag = anInt;
```

See "Referring to Instance Variables," earlier in this chapter, for more on when this syntax is permitted.

3 *Objective C Extensions*

The preceding chapter has all you need to know about Objective C to define classes and design programs in the language. It covers basic Objective C syntax and explains the messaging process in detail.

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

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

The chapter also explains how static typing works and takes up some lesser used features of Objective C, including ways to temporarily overcome its inherent dynamism.

Categories

You can add methods to a class by declaring them in an interface file under a category name and defining them in an implementation file under the same name. The category name indicates that the methods are additions to a class declared elsewhere, not a new 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 Matrix and other NeXTSTEP classes. As in the case of a subclass, you don't need source code for the class you're extending.

The methods the category adds become part of the class type. For example, methods added to the Matrix class in a category will be among the methods the compiler will expect a Matrix instance to have in its repertoire. Methods added to the Matrix class in a subclass would not be included in the Matrix 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 run time, there's no difference. The methods the category adds to the class are inherited by all the class's subclasses, just like other methods.

Adding to a Class

The declaration of a category interface looks very much like a class interface declaration—except the category name is listed within parentheses after the class name and the superclass isn't mentioned. The category must import the interface file for the class it extends:

> **#import** "*ClassName*.**h**"
>
> **@interface** *ClassName* (*CategoryName*)
> *method declarations*
> **@end**

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

> **#import** "*CategoryName*.**h**"
>
> **@implementation** *ClassName* (*CategoryName*)
> *method definitions*
> **@end**

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

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

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

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

How Categories are Used

Categories can be used to extend classes defined by other implementors—for example, you can add methods to the classes defined in the NeXTSTEP software kits. The added methods will be inherited by subclasses and will be indistinguishable at run time from the original methods of the class.

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

- They provide a simple way of grouping related methods. Similar methods defined in different classes can be kept together in the same source file.
- They simplify the management of a large class when more than one developer is contributing 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 different applications, without having to maintain different versions of the same source code.

Categories are also used to declare informal protocols, as discussed under "Protocols" below.

Categories of the Root Class

A category can add methods to any class, including the root Object class. Methods added to Object 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 run-time 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 Object 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 Appendix C, "The Object Class," for more information on class access to root instance methods.

Protocols

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

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

```
- mouseDown:(NXEvent *)theEvent;
- mouseDragged:(NXEvent *)theEvent;
- mouseUp:(NXEvent *)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. NeXTSTEP software uses them heavily to support interprocess communication through Objective C messages.

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

How Protocols are Used

Protocols are useful in at least three different situations:

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

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

Methods for Others to Implement

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

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

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

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

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

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

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.

Anonymous Objects

A protocol can also 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 can't be anonymous to their developers, of course, but they can be anonymous when the developer supplies them to someone else. For example, an anonymous object might be part of a software kit or be located in a remote process:

- Someone who supplies a software kit 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.

- It's possible to send Objective C messages to *remote objects*—objects in other applications. (The next section, "Remote Messaging," 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 won't reveal its class, the object itself will reveal it at run time. A **class** message will return the anonymous object's class. The class object can then be queried with the **name** and **superclass** methods. However, there's usually little point in discovering this extra information; the information in the protocol is sufficient.

Nonhierarchical 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 reimplement the methods in its own way, but the inheritance hierarchy and the common declaration in the abstract class captures the essential similarity between the subclasses.

However, sometimes it's not possible to group common methods in an abstract class. Classes that are unrelated in most respects might nevertheless need to implement some similar methods. This limited similarity may not justify a hierarchical relationship. For example, many different kinds of classes might implement methods to facilitate reference counting:

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

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

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

Informal Protocols

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

```
@interface Object ( RefCounting )
- setRefCount:(int)count;
- (int)refCount;
- incrementCount;
- decrementCount;
@end
```

Informal protocols are typically declared as categories of the Object class, since that broadly associates the method names with any class that inherits from Object. Since 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 run time to see whether an object conforms to the protocol. To get these benefits, you must use a formal protocol.

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 run-time system. For example, the compiler can check for types based on protocols, and objects can introspect at run time to report whether or not they conform to a protocol.

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

@protocol *ProtocolName*
method declarations
@end

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

```
@protocol ReferenceCounting
- setRefCount:(int)count;
- (int)refCount;
- incrementCount;
- decrementCount;
@end
```

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

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

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

Categories adopt protocols in much the same way:

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

Names in the protocol list are separated by commas.

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

It's possible for a class to simply adopt protocols and declare no other methods. For example, this class declaration,

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

adopts the Formatting and Prettifying protocols, but declares no instance variables or methods of its own.

A class or category that adopts a protocol is obligated to implement all the methods the protocol declares. The compiler will issue a warning if it does not. The Formatter class above would define all the methods declared in the two protocols it adopts, in addition to any it might have declared itself.

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

Protocol Objects

Just as classes are represented at run time 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 run time 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 run-time system. They're not allocated and initialized in program source code.

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

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

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

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

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

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

Conforming to a Protocol

A class is said to *conform* to a formal protocol if it adopts the protocol or inherits from a 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 methods declared in the protocols it adopts, and those methods are inherited by its subclasses, saying that a class or an instance conforms to a protocol is tantamount to saying that it has in its repertoire all the methods that the protocol declares.

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

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

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

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

Type Checking

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

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

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

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

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

```
Formatter *anObject;
```

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

Similarly, this declaration,

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

groups all objects that conform to the Formatting protocol into a type, regardless of their positions in the class hierarchy. The compiler can check to be sure that 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 run time, both classes and instances will respond to a **conformsTo:** message.)

Protocols within Protocols

One protocol can incorporate others 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 will also conform to Formatting. Type declarations

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

and **conformsTo:** messages

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

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

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

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

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

```
@interface Pager : Object < 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.

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 run-time, 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, thc clicnt 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 put up an attention panel 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 user interface on-screen and of sorting out which user instructions were the responsibility of which editing tool. 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 run-time system that can establish connections between objects in different address spaces, recognize when a message is intended for a remote address, 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.

NeXTSTEP includes a *distributed objects* architecture that is essentially this kind of extension to the run-time 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.)

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 diagrammed in Figure 13 below, 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:

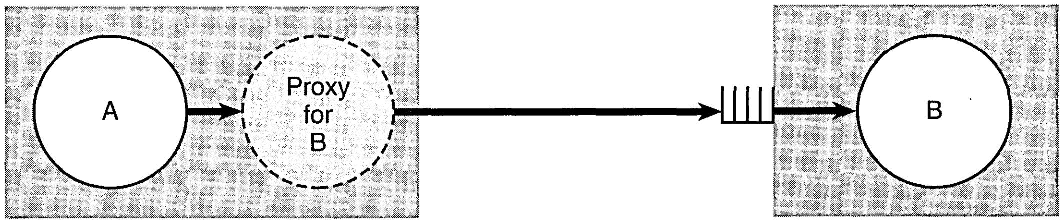


Figure 13. Remote Messages

The sender and receiver are in different tasks and are scheduled independently of each other. So there's no guarantee that the receiver will be 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 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 typically 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 **conformsTo:** 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 NXProxy and NXConnection classes, is documented in the *NeXTSTEP General Reference* manual.

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 five type qualifiers that can be used when declaring methods inside a formal protocol:

oneway
in
out
inout
bycopy

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 five 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:

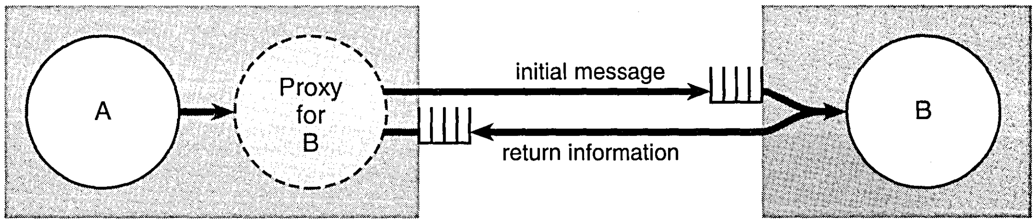


Figure 14. Round-Trip Message

Most remote messages will be, 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 will. 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 run-time system for remote messaging must make some adjustments behind the scenes.

If the argument is used to pass information by reference, the run-time 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 one case, information is passed on the first leg of the round trip. In the other 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 run-time 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 NeXTSTEP distributed objects system takes **inout** to be the default modifier for all pointer arguments except those declared **const**, for which **in** is the default. **inout** is the safest assumption, but also the most time-consuming since it requires passing information in both directions. The only modifier that makes sense for arguments passed by value (nonpointers) is **in**. While **in** can be used with any kind of argument, **out** and **inout** make sense only for pointers.

In C, pointers are sometimes used to represent composite values. For example, a string is represented as a character pointer (**char** *). Although in notation and implementation there's a level of indirection here, in concept there's not. Conceptually, a string is an entity in and of itself, not a pointer to something else.

In cases like this, the distributed objects system automatically dereferences the pointer and passes whatever it points to as if by value. Therefore, the **out** and **inout** modifiers make no sense with simple character pointers. It takes an additional level of indirection in a remote message to pass or return a string by reference:

```
- getTuneTitle:(out char **)theTitle;
```

The same is true of objects:

```
- adjustMatrix:(inout Matrix **)theMatrix;
```

These conventions are enforced at run time, not by the compiler.

Proxies and Copies

Finally, consider a method that takes an object as an argument:

```
- danceWith:aPartner;
```

A **danceWith:** message passes an object **id** to the receiver. If the sender and 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 will need 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;
```

The only type that it makes sense for **bycopy** to modify is an object, whether dynamically typed **id** or statically typed by a class name.

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.

Static Options

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

- The memory for objects is *dynamically allocated* at run time by class methods that create new instances.
- Objects are *dynamically typed.* In source code (at compile time), any object can be of type **id** no matter what its class. The exact class of an **id** variable (and therefore its particular methods and data structure) isn't determined until the program is running.
- Messages and methods are *dynamically bound*, as described under "How Messaging Works" in the previous chapter. A run-time 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 run-time 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 run time back to compile time.

Static Typing

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

```
Matrix  *thisObject;
```

the compiler restricts 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 Matrix 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 run time. Statically typed objects are dynamically allocated by the same class methods that create instances of type **id**. If Mosaic is a subclass of Matrix, the following code would still produce an object with all the instance variables of a Mosaic, not just those of a Matrix:

```
Matrix  *thisObject = [[Mosaic 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 run time as part of the messaging process. A **display** message sent to **thisObject**

```
[thisObject display];
```

will perform the version of the method defined in the Mosaic class, not its Matrix 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 below. The third was covered in the previous chapter under "Defining a Class."

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 check to be sure that 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 can check to be sure that 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. This is illustrated in the example below.

```
View    *aView;
Matrix  *aMatrix;

aMatrix = [[Matrix alloc] init];
aView = aMatrix;
```

Here **aMatrix** can be assigned to **aView** because a Matrix is a kind of View—the Matrix class inherits from View. However, if the roles of the two variables are reversed and **aView** is assigned to **aMatrix**, the compiler will generate a warning; not every View is a Matrix. (For reference, Figure 7 in the previous chapter shows a portion of the class hierarchy including View and Matrix.)

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 check to be sure that a compatible object is returned to a statically typed variable. The following code is error-prone, but is allowed nonetheless:

```
Matrix  *aMatrix;
aMatrix = [[Window alloc] init];
```

Note: This is consistent with the implementation of **void** * (pointer to **void**) in ANSI C. Just as **void** * is a generic pointer that eliminates the need for coercion in assignments between pointers, **id** is a generic pointer to objects that eliminates the need for coercion to a particular class in assignments between objects.

Return and Argument Types

In general, methods that share the same selector (the same name) must also share the same return and argument types. This constraint is imposed by 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 run-time 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 Objects.

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 run time and what will actually happen.

For example, if you statically type a Matrix instance as a View,

```
View *myMatrix = [[Matrix alloc] init];
```

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

```
id cell = [myMatrix selectedCell];
```

the compiler will complain. The **selectedCell** method is defined in the Matrix class, not in View.

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

```
[myMatrix display];
```

the compiler won't complain, even though Matrix overrides the method. At run time, Matrix's version of the method will be 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 run-time system that a **worry** message sent to the object will return a **double**, but at run time it will actually return an **int** and generate 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 Object class, **methodFor:**, you can ask for a pointer to the procedure that implements a method, then use the pointer to call the procedure. The pointer that **methodFor:** 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 **setTag:** method might be called:

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

setter = (id (*)(id, SEL, int))[target methodFor:@selector(setTag:)];
for ( i = 0; i < 1000, i++ )
    setter(targetList[i], @selector(setTag:), i);
```

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

Note that **methodFor:** is provided by the run-time system; it's not a feature of the Objective C language itself.

Getting an Object Data Structure

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

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

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

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

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

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

This technique of turning an object into a structure makes all of its instance variables public, no matter whether they were declared **@private**, **@protected**, or **@public**.

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

Type Encoding

To assist the run-time 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 might also be of use in other contexts and so is made publicly available with the **@encode()** directive. When given a type specification, **@encode()** returns a string encoding that type. The type can be a basic type such as an **int**, a pointer, a tagged structure or union, or a class name—anything, in fact, that can be used as an argument to the C **sizeof()** operator.

```
char  *buf1 = @encode(int **);
char  *buf2 = @encode(struct key);
char  *buf3 = @encode(Matrix);
```

The table below lists the type codes. Note that many of them overlap with the codes used in writing to a typed stream. However, there are codes listed here that you can't use when writing to a typed stream and there are codes that you may want to use when writing to a typed stream that aren't generated by **@encode()**. (See the *NeXTSTEP General Reference* manual and the next chapter of this book for information on typed streams.)

Code	**Meaning**
c	A **char**
i	An **int**
s	A **short**
l	A **long**
C	An **unsigned char**
I	An **unsigned int**
S	An **unsigned short**
L	An **unsigned long**
f	A **float**
d	A **double**
v	A **void**
*	A character string (**char ***)
@	An object (whether statically typed or typed **id**)
#	A class object (Class)
:	A method selector (SEL)
[...]	An array
{...}	A structure
(...)	A union
b*num*	A bitfield of *num* bits
^*type*	A pointer to *type*
?	An unknown type

The type specification 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, this 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 Object class name to **@encode()** yields this encoding:

```
{Object=#}
```

The Object class declares just one instance variable, **isa**, of type Class.

Note: Although the **@encode()** directive doesn't return them, the run-time system also uses these additional encodings for type qualifiers when they're used to declare methods in a protocol:

Code	Meaning
r	**const**
n	**in**
N	**inout**
o	**out**
O	**bycopy**
V	**oneway**

4 The Run-Time System

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

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

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

  It's up to the compiler to produce the data structures that the run-time system requires and to arrange the run-time function calls that carry out language instructions. The data structures capture information found in class and category definitions and in protocol declarations; they include the class and protocol objects discussed earlier, as well as method selectors, instance variable templates, and other information distilled from source code. The principal run-time function is the one that sends messages, as described under "How Messaging Works" in Chapter 2. It's invoked by source-code message expressions.

- Through a method interface defined in the Object class. Every object inherits from the Object class, so every object has access to the methods it defines. Most Object methods interact with the run-time system.

  Some of these methods simply query the system for information. The preceding chapters, for example, mentioned the **class** method, which asks an object to identify its class, **isKindOf:** and **isMemberOf:**, which test an object's position in the inheritance hierarchy, **respondsTo:**, which checks whether an object can accept a particular message, **conformsTo:**, which checks whether it conforms to a protocol, and **methodFor:**, which asks for the address of a method implementation. Methods like these give an object the ability to introspect about itself.

  Other methods set the run-time system in motion. For example, **perform:** and its companions initiate messages, and **alloc** produces a new object properly connected to its class.

  All these methods were mentioned in previous chapters and are described in detail in Appendix C, "The Object Class."

- Through direct calls to run-time functions. The run-time system has a public interface, consisting mainly of a set of functions. Many are functions that duplicate what you get automatically by writing Objective C code or what the Object class provides with a method interface. Others manipulate low-level run-time processes and data structures. These functions make it possible to develop other interfaces to the run-time system and produce tools that augment the development environment; they're not needed when programming in Objective C.

  However, a few of the run-time functions might on occasion be useful when writing an Objective C program. These functions—such as **sel_getUid()**, which returns a method selector for a method name, and **objc_lookUpClass()**, which returns a class object for a class name—are described at various places in the text of this manual.

  All the run-time functions are fully documented in the *NeXTSTEP General Reference* manual.

Because the Object class is at the root of all inheritance hierarchies, the methods it defines are 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 Object class merely defines a framework for how something should be done; it doesn't provide all the necessary code itself.

For example, the Object class defines a **name** method that should return a character string associated with the receiver:

```
if ( !strcmp([anObject name], "Connochaetes taurinus") )
    . . .
```

If you define a class of named objects, you must implement a **name** method to return the specific character string associated with the receiver. Object's version of the method can't know what that name will be, so it merely returns the class name as a default.

This chapter looks at five areas where the Object class provides a framework and defines conventions, but where you may need to write code to fill in the details:

- Allocating and initializing new instances of a class
- Deallocating instances when they're no longer needed
- Forwarding messages to another object
- Dynamically loading new modules into a running program
- Archiving objects—for example, storing them in a file on disk

Other conventions of the Object class are described in Appendix C.

Allocation and Initialization

It takes two steps to create an object in Objective C. You must both:

- Dynamically allocate memory for the new object, and
- Initialize the newly allocated memory to appropriate values.

An object isn't fully functional until both steps have been completed. As discussed in Chapter 2, each step is accomplished by a separate method, but typically in a single line of code:

```
id anObject = [[Matrix 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 in fact controlled and modified.

Allocating Memory for Objects

In Objective C, memory for new objects is allocated using class methods defined in the Object class. Object defines two principal methods for this purpose, **alloc** and **allocFromZone:**.

```
+ alloc;
+ allocFromZone:(NXZone *)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 argument passed to **allocFromZone:** determines where the new object will be located in memory. It permits you to group related objects into the same region of memory for better performance.

Zones

In a multitasking environment like NeXTSTEP, users typically run several applications at once. These applications can easily require more memory than will physically fit on the user's system.

To solve this problem, NeXTSTEP, like most modern systems, makes use of virtual memory—a system for addressing more information than can actually be accommodated in main memory. Whenever an application references some information, the system determines whether the memory page containing that information resides in main memory. If it doesn't, the page with the requested information must be read in. If there's no room for the new page, a page of resident memory must be stored to the disk to make room.

This swapping of pages in and out of main memory, to and from the disk, is much slower than a direct memory reference. It slows the execution of applications, and, in a multitasking environment, can degrade the overall responsiveness of the system. Reducing swapping to a minimum can greatly increase system performance.

One way to reduce swapping is to improve *locality of reference*, the chance that the next piece of information the system needs to reference will be located close to the last piece of information referenced, perhaps on the same page, or at least on a page recently referenced and so still in main memory. The idea is to minimize the number of pages that must be resident for a given operation by putting related information on the same page (or the same few pages) and keeping unrelated, or rarely used, information on other pages.

To this end, NeXTSTEP lets you partition dynamic memory into *zones* and direct which zone objects (and other data structures) should be allocated from.

Zones are recorded in NXZone structures, one per zone. These structures are provided by the system; you don't have to allocate memory for them or make copies. You also don't need to look inside the structure or manipulate its fields. You can simply regard pointers to the structures as zone identifiers.

The system creates one default zone for each application, which is returned by **NXDefaultMallocZone()**.

```
NXZone *defaultZone = NXDefaultMallocZone();
```

Other zones can be created by the **NXCreateZone()** function.

```
NXZone *newZone = NXCreateZone(vm_page_size * 2, vm_page_size, YES);
```

This function takes three arguments:

- The initial size of the zone in bytes.
- The granularity of the zone (how much it should grow or shrink by).
- Whether it's possible to free memory from the zone. For most zones, this normally is YES. However, it can be NO if a zone is to be used temporarily, then destroyed (with **NXDestroyZone()**). Destroying a zone effectively deallocates all the memory within it.

The initial size of a zone and its granularity should be set to small multiples of a page size, since a page is the smallest amount of memory handled by the virtual memory system. The size of a page can vary from installation to installation; its current value is stored in the **vm_page_size** global variable declared in **mach/mach_init.h**.

Ideally, zones should be moderate in size. Large zones may fail to group related data onto a small number of pages; they're prone to the same problem that zone allocation is meant to correct: the fragmentation of data across many pages.

It's also not a good idea to have a large number of zones with very little information in them. The free space in one zone won't be available for allocation from other zones, so an application could end up using more memory than it should.

Allocating from a Zone

The **allocFromZone:** method permits you to cluster related objects (such as a Matrix and its Cells) in the same region of memory. It takes a pointer to a zone as its argument:

```
NXZone *matrixZone = NXCreateZone(vm_page_size, vm_page_size, YES);
id newObject = [[Matrix allocFromZone:matrixZone] init];
```

The **zone** method returns the zone of the receiver and can be used to make sure one object is allocated from the same zone as another object. For example, a Matrix could be allocated from the same zone as the Window it will be displayed in:

```
id aMatrix = [[Matrix allocFromZone:[myWindow zone]] init];
```

The **NXZoneMalloc()** function lets you specify a zone when dynamically allocating memory for data structures that aren't objects. It's arguments are a zone and the number of bytes to be allocated:

```
float *points = (float *)NXZoneMalloc(NXDefaultMallocZone(),
                                       sizeof(float) * numPoints);
```

Allocation methods and functions that don't specify a zone, such as the **alloc** method, take memory from the default zone. The standard C **malloc()** function allocates from the default zone, or from memory outside any zone.

Objects that are commonly used together should be kept together in the same zone, along with any related data structures the objects use. For example, all the objects that contribute to a particular document and its display (the Window object, View objects, text data structures, and so on) could be kept together in the same zone, one created just for the document. When the document isn't open, none of the pages in the zone will clutter main memory.

It's equally important to keep rarely used objects separate from those that are used more frequently. For example, users only occasionally refer to an application's information panel (usually only when first becoming familiar with the application). If the objects that contribute to the panel share pages with objects that are used regularly, they will take up space in main memory even when they're not needed.

If your application often both allocates and frees a certain type of object, there are a couple of considerations to keep in mind. First, freeing tends to fragment memory. It might be best to keep all these objects in the same zone to prevent the fragmentation of other zones. Second, freeing takes a bit of time, because newly freed memory must be coalesced with memory already free. Rather than free each object individually, you might locate them all in a temporary zone that can't free memory, then destroy the whole zone at once (through a call to **NXDestroyZone()**). Such a zone can allocate memory quickly, but can only grow in size, so you should use this technique only if you will soon destroy the zone.

