MACINTOSH® PROGRAMMING SERIES

DISK INCLUDED

OBJECT-ORIENTED PROGRAMMING POWER

for THINK Pascal™ Programmers

- Creating and Using Object Class Hierarchies
- Using a MacApp-like Application Framework
- Building Reusable Software Components

Microsoft PRESS®

CHUCK SPHAR

OBJECT-ORIENTED PROGRAMMING POWER

for THINK Pascal™ Programmers

OBJECT-ORIENTED PROGRAMMING POWER

for THINK Pascal™ Programmers

CHUCK SPHAR

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data

Sphar, Chuck, 1947–
Object-oriented programming power for THINK Pascal programmers /
Chuck Sphar.
p. cm.
Includes index.
ISBN 1-55615-348-1
1. Think Pascal. 2. Macintosh (Computer)--Programming. 3. Object
-oriented programming (Computer science) I. Title.
QA76.8.M3S67 1991
005.265--dc20 91-16715
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 AGAG 6 5 4 3 2 1

Distributed to the book trade in Canada by Macmillan of Canada, a division of
Canada Publishing Corporation.

Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.

Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

British Cataloging-in-Publication Data available.

Acquisitions Editor: Dean Holmes
Project Editor: Erin O'Connor
Technical Editor: Mary DeJong

For

Pam, Max, and the Cats

CONTENTS

ACKNOWLEDGMENTS

Thanks first to Gary Little and his compatriots at Apple Computer. Gary's advice, his moral support, and especially his putting me in contact with the best agent in the business made the back end of the process much easier.

Thanks next to Claudette Moore, my agent, whose tireless efforts not only got the manuscript into Microsoft Press's hands and made publication a reality but also made the book stronger, cleaner, and more effective. Claudette's advice made all the difference.

The technical support people at Symantec, especially Philip Borenstein and Dave Alcott, helped with a number of technical issues.

My list of those to thank at Microsoft Press is long and full of extraordinary professionals who care about their craft. It starts with Dean Holmes, my acquisitions editor, who had the faith and the courage to take on a book that wasn't written about some massively popular DOS product. The book's final quality and appearance come in great measure from the work of art designer Kim Eggleston; word processors Debbie Kem and Judith Bloch; typographers Lisa Iversen and Rodney Cook; layout artist Peggy Herman; and proofreaders Cynthia Riskin, Kathleen Atkins, and Alice Copp Smith. I'd also like to thank the researchers at Microsoft Library, who tracked down a great many bibliographical details.

Most of all, my appreciation goes to my technical editor, Mary DeJong, and my manuscript editor, Erin O'Connor. Mary caught my program bugs, made many invaluable suggestions, and taught me to eat Thai food. Finally, and above all, thanks to Erin, whose wit and love of what she does made the editorial process an actual joy. These are really my coauthors and my friends.

Then thanks also to my friends and colleagues at White Sands Ground Terminal, who cheered me on when I left them to write and who read the ever-changing versions of the manuscript for me. Thanks, Aaron Goodman, Robin Wells, Terry Stambaugh, Bill Shaeffer, Paul Berver, and Toni Giacalone. Thanks also to Kay Patton, Jo Ann Hoffman, and other TDRSS people for their support. Thanks as well to Gary Willis, fellow seeker in the quest for publication. His encouragement and advice meant a lot to me.

Thanks too to my family, especially Max, for a lifetime of encouragement. But thanks most of all to Pam, for assistance, encouragement, reading, encouragement, testing, encouragement, and earning a living while I wasn't. How could I have done it without you?

Chuck Sphar
Las Cruces, New Mexico

INTRODUCTION

If you do any programming at all, you've no doubt paid some attention to the hoopla about object-oriented programming, or OOP as it's usually called.

There are lots of ways to do OOP, of course, and you can program with objects in a variety of languages, from Smalltalk to THINK C to MPW C++ to Prograph. This book-disk package guides you into doing OOP in Pascal—that is, in a modified form of Pascal, called Object Pascal, that preserves every capability you already have in Pascal while adding a parsimonious and rather elegant set of extensions that permit you to do object-oriented programming.

So welcome to OOP. And to Object Pascal. And to this book.

Is This Book for You?