Initializing New Objects

The **alloc** and **allocFromZone:** methods initialize a new 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, a View can be initialized with an **initFrame:** method.

Every class that declares instance variables must provide an **init**... method to initialize them. The Object class declares the **isa** variable and defines an **init** method. However, since **isa** is initialized when memory for a new object is allocated, all Object's **init** method does is return **self**. Object declares the method mainly to establish the naming convention described above.

The Object Returned

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 **initName:** method to initialize new instances. If there can be no more than one object per name**, initName:** 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 **allocFromZone:**. The following code is very dangerous, since it ignores the return of **init**.

```
id anObject = [SomeClass alloc];
[anObject init];
[anObject someOtherMessage];
```

It's recommended that you combine allocation and initialization messages:

```
id anObject = [[SomeClass alloc] init];
[anObject someOtherMessage];
```

If there's a chance that the **init...** method might return **nil**, the return value should be checked 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 **initName: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. One option is to substitute a default value, and to let a null argument explicitly evoke the default.

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:

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

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 Matrix object must be initialized as an Object, a Responder, a View, and a Control before it's initialized as a Matrix. (See Figure 7 in Chapter 2 for the Matrix inheritance hierarchy.)

The connection between the **initName:** method illustrated above and the inherited **init** method it incorporates is diagrammed in the figure below:

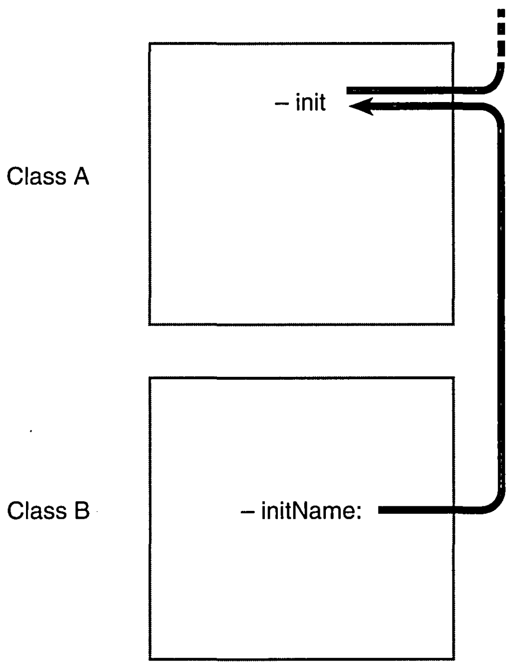


Figure 15. Incorporating an Inherited Initialization Method

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 **initName:** method, as shown in the figure above, B must also make sure that an **init** message will successfully initialize B instances. The easiest way to do that is to replace the inherited **init** method with a version that invokes **initName:**.

```
- init
{
    return [self initName:"default"];
}
```

The **initName:** method would, in turn, invoke the inherited method, as was shown in the example and figure above. That figure can be modified to include B's version of **init**.

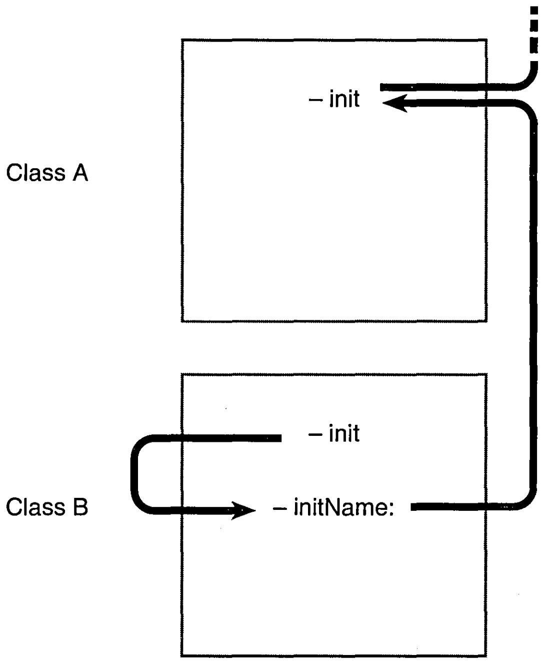


Figure 16. Covering an Inherited Initialization Method

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.

In the example above, **initName:** 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 NeXTSTEP convention that the designated initializer is always the method that allows the most freedom to determine the character of a new instance (the one with the most arguments).

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 **initName:fromFile:** method. In addition to this method, we have to make sure that the inherited **init** and **initName:** methods also work for instances of C. This can be done just by covering B's **initName:** with a version that invokes **initName:fromFile:**.

```
- initName:(char *)string
{
    return [self initName:string fromFile:NULL];
}
```

For an instance of the C class, the inherited **init** method will invoke this new version of **initName:** which will invoke **initName:fromFile:**. The relationship between these methods is diagrammed below.

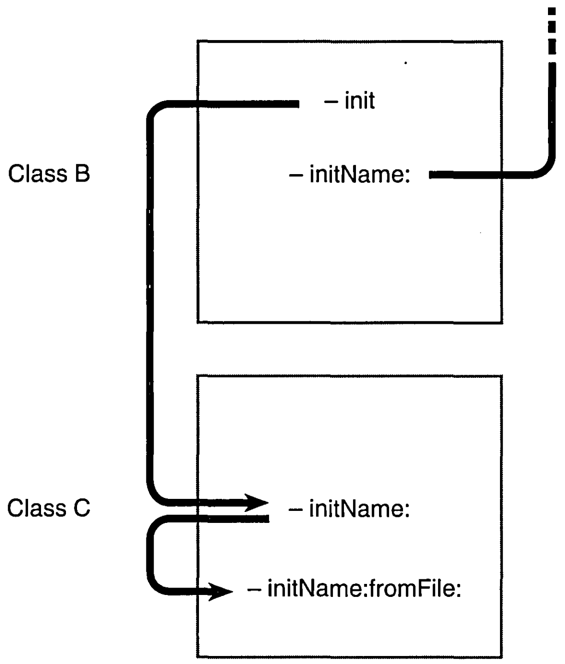


Figure 17. Covering the Designated Initializer

This figure omits an important detail. The **initName:fromFile:** method, being the designated initializer for the C class, will send a message to **super** to invoke an inherited initialization method. But which of B's methods should it invoke, **init** or **initName:**? It can't invoke **init**, for two reasons:

- Circularity would result (**init** invokes C's **initName:**, which invokes **initName:fromFile:**, which invokes **init** again).
- It won't be able to take advantage of the initialization code in B's version of **initName:**.

Therefore, **initName:fromFile:** must invoke **initName:**.

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

The general principle is this:

The designated initializer in one class must, through a message to ***super****, invoke the designated initializer in an inherited class.*

Designated initializers are chained to each other through messages to **super**, while other initialization methods are chained to designated initializers through messages to **self**.

The figure below 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.

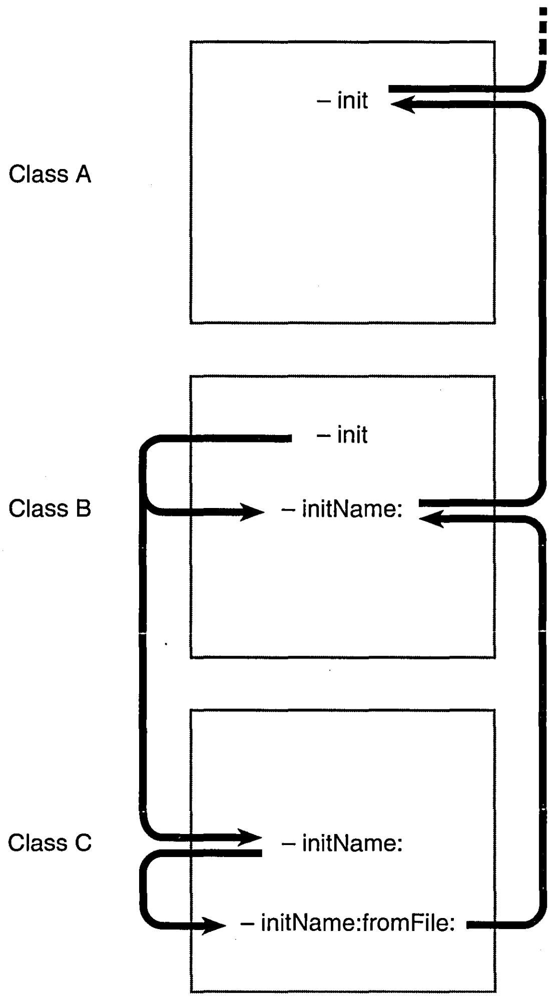


Figure 18. Initialization Chain

Note that B's version of **init** sends a message to **self** to invoke the **initName:** method. Therefore, when the receiver is an instance of the B class, it will invoke B's version of **initName:**, and when the receiver is an instance of the C class, it will invoke C's version.

Combining Allocation and Initialization

The Object class defines a **new** method that combines the two steps of allocating and initializing a new object. Just as you can define **init...** methods with arguments, you can also define **new...** methods that take similar arguments. For example:

```
+ newName:(char *)string
{
    return [[self alloc] initName:string];
}
```

However, there's little point in implementing a **new...** method like this that simply covers for other allocation and initialization methods.

On the other hand, a **new...** method does make sense 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 **newListFromFile:** 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 under "The Object Returned" above, an **init...** method might sometimes substitute another object for the receiver. For example, when **initName:** 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 checks 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 and therefore would not have to free it.

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;
}
```

Note that this method is not named "new" since it rarely returns a new object.

Deallocation

The Object class defines a **free** method that releases the memory that was originally allocated for an object. Because objects are created dynamically at run time, the memory they occupy must be freed when they've outlived their usefulness. This is accomplished by telling the object to free itself:

```
[anObject free];
```

The point of a **free** message is to deallocate all the memory occupied by the receiver. Object's version of the method deallocates the receiver's instance variables, but doesn't follow any variable that points to other memory. If the receiver allocated any additional memory—to store a character string or an array of structures, for example—that memory must also be freed (unless it's shared by other objects). Similarly, if the receiver is served by another object that would be rendered useless in its absence, that object must also be freed.

Therefore, it's necessary to override Object's version of **free** and implement a version that deallocates all the other memory the object occupies. Every class that has its objects allocate additional memory must have its own **free** method. Each version of **free** ends with a message to **super** to perform an inherited version of the method:

```
- free
{
    free(buffer);
    if ( vmMemory )
        vm_deallocate(task_self(), vmMemory, memorySize);
    [servant free];
    return [super free];
}
```

By working its way up the inheritance hierarchy, every **free** message eventually invokes Object's version of the of the method.

Object's version of **free** returns **nil**. If for some reason a **free** method is unable to free the receiver, it should avoid the message to **super** and return **self** instead.

Note: Although **free** returns **nil**, it doesn't automatically change the receiver's **id** to **nil**; it merely makes the **id** invalid. Any further messages sent to the object will produce errors. To be safe, you can assign **free**'s return value to the variable that stores the **id**:

```
myObject = [myObject free];
```

Forwarding

It's an error to send a message to an object that can't respond to it. However, before announcing the error, the run-time system gives the receiving object a second chance to handle the message. It sends the object a **forward::** message with two arguments—the method selector in the original message and a pointer to the arguments that were passed with it. These two arguments fully specify the message the receiver was unable to respond to.

A **forward::** method can be implemented to give a default response to the message, or to avoid the error in some other way. As its name implies, **forward::** 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 **negotiate** message, 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 that you might want to forward. That set might depend on events at run time, and it might change as new methods and classes are implemented in the future.

The second chance offered by a **forward::** 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 run-time system informs the object by sending it a **forward::** message. Every object inherits a **forward::** method from the Object class. However, Object's version of the method simply generates a run-time error due to the unrecognized message. By overriding Object's version and implementing your own, you can take advantage of the opportunity that the **forward::** message provides to forward messages to other objects.

To forward a message, all a **forward::** 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 **performv::** method:

```
- forward:(SEL)aSelector :(marg_list)argFrame
{
    if ( [someOtherObject respondsTo:aSelector] )
        return [someOtherObject performv:aSelector :argFrame];
    else
        . . .
}
```

The original message will return whatever value **forward::** returns. The return type should be **id**.

The **forward::** method's two arguments are the selector in the unrecognized message and the stack frame containing the arguments that were passed in the message. (Even methods like **negotiate** that declare no outward arguments are implemented with the two hidden arguments, **self** and **_cmd**, so the stack frame won't be empty.) Note that **forward::**'s arguments are passed unchanged to **performv::**.

A **forward::** 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 **forward::** method can also consolidate several messages into a single response. What **forward::** 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 **forward::** 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 **forward::**.

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 19 below, an object that responds to a message by forwarding it appears to borrow or "inherit" a method implementation defined in another class.

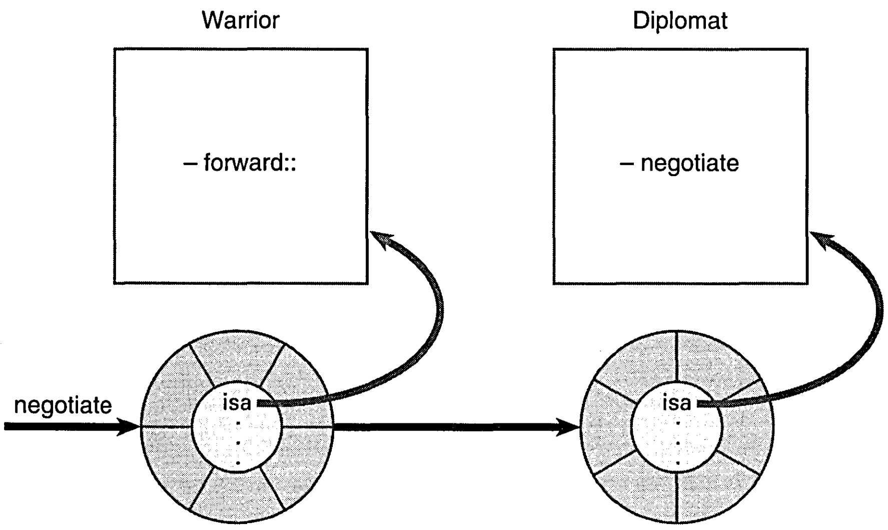


Figure 19. Forwarding

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 will appear as if the Warrior class inherits from Diplomat as well as its own superclass.

Forwarding addresses most needs that lead programmers to value multiple inheritance. However, there's an important difference between the two: Multiple inheritance combines different capabilities in a single object. It tends toward large, multifaceted objects. Forwarding, on the other hand, assigns separate responsibilities to separate 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 under "Remote Messaging" in Chapter 3 is such an object. 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 **forward::** method first receives a message destined for the other object, it would check to be sure 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.

Making Forwarding Transparent

Although forwarding mimics inheritance, the Object class never confuses the two. Methods like **respondsTo:** and **isKindOf:** 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 respondsTo:@selector(negotiate)] )
    . . .
```

the answer will be 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 19 above.)

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 reimplement the **respondsTo:** and **isKindOf:** methods to include your forwarding algorithm:

```
- respondsTo:(SEL)aSelector
{
    if ( [super respondsTo: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 **respondsTo:** and **isKindOf:**, the **instancesRespondTo:** and **isKindOfClassNamed:** methods should also mirror the forwarding algorithm. These two methods round out the set. If protocols are used, the **conformsTo:** methods should likewise be added to the list. Similarly, if an object forwards any remote messages it receives, it should have versions of two other methods, **descriptionForMethod:** and **descriptionForInstanceMethod:**, 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, **forward::** included, call it.

Note: All the methods mentioned above are described in Appendix C, "The Object Class."

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, device drivers written with the NeXTSTEP Device Driver Kit™ (a Release 3.1 addition) are dynamically loaded into the kernel. Adaptors for database servers are dynamically loaded by the Database Kit.

In the NeXTSTEP environment, dynamic loading currently finds its favored use in customizing applications. You can allow others to write modules that your program will load at run time—much as the NeXTSTEP Interface Builder loads custom palettes, the Preferences application loads custom displays, and the Workspace Manager loads data inspectors. 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.

Dynamically loaded modules that customize an application generally come with their own user interface—perhaps their own windows, but more likely objects that draw in windows you provide. When the code is loaded and objects are instantiated or unarchived, the interface to the custom portion of the application is presented on-screen along with the rest of the user interface.

The Preferences application, for example, has a window with a scrollable list of buttons along the top, plus a display area beneath the buttons. Each button controls the presentation of a different set of options within the display area; clicking a button causes its options to be displayed. Each dynamically loadable module provides a display that can be presented in the window, along with an image for the button and the code to handle user actions and set preferences. The Preferences window illustrated below shows the localization button highlighted and, beneath it, the display of localization options.

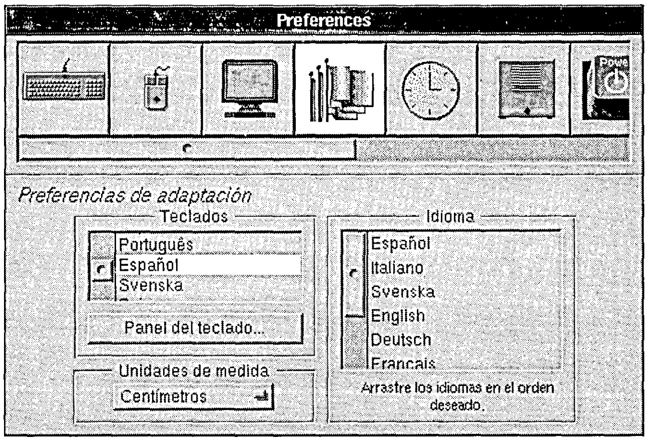


Figure 20. The Preferences Application

Bundles

Classes and categories are dynamically loaded and linked by calling the **objc_loadModules()** function. They can be unlinked and unloaded again by calling **objc_unloadModules()**. However, once code is loaded, it typically remains in place; there's little reason to unload it.

These two functions are part of the Objective C run-time system and provide the basic methodology for dynamic loading and unloading. They're documented in the *NeXTSTEP General Reference* manual. However, the NeXTSTEP environment also provides another, more convenient interface for dynamic loading—one that's object-oriented and integrated with related services. The loading task can be assigned to an NXBundle object.

NXBundles correspond to directories where programs store resources they'll refer to at run time. Each object manages one directory. The directory "bundles" the resources and makes them available to the NXBundle object and, through the object, to the program. The directory might contain image data, sound files, objects that were archived into so-called "nib files" by Interface Builder, tables of character strings, and other resources. It can also contain a file of executable code.

An NXBundle object is initialized to a bundle directory with the **initForDirectory:** method as shown below (though, of course, you'd rarely use a hard-wired path like this):

```
char *path = "/LocalLibrary/Preferences/Music.preferences";
NXBundle *myBundle = [[NXBundle alloc] initForDirectory:path];
```

An NXBundle can do two things with the information stored in the bundle directory:

- Dynamically load the executable code and return class objects for the newly loaded classes

- Find resources that match the user's language preference and make them available to the application

These two things go together; the whole point of an NXBundle is to combine them in a single facility. Dynamically loaded code doesn't stand on its own. It typically requires the support of various resource files—archived instances of the bundled classes, character strings that the code displays to users, bitmap images to place within the display, and so on. Code and resources are grouped together in the same directory and are managed together by the same NXBundle object.

Localized Resources

Typically, a bundle directory packages a file of loadable code with all the resources that the code requires. Some resources in the bundle might occur in various alternative forms "localized" to a particular language or region of the world. For example, the English string "Select All" might have counterparts in Spanish ("Seleccionar todo"), German ("Alles auswählen"), French ("Tout sélectionner"), Swedish ("Markera allt"), and other languages.

Localized resources are kept in subdirectories of the bundle directory. Each subdirectory is named after a language and carries a ".lproj" extension (for "language project"). For example, there might be **Swedish.lproj**, **English.lproj**, and **Tagalog.lproj** subdirectories. Each subdirectory has a matched set of files. If the user sets the language preference to Swedish, the application will use the files in **Swedish.lproj**. If the preference is set to English, **English.lproj** files will be used. When asked for a resource, an NXBundle looks in the subdirectory that matches the current preference.

The figure below illustrates the layout of a possible bundle for the Preferences application. The **Music** file holds the loadable code and **Music.tiff** holds the bitmap image (in Tag Image File Format) that will be displayed on the button at the top of the window. The rest of the files are localized and located in ".lproj" subdirectories.

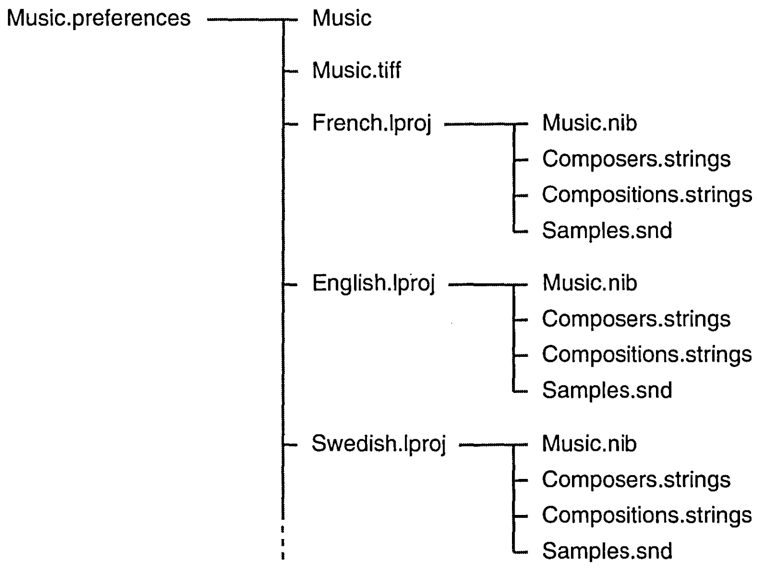


Figure 21. A Bundle Directory

Note: Language preferences are set using the Preferences application, as shown above in Figure 20. The application not only sets the preference, it is itself localized and reflects the current choice. In the figure, the language preference is set to Spanish and, accordingly, Spanish is displayed in the window.

Loadable Code

The loadable code in a bundle directory must be in a file with the same name as the directory (minus any extension on the directory name) and it must contain nothing but compiled class and category definitions.

Bundled code is not localized. Rather, it's kept free of any content that would vary depending on the language or location where the software is used. This content is extracted from the code and put in resource files within the ".lproj" subdirectories. The same NXBundle object that loads the executable code can find the required resources at run time.

Dynamic loading therefore should not be seen as the isolated task of loading and linking class and category code. It also includes loading the objects, images, strings, sounds, and other resources that are required at run time. The decision of which resources to load must be dynamic for it depends on information available only at run time—the user's language preference.

Loading Bundled Code

When requested, an NXBundle returns class objects for the classes bundled within its directory. It waits until it receives the first request to load the bundled code. This message, for examplc, asks an NXBundle for the Mozart class:

```
Class composer = [myBundle classNamed:"Mozart"];
```

If the executable code stored in the bundle had not yet been loaded, this message would load it. All bundled classes are located in one file and are loaded at the same time. If the file doesn't contain the requested class, **classNamed:** returns **nil**.

The **classNamed:** method finds a specific class within the bundle, one that you request by name. Typically, however, an application needs to find only one class from the dynamically loaded file (at least initially), and it won't know or care what the class is named. The **principalClass** method returns this class:

```
Class head = [myBundle principalClass];
```

Like **classNamed:**, **principalClass** dynamically loads the bundled code if it hasn't already been loaded.

A set of bundled classes often supports a small subnetwork of objects that can be attached to the larger object network already in place. The connection is established through just one object, an instance of the principal class. That object might have methods to return other objects that the application can talk to, but typically all messages from the application to the subnetwork are funneled through the one instance.

The NXBundle expects the principal class to be the first one encountered in the executable file. When several classes are linked into a dynamically loadable file, the principal class should be the first one listed on the **ld** command line. For example, this command makes DiscJockey the principal class in the Music bundle:

```
ld -o Music -r DiscJockey.o Bach.o Mozart.o Coltrane.o . . .
```

Each application has a choice to make regarding when to load bundled code. It can load the code at start-up before the interaction with the user begins, or it can wait until the user requests it. There are benefits to waiting. For example, when it encounters a bundle of loadable code, the Preferences application immediately creates a button for it and puts the button in the scrollable list at the top of the window. It takes the image for the button from the bundle directory (**Music.tiff** in Figure 21 above). However, it doesn't dynamically load the bundled code (the **Music** file Figure 21) until the user clicks the button. Code that isn't used isn't loaded.

For some applications, the user's request can be even more explicit. For example, you might present the user with a panel that displays file icons for each bundle containing dynamically loadable code. The panel might even display some information about each bundle, information supplied by the customizer in the bundle directory. When the user double-clicks an icon, the application would load it and display its user interface on-screen.

No matter what triggers dynamic loading, when your application is ready to start using the bundled code, the **principalClass** method will load it.

Organizing for Customization

To allow others to customize your application, you need to set up a framework for finding, loading, and executing their code. So that customizers can prepare their code to work within your framework, you'll need to let them know what the framework is and what you expect of them.

You are, in effect, entering into a contract of sorts with the customizer. The framework lets you carry out your end of the bargain; the information you provide lets them carry out theirs. You need to furnish customizers with at least the following information:

- The path where your application will search for their code
- The naming conventions you expect their bundle directories to follow
- A protocol that they should implement so that your objects can talk to one of theirs
- A protocol that you implement so that their objects can talk to one of yours
- The interface their code can be linked against when it's loaded

The following sections discuss these matters in more detail.

Search Path

Code must be located before it can be loaded. If you're setting up a framework that lets others add to your application, the first requirement is to publish the places where the application will look for bundles with dynamically loadable code. This usually takes the form of a search path. For example, the Preferences application looks for bundles inside a Preferences subdirectory located in one of the "Library" directories. It looks in this order:

- First, in the user's home directory, **~/Library/Preferences**,
- Next, in the site-specific library, **/LocalLibrary/Preferences**,
- Then, in the NeXT-supplied library, **/NextLibrary/Preferences**, and
- Finally, in the application file package itself, **/NextApps/Preferences.app**.

This kind of path prefers the user version of a bundle over the site version, and the site version over one supplied by NeXT. If two or more bundles have the same name, the one that's encountered first is the one that's chosen, and those encountered afterward ignored.

Preferences includes its own file package in the search path because all of the buttons and preference options it displays are loaded from bundles at run time. The file package stores the bundles that come with the application.

Note: The fact that Preferences dynamically loads its own code has benefits both for the user and for the programmer. For the user, it means reduced launch times. For the programmer, it means faster link times. It also means that Preferences can have a simpler design—a single paradigm is used for interacting with all preference modules. Moreover, the customization framework is thoroughly tested in Preferences' own development.

Bundle Name

In addition to the search path, you'll also need to publish any constraints you impose on the name of the bundle directory. Preferences expects its bundles to carry a specific extension (".preferences"), but doesn't restrict the rest of the name. This seems like a good precedent to follow. The name shouldn't be so constrained that it limits the number of loadable modules, but it must be recognizable as being intended for a particular application.

If you register the extension as one that denotes documents belonging to your application, the Workspace Manager will treat the bundle directory as a file package; it will display the directory as if it were a file. Just as users don't look inside files, they don't normally look inside file packages. This hides the contents of the bundle directory, simplifies its interface, and ensures its integrity.

Communication Protocols

Bundled code is loaded simply by asking the NXBundle for the principal class, as mentioned above. But getting the principal class is just the first step. Once your program has the class, it needs to create an instance and talk to it.

To be able to write code that can communicate with an instance of a dynamically loaded class, you need to know what messages the new instance will respond to. To arrange this communication, you'll need to publish a protocol of methods that you expect customizers to implement in the principal class. This protocol should include an initialization method, so that your code can properly initialize a new instance.

If you want the dynamically loaded code to be able to talk to your objects (often a good idea), you'll also need to publish another protocol, one that's implemented by an object in your program. One of the first things your code should do is pass this object to the new instance of the principal class.

For example, if the protocol you implement is called ApplicationResponsibilities,

```
@protocol ApplicationResponsibilities
. . .
@end
```

the protocol you expect the principal class to implement could include an **initContact:** method that would both (a) initialize a new instance of the principal class and (b) pass that instance an object that conforms to the ApplicationResponsibilities protocol:

```
@protocol CustomizerResponsibilities
- initContact:(id <ApplicationResponsibilities>)anObject;
. . .
@end
```

Both protocols would be declared in a header file that you make available to potential customizers.

Before your application begins interacting with the customizer's code, it's a good idea to check whether the principal class conforms to the expected protocol:

```
if ( [[theBundle principalClass]
            conformsTo:@protocol(CustomizerResponsibilites)] ) {
    . . .
}
```

If it does conform, you can create an instance of the principal class and begin talking to it:

```
id adjunct = [[[theBundle principalClass] alloc] initContact:myAgent];
if ( adjunct )
    . . .
```

The object passed as **initContact:**'s argument is, in a sense, a counterpart to the new instance of the principal class. It receives messages from dynamically loaded code, just as the new instance receives messages sent from application code.

As an alternative to the two protocols, you could provide an interface to an abstract class that developers should subclass. The abstract class would declare methods—equivalent to the CustomizerResponsibilites protocol—that customizers should implement in their subclasses. It could also implement methods—equivalent to the ApplicationResponsibilities protocol—that customizers could use in their code to communicate with your application. The abstract class would reside in your application, rather than in a library. The superclass-subclass relationship is resolved when the customizer's subclass is loaded and linked dynamically.

Note: Preferences and other applications that come with NeXTSTEP choose an abstract class over the two protocols. The class gives them the opportunity to lighten the customizer's burden by providing, through inheritance, part of the customization code.

Development Interfaces

Most NeXTSTEP software resides in shared libraries, principally **libNeXT_s** and **libsys_s**. Linking to a shared library gives an application access to library code, but it doesn't actually incorporate the code into the application. At run time, one copy of the code is shared by all applications linked to the library.

An application can be linked to a shared library, but the code it dynamically loads can't be. Bundled code must be self-contained; any unresolved symbols within it must be resolved by symbols found within the application it's dynamically linked to. It can't maintain an independent connection to a shared library. Therefore, if others are to customize your application, you'll need to inform them of the interface they can link against.

Normally, dynamically loaded code can link against any program symbols that are global to the application or any shared library symbols the application actually references. It can't use facilities in the shared library that the application itself doesn't use. However, if you link your application to shared libraries with the **–ObjC** flag (the Project Builder default), all classes in the libraries will be available to customizers, whether or not the application also uses them. In addition, if you list the names of shared libraries after the **–u** option,

```
ld -o myApp -ObjC -u libNeXT_s -u libsys_s . . .
```

all symbols defined in the listed libraries will be available to dynamically loaded code.

The **strip** utility can be used to remove extraneous symbols from the final version of your application. You should be careful to leave symbols that might be needed by dynamically loaded code. The **–A** option leaves all global symbols from a shared library and all Objective C class symbols:

```
strip -s appSymbolsToSave -A myApp
```

In summary, it's important to let customizers know what programming interface they can rely on when developing dynamically loadable code, including:

- Whether your application defines a programming interface that they can use,
- Which libraries the application is linked against, and
- Which part of the library interface is available to them.

Archiving

NeXTSTEP software and the Object class support *archiving*, copying an object's data structure from dynamic memory to some other location. The other location can be a file where the object is stored until it's later reactivated, another application that will use the copy in its own way, the pasteboard, a port, or some other destination. Archiving permits objects to persist between sessions of the same application, and to be passed from one application to another.

Only the object data structure is archived (only its instance variables), not any of the class code that includes the object's methods and other information it needs to function properly. For an object to be of any use once it's read from the archive, the program that reads it must have access to this code. The class must have been loaded, either at startup or dynamically later on, before the object is unarchived.

Interface Builder is perhaps the most dramatic example of how archiving can be used. It permits you to design an application on-screen using graphical tools. You choose objects from a palette, manipulate them graphically to initialize them, and connect them to each other to form a program network. Interface Builder archives these objects in a file; they can then be unarchived as part of your application, or again in Interface Builder to further modify the design.

Typed Streams

Objects are archived by writing them to a special kind of data stream, a *typed stream*, that accepts not only the object's data but also information about what type of data it is and what class the object belongs to. The class information is needed to reconnect the object to its class when it's unarchived.

Data written to the stream might go to a file, to the pasteboard, to memory somewhere, or to some other destination. It doesn't matter; the archiving mechanism doesn't distinguish among destinations. (In this discussion, it will be assumed that the stream is connected to a file.)

Objects are unarchived in similar fashion by reading them from a typed stream. Again, the mechanism is oblivious to the source of the data. The stream might be connected to a file, to memory, to the pasteboard or a port, or to some other source. However, typed streams impose a format on the data in the archive. Therefore, you can read data using a typed stream only if the data was previously written using a typed stream.

Because the archiving mechanism is independent of the destination or source of the data, the same code can be reused in a variety of situations. Unarchiving an object from an Interface Builder file, for example, uses the same code as retrieving it from the pasteboard or receiving it as an argument in a remote message.

Typed streams are recorded in NXTypedStream structures. The functions that open a stream for reading or writing return an NXTypedStream pointer. For example, **NXOpenTypedStreamForFile()** opens a typed stream on a file:

```
NXTypedStream *stream;
stream = NXOpenTypedStreamForFile("/home/archive", NX_READWRITE);
```

The two arguments to **NXOpenTypedStreamForFile()** are a file name and a specification for how the stream will be used (NX_READWRITE, NX_READONLY, or NX_WRITEONLY).

Like NXZone and FILE structures, you don't have to allocate memory for an NXTypedStream or look inside the structure; a pointer to the structure can be regarded as a stream identifier.

Reading and Writing

Archiving is initiated by calling a function that writes an object to a typed stream, typically **NXWriteRootObject()**:

```
NXWriteRootObject(stream, anObject);
```

This generates a **write:** message to the object, telling it to write its instance variables to the stream. (The "root object" in the function name refers not to the root of the inheritance hierarchy, but to the object passed as the argument. Since archiving this object might indirectly cause other objects to be archived, it's the one that "roots" (initiates) the archiving sequence.)

Objects are unarchived by the inverse of this process, usually initiated by calling **NXReadObject()**:

```
id anObject = NXReadObject(stream);
```

This function allocates memory for the object, reconnects it to its class, and generates a **read:** message to the object telling it to reinitialize its instance variables from the stream.

The write: and read: Methods

The Object class defines default versions of the **write:** and **read:** methods. However, Object's versions can't know about instance variables declared in other classes. Thus, any class that declares instance variables must supply its own versions of **write:** and **read:** to archive and unarchive them. Every class has responsibility for reading and writing its own instance variables.

So that **write:** and **read:** messages will archive or unarchive all of an object's instance variables, each version of either method should incorporate, through a message to **super**, the version it overrides:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    . . .
}

- read:(NXTypedStream *)stream
{
    [super read:stream];
    . . .
}
```

Because the message to **super** comes first, instance variables are archived in the order of inheritance and unarchived in the same order. Those declared in a superclass are handled before those declared in a subclass.

Every **write:** method must be matched by a **read:** method. The two methods are the inverse of each other; whatever **write:** writes, **read:** reads.

However, **write:** and **read:** don't have to account for every instancc variable. It's more efficient to skip over instance variables that fall into one of the following categories:

- Instance variables that are not essential to the character of the object
- Those that would be invalid when the object is unarchived in a different context
- Those that can be recalculated from scratch when the object is unarchived

For example, it would be better not to archive a variable that records a transitory state—one that changes often as the program runs (such as the current selection in a body of text). Rather, the variable could simply be set to an arbitrary initial value when the object is unarchived.

The Archiving Functions

After the message to **super**, a **write:** method must get down to the business of archiving a set of instance variables. A number of functions are provided for this purpose, the most general of them being **NXWriteType()**, which archives a single variable, possibly a structure, and **NXWriteTypes()**, which archives a series of variables, none of which can be structures. As their first argument, both functions take a pointer to the stream. As their second argument, both take a string of characters encoding all the types to be written, including all the types within a structure. These type codes are almost identical to those provided by the **@encode()** compiler directive. (Differences are explained in the *NeXTSTEP General Reference* manual.) Pointers to the data to be archived follow the type codes.

For example, imagine a class that declares these six instance variables,

```
{
    struct key info;
    char *name;
    double factor;
    unsigned int mask;
    unsigned int flags;
    int state;
}
```

where the **key** structure consists of just an integer and a string:

```
struct key {
    int i;
    char *s;
};
```

It might archive the instance variables as follows:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    NXWriteType(stream, "{i*}", &info);
    NXWriteTypes(stream, "*dII", &name, &factor, &mask, &flags);
    return self;
}
```

Note that this method ignores the **state** variable.

The archiving functions follow a character pointer (coded '*') to archive the string it points to, and an object pointer (coded '@') to archive the object. However, they don't follow other kinds of pointers. You must explicitly archive any data that the object refers to but that resides in memory outside the object.

The **NXReadType()** and **NXReadTypes()** functions read back what **NXWriteType()** and **NXWriteTypes()** write:

```
- read:(NXTypedStream *)stream
{
    [super read:stream];
    NXReadType(stream, "{*i}", &info);
    NXReadTypes(stream, "*dII", &name, &factor, &mask, &flags);
    state = 1;
    return self;
}
```

A **read:** method might also reinitialize instance variables that were not archived. Here it resets **state** to 1;

Outlet Instance Variables

Most objects have instance variables that point to other objects. As explained in Chapter 1, these *outlets* let an object keep track of the other objects it needs to communicate with. They serve to define the connections between objects in a program.

Outlets raise a question for archiving: When one object is archived, should the objects that its outlets point to also be archived?

The answer to this question can't be a universal "yes." If it were, archiving one object might result in archiving a whole series of unwanted objects. Archiving a Matrix, for example, would also archive the Window object for the window where the matrix is drawn, along with every other object that draws in the window, all objects those objects are connected to, and so on. The answer also can't always be "no." If it were, essential elements of an archived object might be missing when it was unarchived. You'd get back a Matrix without its Cells, for example.

How an outlet instance variable is archived depends on the nature of the connection it represents. In NeXTSTEP, you have three options:

- If an outlet object is the private servant of the object being archived and can be recreated from scratch without losing information, it doesn't need to be archived. The **write:** method can simply ignore the object and **read:** can produce a new one to take its place.

  ```
  - read:(NXTypedStream *)stream
  {
      [super read:stream];
      anOutlet = [[SomeClass alloc] init];
      . . .
  }
  ```

- If an outlet object is intrinsic to the object being archived, crucial to its character, or required for its operation, the **write:** method should archive it. For example, a Matrix archives its Cells. **write:** can archive an object by calling **NXWriteType()** or **NXWriteTypes()** and passing it the '@' type code that designates an object:

  ```
  - write:(NXTypedStream *)stream
  {
      [super write:stream];
      NXWriteType(stream, "@", &anOutlet);
      . . .
  }
  ```

  It's even simpler to use the shorthand **NXWriteObject()** method:

  ```
  - write:(NXTypedStream *)stream
  {
      [super write:stream];
      NXWriteObject(stream, anOutlet);
      . . .
  }
  ```

  These three functions have equivalent results; they each initiate a **write:** message to the outlet object.

  NXWriteObject() should be balanced by a call to **NXReadObject()** in the **read:** method. **NXWriteType()** is balanced by **NXReadType()** and **NXWriteTypes()** is balanced by **NXReadTypes()**.

- If an outlet object is only peripherally connected to the object being archived, the **write:** method can call **NXWriteObjectReference()** to ask that a reference to it be maintained in case the object is archived for some other reason:

  ```
  - write:(NXTypedStream *)stream
  {
      [super write:stream];
      NXWriteObjectReference(stream, anOutlet);
      . . .
  }
  ```

  This function doesn't archive the object. However, it's possible that the object will be archived anyway, perhaps because some other **write:** method requests it. If so, **NXWriteObjectReference()** guarantees that the outlet connection will be reestablished when the objects are unarchived. If not, it makes sure the outlet instance variable is set to **nil**.

  Like **NXWriteObject()**, **NXWriteObjectReference()** is balanced by calling **NXReadObject()** in the **read:** method:

  ```
  - read:(NXTypedStream *)stream
  {
      [super read:stream];
      anOutlet = NXReadObject(stream);
      . . .
  }
  ```

  For example, a Matrix writes a reference to its Window object. If just the Matrix is archived, **NXReadObject()** will set the outlet that points to the Window to **nil**. However, if the entire Window is archived, the Matrix will also be archived and **NXReadObject()** will reestablish its connection to the Window.

In addition to being used inside a **write:** method, **NXWriteObject()** can be used instead of **NXWriteRootObject()** to initiate archiving. However, **NXWriteObject()** fails in this role if any object it seeks to archive calls **NXWriteObjectReference()**. Only **NXWriteRootObject()** knows how to write a reference to an object. It makes two passes over the network of objects being archived. On the first pass, it maps out the connections between objects, taking note of which ones are referred to and where. On the second pass, it archives the objects. (Therefore, all **write:** methods should be able to be invoked twice with no side effects.)

NXWriteRootObject() can be used only to initiate archiving, never inside a **write:** method.

Both **NXWriteRootObject()** and **NXWriteObject()** make sure that no object is archived more than once, no matter how may **write:** methods request that it be archived.

Final Steps

Immediately after an object has been unarchived with the **read:** method, **NXReadObject()** sends it an **awake** message. The inherited version of **awake** defined in the Object class does nothing but return **self**. But a class can define an **awake** method of its own to reinitialize its instances and make sure they're in a usable state before they receive any other messages.

Like **write:** and **read:** methods, **awake** methods should be chained together through messages to **super**.