If you program the Macintosh in Pascal and you're intrigued by what you've been hearing about object-oriented programming, this book is for you. It will deliver what you want to know—in your native language. If you've heard about Apple's MacApp application framework and you're interested in learning to write applications with it, this book will teach you the fundamentals of object-oriented programming on which MacApp is based. The book doesn't teach MacApp, but it does teach you the skills you need to get into MacApp. You don't have to be interested in MacApp to benefit from this book, but you probably will be by the time you finish it.

What Equipment Will You Need?

You'll need a Macintosh with 1 megabyte of RAM or more—a Macintosh Plus, Classic, or LC, or any member of the Macintosh SE and Macintosh II lines. You'll need Apple's MPW Pascal 2.0, TML Systems' TML Pascal 2.0 (now discontinued), or Symantec's THINK Pascal 2.0 (formerly Lightspeed)—or later versions. To use any of these Pascals effectively, you'll need a hard disk. See Appendix A for a discussion of the relative merits of these Pascals and for buying information. I've written the book's code with THINK Pascal 3.0.

What Should You Already Know?

At a minimum, you should know Pascal. There's too little space in this book to teach object-oriented programming and Pascal.

And the more you know about Macintosh programming, the better, but we'll do relatively little that requires a deep knowledge of the Mac's User Interface Toolbox. We'll focus on OOP, not on patching traps, dragging gray regions, and using the Notification Manager. Many of our objects will use Toolbox calls, and I'll refer you at times to *Inside Macintosh* (Apple Computer, Incorporated 1985–). You might also want to consult Stephen Chernicoff's *Macintosh Revealed* (Chernicoff 1984–1990). But you don't have to be an ace Mac hacker to make it through this book.

You'll find it helpful to know your way around in one of the Object Pascal systems. I've written the examples in the book with THINK Pascal, but they should be readily usable in the Macintosh Programmer's Workshop, with either MPW Pascal or TML Pascal. In Appendix A you'll find some suggestions on porting the examples to those environments.

What Will You Get?

From this book-disk package you'll get

- A painless introduction to object-oriented programming concepts and techniques, as implemented in Object Pascal
- Instruction in designing applications—some partially and some highly object-oriented
- A gentle initiation into the concept of application frameworks as exemplified by MacApp and the THINK Class Library for Pascal
- A demonstration of using an application framework, featuring PicoApp (peeko-app), the book's own tiny framework
- Reusable software components that you can plug into any application—and encouragement to develop more of them

And you'll get lots of sample object-oriented code. You'll find demonstrations and test programs for most of the chapters, including several sizable applications, on the disk that comes with this book.

How Will You Get It?

In Part 1, we'll go into the fundamentals of OOP and Object Pascal. In Part 2, we'll develop two sample OOP applications that illustrate many OOP techniques and the fundamentals of application frameworks like MacApp. In Part 3, we'll delve into the mysteries of polymorphism through an OOP software component—a highly general, reusable linked list. Along the way, we'll look at many object examples and several small OOP programs.

Part 1. About OOP

Part 1 is an introduction for Pascal programmers who want to learn OOP and a source, in its later chapters, of considerable technical information for programmers who already have a good grounding in OOP concepts.

If you're completely unfamiliar with object-oriented programming, read Chapters 1 and 2, in which an overview and an OOP tutorial touch on all the fundamental concepts, using Object Pascal examples. Spend some time with Chapters 3, 4, 5, and 6, where accessible example programs introduce the syntax of Object Pascal and the techniques of OOP. Study the code, and try compiling the examples yourself. All the files you'll need come with either your Pascal compiler or the disk that accompanies this book.

If you have a pretty good grasp of the basic concepts of OOP, you can skim Part 1's first six chapters and move on to Chapters 7, 8, and 9, more technical chapters that take you deeper into OOP and Object Pascal. Read Chapters 7, 8, and 9, and use them later for reference.

Part 2. Object-Oriented Applications

Part 2 will teach you some fundamentals of object-oriented application design and show you how application frameworks operate. We'll design and implement some highly object-oriented application programs—a simple game called Crapgame, based on the popular dice game, and a simple drawing demonstration called PicoSketch. From Crapgame, we'll derive and build PicoApp, our small application framework modeled on Apple's MacApp.

After we design Crapgame, we'll "abstract out" all the standard Macintosh user interface parts, separating the user interface from the game. Then, to create PicoApp, our down-sized version of MacApp, we'll turn the components of the abstracted user interface into objects: an application object, a document object, a window object, view objects, and others.

Then we'll rebuild Crapgame by adding the game components to the PicoApp framework as objects. PicoApp, which handles menus and mouse clicks, updates the cursor, draws the window, and does many other standard chores, passes commands to the game objects. Because PicoApp does so many of the hard standard chores for you, creating an application like Crapgame becomes quite easy. Combining the PicoApp framework and the Crapgame objects gives you a complete application.

We'll round out Part 2 by developing PicoSketch, our second small "PicoApplication." The development of PicoSketch demonstrates one way to engineer mouse-driven drawing, as in a paint program.

By the end of Part 2, you'll have seen a range of ways to use objects to build a Macintosh application, including ways to derive your application from a generic application framework. You'll understand the fundamental structure of frameworks like

MacApp and the THINK Class Library (TCL), and you'll know the basic techniques for using them. This doesn't mean that you'll then know how to program in MacApp or TCL, but you'll find learning to use MacApp or TCL a much easier task after you've read Part 2.

Part 3. Polymorphic Software Components

Part 3 emphasizes polymorphism and working with class hierarchies. Its chief example is a fairly complex and complete reusable software component, a list capable of holding any object. We'll look at the role of such components in class libraries and applications and at a number of other, less thoroughly developed, components.

The list software component we develop at length in Part 3 is designed to hold almost any kind of object. That makes it polymorphic, or generic. The list also illustrates the concept of encapsulation by hiding details from the portion of the program that uses the component. A programmer using the list in a program knows what the list does but not how it does it. In much of Part 3, we'll explore the uses and difficulties of polymorphism and encapsulation. In particular, we'll look at problems such as searching a general list that could contain anything and whose innards are invisible to us.

In the rest of Part 3, we'll move more quickly through several other reusable software components in order to get a sense of the many things we can do with them. We'll look at a highly general binary search tree object and at array objects that can help solve Pascal's problems with character-array parameters of varying lengths. Finally, we'll briefly look at the idea of input/output stream objects that simplify file and other kinds of I/O and let you store objects in files. And we'll look at several kinds of objects used to "filter" streams to alter their input or output.

Projects

At the end of most chapters, I'll suggest projects you can do that relate to what we've covered in the chapter. The projects range from the simple, especially early in the book, through the more challenging.

Try at least some of the projects as a way to reinforce what you've learned in the chapters and to begin building your own skills as an object-oriented programmer.

Most of the projects are open-ended enough to let you be creative and experimental. Except in a few cases, answers are not provided.

Appendixes

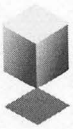
The THINK Pascal 3.0 examples we use throughout this book require slight changes for compiling under Apple's MPW using MPW Pascal or TML Pascal. Changes might also be required for compiling with earlier versions of THINK Pascal. Appendix A, on porting THINK Pascal programs to other Pascals, goes into the compiler directives, segmentation directives, set constants, L-value function calls, and other program features you might have to change to get our example programs to compile and run under these products. We'll look briefly at other languages and even compilers for other machines in this context. Appendix A also tells you where you can buy the object-oriented Pascal compilers I talk about in the book.

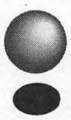
The annotated bibliography in Appendix B tells you about some of the books and articles I found most helpful as I wrote this book. In the text, references to entries in the bibliography appear in parentheses. Use the books and articles in the bibliography to pick up ideas for programming projects and to further explore the rich concept of OOP.

The glossary in Appendix C defines OOP terms and includes useful organizations and publications of interest to OOP programmers on the Mac.

PART 1

ABOUT OOP





In Part 1, we'll look at the fundamental concepts of object-oriented programming: classes, objects, methods, inheritance, subclassing, overriding, polymorphism, and runtime binding.

We'll get rolling in Chapter 1, and in Chapter 2, we'll move through a spiraling survey of the concepts at a fairly general level. In Chapters 3, 4, 5, and 6, simple code examples will illustrate the concepts. In Chapter 3, "Records Plus Procedures Equals Objects," we'll look at the basic techniques of declaring, implementing, using, and disposing of objects in a very simple program. In Chapter 4, "MacObjects," we'll write a serious class, a Macintosh button class called *TButton*. In Chapter 5, "Programming by Differences," we'll take up the concept of inheritance and how to make efficient use of it through subclassing and overriding, and we'll derive several new kinds of button classes from our *TButton* class. In Chapter 6, "Many Shapes," we'll look at an example of how to use polymorphism and runtime binding. The example is from a conventional data processing setting: an auto parts inventory application.