```
- awake
{
    [super awake];
    . . .
}
```

After the **awake** message, **NXReadObject()** sends each unarchived object that can respond a **finishUnarchiving** message. This message gives Objective C programs a chance to free the unarchived object and substitute another object for it. For example, if a class of named objects requires each name to be unique, and the newly unarchived object has a name that's already in use, **finishUnarchiving** might replace the new object with the existing one.

finishUnarchiving should return **nil** if there's no substitution, and the replacement object if there is. The Object class declares a prototype for this method, but doesn't implement it.

5 Programming in Objective C

When you write a program in an object-oriented language, you're almost certainly not doing it alone. You'll be using classes developed by others and perhaps a software kit or two. A kit provides a set of mutually dependent classes that work together to structure a portion, often a substantial portion, of your program.

The NeXTSTEP development environment contains a number of software kits, including:

- The Application Kit for running an interactive and graphical user interface
- The Database Kit for operating a connection to a database server
- The 3D Graphics Kit for drawing in three dimensions
- The Sound Kit for recording, editing, and playing sounds
- The Indexing Kit™ for managing large amounts of textual data

Using a library of kit classes differs somewhat from using a library of C functions. You can pretty much pick and choose which library functions to use and when to use them depending on the program you're designing. A kit, on the other hand, imposes a design on your program (at least on the part the kit is concerned with). When you use a kit, you'll find yourself relying on library methods to do much of the work of the program. To customize the kit and adapt it to your needs, you'll implement methods that the kit will invoke at the appropriate time. These kit-designated methods are "hooks" where your own code can be introduced into the kit design. In a sense, the usual roles of program and library are reversed. Instead of incorporating library code into the program, program code is incorporated into the kit.

This chapter discusses what it's like to write an Objective C program, especially one based on a software kit. It discusses some of the programming techniques that come into play, and, as an example, explains something of how the NeXTSTEP Application Kit works. The final section takes up the question of combining Objective C with C++.

Starting Up

Objective C programs begin where C programs do, with the **main()** function. The job of an Objective C **main()** is quite simple. Its twofold task is to:

- Set up a core group of objects, and
- Turn program control over to them.

Objects in the core group might create other objects as the program runs, and those objects might create still other objects. From time to time, the program might also load classes, unarchive instances, connect to remote objects, and find other resources as they're needed. However, all that's required at the outset is enough structure (enough of the object network) to handle the program's initial tasks. **main()** puts this initial framework in place and gets it ready to go to work.

Typically, one of the core objects has responsibility for overseeing the program or controlling its input When the core structure is ready, **main()** sends this object a message to set the program in motion:

- If the program is launched from a shell and takes direction from the command line, **main()**'s message could simply tell it to begin. It might also pass along crucial command-line arguments. For example, a utility that reformats files might take a target file name as an argument on the command line. After setting up the objects the program requires, **main()** could begin the reformatting process by passing this name to one of the objects.

  (In NeXTSTEP, the **argc** and **argv** arguments passed to **main()** are stored in global variables, NXArgc and NXArgv, to make them available to all parts of the program.)

- If the program has no user interface, but exists only to run in the background and serve other applications, **main()**'s message has it begin listening for remote input. A number of services might be implemented as this kind of background process—for example, a sorting service, a service that checks for spelling errors, a utility that translates documents from one data format to another, or one that compresses and decompresses files.

- If the program is launched from the workspace, not a shell, and presents a graphical interface to the user, **main()**'s message has it begin responding to user input.

Most NeXTSTEP applications belong in this last category. They display windows and menus on-screen, and may have various kinds of buttons to click, text fields to type into, icons to drag, and other control devices to manipulate. Such an interface invites user actions on the keyboard and mouse.

The core group of objects that **main()** sets up must include some that draw the user interface, and **main()** must make sure that at least part of this interface—perhaps just a menu—is placed on-screen. (This typically is taken care of as a by-product of loading core objects from an Interface Builder archive.)

Once the initial interface is on-screen, the application will be driven not by command-line arguments or even remote messages, but by external *events*, most notably user actions on the keyboard and mouse. For example, when the user clicks a menu item, two events are generated—a mouse-down event when the mouse button is pressed and a mouse-up event when the button is released again. Similarly, typing on the keyboard generates key-down and key-up events, moving the mouse with a button pressed generates mouse-dragged events, and so on. Each event is reported to the application with a good deal of information about the circumstances of the user action—for example, which key or mouse button was pressed, where the cursor was located, and which window was affected.

An application gets an event, looks at it, responds to it, then waits for another event. It keeps getting one event after another, as long as the user produces them. From the time it's launched to the time it terminates, almost everything the application does will derive from user actions in the form of events.

The mechanism for getting and responding to events is the *main event loop* (called "main" because an application can also set up subordinate event loops for brief periods of time). One object in the core group has responsibility for running the main event loop—getting an event, generating a message that initiates the application's response to the event, then getting (or waiting for) the next event.

The Application Kit defines an object to do this work; it's an instance of the Application class and is assigned to the global variable NXApp. After creating this object (and other objects in the core framework), the **main()** function sends it a **run** message:

```
main()
{
    . . .
    NXApp = [Application new];
    . . .
    [NXApp run];
    [NXApp free];
    exit(0);
}
```

With the **run** message, **main()**'s work is essentially done. The **run** method puts the application in the main event loop and has it begin responding to events. It remains in the loop until the user quits the application.

While in the main event loop, an application can also receive input from other sources. For example, a music application might receive input from a MIDI (Musical Instrument Digital Interface) keyboard, an application that's hooked up to a telephone line might receive data over the line, and almost any application might receive remote messages from other applications. This kind of remote input is scheduled between events. In some cases (MIDI, for example), it might far outweigh user activity on the computer keyboard and mouse.

Using a Software Kit

Library functions impose few restrictions on the programs that use them; they can be called whenever they're needed. The methods in an object-oriented library, on the other hand, are tied to class definitions and can't be used unless you create an object that has access to them. The object must be connected to at least one other object in the program so that it can operate in the program network. A class defines a program component; to avail yourself of its services, you need to craft it into the structure of your application.

Nevertheless, for some classes, using a library object is pretty much the same as using a library function, though on a grander scale. You can simply create an instance, initialize it, and insert it into an awaiting slot in your application. For example, NeXTSTEP includes a HashTable class that provides a hashing service that you can take advantage of as needed.

In general, however, object-oriented libraries contain more than single classes that offer individual services. They contain kits, collections of classes that structure a problem space and present an integrated solution to it. Instead of providing services that you use as needed, a kit provides an entire program structure, a framework, that your own code must adapt to. It's a generic program model that you specialize to the requirements of your particular application. Rather than design a program that you plug library functions into, you plug your own code into the design provided by the kit.

To use a kit, you must accept the program model it defines and employ as many of its classes as necessary to implement the model in your program. The classes are mutually dependent and come as a group, not individually.

The classes in a software kit deliver their services in four ways:

- Some kit classes define "off-the-shelf" objects. You simply create instances of the class and initialize them as needed. The Matrix, ButtonCell, and TextFieldCell classes in the Application Kit are examples of this kind of class. Off-the-shelf objects are typically created and initialized using Interface Builder.

- Some kit objects are created for you behind the scenes; you don't need to allocate and initialize them. Behind-the-scenes objects are usually anonymous; a protocol specifies what messages they can respond to.

- Some kit classes are generic. The kit may provide some concrete subclasses that you can use unchanged and off-the-shelf, but you can—and in some cases are required to—define your own subclasses and complete the implementation of certain methods.

- Sometimes, a kit object is prepared to keep another object informed of its actions and even delegate certain responsibilities to it. The messages the kit object is prepared to send are declared in a protocol. If you implement the protocol in a class of your own design and register an instance of the class with the kit object, your code will be connected to the kit.

The last two items on this list—subclassing and delegation—are ways of specializing the kit design and adapting it to the needs of your program. The next sections look in more detail at these two ways of using a kit.

Inheriting from Kit Classes

A kit defines a program framework that many different kinds of applications can share. Since the framework is generic, it's not surprising that some kit classes are incomplete or abstract. A class can often do most of the work in low-level and common code, but nevertheless will require application-specific additions.

These additions are made in subclasses of the kit class. The point of defining a subclass is to fill in pieces the kit class is missing. This is done by implementing a specific set of methods. The kit designer declares these methods, sometimes in a protocol, but typically in the kit class itself. The subclass simply overrides the kit version of the methods.

Implementing Your Own Version of a Method

Most methods defined in a kit class are fully implemented; they exist so that you can invoke the services the class provides. In some cases, these methods should never be changed by the application. The kit depends on them doing just what they do—nothing more and nothing less. In other cases, the methods can be overridden, but there's no real reason to do so. The kit's version does the job. But, just as you might implement your own version of a string comparison function rather than use **strcmp()**, you can choose to override the kit method if you want to.

However, a few kit methods are designed to be overridden; they exist so that you can add specific behavior to the kit. Often, the kit-defined method will do little or nothing that's of use to your application, but will appear in messages initiated by other methods. To give content to the method, your application must implement its own version.

It's possible to distinguish four different kinds of methods that you might define in a subclass:

- Some methods are fully implemented by the kit, and are also invoked by the kit; you wouldn't invoke them in the code you write. These methods exist in the interface for just one reason—so that you can override them if you want to. They give you an opportunity to substitute your own algorithm for the one used by the kit.

  For example, the **placePrintRect:offset:** method is invoked to position an image on the printed page. The kit version of this method works fine and is rarely overridden. But if you want to do it differently, you can replace the kit version with your own.

- Some methods make object-specific decisions. The kit may implement a default version of the method that makes the decision one way, but you'll need to implement your own versions to make a different decision where appropriate. Sometimes, it's just a matter of returning YES instead of NO, or of calculating a specific value rather than the default.

  For example, Views in the Application Kit are sent **acceptsFirstResponder** messages asking, among other things, if they can display the user's typing. By default, the View method returns NO—most Views don't accept typed input. However, some View subclasses (such as the Text class) do; they have to override the method to return YES.

- Some methods must be overridden, but only to add behavior, not to alter what the kit-defined method does. When your application implements one of these methods, it's important that it incorporate the very method it overrides. This is done by messaging **super** to perform the kit-defined version of the method.

  Occasionally, the kit method will have generic code that sets up the specific work to be accomplished in the subclass version of the method, and so must be included with the code you write.

  More often, the method is one that every class is expected to contribute to. For example, Chapter 4 discussed how initialization (**init**...) methods are chained together through messages to **super**, and also how it was necessary to implement versions of the **write:** and **read:** methods to archive and unarchive instance variables declared in the class. So that a **write:** message will archive all of an object's instance variables, not just those declared in the subclass, each version of the method begins by incorporating the version it overrides.

- Some kit methods do nothing at all, except return **self**. These are methods that the kit can't define even in rudimentary form since they carry out tasks that are entirely application-specific. There's no need to incorporate the kit implementation of the method in the subclass version.

  Most methods that are to be overridden in a subclass belong in this group. It includes, among others, the principal methods you implement to draw (**drawSelf::**) and respond to events (**mouseDown:** and others). To keep the interface simple, kit classes generally try to isolate the responsibilities of their subclasses in methods unencumbered by superclass code.

It's important to note that you're not on entirely on your own when you implement your own version of a method. Subclass methods can often be built from facilities provided in the kit. An event-handling method, for example, can call on other kit methods and functions to do much of the work.

Implementing Methods You Don't Invoke

The kit methods you override in a subclass generally won't be ones that you'll invoke yourself, at least not directly. You simply implement the method and leave the rest up to the kit.

In fact, the more likely you are to write an application-specific version of a method, the less likely you are to use it in your own code. There's good reason for this. There are really only two reasons for a kit class to declare a method. Methods are provided so that you can either:

- Invoke them to avail yourself of the services the class provides, or
- Override them to introduce your own code into the kit.

If a method is one that you can invoke, it's generally fully defined by the kit and doesn't need to be redefined in your code. If the method is one that you need to implement, the kit has a particular job for it to do and so will invoke the method itself at the appropriate times.

Much of the task of programming an object-oriented application is implementing methods that you use only indirectly, through messages arranged by the kit.

Inheriting the Framework

New instances of a subclass are ready to take their place in the network of objects the kit defines. They inherit the ability to work with other objects from the kit. For example, if you define a Cell subclass, instances of the new class will be able to connect with a Matrix just like ButtonCells, TextFieldCells, and other kinds of kit-defined Cell objects.

The kit superclass defines a set of outlet connections to other objects and provides a mechanism for setting those connections, sometimes automatically. Instances of the subclass fit into the kit framework as if they were defined in the kit.

Connecting to Kit Objects

A kit framework can never be complete. It can cover much of the terrain, but it can't anticipate all the details of every application or what additional structure they'll need. Therefore, kits generally provide ways for you to hook your own objects up to kit objects.

This is usually done by implementing an object that can respond to messages declared in a protocol and registering the object with a kit object. For example, several classes in the NeXTSTEP software kits permit you to register a delegate:

```
[myWindow setDelegate:myObject];
```

Delegation is one of the principal ways that the objects you design to do the basic work of your application can be connected to a kit.

Delegation

Delegates mainly receive two kinds of messages:

- Messages that notify the delegate of what the kit object is doing
- Messages that assign some specific task to the delegate

Notification messages are easy to recognize in the interface; the methods that respond to them are typically named for the action the kit object took or is about to take, not for what the method will be implemented to do. For example, names like **browserDidScroll:** or **textWillConvert:fromFont:toFont:** are typical of notification methods in the Application Kit. The method can be implemented to do anything the application needs to keep current with the activity of the kit object.

Notifications after the fact allow the delegate to coordinate other activities with the actions of the kit object, but they don't give the delegate any control over the kit. For example, a Window object's delegate receives **windowDidResize:** messages after the user resizes the window.

Prior notifications permit the same kind of coordination, but in addition may give the delegate a chance to approve or disapprove of the impending action, or to modify it in some way. For example, a Window sends its delegate **windowWillResize:toSize:** messages as the user drags an outline of the window to resize it. The message gives the delegate a chance to constrain the size of the window.

True delegation messages, those that assign a specific responsibility to the delegate, use a different naming scheme. The methods that respond to these messages are, like most methods, named for what the method is supposed to do, not for what the kit did. For example, a kit object might send a **browser:loadCell:atRow:inColumn:** message to its delegate when it needs data to display in a browser on-screen, or an **app:openFile:type:** message when it needs help in opening a file.

Other Kit Outlets

Assigning a delegate is just one way of making a connection to a kit object. In the Application Kit, for example, you can also assign owners to Pasteboard objects and targets to control devices (button, sliders, and the like). These different kinds of connections—delegates, targets, owners, and others—name the various kinds of outlet connections that kit objects maintain to the objects you invent. Each specifies the kind of role your object will play in the kit design.

Different kinds of outlets have different responsibilities. The pasteboard's owner, for example, is responsible for providing data when it's needed for a Paste operation. A target is entrusted with carrying out the command of a control device.

Each kit defines its own group of outlets, plus the methods for setting them. Outlets often can be set graphically within Interface Builder while programming the user interface. Although there are **setDelegate:** and **setTarget:** methods, for example, delegates and targets are almost always set in Interface Builder.

Programming with the Application Kit

Instead of talking about software kits in the abstract, it's helpful to turn to a specific example—the NeXTSTEP Application Kit. Each NeXTSTEP kit covers a certain terrain. For the Application Kit, it's the user interface and the attendant tasks of drawing on the screen, printing, and organizing the interaction with the user. To show the logic of the Kit and of the framework it defines, the following sections start from the ground up. They build a rationale for the kit, then look at what tasks it takes on, how it's structured, and how your own code can be made to fit within that structure.

For this example, it's necessary to shift gears to some extent and examine, in general terms, the architecture of NeXTSTEP applications. What follows is an overview. Don't be concerned about every method or line of code that's mentioned here—there are other manuals that go into greater detail—but concentrate instead on the general concepts introduced, and especially on the division of labor between the Application Kit and the code that you would write.

NeXTSTEP Applications

The Application Kit is the basic kit used by all NeXTSTEP applications. It's designed for interactive applications that cooperate with each other in a multitasking environment and present a graphical interface to the user.

- NeXTSTEP applications cooperate by sharing the screen and other resources, and also by sharing the work. They're able to split up tasks and make use of the services provided by other applications.
- NeXTSTEP applications have a graphical interface that structures the user's interaction with the computer, making it easy and intuitive. Rather than use the command line or a multitude of function keys, NeXTSTEP applications employ software control devices—buttons, scrollers, sliders, and the like—to get information from the user. They're more likely to use graphical devices that the user can directly manipulate, like icons and scrollers, than require indirect or typed instructions.
- NeXTSTEP applications are interactive—they take instructions from the user through the keyboard and mouse and respond visually through the graphical interface. (They can also respond in other ways, but a visual component in always present.) The interaction between the user and the application is immediate and ongoing—like the give and take of a friendly conversation.

This kind of user interface must be a fundamental part of each application's design, not just something that's tacked on at the end. The more an application is able to take advantage of the graphical possibilities of the screen and the more effectively it puts the user in charge, the more successful it will be. Thus, in large part, applications need to be designed around the user interface and the two principal tasks it imposes:

- Getting user input from the keyboard and mouse, interpreting it, and responding to it without delay.
- Drawing the application's interface on the screen and continually updating the display in response to user actions.

These can be formidable tasks. While easy and intuitive for the user, an interactive graphical interface can be quite intricate from the programmer's perspective. They demand a lot of time and a lot of code.

The Application Kit is designed to reduce this effort. The Kit takes over much of the user-interface work, so your application doesn't need to be concerned with it at all. For example, it provides ready-made windows and a set of software control devices with built-in code for drawing and responding to user actions.

Of course, the Kit can't handle every user action or do all the drawing. If it did, there would be no way for you to develop an application—to put your own interpretation on input from the user or put your own output on the screen. The goal of the Kit is not to constrain what an application can do, but to set up a structure that makes it easy for it to draw and interpret user input. The Kit assumes most of the low-level burden of these tasks, while allowing you complete freedom to construct any kind of interactive application you want.

Since getting user input and providing drawing output are fundamental to the design of an application, the way the Kit structures these tasks becomes the core of a basic program structure for the entire application. This structure is built around the event cycle.

The Event Cycle

NeXTSTEP applications are controlled by the actions of the user on the keyboard and mouse. These actions are reported to the application as *events*—discrete packets of information that record what the user did along with other relevant information (such as where the cursor was located at the time and which window was affected). For example, a key-down event is generated when the user presses a key on the keyboard, and a key-up event when the key is released again. Mouse-down, mouse-dragged, and mouse-up events are generated when the user presses a mouse button, moves the cursor on-screen with the button down, and then releases the button. Each event records one atomic user action.

Almost everything the application does, it does on direct instructions from the user in the form of events. This is what's meant when it's said that an application is "driven by events"—the user drives the application. An application can choose just what events it wants to receive and just how to react to them, but the cycle of getting an event, responding to it, then getting another event is universal.

Applications invite user actions by the graphical user interface they draw on-screen. An empty text field with a blinking caret, for example, lets the user know it's appropriate to enter text. A button or menu command solicits a click. The knob of a scroller looks like something that might be grabbed and moved along the bar.

Once the user acts and an event is received, an application must draw again to keep up its end of the conversation with the user—to provide a visible response to the user's actions. It's useful, in fact, to think of applications drawing for three overlapping reasons, all of them related to the cycle of getting and responding to events. An application draws:

- To present itself to the user. At start up, an application draws its user interface on-screen, then prepares to receive events. As it runs, it continues to update its user interface in response to events.

- To provide immediate feedback that an event has been received. For example, a button is highlighted and a typed character appears in place. This feedback lets users know that the application is responsive and paying attention.

- To display the results of the user's work. The reason for using a computer is not to experience the user interface or watch buttons highlight, but to get something done. The user's work is captured as data. Only when it displays this data—whether it's an edited document, a scanned image, a graph picturing some information, or the current configuration of pieces on a game board—does the screen become a real "workspace."

Thus, at start up, an application draws its interface on the screen. The user chooses what to do and generates an event. The application responds to the event and alters the display, while the user again acts to generate another event.

This cycle of drawing and events is illustrated below:

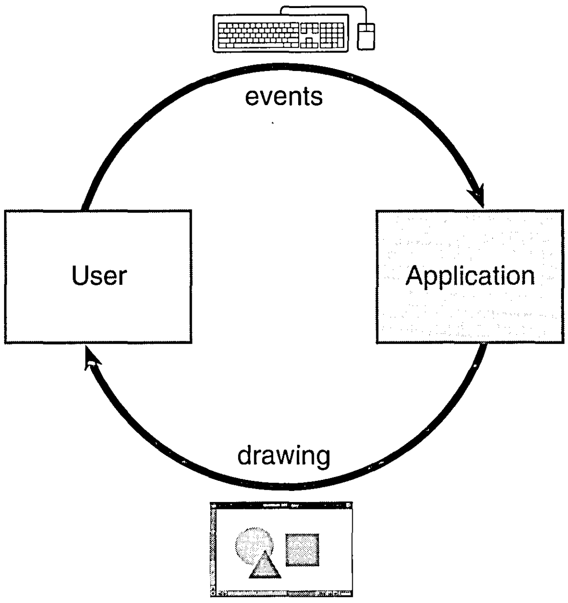


Figure 22. The Event Cycle

The Window Server

All of an application's event input and drawing output flows through a single process—the Window Server. The Server is a low-level process that runs in the background; it doesn't have or need a user interface of its own. Its primary function is to provide client applications with windows where they can draw. It can serve any number of clients; there's just one Server for all the applications running on any given machine.

At start up, every NeXTSTEP application establishes a connection to the Window Server. The application and the Server run independently of each other, but maintain a two-way communication channel. As the application runs, it requests the windows it needs, and the Server provides them. Each client application has an independent operating context within the Server and its own set of windows.

All of a client application's drawing output is confined within window boundaries; it's impossible to draw directly on-screen outside a window. So the application presents itself—its user interface—within windows, and users do their work within windows. It's natural, therefore, for windows to be the focus of user actions. Consequently, the Window Server plays a principal role for both drawing output and event input:

- It interprets the client's drawing code and renders it.
- It monitors the keyboard and mouse, and turns the user's actions on those devices into events for the intended application.

Putting the Window Server into the event cycle shown in Figure 22 above, the picture looks something like this:

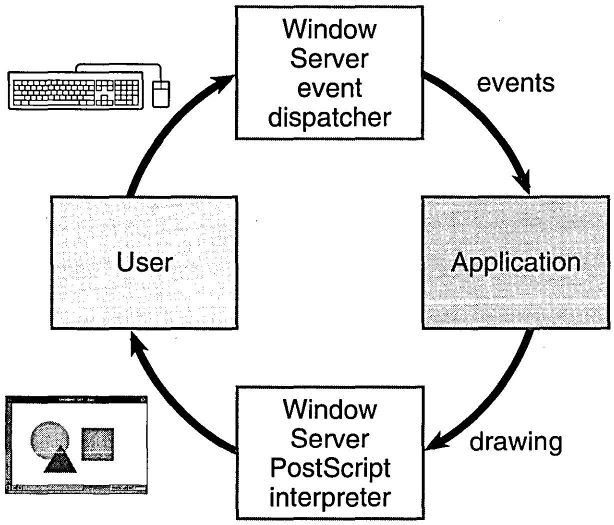

Window Server event dispatcher
events
User
Application
Window Server PostScript interpreter
drawing

Figure 23. The Window Server in the Event Cycle

Drawing instructions are typically encoded in the PostScript® language. (Three dimensional images from the 3D Graphics Kit are encoded using RenderMan®.) As the illustration above shows, the Window Server includes a PostScript interpreter that receives drawing code and renders it.

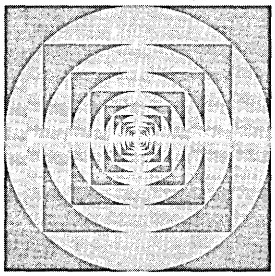
PostScript code is captured within Objective C by a set of C functions that correspond to PostScript operators. For example, this PostScript code

```
/radius 50 def
10 {
    0.333 setgray
    radius neg radius neg radius 2 mul radius 2 mul rectfill
    0.667 setgray
    0 0 radius 0 360 arc
    fill
    /radius radius 0.707 mul def
} repeat
```

can be written in C as follows:

```
float radius = 50.0;
for ( i = 0; i < 10 ; i++ ) {
    PSsetgray(0.333);
    PSrectfill(0.0-radius, 0.0-radius, radius*2.0, radius*2.0);
    PSsetgray(0.667);
    PSarc(0.0, 0.0, radius, 0.0, 360.0);
    PSfill();
    radius = 0.707 * radius;
}
```

This code draws a series of ten inset circles and squares that looks like this:



In the C version, the control loop and calculations are done in C. PostScript code is delivered to the Window Server only to produce the image. This is accomplished by the **PSsetgray()**, **PSrectfill()**, **PSarc()**, and **PSfill()** functions, corresponding to the **setgray**, **rectfill**, **arc**, and **fill** operators.

In addition to the library of single-operator "PS" functions, a utility called **pswrap** creates C functions that send definable bundles of PostScript code to the interpreter in the Window Server. The following declaration will compile to a single function, **drawSquaresAndCircles()**, that, when called with the proper argument, will draw the same figure as the code shown above.