Chapters 7 through 9 are reference and technical chapters. In Chapter 7, "OOP Cookbook," you'll find a summary of OOP programming techniques. In Chapter 8, "The Family Tree," you'll find a discussion of the structure, uses, and pitfalls of object hierarchies. In Chapter 9, "Inside Object Pascal," we'll go through the syntax of Object Pascal and the particulars of how Object Pascal is implemented in compilers, and we'll look at topics such as efficiency, optimizing, and compiling libraries containing objects.

1

INTRODUCTION TO OOP

In Object Pascal, our statement is something like this:

```
if (you.ReadyToLearn('object-oriented programming')) then
  if (you.LanguageOf = Pascal) and (you.MachineOf = Macintosh) then
    you.Need('this book');
```

where *you* is an "object" whose type declaration looks like this:

```
type
  TProgrammer = object(TPerson);

      fReadiness: Boolean;
      fLanguage: Languages;
      fMachine: Machines;
      fBookNeeded: Str255;

      procedure IProgrammer(readiness: Boolean; language:
          Languages; machine: Machines);
      function ReadyToLearn(whatSubject: Str255): Boolean;
      function LanguageOf: Languages;
      function MachineOf: Machines;
      procedure Need(whatBook: Str255);
    end;  { Class TProgrammer }
```

and the object itself, a variable, is declared, allocated, and initialized as

```
var
  you: TProgrammer;
  ⋮
  { Allocate heap space for the object }
  New(you);                                  { If you buy this book and use it }
  you.IProgrammer(true, Pascal, Macintosh);  { Initialize object }
  ⋮
  if ...                                     { The if statement from above }
```

You've just seen the essence of object-oriented code in Pascal.

What Is Object-Oriented Programming?

Basically, OOP (as object-oriented programming is inevitably called) is a style of programming in which you model real-world objects—buttons, windows, spheres, dice, lists, auto parts, employees—with software "objects." As a style of programming, OOP can be done, laboriously, even in conventional programming languages such as C or FORTRAN. But other languages provide direct support for objects, making it a lot easier to adopt the style.

Some languages provide more support than others, of course. Smalltalk is a thoroughly object-oriented programming language (an environment, really), in which you must use objects to perform any task. Object Pascal provides for a simpler style of OOP with a simpler syntax and no doubt somewhat less power. But you can do a lot with objects using Object Pascal, as you'll see in this book. Here's a very brief synopsis. The chapters to come will provide much more detail and plenty of examples.

From the OOP Perspective

The fundamental entities in OOP programs are object classes, simply called "classes," and the "objects" that are members of them. The button class, for instance, is the class of all buttons. All buttons have certain things in common, and the class captures that commonality. To use a button in a program, you "instantiate" an object from the class, which is a kind of template. An object, or "instance," is a particular button that fills the template—with a shape, a title, a location, and so on.

You can derive new, related classes from existing classes by "subclassing." The subclass, or descendant, "inherits" all the data and capabilities of its ancestor. However, the subclass can also "override" what it inherits and add new data and capabilities. Inheritance often extends down through a whole hierarchy of classes and subclasses.

What are the characteristics of, say, a button class? Like a Pascal record type, a button class has data fields (called "instance variables" in OOP) of various kinds. For example, a button class might include a field to hold the coordinates of a display rectangle. Every instance of that class (every button object) would have that field in which to store its display rectangle.

But unlike a Pascal record type, the button class also includes "methods." In Object Pascal, methods are procedures and functions. Thus, an object in a class contains both data and operations. A button class would have methods to display, hide, and dim a button, to test whether a button has been clicked, and so on. Think of an object in a button class as a capsule that "knows" how to behave the way a Macintosh button is supposed to. When a subclass inherits, it inherits all of its class ancestor's data fields and methods except for the methods it overrides.

Messages and methods

Objects—the instances of classes (actual buttons, for example)—work by sending each other "messages." An object receiving a message invokes its own corresponding method in response. The Display message, for example, causes an object to invoke its corresponding *Display* method.