```
defineps drawSquaresAndCircles(float initial)
    /radius initial def
    10 {
        0.333 setgray
        radius neg radius neg radius 2 mul radius 2 mul rectfill
        0.667 setgray
        0 0 radius 0 360 arc
        fill
        /radius radius 0.707 mul def
    } repeat
endps
```

This version makes the initial radius (one-half the side of the outer square) a parameter of the function. The values passed to **setgray** and the other operators could also have been parameterized to make the function more generally useful. Note that this version keeps the control loop and all calculations within the PostScript code. Generally, it's more efficient to do that kind of work in compiled C code, and just do the drawing in interpreted PostScript.

Calling either a single-operator function or a **pswrap**-generated function delivers PostScript code to the Window Server, where it will be interpreted and the image it describes rendered.

That's just what Application Kit objects do. The Kit defines objects that draw most of the user-interface devices (such as buttons, sliders, window title bars, and the like) that your application will need. There's also a Text object to draw editable text. Your application can concentrate on drawing just what's unique to it.

In addition to functions that wrap around PostScript code, the Kit has other facilities to aid in the drawing code you write. For example, NXImage objects can manage the presentation of images produced from various kinds of data.

Event Loops

After setting itself up and placing its initial user interface on-screen, an application begins the cycle of getting and responding to events. In code, the cycle is manifested as the *main event loop*. In outline form, it looks something like this:

```
BOOL running = YES;
while ( running ) {
    /* get an event */
    /* respond to it */
}
```

As long as an application is running, it stays in the main event loop and continues to get and respond to events as they're generated. It remains in the loop even when it's inactive—when the user turns to another application, for example, or takes a nap. While it waits for events, the application doesn't consume system resources or compete with other applications for processing time.

As part of its response to an event, an application can set up a subordinate event loop for a short period of time. Like the main event loop, a subordinate loop gets and responds to events, but is typically focused on only a small subset of events. The loop puts the application into a temporary mode that's broken by an appropriate user action; subordinate loops are therefore referred to as *modal event loops*.

In the NeXTSTEP user interface, modal event loops are used in only a limited number of situations:

- For attention panels. When an attention panel is on-screen, the user's actions within the application are limited. Only actions that affect the panel are permitted. The loop is broken when the user dismisses the panel. Like the main event loop, modal loops for attention panels are implemented by the Application Kit.
- For coordinating events. These are typically "spring-loaded" event loops that last only as long as the user holds down a mouse button or a key. For example, when an application gets a mouse-down event while the cursor is over a button or menu command, it sets up a temporary event loop that tracks the movement of the cursor through mouse-dragged events while waiting for a mouse-up event. When the inevitable mouse-up event is received, the loop is broken and the action of the button or menu command is performed (provided the cursor hasn't moved away).

  A coordinating event loop can also be devised to collect a number of similar events, so they can be handled together. For example, while the user is typing, an application might collect a small number of key-down events before rendering the characters on-screen. The loop is broken when the user stops generating the expected type of event (stops typing or hesitates momentarily) or enough events have been collected to handle efficiently as a group.

Each modal event loop operates inside another event loop and, ultimately, inside the main event loop. For example, a spring-loaded event loop might operate directly inside the main event loop or inside a modal event loop for an attention panel. When the inner loop is broken, the outer loop will get the next event. The main event loop isn't broken until the application terminates.

The main event loop works identically in all applications; it's common code that can be implemented by the Application Kit. The Kit also defines a number of off-the-shelf objects that run modal event loops, including attention panels and "spring-loaded" event loops that track the cursor. You can invent your own objects to do the same.

Other Input

Event loops embody the principle that "applications are driven by events." But applications aren't driven only by events, so event loops must also accommodate other kinds of input:

- To be cooperative in a multitasking environment like NeXTSTEP, applications need to communicate not just with the user, but with each other; they need to be able to respond to remote messages. For example, an application might receive a message asking it to supply some data in text form to a word processor, or it might get a message from the Workspace Manager requesting it to open another file.
- Some applications might expect input from an external device other than the keyboard or mouse, or might need to read data that accumulates in a file. For example, an application might monitor changes to a central database.
- In addition, an application might want to do something at regular intervals, and so might regard the mere passage of time as sufficient reason to take action. It needs to respond to *timed entries*, procedures that are called periodically. For example, an application could register a timed entry to animate a display, or to cause files to be saved at regular intervals.

An application responds to these other types of input between events. When it's time to get the next event, the application first checks whether any remote messages have been received, any data is waiting at a monitored port or file, or it's time to call a timed entry. If so, the next event (if one is waiting) can be postponed.

Each additional source of input has an assigned priority. The choice of what to respond to next is made by weighing the various priorities against the threshold importance of getting the next event. If an event and a remote message are both waiting for a response, the application could pick the remote message first and postpone the event, or do it the other way around and pick the event first. For example, while in a modal event loop, an application might not want to be interrupted by a remote message that it would be willing to receive between events in the main event loop. Once your application sets the priorities, the Application Kit manages this decision-making process for you.

Events are the most important input for almost all applications, both because they're generally more common than the other types of input and because they convey instructions directly from the user.

But, like events, the other types of input are also usually due to user activity; they indirectly derive from events. A remote message is typically prompted by an event in another application, data is received at monitored ports and files due to user activity elsewhere, and a timed entry is usually registered because of the events the application receives.

Application Kit Objects

The Application Kit defines objects that play critical roles in every part of the event cycle. These objects take over the elementary work of running the main event loop, managing windows, and drawing in them. They structure the event cycle and, in so doing, also structure the application.

- The Application object runs the application's connection to the Window Server and initiates its main event loop. It gets events from the Server and distributes them to the objects that will respond.
- Window objects correspond to the application's windows. Each object communicates with the Window Server to create and manage a window and responds to events that concern the window.
- View objects draw within windows. Each object controls a particular region within a window and handles events associated with that region.

Each application has just one Application object, several Windows, and many Views. The Application object keeps a list of all the Windows, and each Window organizes a set of Views. When an event is received, the Application object decides which window it matters to, and passes it to the Window object for that window. The Window decides which View it concerns and hands it to the View.

This could be the View that's displaying the current selection and handling typing within the key window, and so needs to receive the key-down events that the typing generates. Or it might be a View that drew an icon that the user clicked, and so must get the mouse-down event for the click.

When a View gets an event, it responds to it, at least in part, by altering the display—for example, by inserting new characters into the stream of text or highlighting the icon that was clicked.

An application in the event cycle looks something like this:

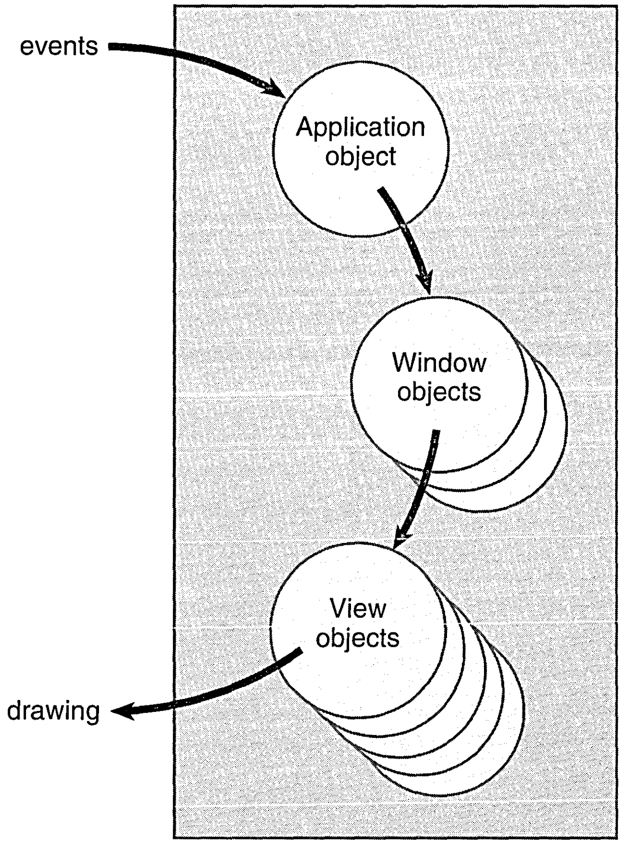


Figure 24. Inside an Application in the Event Cycle

The Application Object

Every program must have an Application object to act as its contact with the Window Server. The Application object has four principal tasks:

- It supervises the entire program, receiving events from the Server and dispatching them to the appropriate Window objects for distribution to their Views.
- It manages all the application's Windows (much as each Window object manages its Views).
- It handles changes to the application's status. It can make it the currently active application, hide and unhide it, and terminate it when the user quits.
- It keeps global information shared by other objects. So that all objects can readily take advantage of its services, it's assigned to the global variable NXApp.

The Application class is not abstract; it defines an off-the-shelf object that you can use without subclassing. To coordinate your own code with the Application object, you can assign it a delegate of your own design.

Window Objects

Every window the user sees on-screen is managed by a separate Window object. At the lowest level, windows are implemented by the Window Server. Generally, when a new Window object is created, the Server produces the window it will manage. However, to conserve memory and reduce start-up time, you can delay creating the window until it's placed on-screen; you can also arrange for the Window object to destroy the window when it's removed from the screen and get a new one when it's needed again. So it's possible for a Window object to be temporarily without a window and to be paired, at various times, with a variety of different window devices. The object corresponds to the user's conception of a window, not necessarily to its lower-level implementation.

The Window object takes care of drawing the window's title bar and frame, and it responds to user actions that move the window, miniaturize and close it. It handles all window-specific communication with the Window Server.

In addition to the Window class, the Application Kit provides a number of more specific classes that inherit from Window:

- Some, such as Panel, Menu, and PopUpList, provide the specialized form and behavior expected of panels, menus, and pop-up lists in the user interface. But their specific contents can be set by the application.
- Others, such as FontPanel and OpenPanel, have fixed contents and fulfill specific roles in the user interface.

Instances of the Window class, on the other hand, are more generic; they can be assigned any content the application requires. Typically, they're used to display the work of the user, such as text, graphics, a game board, or a form for entering information in a database.

You give content to a Window by assigning it Views. Each Window contains a set of hierarchically arranged Views; you can place a View anywhere in the hierarchy. At the top of the hierarchy is the *content view*, which fills the entire content area of the window inside the frame and title bar. Other Views are in charge of smaller areas (such as a particular text field or scroller), or larger areas (such as an entire document) that are clipped to the visible area within the window.

Like the Application object, Windows can be taken off-the-shelf and used without subclassing, though you might define a subclass of Panel (Window's subclass) to set up a particular kind of panel. To connect any Window object (including a Panel, a Menu, or a PopUpList) with application-specific code, you can assign it a delegate. Other kinds of Windows inherit the ability to have a delegate from the Window class.

Views

Views are the objects in charge of drawing and handling keyboard and mouse events. Each View owns a rectangular region associated with a particular window and is responsible for drawing one part of a window's contents. It produces images within its rectangle—typically by calling C functions that wrap around PostScript code—and responds to events associated with the images it draws. Its drawing is clipped to the rectangle.

Views adopt coordinate systems that are convenient for the drawing and event handling they do. Typically, the coordinate origin is at the lower (or upper) left corner of the View rectangle. This is illustrated in the figure below:

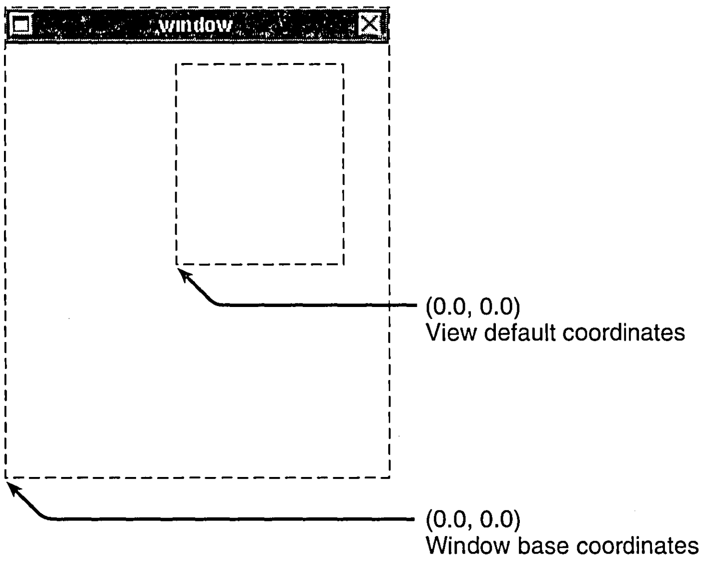


Figure 25. A View in a Window

Views can be moved and resized, much as windows can. Each View is part of a hierarchy that includes all the Views associated with the window. One View can be made up of other Views.

The View class is abstract. It implements the overall mechanism for drawing and event handling—organizing the hierarchy of Views, making sure they're in focus before they draw, getting events to the correct View, and so on—but the specific content of methods that draw and respond to events is left up to the application. They have to be implemented in subclasses.

The objects defined by View subclasses fall into three major groups:

- Views that display data and enable the user to change and manipulate it. The Views that display editable text for a word processor or images for a graphics editor fall into this group, as do the Views that display data in a spreadsheet or the game board for a chess program.
- Views that capture instructions from the user and pass them on to other objects. Views that implement control devices like buttons, sliders, and text fields belong in this group. Their job is to interpret a user action and translate it into a more specific instruction for the application.
- Views that are used in conjunction with other Views, either to enhance or to regulate the display. This group includes Views that scroll other Views, split the display into two or more resizable sections, or simply enclose other Views within a frame.

Views that belong to the first group are wholesale consumers of events; they display the direct results of the user's actions. Views in the second group act as intermediaries for actions that ultimately are intended to affect other objects. (Because of their importance to program structure, these Views are discussed in a little more detail under "Controls" below.) Views in the third group give users some control over what they see on-screen.

The Application Kit defines off-the-shelf Views in each group—the Text object in the first group, Matrix, Form, and other controls in the second, and ScrollView, ClipView, and others in the third. However, most programs need to add to these objects with their own customized View subclasses.

Unlike Windows and the Application object, Views don't have delegates (though you could implement a View subclass that had one). Since you must define a View subclass to hold application-specific code anyway, any coordinating code that would otherwise be placed in the delegate can go in the subclass instead.

Fitting into the Event Cycle

The Application, Window, and View classes set up a basic framework for handling the application's end of the event cycle. The application must fill in this framework by giving it specific drawing and event-handling content. You may also need to coordinate the activities of your own objects with the objects defined in the kit.

Handling Events

Events are delivered to Views in messages named after the event. For example, a mouse-down event is delivered as a **mouseDown:** message and a kcy-down event as a **keyDown:** message. Each message carries a pointer to a record of the event (a structure of type NXEvent).

To handle an event, a View must have a method that can respond to the *event message* that delivers it. Since each application, and each kind of View, responds to events differently, the implementation of these methods is left to View subclasses.

Methods that respond to event messages sometimes set up modal event loops. For example, to coordinate a mouse-down event with the subsequent mouse-up event, or to track the cursor while the user holds the mouse button down, a **mouseDown:** method might get mouse-dragged and mouse-up events directly from the Application object. The modal loop is broken when the mouse-up event arrives. Such a method might look something like this:

```
- mouseDown:(NXEvent *)thisEvent
{
    int             shouldLoop = YES;
    int             oldMask;
    NXEvent        *nextEvent;

    /* Make the initial response to the mouse-down event here. */
    oldMask = [window addToEventMask:NX_MOUSEDRAGGEDMASK];
    while ( shouldLoop ) {
        nextEvent = [NXApp getNextEvent:(NX_MOUSEUPMASK |
                                         NX_MOUSEDRAGGEDMASK)];
        switch ( nextEvent->type ) {
        case NX_MOUSEUP:
            shouldLoop = NO;
            break;
        case NX_MOUSEDRAGGED:
            /* Track the position of the cusor as     *
             * reported in mouse-dragged events here. */
            break;
        default:
            break;
        }
    }
    /* Respond to the mouse-up event that broke the loop here. */
    [window setEventMask:oldMask];
    return self;
}
```

Briefly, this method responds to a mouse-down event by resetting the window's event mask to include mouse-dragged events. It then sets up a subordinate event loop that looks only for mouse-dragged and mouse-up events. While waiting for the mouse-up event that will break the loop, it tracks the position of the cursor through the mouse-dragged events it receives. (All of the programming elements used in this example are defined in the Application Kit and documented in the *NeXTSTEP General Reference*.)

Such a method might respond to a click by first highlighting the image that was clicked (on the mouse-down event) then taking the required action (on the mouse-up event). Or it might continually update the position of an image the user is dragging, or highlight a range of text as the user drags over it.

The Application Kit's event-handling mechanism makes sure the event message gets to the View. All the View subclass needs to do is implement the method.

Drawing

Views draw at two different times and in two different ways. They draw proactively to present themselves to the user (that is, to present what the display within the View rectangle currently looks like) and reactively in response to events. Reactive drawing is temporary—the highlighting of a button while the cursor is over it or of text as the user drags across it. It gives immediate feedback to the user. Proactive drawing is more permanent. It can change as the result of events, of course, but it doesn't track the user's action.

For example, as the user drags to select a range of text, the amount of highlighted text changes as the cursor moves. This drawing reacts to the movement of the cursor as reported in mouse-dragged events. When the user releases the mouse button to finish the selection, the text stays highlighted. The highlighting has become an inherent part of the text display (at least until the next event), and part of what the View will proactively present when asked to display itself.

Reactive drawing happens in methods that respond to event messages, such as the **mouseDown:** method illustrated above. Proactive drawing happens as the result of display messages. A View can be asked to display its contents at any time. For example, a View that draws editable text would receive a display message when its window first comes on-screen. Later it might receive display messages when the text is scrolled, after the user makes some editing changes, or when the image is magnified.

The Application Kit's display mechanism makes sure that Views receive display messages when they need to refresh their images on-screen. However the Kit can't know what image to draw. To do the actual drawing, a display message invokes the **drawSelf::** method of each View being displayed. Every View is required to supply its own **drawSelf::** method to proactively present itself to the user.

Before sending a **drawSelf::** message to a View, the Kit brings the View into focus (makes its coordinate system the current coordinate system for drawing). The message itself passes the View two arguments specifying the regions where it's expected to draw. These regions might be smaller that the View rectangle; by paying attention to them, **drawSelf::** can avoid generating unneeded drawing instructions.

A View can draw using wrapped PostScript code, as discussed earlier under "The Window Server," or it can use the imaging facilities provided in the NXImage class, or even draw in three dimensions using the 3D Graphics Kit. The 3D Kit defines a framework for building 3D images and presenting them within a View.

A View can also divide its drawing area into small sections and put another object in charge of each section. This is exactly what a Matrix does; it uses Cell objects to draw and handle events for it. A Matrix is a kind of View, so it can take care of all external relations—positioning itself relative to other Views, getting events, setting up a coordinate system for drawing, and so on. The Cells take care of internal matters. When the Matrix receives a display message, its **drawSelf::** method sends **drawSelf:inView:** messages to the Cells. When it gets a mouse-down event, its **mouseDown:** method tracks the cursor over the Cells and informs the Cell the cursor is pointing to.

Controls

Event messages deliver the user's instructions directly to Views. However, because events are closely tied to hardware devices—the keyboard and mouse—there can be but a handful of different event types. Events alone don't give an application much information about the user's intentions.

The user interface, therefore, needs to establish well-understood paradigms for interpreting events. In text, for example, a key-down event for a backspace deletes the previous character, other key-down events insert characters into the stream of text, double-clicking selects a word, and so on.

Graphical user interfaces often borrow paradigms from the real world by emulating hardware devices in software. It's not possible (or at least not very practical) to attach a lot of control panels with buttons, knobs, sliders and other devices to the computer alongside the keyboard and mouse. But it is possible to provide those controls on the screen where they can be manipulated indirectly through mouse and keyboard events.

It's the job of software control devices to interpret events and ask other objects to respond to them. Graphically, a control presents the user with an image of a recognizable device that can be manipulated with the keyboard and mouse—a button that can be pressed, a slider with a knob that can be dragged, a text field where data can be entered. Functionally, it translates the event messages it receives into application-specific *action messages* for another object. The object that receives an action message is the control's *target.*

The Application Kit defines several standard control devices. Most are implemented as Views, others as Panels. In a Matrix, each ButtonCell or SliderCell might be thought of as a separate control, or the whole Matrix might act as an independent control device.

The Kit lets you set the target of a control,

```
   [myDevice setTarget:anObject];
```

and also the selector of the action message:

```
   [myDevice setAction:@selector(dimLights:)];
```

There's a limited set of events, but a virtually unlimited number of action messages.

In some cases, the target might be an object defined in the Kit. The Text object, for example, can respond to **copy:**, **paste:**, and **selectAll:** action messages, among others. In most cases, however, you'll need to define the target object and implement a methods to respond to the action messages it receives.

Action messages take a single argument, the **id** of the object that sends the message. If a target needs more information to accurately respond to an action message, it can send a message back to the control requesting the information it needs. For example, if a target receives a **changeName:** message from a Matrix, it can send a **stringValue** message back to the Matrix to find the name the user selected.

Assigning a target and action message to a control device gives it specific meaning. It's something like buying a generic switch at the hardware store and hooking it up to a particular piece of equipment in your home. Using kit-defined controls, you can assemble most of your application's user interface from off-the-shelf objects. The objects you design to do the basic work of your application can be connected to the user interface by being made targets of controls.

Interface Builder lets you set targets and actions graphically. You can name your own action methods while you design the user interface, then implement the methods later.

Coordinating with the Application and its Windows

Much of the activity of an application centers on the Application object, which represents the application as a whole, and on Window objects, which represent each of the windows the application uses. To coordinate other parts of the program with these objects, you can assign them a delegate and implement methods that the Application and Window classes declare in informal protocols. You can select which messages you want the delegate to receive; a message is sent only if the delegate implements a method that can respond.

Most methods implemented by Window and Application delegates respond to simple notifications. For example, when the user moves a window, the Window's delegate can be notified with a **windowDidMove:** message. Applications mostly don't care where their windows are located on-screen, but they can take note of the new location by implementing this method.

When the window becomes the key window (the window the user is about to work in and the one marked by a black title bar), the delegate is sent a **windowDidBecomeKey:** message, and when another window takes over as key window, it gets a **windowDidResignKey:** message. The Window object takes care of all the required changes when a window gains and loses key-window status (such as bringing the window to the front of the screen and highlighting and unhighlighting its title bar), but the delegate can take note of these status changes by implementing these methods.

The Application object can send its delegate a variety of messages. Many are pure notifications. For example, just after it finishes all its initialization tasks and before it enters the main event loop, the Application object sends its delegate an **appDidInit:** message. When the application becomes the active application (the one the user is about to work in), the delegate is sent an **appDidBecomeActive:** message and, if it was hidden, an **appDidUnhide:** message. When the application is hidden (its windows removed from the screen), the delegate is sent an **appDidHide:** message and an **appDidUnhide:** message when it returns to the screen. When the user quits the application, the delegate gets an **appWillTerminate:** message.

These messages give you a chance to coordinate with user actions. For example, **appDidInit:** could open an empty window for the user to work in, if the application normally opens and displays a file but the user launched it by double-clicking an application icon rather than a file icon. If quitting would destroy some of the user's work, **appWillTerminate:** could put up an attention panel reminding the user of that fact, then return **nil** to prevent the termination if the user cancels the Quit command.

Some messages to the Application object's delegate assign it specific tasks. For example, if the application receives a remote message asking it to open a particular file, the Application object will determine whether the application is able to open another file and, if it is, find the file and pass its pathname to its delegate in an **app:openFile:type:** message. The Application object can't itself open the file and display its contents in a window. Different applications store their data differently. Being generic, the Application Kit can't know about application-specific data formats and conventions. Therefore, when it receives a request to open a file, it sends its delegate an **app:openFile:type:** message. The delegate can implement this method to open the file, create a Window and a View, and display the contents of the file on-screen.

See the *NeXTSTEP General Reference* manual for more detailed information on these methods, their arguments, and return values.

Subclass Clusters

There's usually a one-to-one correspondence between concepts in the program design and class definitions. Matrices are defined by the Matrix class, windows by the Window class, and so on. However, given the basic tenets of object-oriented programming, there's no reason why a single conceptual "object" can't be implemented in more than one class. This possibility follows from the fact that:

- A class interface lies in its methods. The data structure it defines is a matter of implementation only; it should never be of much concern to anyone who uses the class.
- Different classes can have different implementations of the same method.

It shouldn't be surprising, therefore, to find a group of classes that have differing, and alternative, implementations of the same interface. Each implementation is optimized for a different set of circumstances. Instead of a program component being implemented in just one class, it's implemented in a cluster of classes.

To keep differences between classes in the cluster a matter of implementation only, it's important that they all share the same interface. Typically, all the implementing classes are grouped under an abstract class that declares their common interface. Program users can regard an instance of any class in the cluster as if it was an instance of the abstract class.

A cluster of classes can be used to optimize data storage. Suppose, for example, that your program will make extensive use of an object with a long list of instance variables. Some variables will be used frequently, but many are for specialized situations that rarely come up. Perhaps some variables declared as **double**s will most often hold small integers, but they must be declared as floating point numbers for the exceptional case.

Moreover, because there will be many instances of this kind of object in your program, you're concerned with the amount of memory they'll consume.

In this circumstance, you might save memory by implementing a cluster of classes instead of a single class. An abstract class would declare a common interface that all its subclasses would adhere to; it would declare no instance variables of its own. One subclass could declare a small subset of instance variables sufficient to handle simple cases. It would have **char**s or **short**s instead of **double**s and avoid some of the more esoteric fields that would be infrequently used. Other subclasses could be optimized for special circumstances. One subclass, of course, would have to provide the full set of instance variables that you originally contemplated. Each subclass would override superclass methods and implement them in a way that's appropriate for its particular kind of data storage.

A cluster of classes can also be used to optimize methods for different patterns of usage. For example, suppose that you want to define an interface for data storage and retrieval. The way you'd implement the methods would vary depending on the anticipated size of the data elements and the number being stored. In this circumstance, you could provide more than one implementation of the interface, optimizing each for a particular size and amount of data.

The choice of which implementation (which subclass in the cluster) to use can be handled in a variety of ways. Allocation and initialization methods in the superclass might choose the subclass that best fits the circumstances and create an instance of that class—trying always for instances that occupy as little memory as possible and that have access to the most efficient algorithms for the data.

A subclass cluster obviously works best for objects that, once initialized, are rarely or never modified. However, if an instance ever needs to change its implementation to that of another subclass (one that can store more information, for example), the change might be made almost automatically. A special "copy" method would allocate an instance of the alternative class, initialize and return it. The original instance could then be freed.

Subclass clusters take to heart the idea that what matters most about an object is its method interface, not its implementation. By keeping the same interface for an assortment of alternative implementations, they serve to simplify the overall interface and make it easier to understand. Instead of a variety of different data-storage schemes, for example, there's just one, with a variety of different implementations.

Using C++ with Objective C

The Objective C language is conceived as a set of extensions to a base language, which happens to be C. These extensions don't alter the semantics of the base language; they simply add a few syntactic constructions to give it an object-oriented capability. Other programming languages may similarly be made "Objective," as long as any syntactic conflicts between the base language and the extensions are properly handled.

NeXT has taken the logical step and added the Objective C extensions to C++. Objective C and C++ can be combined in the same source file and compiled together. With C++ as the base for the Objective C extensions, you can mix features from both languages, using whichever ones are appropriate for the task. For example, you can take advantage of Objective C's dynamic binding and still use C++'s stronger type checking and compile-time binding as needed. Modules using features from both languages can translate between modules written in "straight" C++ and Objective C code.

NeXT's primary goal in integrating the C++ and Objective C programming languages is to make it possible for you to use an existing base of C++ code with the NeXTSTEP software kits. You can also take advantage of this integration to write mixed code if you have a strong need for some features of the C++ language while programming in Objective C.

Writing Mixed Code

NeXT's C++ compiler allows C++ and Objective C expressions to be mixed in almost any manner. For example, Objective C messages can be sent within the member functions of C++ classes, and C++ member functions can be called within Objective C methods. Objects defined in either language can create and use objects defined in the other language; a C++ class can declare an **id** as a data member, while an Objective C class can declare a C++ object as an instance variable. An Objective C message can pass a C++ object as an argument, and a C++ function can take an Objective C object as an argument.

However, classes defined in each languages retain their own character. You can't define an hybrid object. Objective C messages can be sent only to Objective C receivers, and C++ member functions are called only through C++ objects. A class defined in one language can't inherit from a class defined in the other language. Moreover, operators specific to C++ can't be applied to Objective C objects. It won't work, for example, to use the C++ **new** and **delete** operators to allocate and free Objective C objects.

When mixing C++ and Objective C code, the keywords of both languages should be respected. In most cases, there's no problem since both languages derive from C and most keywords are shared. However, the C++ keyword **new** is also a common method name in Objective C. To handle this potential conflict, the compiler allows an Objective C method to have the same name as a C++ keyword and decides which is which based on context. For example, if "new" appears outside square brackets, it's treated as the C++ keyword for allocation from the free store; if it appears within square brackets in the position of a message name, it's treated as such. It's even legal to use "new" within a message expression as a C++ keyword:

```
[anObject useCPlusPlusObject:new cPlusPlusClass];
```

Since this style of code can be hard to read, it's not recommended that you use it.

Bridge Classes

A common method of mixing C++ and Objective C code is to write most modules purely in one language or the other, and then connect them by creating compound "bridge" objects—one defined as an Objective C object containing a C++ object as an instance variable, and if needed, a corresponding C++ object containing an Objective C object as a data member. The Objective C object responds to messages by calling one of the C++ object's member functions, and the C++ object implements its member functions to send messages to its Objective C counterpart. If the communication must be two-way, the C++ **class** and the Objective C **@class** declarations can be used to avoid circular references in header files.

When a project is structured this way, only the bridge classes need to contain both Objective C and C++ expressions. This reduces the potential for conflicts and makes the rest of the source code easier to read.

Suppose, for example, that a project written primarily in the Objective C language needs to use a C++ object for some of its calculations. A special Objective C class can be defined to translate Objective C messages into C++ function calls. If the Objective C object stores the C++ object as an instance variable called **calculator**, one of its methods might look like this:

```
- (int)calcInt:(int)anInt withFloat:(float)aFloat
{
    return calculator.calcIntWithFloat(anInt, aFloat);
}
```

Similarly, a C++ class could be implemented to use an Objective C object for displaying information to the user. If the C++ object stores the Objective C object as a data member called **displayer**, one of its member functions might look like this:

```
void collaborator::showResult(char *aString)
{
    [displayer showResult:aString];
    return;
}
```

Using Objective C Libraries with C++ Code

Files written in C++ that use standard C libraries must use an **extern** linkage directive when including header files for those libraries, for example:

```
extern "C" {
#include <stdio.h>
. . .
}
```

Similarly, C++ files that include Objective C header files require the "Objective-C" language to be specified (note that this form of the name is hyphenated):

```
extern "Objective-C" {
#import <objc/Object.h>
. . .
}
```

When Objective C is added to C++, the meaning of the C++ linkage directive is slightly different from its meaning in standard C++. Normally, the linkage directive merely specifies that the code in the directive's scope is to be linked according to rules defined for the named programming language; the code doesn't actually have to be written in the language specified. However, when the specified language is "Objective-C," all the code within the scope of the directive must be legal Objective C.

Run-Time Sequencing

The Objective C language uses an extensive run-time system to support its dynamic allocation, typing, binding, and loading. C++, on the other hand, needs very little run-time support. Statically declared C++ objects are initialized before the Objective C run-time system is. For this reason, C++ code should not refer to Objective C objects in static initializers or constructors. Doing so will result in launch-time errors.

A Objective C 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 run-time system. This appendix lists all the additions to the language, but doesn't go into great detail. For more information, see Chapters 2 and 3 of this manual. For a more formal presentation of Objective C syntax, see Appendix B, "Reference Manual for the Objective C Language," which follows this summary.

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:

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:

nil	A null object pointer, (**id**)0
Nil	A null class pointer, (Class)0

Preprocessor Directives

The preprocessor understands these new notations:

#import	Imports a header file. This directive is identical to **#include**, except that it won'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:

@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:

@private	Limits the scope of an instance variable to the class that declares it
@protected	Limits instance variable scope to declaring and inheriting classes
@public	Removes restrictions on the scope of instance variables

The default is **@protected**.

In addition, there are directives for these particular purposes:

@class	Declares the names of classes defined elsewhere
@selector(*method*)	Returns the compiled selector that identifies *method*
@protocol(*name*)	Returns the *name* protocol (an instance of the Protocol class)
@encode(*spec*)	Yields a character string that encodes the type structure of *spec*
@defs(*classname*)	Yields the internal data structure of *classname* instances

Classes

A new class is declared with the **@interface** directive. It imports the interface file for its superclass:

```
#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 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. It imports the interface file that declares the class:

> **#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 category definition imports its own interface:

> **#import** "*CategoryName*.**h**"
>
> **@implementation** *ClassName* (*CategoryName*)
> *method definitions*
> **@end**

Formal Protocols

Formal protocols are declared using the **@protocol** directive:

> **@protocol** *ProtocolName* < *protocol list* >
> *method declarations*
> **@end**

The list of incorporated protocols and the method declarations are optional. The protocol must import the header files that declare any protocols it incorporates.

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 above)
- In a class or category declaration, to adopt the protocol (as shown under "Classes" and "Categories" above)

- In a type specification, to limit the type to objects that conform to the protocol

Within protocol declarations, these type qualifiers support remote messaging:

oneway	The method is for asynchronous messages and has no valid return.
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.

Method Declarations

The following conventions are used in method declarations:

- A "+" precedes declarations of class methods.
- A "–" precedes declarations of instance methods.
- Arguments are declared after colons (**:**). Typically, a label describing the argument precedes the colon. Both labels and colons are considered part of the method name.
- Argument and return types are declared using the C syntax for type casting.
- The default return and argument type for methods is **id**, not **int** as it is for functions. (However, the modifier **unsigned** when used without a following type always means **unsigned int**)

Method Implementations

Each method implementation is passed two hidden arguments:

- The receiving object (**self**)
- The selector for the method (**_cmd**)

Within the implementation, both **self** and **super** refer to the receiving object. **super** replaces **self** as the receiver of a message to indicate that only methods inherited by the implementation should be performed in response to the message.

Methods with no other valid return typically return **self**.

Naming Conventions

The names of files that contain Objective C source code have a ".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.

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 variables and defined types. A program can't have a global variable with the same name as a class.

B *Reference Manual for the Objective C Language*

This appendix presents a formal grammar for the Objective C extensions to the C language—as the Objective C language is implemented for the NeXTSTEP development environment. It adds to the grammar for ANSI standard C found in Appendix A of *The C Programming Language* (second edition, 1988) by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall, and should be read in conjunction with that book.

The Objective C extensions introduce some new symbols (such as *class-interface*), but also make use of symbols (such as *function-definition*) that are explained in the standard C grammar. The symbols mentioned but not explained here are listed below:

compound statement
constant
declaration
declaration-list
enum-specifier
expression
function-definition
identifier
parameter-type-list
string
struct-declaration-list
struct-or-union
typedef-name
type-name

Of these, *identifier* and *string* are undefined terminal symbols. Objective C adds no undefined terminal symbols of its own.

Two notational conventions used here differ from those used in *The C Programming Language*:

- Literal symbols are shown in **bold** type.
- Brackets enclose optional elements and are in *italic* type. Literal brackets, like other literal symbols, are nonitalic and bold.

Otherwise, this appendix follows the conventions of the C reference manual. Each part of the grammar consists of a symbol followed by a colon and an indented list of mutually-exclusive possibilities for expanding the symbol. For example:

> *receiver:*
>   *expression*
>   *class-name*
>   **super**

However, there is an exception: Even though they're not mutually exclusive, the constituents of classes, categories, and protocols are listed on separate lines to clearly show the ordering of elements. For example:

> *protocol-declaration:*
>   **@protocol** *protocol-name*
>   *[protocol-reference-list]*
>   *[interface-declaration-list]*
>   **@end**

This exception to the general rule is easily recognized since each list terminates with **@end**.

There are just four entry points where the Objective C language modifies the rules defined for standard C:

- External declarations
- Type specifiers
- Type qualifiers
- Primary expressions

This appendix is therefore divided into four sections corresponding to these points. Where a rule in the standard C grammar is modified by an Objective C extension, the entire rule is repeated in its modified form.

External Declarations

external-declaration:
    function-definition
    declaration
    class-interface
    class-implementation
    category-interface
    category-implementation
    protocol-declaration
    class-declaration-list

class-interface:
    @interface *class-name* [**:** *superclass-name*]
        [*protocol-reference-list*]
        [*instance-variables*]
        [*interface-declaration-list*]
    @end

class-implementation:
    @implementation *class-name* [**:** *superclass-name*]
        [*instance-variables*]
        [*implementation-definition-list*]
    @end

category-interface:
    @interface *class-name* (*category-name*)
        [*protocol-reference-list*]
        [*interface-declaration-list*]
    @end

category-implementation:
    @implementation *class-name* (*category-name*)
        [*implementation-definition-list*]
    @end

protocol-declaration:
    @protocol *protocol-name*
        [*protocol-reference-list*]
        [*interface-declaration-list*]
    @end

class-declaration-list:
  @class *class-list* **;**

class-list:
  class-name
  class-list **,** *class-name*

protocol-reference-list:
  < *protocol-list* **>**

protocol-list:
  protocol-name
  protocol-list **,** *protocol-name*

class-name:
  identifier

superclass-name:
  identifier

category-name:
  identifier

protocol-name:
  identifier

instance-variables:
  { *[visibility-specification] struct-declaration-list [instance-variables]* **}**

visibility-specification:
  @private
  @protected
  @public

interface-declaration-list:
  declaration
  method-declaration
  interface-declaration-list declaration
  interface-declaration-list method-declaration

method-declaration:
  class-method-declaration
  instance-method-declaration

class-method-declaration:
  + *[method-type] method-selector* **;**

instance-method-declaration:
  – *[method-type] method-selector* **;**

implementation-definition-list:
  function-definition
  declaration
  method-definition
  implementation-definition-list function-definition
  implementation-definition-list declaration
  implementation-definition-list method-definition

method-definition:
  class-method-definition
  instance-method-definition

class-method-definition:
  + *[method-type] method-selector [declaration-list] compound-statement*

instance-method-definition:
  – *[method-type] method-selector [declaration-list] compound-statement*

method-selector:
  unary-selector
  keyword-selector [**, ...** *]*
  keyword-selector [**,** *parameter-type-list]*

unary-selector:
  selector

keyword-selector:
  keyword-declarator
  keyword-selector keyword-declarator

keyword-declarator:
  : *[method-type] identifier*
  selector **:** *[method-type] identifier*

selector:
  identifier

method-type:
  (*type-name* **)**

Type Specifiers

type-specifier:
void
char
short
int
long
float
double
signed
unsigned
id *[protocol-reference-list]*
class-name [protocol-reference-list]
struct-or-union-specifier
enum-specifier
typedef-name

struct-or-union-specifier:
struct-or-union [identifier] { *struct-declaration-list* }
struct-or-union [identifier] { **@defs** (*class-name*) }
struct-or-union identifier

Type Qualifiers

type-qualifier:
const
volatile
protocol-qualifier

protocol-qualifier:
in
out
inout
bycopy
oneway

Primary Expressions

primary-expression:
  identifier
  constant
  string
  (*expression*)
  self
  message-expression
  selector-expression
  protocol-expression
  encode-expression

message-expression:
  [*receiver* *message-selector*]

receiver:
  expression
  class-name
  super

message-selector:
  selector
  keyword-argument-list

keyword-argument-list:
  keyword-argument
  keyword-argument-list *keyword-argument*

keyword-argument:
  selector **:** *expression*
  : *expression*

selector-expression:
  @selector (*selector-name*)

selector-name:
  selector
  keyword-name-list

keyword-name-list:
  keyword-name
  keyword-name-list *keyword-name*

keyword-name:
  selector **:**
  :

protocol-expression:
  @protocol (*protocol-name*)

encode-expression:
  @encode (*type-name*)

C The Object Class

Inherits From: none *(Object is the root class)*

Declared In: /NextDeveloper/Headers/objc/Object.h

Class Description

Object is the root class of all ordinary Objective C inheritance hierarchies; it's the one class that has no superclass. From Object, other classes inherit a basic interface to the run-time system for the Objective C language. It's through Object that instances of all classes obtain their ability to behave as objects.

Among other things, the Object class provides inheriting classes with a framework for creating, initializing, freeing, copying, comparing, and archiving objects, for performing methods selected at run-time, for querying an object about its methods and its position in the inheritance hierarchy, and for forwarding messages to other objects. For example, to ask an object what class it belongs to, you'd send it a **class** message. To find out whether it implements a particular method, you'd send it a **respondsTo:** message.

The Object class is an abstract class; programs use instances of classes that inherit from Object, but never of Object itself.

Initializing an Object to Its Class

Every object is connected to the run-time system through its **isa** instance variable, inherited from the Object class. **isa** identifies the object's class; it points to a structure that's compiled from the class definition. Through **isa**, an object can find whatever information it needs at run time—such as its place in the inheritance hierarchy, the size and structure of its instance variables, and the location of the method implementations it can perform in response to messages.

Because all objects directly or indirectly inherit from the Object class, they all have this variable. The defining characteristic of an "object" is that its first instance variable is an **isa** pointer to a class structure.

The installation of the class structure—the initialization of **isa**—is one of the responsibilities of the **alloc**, **allocFromZone:**, and **new** methods, the same methods that create (allocate memory for) new instances of a class. In other words, class initialization is part of the process of creating an object; it's not left to the methods, such as **init**, that initialize individual objects with their particular characteristics.

Instance and Class Methods

Every object requires an interface to the run-time system, whether it's an instance object or a class object. For example, it should be possible to ask either an instance or a class about its position in the inheritance hierarchy or whether it can respond to a particular message.

So that this won't mean implementing every Object method twice, once as an instance method and again as a class method, the run-time system treats methods defined in the root class in a special way:

> *Instance methods defined in the root class can be performed both by instances and by class objects.*

A class object has access to class methods—those defined in the class and those inherited from the classes above it in the inheritance hierarchy—but generally not to instance methods. However, the run-time system gives all class objects access to the instance methods defined in the root class. Any class object can perform any root instance method, provided it doesn't have a class method with the same name.

For example, a class object could be sent messages to perform Object's **respondsTo:** and **perform:with:** instance methods:

```
SEL method = @selector(riskAll:);

if ( [MyClass respondsTo:method] )
    [MyClass perform:method with:self];
```

When a class object receives a message, the run-time system looks first at the receiver's repertoire of class methods. If it fails to find a class method that can respond to the message, it looks at the set of instance methods defined in the root class. If the root class has an instance method that can respond (as Object does for **respondsTo:** and **perform:with:**), the run-time system uses that implementation and the message succeeds.

Note that the only instance methods available to a class object are those defined in the root class. If MyClass in the example above had reimplemented either **respondsTo:** or **perform:with:**, those new versions of the methods would be available only to instances. The class object for MyClass could perform only the versions defined in the Object class. (Of course, if MyClass had implemented **respondsTo:** or **perform:with:** as class methods rather than instance methods, the class would perform those new versions.)

Interface Conventions

The Object class defines a number of methods that other classes are expected to override. Often, Object's default implementation simply returns **self**. Putting these "empty" methods in the Object class serves two purposes:

- It means that every object can readily respond to certain standard messages, such as **awake** or **init**, even if the response is to do nothing It's not necessary to check (using **respondsTo:**) before sending the message.

- It establishes conventions that, when followed by all classes, make object interactions more reliable. These conventions are explained in full under the method descriptions.

Sometimes a method is merely declared in the Object class; it has no implementation, not even the empty one of returning **self**. These "unimplemented" methods serve the same purpose—defining an interface convention—as Object's "empty" methods. When implemented, they enable objects to respond to system-generated messages.

Instance Variables

Class **isa**;

isa	A pointer to the instance's class structure.

Method Types

Initializing the class	+ initialize
Creating, copying, and freeing instances	
	+ alloc
	+ allocFromZone:
	+ new
	– copy
	– copyFromZone:
	– zone
	– free
	+ free
Initializing a new instance	– init
Identifying classes	+ name
	+ class
	– class
	+ superclass
	– superclass
Identifying and comparing instances	
	– isEqual:
	– hash
	– self
	– name
	– printForDebugger:
Testing inheritance relationships	
	– isKindOf:
	– isKindOfClassNamed:
	– isMemberOf:
	– isMemberOfClassNamed:

Testing class functionality	– respondsTo:
	+ instancesRespondTo:
Testing for protocol conformance	
	+ conformsTo:
	– conformsTo:
Sending messages determined at run time	
	– perform:
	– perform:with:
	– perform:with:with:
Forwarding messages	– forward::
	– performv::
Obtaining method information	– methodFor:
	+ instanceMethodFor:
	– descriptionForMethod:
	+ descriptionForInstanceMethod:
Posing	+ poseAs:
Enforcing intentions	– notImplemented:
	– subclassResponsibility:
Error handling	– doesNotRecognize:
	– error:
Dynamic loading	+ finishLoading:
	+ startUnloading
Archiving	– read:
	– write:
	– startArchiving:
	– awake
	– finishUnarchiving
	+ setVersion:
	+ version

Class Methods

alloc

+ alloc

Returns a new instance of the receiving class. The **isa** instance variable of the new object is initialized to a data structure that describes the class; memory for all other instance variables is set to 0. A version of the **init** method should be used to complete the initialization process. For example:

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

Other classes shouldn't override **alloc** to add code that initializes the new instance. Instead, class-specific versions of the **init** method should be implemented for that purpose. Versions of the **new** method can also be implemented to combine allocation and initialization.

Note: The **alloc** method doesn't invoke **allocFromZone:**. The two methods work independently.

See also: + allocFromZone:, – init, + new

allocFromZone:

+ allocFromZone:(NXZone *)*zone*

Returns a new instance of the receiving class. Memory for the new object is allocated from *zone*.

The **isa** instance variable of the new object is initialized to a data structure that describes the class; memory for its other instance variables is set to 0. A version of the **init** method should be used to complete the initialization process. For example:

```
id newObject = [[TheClass allocFromZone:someZone] init];
```

The **allocFromZone:** method shouldn't be overridden to include any initialization code. Instead, class-specific versions of the **init** method should be implemented for that purpose.

When one object creates another, it's often a good idea to make sure they're both allocated from the same region of memory. The **zone** method can be used for this purpose; it returns the zone where the receiver is located. For example:

```
id myCompanion = [[TheClass allocFromZone:[self zone]] init];
```

See also: + alloc, – zone, – init

class

+ **class**

Returns **self**. Since this is a class method, it returns the class object.

When a class is the receiver of a message, it can be referred to by name. In all other cases, the class object must be obtained through this, or a similar method. For example, here SomeClass is passed as an argument to the **isKindOf:** method:

```
BOOL test = [self isKindOf:[SomeClass class]];
```

See also: – name, – class

conformsTo:

+ (BOOL)**conformsTo:**(Protocol *)*aProtocol*

Returns YES if the receiving class conforms to *aProtocol*, and NO if it doesn't.

A class is said to "conform to" a protocol if it adopts the protocol or inherits from another class that adopts it. Protocols are adopted by listing them within angle brackets after the interface declaration. Here, for example, MyClass adopts the imaginary AffiliationRequests and Normalization protocols:

```
@interface MyClass : Object <AffiliationRequests, Normalization>
```

A class also conforms to any protocols that are incorporated in the protocols it adopts or inherits. Protocols incorporate other protocols in the same way that classes adopt them. For example, here the AffiliationRequests protocol incorporates the Joining protocol:

```
@protocol AffiliationRequests <Joining>
```

When a class adopts a protocol, it must implement all the methods the protocol declares. If it adopts a protocol that incorporates another protocol, it must also implement all the methods in the incorporated protocol or inherit those methods from a class that adopts it. In the example above, MyClass must implement the methods in the AffiliationRequests and Normalization protocols and, in addition, either inherit from a class that adopts the Joining protocol or implement the Joining methods itself.

When these conventions are followed and all the methods in adopted and incorporated protocols are in fact implemented, the **conformsTo:** test for a set of methods becomes roughly equivalent to the **respondsTo:** test for a single method.

However, **conformsTo:** judges conformance solely on the basis of the formal declarations in source code, as illustrated above. It doesn't check to see whether the methods declared in the protocol are actually implemented. It's the programmer's responsibility to see that they are.

The Protocol object required as this method's argument can be specified using the **@protocol()** directive:

```
BOOL canJoin = [MyClass conformsTo:@protocol(Joining)];
```

The Protocol class is documented in the *NeXTSTEP General Reference* manual.

See also: **– conformsTo:**

descriptionForInstanceMethod:

+ (struct objc_method_description *)
descriptionForInstanceMethod:(SEL)*aSelector*

Returns a pointer to a structure that describes the *aSelector* instance method, or NULL if the *aSelector* method can't be found. To ask the class for a description of a class method, or an instance for the description of an instance method, use the **descriptionForMethod:** instance method.

See also: **– descriptionForMethod:**

finishLoading:

+ **finishLoading:**(struct mach_header *)*header*

Implemented by subclasses to integrate the class, or a category of the class, into a running program. A **finishLoading:** message is sent immediately after the class or category has been dynamically loaded into memory, but only if the newly loaded class or category implements a method that can respond. *header* is a pointer to the structure that describes the modules that were just loaded.

Once a dynamically loaded class is used, it will also receive an **initialize** message. However, because the **finishLoading:** message is sent immediately after the class is loaded, it always precedes the **initialize** message, which is sent only when the class receives its first message from within the program.

A **finishLoading:** method is specific to the class or category where it's defined; it's not inherited by subclasses or shared with the rest of the class. Thus a class that has four categories can define a total of five **finishLoading:** methods, one in each category and one in the main class definition. The method that's performed is the one defined in the class or category just loaded.

There's no default **finishLoading:** method. The Object class declares a prototype for this method, but doesn't implement it.

See also: **+ startUnloading**

free

+ free

Returns **nil**. This method is implemented to prevent class objects, which are "owned" by the run-time system, from being accidentally freed. To free an instance, use the instance method **free**.

See also: **– free**

initialize

+ initialize

Initializes the class before it's used (before it receives its first message). The run-time system generates an **initialize** message to each class just before the class, or any class that inherits from it, is sent its first message from within the program. Each class object receives the **initialize** message just once. Superclasses receive it before subclasses do.

For example, if the first message your program sends is this,

```
[Application new]
```

the run-time system will generate these three **initialize** messages,

```
[Object initialize];
[Responder initialize];
[Application initialize];
```

since Application is a subclass of Responder and Responder is a subclass of Object. All the **initialize** messages precede the **new** message and are sent in the order of inheritance, as shown.

If your program later begins to use the Text class,

```
[Text instancesRespondTo:someSelector]
```

the run-time system will generate these additional **initialize** messages,

```
[View initialize];
[Text initialize];
```

since the Text class inherits from Object, Responder, and View. The **instancesRespondTo:** message is sent only after all these classes are initialized. Note that the **initialize** messages to Object and Responder aren't repeated; each class is initialized only once.

You can implement your own versions of **initialize** to provide class-specific initialization as needed.

Because **initialize** methods are inherited, it's possible for the same method to be invoked many times, once for the class that defines it and once for each inheriting class. To prevent code from being repeated each time the method is invoked, it can be bracketed as shown in the example below:

```
+ initialize
{
    if ( self == [MyClass class] ) {
        /* put initialization code here */
    }
    return self;
}
```

Since the run-time system sends a class just one **initialize** message, the test shown in the example above should prevent code from being invoked more than once. However, if for some reason an application also generates **initialize** messages, a more explicit test may be needed:

```
+ initialize
{
    static BOOL tooLate = NO;
    if ( !tooLate ) {
        /* put initialization code here */
        tooLate = YES;
    }
    return self;
}
```

See also: – init, – class

instanceMethodFor:

+ (IMP)**instanceMethodFor:**(SEL)*aSelector*

Locates and returns the address of the implementation of the *aSelector* instance method. An error is generated if instances of the receiver can't respond to *aSelector* messages.

This method is used to ask the class object for the implementation of an instance method. To ask the class for the implementation of a class method, use the instance method **methodFor:** instead of this one.

instanceMethodFor:, and the function pointer it returns, are subject to the same constraints as those described for **methodFor:**.

See also: – methodFor:

instancesRespondTo:

+ (BOOL)**instancesRespondTo:**(SEL)*aSelector*

Returns YES if instances of the class are capable of responding to *aSelector* messages, and NO if they're not. To ask the class whether it, rather than its instances, can respond to a particular message, use the **respondsTo:** instance method instead of **instancesRespondTo:**.

If *aSelector* messages are forwarded to other objects, instances of the class will be able to receive those messages without error even though this method returns NO.

See also: – **respondsTo:**, – **forward::**

name

+ (const char *)**name**

Returns a null-terminated string containing the name of the class. This information is often used in error messages or debugging statements.

See also: – **name**, + **class**

new

+ **new**

Creates a new instance of the receiving class, sends it an **init** message, and returns the initialized object returned by **init**.

As defined in the Object class, **new** is essentially a combination of **alloc** and **init**. Like **alloc**, it initializes the **isa** instance variable of the new object so that it points to the class data structure. It then invokes the **init** method to complete the initialization process.

Unlike **alloc**, **new** is sometimes reimplemented in subclasses to have it invoke a class-specific initialization method. If the **init** method includes arguments, they're typically reflected in the **new** method as well. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    return [[self alloc] initArg:tag arg:data];
}
```

However, there's little point in implementing a **new...** method if it's simply a shorthand for **alloc** and **init...**, like the one shown above. Often **new...** methods will do more than just allocation and initialization. In some classes, they manage a set of instances, returning the one with the requested properties if it already exists, allocating and initializing a new one only if necessary. For example:

```
+ newArg:(int)tag arg:(struct info *)data
{
    id theInstance;

    if ( theInstance = findTheObjectWithTheTag(tag) )
        return theInstance;
    return [[self alloc] initArg:tag arg:data];
}
```

Although it's appropriate to define new **new...** methods in this way, the **alloc** and **allocFromZone:** methods should never be augmented to include initialization code.

See also: – init, + alloc, + allocFromZone:

poseAs:

+ **poseAs:***aClassObject*

Causes the receiving class to "pose as" its superclass, the *aClassObject* class. The receiver takes the place of *aClassObject* in the inheritance hierarchy; all messages sent to *aClassObject* will actually be delivered to the receiver. The receiver must be defined as a subclass of *aClassObject*. It can't declare any new instance variables of its own, but it can define new methods and override methods defined in the superclass. The **poseAs:** message should be sent before any messages are sent to *aClassObject* and before any instances of *aClassObject* are created.

This facility allows you to add methods to an existing class by defining them in a subclass and having the subclass substitute for the existing class. The new method definitions will be inherited by all subclasses of the superclass. Care should be taken to ensure that this doesn't generate errors.

A subclass that poses as its superclass still inherits from the superclass. Therefore, none of the functionality of the superclass is lost in the substitution. Posing doesn't alter the definition of either class.

Posing is useful as a debugging tool, but category definitions are a less complicated and more efficient way of augmenting existing classes. Posing admits only two possibilities that are absent for categories:

- A method defined by a posing class can override any method defined by its superclass. Methods defined in categories can replace methods defined in the class proper, but they cannot reliably replace methods defined in other categories. If two categories define the same method, one of the definitions will prevail, but there's no guarantee which one.

- A method defined by a posing class can, through a message to **super**, incorporate the superclass method it overrides. A method defined in a category can replace a method defined elsewhere by the class, but it can't incorporate the method it replaces.

If successful, this method returns **self**. If not, it generates an error message and aborts.

setVersion:

+ **setVersion:**(int)*aVersion*

Sets the class version number to *aVersion*, and returns **self**. The version number is helpful when instances of the class are to be archived and reused later. The default version is 0.

See also: + version

startUnloading

+ **startUnloading**

Implemented by subclasses to prepare for the class, or a category of the class, being unloaded from a running program. A **startUnloading** message is sent immediately before the class or category is unloaded, but only if the class or category about to be unloaded implements a method that can respond.

A **startUnloading** method is specific to the class or category where it's defined; it isn't inherited by subclasses or shared with the rest of the class. Thus a class that has four categories can define a total of five **startUnloading** methods, one in each category and one in the main class definition. The method that's performed is the one defined in the class or category that will be unloaded.

There's no default **startUnloading** method. The object class declares a prototype for this method but doesn't implement it.

See also: + finishLoading:

superclass

+ **superclass**

Returns the class object for the receiver's superclass.

See also: + class, – superclass

version

+ (int)**version**

Returns the version number assigned to the class. If no version has been set, this will be 0.

See also: + **setVersion:**

Instance Methods

awake

– **awake**

Implemented by subclasses to reinitialize the receiving object after it has been unarchived (by **read:**). An **awake** message is automatically sent to every object after it has been unarchived and after all the objects it refers to are in a usable state.

The default version of the method defined here merely returns **self**.

A class can implement an **awake** method to provide for more initialization than can be done in the **read:** method. Each implementation of **awake** should limit the work it does to the scope of the class definition, and incorporate the initialization of classes farther up the inheritance hierarchy through a message to **super**. For example:

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

All implementations of **awake** should return **self**.

Note: Not all objects loaded from a nib file (created by Interface Builder) are unarchived; some are newly instantiated. Those that are unarchived receive an **awake** message, but those that are instantiated do not. See the Interface Builder documentation in *NeXTSTEP Development Tools* for more information.

See also: – **read:**, – **finishUnarchiving**, – **awakeFromNib** (NXNibNotification protocol in the Application Kit chapter of the *NeXTSTEP General Reference* manual), – **loadNibFile:owner:** (Application class in the Application Kit chapter of the *NeXTSTEP General Reference* manual)

class

– **class**

Returns the class object for the receiver's class.

See also: + **class**

conformsTo:

– (BOOL)**conformsTo:**(Protocol *)*aProtocol*

Returns YES if the class of the receiver conforms to *aProtocol*, and NO if it doesn't. This method invokes the **conformsTo:** class method to do its work. It's provided as a convenience so that you don't need to get the class object to find out whether an instance can respond to a given set of messages.

See also: + **conformsTo:**

copy

– **copy**

Returns a new instance that's an exact copy of the receiver. This method creates only one new object. If the receiver has instance variables that point to other objects, the instance variables in the copy will point to the same objects. The values of the instance variables are copied, but the objects they point to are not.

This method does its work by invoking the **copyFromZone:** method and specifying that the copy should be allocated from the same memory zone as the receiver. If a subclass implements its own **copyFromZone:** method, this **copy** method will use it to copy instances of the subclass. Therefore, a class can support copying from both methods just by implementing a class-specific version of **copyFromZone:**.

See also: – **copyFromZone:**

copyFromZone:

– **copyFromZone:**(NXZone *)*zone*

Returns a new instance that's an exact copy of the receiver. Memory for the new instance is allocated from *zone*.

This method creates only one new object. If the receiver has instance variables that point to other objects, the instance variables in the copy will point to the same objects. The values of the instance variables are copied, but the objects they point to are not.

Subclasses should implement their own versions of **copyFromZone:**, not **copy**, to define class-specific copying.

See also: – copy, – zone

descriptionForMethod:

– (struct objc_method_description *)**descriptionForMethod:**(SEL)*aSelector*

Returns a pointer to a structure that describes the *aSelector* method, or NULL if the *aSelector* method can't be found. When the receiver is an instance, *aSelector* should be an instance method; when the receiver is a class, it should be a class method.

The **objc_method_description** structure is declared in **objc/Protocol.h**, and is mostly used in the implementation of protocols. It includes two fields—the selector for the method (which will be the same as *aSelector*) and a character string encoding the method's return and argument types:

```
struct objc_method_description {
    SEL name;
    char *types;
};
```

Type information is encoded according to the conventions of the **@encode()** directive, but the string also includes information about total argument size and individual argument offsets. For example, if **descriptionForMethod:** were asked for a description of itself, it would return this string in the **types** field:

```
^{objc_method_description=:*}12@8:12:16
```

This records the fact that **descriptionForMethod:** returns a pointer ('^') to a structure ("{...}") and that it pushes a total of 12 bytes on the stack. The structure is called "objc_method_description" and it consists of a selector (':') and a character pointer ('*'). The first argument, **self**, is an object ('@') at an offset of 8 bytes from the stack pointer, the second argument, **_cmd**, is a selector (':') at an offset of 12 bytes, and the third argument, *aSelector*, is also a selector but at an offset of 16 bytes. The first two arguments—**self** for the message receiver and **_cmd** for the method selector—are passed to every method implementation but are hidden by the Objective C language.

The type codes used for methods declared in a class or category are:

Meaning	**Code**
id	'@'
Class	'#'
SEL	':'
void	'v'
char	'c'
unsigned char	'C'
short	's'
unsigned short	'S'
int	'i'
unsigned int	'I'
long	'l'
unsigned long	'L'
float	'f'
double	'd'
char *	'*'
any other pointer	'^'
an undefined type	'?'
a bitfield	'b'
begin an array	'['
end an array	']'
begin a union	'('
end a union	')'
begin a structure	'{'
end a structure	'}'

The same codes are used for methods declared in a protocol, but with these additions for type modifiers:

const	'r'
in	'n'
inout	'N'
out	'o'
bycopy	'O'
oneway	'V'

See also: + descriptionForInstanceMethod:

doesNotRecognize:

– **doesNotRecognize:**(SEL)*aSelector*

Handles *aSelector* messages that the receiver doesn't recognize. The run-time system invokes this method whenever an object receives an *aSelector* message that it can't respond to or forward. This method, in turn, invokes the **error:** method to generate an error message and abort the current process.

doesNotRecognize: messages should be sent only by the run-time system. Although they're sometimes used in program code to prevent a method from being inherited, it's better to use the **error:** method directly. For example, an Object subclass might renounce the **copy** method by reimplementing it to include an **error:** message as follows:

```
- copy
{
    [self error:"  %s objects should not be sent '%s' messages\n",
           [[self class] name], sel_getName(_cmd)];
}
```

This code prevents instances of the subclass from recognizing or forwarding **copy** messages—although the **respondsTo:** method will still report that the receiver has access to a **copy** method.

(The **_cmd** variable identifies the current selector; in the example above, it identifies the selector for the **copy** method. The **sel_getName()** function returns the method name corresponding to a selector code; in the example, it returns the name "copy".)

See also: **– error:**, **– subclassResponsibility:**, **+ name**

error:

– **error:**(const char *)*aString*, **...**

Generates a formatted error message, in the manner of **printf()**, from *aString* followed by a variable number of arguments. For example:

```
[self error:"index %d exceeds limit %d\n", index, limit];
```

The message specified by *aString* is preceded by this standard prefix (where *class* is the name of the receiver's class):

```
"error: class "
```

This method doesn't return. It calls the run-time **_error** function, which first generates the error message and then calls **abort()** to create a core file and terminate the process.

See also: **– subclassResponsibility:**, **– notImplemented:**, **– doesNotRecognize:**

finishUnarchiving

– **finishUnarchiving**

Implemented by subclasses to replace an unarchived object with a new object if necessary. A **finishUnarchiving** message is sent to every object after it has been unarchived (using **read:**) and initialized (by **awake**), but only if a method has been implemented that can respond to the message.

The **finishUnarchiving** message gives the application an opportunity to test an unarchived and initialized object to see whether it's usable, and, if not, to replace it with another object that is. This method should return **nil** if the unarchived instance (**self**) is OK; otherwise, it should free the receiver and return another object to take its place.

There's no default implementation of the **finishUnarchiving** method. The Object class declares this method, but doesn't define it.

See also: – read:, – awake, – startArchiving:

forward::

– **forward:**(SEL)*aSelector* **:**(marg_list)*argFrame*

Implemented by subclasses to forward messages to other objects. When an object is sent an *aSelector* message, and the run-time system can't find an implementation of the method for the receiving object, it sends the object a **forward::** message to give it an opportunity to delegate the message to another receiver. (If the delegated receiver can't respond to the message either, it too will be given a chance to forward it.)

The **forward::** message thus allows an object to establish relationships with other objects that will, for certain messages, act on its behalf. The forwarding object is, in a sense, able to "inherit" some of the characteristics of the object it forwards the message to.

A **forward::** message is generated only if the *aSelector* method isn't implemented by the receiving object's class or by any of the classes it inherits from.

An implementation of the **forward::** method has two tasks:

- To locate an object that can respond to the *aSelector* message. This need not be the same object for all messages.

- To send the message to that object, using the **performv::** method.

In the simple case, in which an object forwards messages to just one destination (such as the hypothetical **friend** instance variable in the example below), a **forward::** method could be as simple as this:

```
- forward:(SEL)aSelector :(marg_list)argFrame
{
    if ( [friend respondsTo:aSelector] )
        return [friend performv:aSelector :argFrame];
    [self doesNotRecognize:aSelector];
}
```

argFrame is a pointer to the arguments included in the original *aSelector* message. It's passed directly to **performv::** without change. (However, *argFrame* does not correctly capture variable arguments. Messages that include a variable argument list—for example, messages to perform Object's **error**: method—cannot be forwarded.)

The *aSelector* message will return the value returned by **forward::**. (Note in the example that **forward::** returns unchanged the value returned by **performv::**.) Since **forward::** returns a pointer, specifically an **id**, the *aSelector* method must also be one that returns a pointer (or **void**). Methods that return other types cannot be reliably forwarded.

Implementations of the **forward::** method can do more than just forward messages. **forward::** can, for example, be used to consolidate code that responds to a variety of different messages, thus avoiding the necessity of having to write a separate method for each selector. A **forward::** method might also involve several other objects in the response to a given message, rather than forward it to just one.

The default version of **forward::** implemented in the Object class simply invokes the **doesNotRecognize:** method; it doesn't forward messages. Thus, if you choose not to implement **forward::**, unrecognized messages will generate an error and cause the task to abort.

Note: If it's necessary for a **forward::** method to reason about the arguments passed in *argFrame*, it can get information about what kinds of arguments they are by calling the **method_getNumberOfArguments()**, **method_getSizeOfArguments()**, and **method_getArgumentInfo()** run-time functions. It can then examine and alter argument values with the **marg_getValue()**, **marg_getRef()**, and **marg_setValue()** macros. These functions and macros are documented in the *NeXTSTEP General Reference*.

See also: – performv::, – doesNotRecognize:

free

– **free**

Frees the memory occupied by the receiver and returns **nil**. Subsequent messages to the object will generate an error indicating that a message was sent to a freed object (provided that the freed memory hasn't been reused yet).

Subclasses must implement their own versions of **free** to deallocate any additional memory consumed by the object—such as dynamically allocated storage for data, or other objects that are tightly coupled to the freed object and are of no use without it. After performing the class-specific deallocation, the subclass method should incorporate superclass versions of **free** through a message to **super**:

```
- free {
    [companion free];
    free(privateMemory);
    vm_deallocate(task_self(), sharedMemory, memorySize);
    return [super free];
}
```

If, under special circumstances, a subclass version of **free** refuses to free the receiver, it should return **self** instead of **nil**. Object's default version of this method always frees the receiver and always returns **nil**. It calls **object_dispose()** to accomplish the deallocation.

hash

– (unsigned int)**hash**

Returns an unsigned integer that's derived from the **id** of the receiver. The integer is guaranteed to always be the same for the same **id**.

See also: – isEqual:

init

– init

Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated. An **init** message is generally coupled with an **alloc** or **allocFromZone:** message in the same line of code:

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

An object isn't ready to be used until it has been initialized. The version of the **init** method defined in the Object class does no initialization; it simply returns **self**.

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

Every class must guarantee that the **init** method returns a fully functional instance of the class. Typically this means overriding the method to add class-specific initialization code. Subclass versions of **init** need to incorporate the initialization code for the classes they inherit from, through a message to **super**:

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

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

Subclasses often add arguments to the **init** method to allow specific values to be set. The more arguments a method has, the more freedom it gives you to determine the character of initialized objects. Classes often have a set of **init...** methods, each with a different number of arguments. For example:

```
- init;
- initArg:(int)tag;
- initArg:(int)tag arg:(struct info *)data;
```

The convention is that at least one of these methods, usually the one with the most arguments, includes a message to **super** to incorporate the initialization of classes higher up the hierarchy. This method is the *designated initializer* for the class. The other **init...** methods defined in the class directly or indirectly invoke the designated initializer through messages to **self**. In this way, all **init...** methods are chained together. For example:

```
- init
{
    return [self initArg:-1];
}

- initArg:(int)tag
{
    return [self initArg:tag arg:NULL];
}

- initArg:(int)tag arg:(struct info *)data
{
    [super init. . .];
    /* class-specific initialization goes here */
}
```

In this example, the **initArg:arg:** method is the designated initializer for the class.

If a subclass does any initialization of its own, it must define its own designated initializer. This method should begin by sending a message to **super** to perform the designated initializer of its superclass. Suppose, for example, that the three methods illustrated above are defined in the B class. The C class, a subclass of B, might have this designated initializer:

```
- initArg:(int)tag arg:(struct info *)data arg:anObject
{
    [super initArg:tag arg:data];
    /* class-specific initialization goes here */
}
```

If inherited **init...** methods are to successfully initialize instances of the subclass, they must all be made to (directly or indirectly) invoke the new designated initializer. To accomplish this, the subclass is obliged to cover (override) only the designated initializer of the superclass. For example, in addition to its designated initializer, the C class would also implement this method:

```
- initArg:(int)tag arg:(struct info *)data
{
    return [self initArg:tag arg:data arg:nil];
}
```

This ensures that all three methods inherited from the B class also work for instances of the C class.

Often the designated initializer of the subclass overrides the designated initializer of the superclass. If so, the subclass need only implement the one **init...** method.

These conventions maintain a direct chain of **init...** links, and ensure that the **new** method and all inherited **init...** methods return usable, initialized objects. They also prevent the possibility of an infinite loop wherein a subclass method sends a message (to **super**) to perform a superclass method, which in turn sends a message (to **self**) to perform the subclass method.

This **init** method is the designated initializer for the Object class. Subclasses that do their own initialization should override it, as described above.

See also: + **new**, + **alloc**, + **allocFromZone:**

isEqual:

– (BOOL)**isEqual:***anObject*

Returns YES if the receiver is the same as *anObject*, and NO if it isn't. This is determined by comparing the **id** of the receiver to the **id** of *anObject*.

Subclasses may need to override this method to provide a different test of equivalence. For example, in some contexts, two objects might be said to be the same if they're both the same kind of object and they both contain the same data:

```
- (BOOL)isEqual:anObject
{
    if ( anObject == self )
        return YES;
    if ( [anObject isKindOf:[self class]] ) {
        if ( !strcmp(stringData, [anObject stringData]) )
            return YES;
    }
    return NO;
}
```

isKindOf:

– (BOOL)**isKindOf:***aClassObject*

Returns YES if the receiver is an instance of *aClassObject* or an instance of any class that inherits from *aClassObject*. Otherwise, it returns NO. For example, in this code **isKindOf:** would return YES because, in the Application Kit, the Menu class inherits from Window:

```
id  aMenu = [[Menu alloc] init];
if ( [aMenu isKindOf:[Window class]] )
    . . .
```

When the receiver is a class object, this method returns YES if *aClassObject* is the Object class, and NO otherwise.

See also: **– isMemberOf:**

isKindOfClassNamed:

– (BOOL)**isKindOfClassNamed:**(const char *)*aClassName*

Returns YES if the receiver is an instance of *aClassName* or an instance of any class that inherits from *aClassName*. This method is the same as **isKindOf:**, except it takes the class name, rather than the class **id**, as its argument.

See also: **– isMemberOfClassNamed:**

isMemberOf:

– (BOOL)**isMemberOf:***aClassObject*

Returns YES if the receiver is an instance of *aClassObject*. Otherwise, it returns NO. For example, in this code, **isMemberOf:** would return NO:

```
id  aMenu = [[Menu alloc] init];
if ([aMenu isMemberOf:[Window class]])
    . . .
```

When the receiver is a class object, this method returns NO. Class objects are not "members of" any class.

See also: **– isKindOf:**

isMemberOfClassNamed:

– (BOOL)**isMemberOfClassNamed:**(const char *)*aClassName*

Returns YES if the receiver is an instance of *aClassName*, and NO if it isn't. This method is the same as **isMemberOf:**, except it takes the class name, rather than the class **id**, as its argument.

See also: **– isKindOfClassNamed:**

methodFor:

– (IMP)**methodFor:**(SEL)*aSelector*

Locates and returns the address of the receiver's implementation of the *aSelector* method, so that it can be called as a function. If the receiver is an instance, *aSelector* should refer to an instance method; if the receiver is a class, it should refer to a class method.

aSelector must be a valid, nonNULL selector. If in doubt, use the **respondsTo:** method to check before passing the selector to **methodFor:**.

IMP is defined (in the **objc/objc.h** header file) as a pointer to a function that returns an **id** and takes a variable number of arguments (in addition to the two "hidden" arguments—**self** and **_cmd**—that are passed to every method implementation):

```
typedef id (*IMP)(id, SEL, ...);
```

This definition serves as a prototype for the function pointer that **methodFor:** returns. It's sufficient for methods that return an object and take object arguments. However, if the *aSelector* method takes different argument types or returns anything but an **id**, its function counterpart will be inadequately prototyped. Lacking a prototype, the compiler will promote **float**s to **double**s and **char**s to **int**s, which the implementation won't expect. It will therefore behave differently (and erroneously) when called as a function than when performed as a method.

To remedy this situation, it's necessary to provide your own prototype. In the example below, the declaration of the **test** variable serves to prototype the implementation of the **isEqual:** method. **test** is defined as pointer to a function that returns a BOOL and takes an **id** argument (in addition to the two "hidden" arguments). The value returned by **methodFor:** is then similarly cast to be a pointer to this same function type:

```
BOOL (*test)(id, SEL, id);
test = (BOOL (*)(id, SEL, id))[target methodFor:@selector(isEqual:)];

while ( !test(target, @selector(isEqual:), someObject) ) {
    . . .
}
```

In some cases, it might be clearer to define a type (similar to IMP) that can be used both for declaring the variable and for casting the function pointer **methodFor:** returns. The example below defines the **EqualIMP** type for just this purpose:

```
typedef BOOL (*EqualIMP)(id, SEL, id);
EqualIMP test;
test = (EqualIMP)[target methodFor:@selector(isEqual:)];

while ( !test(target, @selector(isEqual:), someObject) ) {
    . . .
}
```

Either way, it's important to cast **methodFor:**'s return value to the appropriate function type. It's not sufficient to simply call the function returned by **methodFor:** and cast the result of that call to the desired type. This can result in errors.

Note that turning a method into a function by obtaining the address of its implementation "unhides" the **self** and **_cmd** arguments.

See also: + **instanceMethodFor:**

name

– (const char *)**name**

Implemented by subclasses to return a name associated with the receiver.

By default, the string returned contains the name of the receiver's class. However, this method is commonly overridden to return a more object-specific name. You should therefore not rely on it to return the name of the class. To get the name of the class, use the class **name** method instead:

```
const char *classname = [[self class] name];
```

See also: + **name**, + **class**

notImplemented:

– **notImplemented:**(SEL)*aSelector*

Used in the body of a method definition to indicate that the programmer intended to implement the method, but left it as a stub for the time being. *aSelector* is the selector for the unimplemented method; **notImplemented:** messages are sent to **self**. For example:

```
- methodNeeded
{
    [self notImplemented:_cmd];
}
```

When a **methodNeeded** message is received, **notImplemented:** will invoke the **error:** method to generate an appropriate error message and abort the process. (In this example, **_cmd** refers to the **methodNeeded** selector.)

See also: – **subclassResponsibility:**, – **error:**

perform:

– perform:(SEL)*aSelector*

Sends an *aSelector* message to the receiver and returns the result of the message. This is equivalent to sending an *aSelector* message directly to the receiver. For example, all three of the following messages do the same thing:

```
id myClone = [anObject copy];
id myClone = [anObject perform:@selector(copy)];
id myClone = [anObject perform:sel_getUid("copy")];
```

However, the **perform:** method allows you to send messages that aren't determined until run time. A variable selector can be passed as the argument:

```
SEL myMethod = findTheAppropriateSelectorForTheCurrentSituation();
[anObject perform:myMethod];
```

aSelector should identify a method that takes no arguments. If the method returns anything but an object, the return must be cast to the correct type. For example:

```
char *myClass;
myClass = (char *)[anObject perform:@selector(name)];
```

Casting generally works for pointers and for integral types that are the same size as pointers (such as **int** and **enum**). Whether it works for other integral types (such as **char**, **short**, or **long**) is machine dependent. Casting doesn't work if the return is a floating type (**float** or **double**) or a structure or union. This is because the C language doesn't permit a pointer (like **id**) to be cast to these types.

Therefore, **perform:** shouldn't be asked to perform any method that returns a floating type, structure, or union, and should be used very cautiously with methods that return integral types. An alternative is to get the address of the method implementation (using **methodFor:**) and call it as a function. For example:

```
SEL aSelector = @selector(backgroundGray);
float aGray = ( (float (*)(id, SEL))
                [anObject methodFor:aSelector] )(anObject, aSelector);
```

See also: – perform:with:, – perform:with:with:, – methodFor:

perform:with:

– **perform:**(SEL)*aSelector* **with:***anObject*

Sends an *aSelector* message to the receiver with *anObject* as an argument. This method is the same as **perform:**, except that you can supply an argument for the *aSelector* message. *aSelector* should identify a method that takes a single argument of type **id**.

See also: – **perform:**, – **perform:with:afterDelay:cancelPrevious:** (Application Kit chapter of the *NeXTSTEP General Reference* manual)

perform:with:with:

– **perform:**(SEL)*aSelector*
  with:*anObject*
  with:*anotherObject*

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. This method is the same as **perform:**, except that you can supply two arguments for the *aSelector* message. *aSelector* should identify a method that can take two arguments of type **id**.

See also: – **perform:**

performv::

– **performv:**(SEL)*aSelector* **:**(marg_list)*argFrame*

Sends the receiver an *aSelector* message with the arguments in *argFrame*. **performv::** messages are used within implementations of the **forward::** method. Both arguments, *aSelector* and *argFrame*, are identical to the arguments the run-time system passes to **forward::**. They can be taken directly from that method and passed through without change to **performv::**.

performv:: should be restricted to implementations of the **forward::** method. Because it doesn't restrict the number of arguments in the *aSelector* message or their type, it may seem like a more flexible way of sending messages than **perform:**, **perform:with:**, or **perform:with:with:**. However, it's not an appropriate substitute for those methods. First, it's more expensive than they are. The run-time system must parse the arguments in *argFrame* based on information stored for *aSelector*. Second, in future releases, **performv::** may not work in contexts other than the **forward::** method.

See also: – **forward::**, – **perform:**

printForDebugger:

– (void)**printForDebugger:**(NXStream *)*stream*

Implemented by subclasses to write a useful description of the receiver to *stream*. Object's default version of this method provides the class name and the hexadecimal address of the receiver, formatted as follows:

<*classname*: 0x*address*>

Debuggers can use this method to ask objects to identify themselves.

read:

– **read:**(NXTypedStream *)*stream*

Implemented by subclasses to read the receiver's instance variables from the typed stream *stream*. You need to implement a **read:** method for any class you create, if you want its instances (or instance of classes that inherit from it) to be archivable.

The method you implement should unarchive the instance variables defined in the class in a manner that matches they way they were archived by **write:**. In each class, the **read:** method should begin with a message to **super**:

```
- read:(NXTypedStream *)stream
{
    [super read:stream];
    /* class-specific code goes here */
    return self;
}
```

This ensures that all inherited instance variables will also be unarchived.

All implementations of the **read:** method should return **self**.

After an object has been read, it's sent an **awake** message so that it can reinitialize itself, and may also be sent a **finishUnarchiving** message.

See also: – awake, – finishUnarchiving, – write:

respondsTo:

– (BOOL)**respondsTo:**(SEL)*aSelector*

Returns YES if the receiver implements or inherits a method that can respond to *aSelector* messages, and NO if it doesn't. The application is responsible for determining whether a NO response should be considered an error.

Note that if the receiver is able to forward *aSelector* messages to another object, it will be able to respond to the message, albeit indirectly, even though this method returns NO.

See also: – **forward::**, + **instancesRespondTo:**

self

– **self**

Returns the receiver.

See also: + **class**

startArchiving:

– **startArchiving:**(NXTypedStream *)*stream*

Implemented by subclasses to prepare an object for being archived—that is, for being written to the typed stream *stream*. A **startArchiving:** message is sent to an object just before it's archived—but only if it implements a method that can respond. The message gives the object an opportunity to do anything necessary to get itself, or the stream, ready before a **write:** message begins the archiving process.

There's no default implementation of the **startArchiving:** method. The Object class declares the method, but doesn't define it.

See also: – **awake**, – **finishUnarchiving**, – **write:**

subclassResponsibility:

– **subclassResponsibility:**(SEL)*aSelector*

Used in an abstract class to indicate that its subclasses are expected to implement *aSelector* methods. If a subclass fails to implement the method, it will inherit it from the abstract superclass. That version of the method generates an error when it's invoked. To avoid the error, subclasses must override the superclass method.

For example, if subclasses are expected to implement **doSomething** methods, the superclass would define the method this way:

```
- doSomething
{
    [self subclassResponsibility:_cmd];
}
```

When this version of **doSomething** is invoked, **subclassResponsibility:** will—by in turn invoking Object's **error:** method—abort the process and generate an appropriate error message.

(The **_cmd** variable identifies the current method selector, just as **self** identifies the current receiver. In the example above, it identifies the selector for the **doSomething** method.)

Subclass implementations of the *aSelector* method shouldn't include messages to **super** to incorporate the superclass version. If they do, they'll also generate an error.

See also: – doesNotRecognize:, – notImplemented:, – error:

superclass

– **superclass**

Returns the class object for the receiver's superclass.

See also: + superclass

write:

– **write:**(NXTypedStream *)*stream*

Implemented by subclasses to write the receiver's instance variables to the typed stream *stream*. You need to implement a **write:** method for any class you create, if you want to be able to archive its instances (or instances of classes that inherit from it).

The method you implement should archive only the instance variables defined in the class, but should begin with a message to **super** so that all inherited instance variables will also be archived:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    /* class-specific archiving code goes here */
    return self;
}
```

All implementations of the **write:** method should return **self.**

During the archiving process, **write:** methods may be performed twice, so they shouldn't do anything other than write instance variables to a typed stream.

See also: – read:, – startArchiving:

zone

– (NXZone *)**zone**

Returns a pointer to the zone from which the receiver was allocated. Objects created without specifying a zone are allocated from the default zone, which is returned by **NXDefaultMallocZone()**.

See also: + allocFromZone:, + alloc, + copyFromZone:

Suggested Reading on Object-Oriented Programming

Designing Object-Oriented Software. Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. Prentice Hall, 1990.

An Introduction to Object-Oriented Programming. Timothy A. Budd. Addison-Wesley, 1991.

Object Orientation: Concepts, Languages, Databases, User Interfaces. Setrag Khoshafian and Razmik Abnous. John Wiley and Sons, 1990.

Object-Oriented Design: with Applications. Grady Booch. Benjamin/Cummings, 1991.

Object-Oriented Software Construction. Bertrand Meyer. Prentice Hall International Series in Computer Science, 1988.

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.*

action message
In the Application Kit, a message sent by an object (such as a Button or Slider) in response to a user action (such as clicking the button or dragging the slider's knob). The message translates the user's action into a specific instruction for the application. See also *target.*

active application
The application associated with keyboard events, the one the user is currently working in. Menus are visible on-screen only for the active application, and only the active application can have the current key window.

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
The Objective C classes and C functions available for implementing the NeXTSTEP window-based user interface in an application. 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. Archiving involves writing data to a special kind of data stream, called a typed stream. See also *typed stream.*

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.*

attention panel
A panel that demands the user's attention. Until the user acts to dismiss the panel from the screen, only actions affecting the panel are permitted. Attention panels permit the user to rescind a command (such as Close), ask the user to complete a command (such as Save As), and give warnings that the user must acknowledge. See also *panel.*

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 be used by the class object rather than by 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.

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 adopts the protocol or inherits from a class that adopts it. 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 View object that's associated with the content area of a window—all the area in the window excluding the title bar, resize bar, and border. All other Views in the window are arranged in a hierarchy beneath the content view.

controls
Graphical objects—such as buttons, sliders, text fields, and scrollers—that the user can operate to give instructions to an application.

cursor
The small image (usually an arrow) that moves on the screen and is controlled by moving the mouse.

delegate
In the NeXTSTEP software kits, an object that acts on behalf of another object. Window, Application, Text, Listener, NXBrowser, NXImage, and other objects can be assigned delegates.

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.

dynamic binding
Binding a method to a message—that is, finding the method implementation to invoke in response to the message—at run time, rather than at compile time.

dynamic typing
Discovering the class of an object at run time rather than at compile time.

event
The direct or indirect report of external activity, especially user activity on the keyboard and mouse.

event message
In the Application Kit, a message to perform a method named after an event or subevent. Event messages are used to dispatch events to the objects that will respond to them. See also *action message.*

factory
Same as *class object.*

factory method
Same as *class method.*

factory object
Same as *class object.*

file package
A directory that the Workspace Manager presents as a file, allowing the user to manipulate a group of files as if they were one file. A file package for an application executable has the same name as the executable file, plus a ".app" extension. File packages for documents and bundles bear an extension that's recognized as belonging to a particular application.

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 run time when asked if they conform to a formal protocol, and instances can be typed by the formal protocols they conform to.

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.

informal protocol
In the Objective C language, a protocol declared as a category, usually as a category of the Object 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 Object, which is at the root of the hierarchy) 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 run time 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 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. The title bar of the key window is highlighted in black.

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" as the last item in the main 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 Server and responds to them, waiting between events if the next event isn't ready. In the Application Kit, the Application 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.

modal event loop
A temporary event loop that's set up to get events directly from the event queue, bypassing the main event loop. Typically, a mouse-down event initiates the modal loop and the following mouse-up event ends it. The loop gets mouse-dragged events (or mouse-entered and mouse-exited events) to track the cursor's movement while the user holds the mouse button down.

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.

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

NeXTSTEP
NeXT's application development and user environment, consisting of the Workspace Manager, the Window Server, various software kits such as the Application Kit and the Database Kit, various applications such as Project Builder and Interface Builder, and other software.

nib file
A file (actually a file package) that stores the specifications for all or part of an application's interface. Nib files are created using Interface Builder and can contain archived objects, information about connections between objects, and sound and image data.

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.

panel
A window that holds objects that control what happens in other windows (such as a Font panel) or in the application generally (such as a Preferences panel), or a window that presents information about the application to the user (such as an information panel). See also *attention panel.*

polymorphism
In object-oriented programming, the ability of different objects to respond, each in its own way, to the same message.

pop-up list
A menu-like list of items that appears over (or next to) an on-screen button when the button is pressed. The user can choose an item by dragging to it and releasing the mouse button. When the mouse button is released, the pop-up list disappears.

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.

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.

run time
The time after a program is launched and while it's running. Decisions made at run time 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 acknowledgement or return information from the receiving application, the two applications are kept "in sync." See also *asynchronous message.*

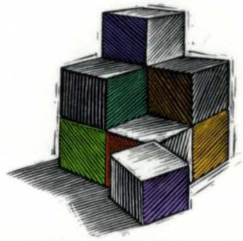
target
In the Application Kit, the object that receives action messages from a Control.

typed stream
A specialized data stream used for archiving. When a typed stream is used, the type of the data is archived along with the data and an object's class hierarchy and version are archived with the object. See also *archiving.*

Window Server
A process that dispatches user events to applications and renders PostScript code on behalf of applications.

zone
A particular region of dynamic memory. Zones are set up in program code and are passed to allocation methods and functions to specify that the allocated memory should come from a particular zone. Allocating related data structures from the same zone can improve locality of reference and overall system performance. For example, all the Views that are displayed in the same window might be clustered in the same zone.

Index



NEXTSTEP OBJECT-ORIENTED PROGRAMMING AND THE OBJECTIVE C LANGUAGE: RELEASE 3

NeXTSTEP is the object-oriented programming environment that speeds the development of all kinds of software—from mission-critical custom applications for business to advanced research projects for academia. NeXTSTEP offers building blocks that implement essential behavior in a variety of application areas—including database management, telecommunications and networking, and high-quality 2D and 3D graphics.

NeXTSTEP Object-Oriented Programming and the Objective C Language describes the Objective C language as it is implemented for NeXTSTEP. The Objective C language adds a small amount of syntax to standard ANSI C to support object-oriented programming techniques—including full dynamic binding, message forwarding, and remote messaging between objects in different applications. Standard C and C++ can be freely mixed with Objective C code.

Topics covered in this book include:

- The principles of object-oriented programming
- The Objective C language
- The run-time system for the Objective C language
- Using Objective C with C++

NEXTSTEP

Object Oriented Software

52495

9 780201 632514

ISBN 0-201-63251-9

US $24.95
CANADA $31.95

Addison-Wesley Publishing Company