Polymorphism

An object can send the same message—Display in our example—to a dozen other objects of very different classes (buttons, integers, employees, for example), and the receiving object in each class will invoke its own *Display* method. Each object will respond to the Display message in its own particular way. An object in the button class will draw a button image on the screen, an object in the integer class will print out the object's integer value, and an object in the employee class will list the employee's address, job title, supervisor's name, and salary information. This ability of objects in different classes to respond individually, with their classes' own methods, to the same message is the essence of "polymorphism"—Greek for "many shapes."

Runtime binding

Another feature makes polymorphism even more powerful. In Object Pascal, your program can assign an object of one class to a variable of any of that class's ancestor classes. Thus, an object of the *TRadioButton* class can be assigned to a variable of the class *TButton*, where *TRadioButton* is a subclass of *TButton*. The variable of the *TButton* class can thus take on many shapes. Given such an assignment, your program can blindly send a Display message to whatever object the variable refers to at runtime, and the object of the *TRadioButton* class actually there will respond by displaying itself as a radio button should. This late binding of the message to a particular object's method is called "runtime binding" of method calls to the actual code invoked. Normally in Pascal, binding takes place at compile time. Polymorphism and runtime binding let you create highly general polymorphic data structures, such as linked lists that can hold multiple data types (even at the same time). We'll explore polymorphism and runtime binding thoroughly in this book.

In OOP, you design and implement new classes—often whole hierarchies of classes. Then you instantiate the classes with objects and start manipulating the data in the new objects with the objects' own methods. OOP is a new, rather data-centered way to organize programs. But Object Pascal is also still Pascal, and most of it will look quite familiar to you.

From the Pascal Perspective

In Pascal terms, a class is simply a type. Pascal programmers know how to use a wide variety of built-in and composite types: records, arrays, sets, files, arrays of records, records with array fields, pointer-based dynamic data structures, and so on. The class is simply a new type.

As you do any type, you have to declare an object type (class), specifying its name, its fields, its procedure and function headings (method headings), and—optionally in Object Pascal—the name of its immediate ancestor. You do this in the type declaration (class declaration) part at the global level of your program or of a unit. As we've noted, an object type resembles a Pascal record type. The elements of an object type declaration, though, include not only data fields but also method headings. An object type is like a record type with operations built into it.

Once you've declared an object type, you can declare object variables of this type in any variable declaration part anywhere in your program.

An object variable resembles a pointer (actually, a Macintosh handle). It's a reference to a block of storage dynamically allocated in the application heap. In fact, Object Pascal implements an object variable, also called an "object reference," as a handle. As you do with a pointer, you call Pascal's *New* procedure (not the Toolbox procedure *NewHandle*) to allocate memory for an object. Unlike with a pointer, though, you never dereference an object variable to access it. (Not on the Mac, anyway—the MS-DOS Turbo Pascal version of Object Pascal is different.) You access an object simply by calling it:

```
sphere1.fRadius := 3.0;       { Accessing an object's data }
```

or

```
sphere1.SetRadius(3.0);       { Calling an object's method }
```

This resembles the dot notation you'd use to access a record variable, but it has no caret (^) symbols to dereference the underlying handle. When it comes to objects, the compiler does the dereferencing for you.

In a sense, an object class resembles a Pascal unit. Like a unit, an object class has an interface part—the object class declaration's field list and method list. And, like a unit, the object class has an implementation part. The bodies of the class's methods, which are really only procedures and functions, get implemented somewhere below the class declaration—outside it. We'll go into where later.

From the Structured Programming Perspective

In structured programming terms, an object is a module. An object contains both the data and the code to operate on the data, "encapsulating" the data to protect it from outside access except by means of method calls. Your program accesses the object through a well-defined interface, just as a Pascal program accesses the code inside a procedure through a precise interface. "Sending messages" to objects really amounts to calling methods—very nearly the same as calling ordinary procedures.

Object-oriented Pascal is familiar in many ways, but it gives you a very different way of looking at programs and programming, a new way to divide up your programming problem—by data rather than by function. Still, Object Pascal is "good old Pascal," with a few simple extensions—and many new powers.

What's the Ancestry of OOP?

OOP has been around, in various permutations, since the 1960s—starting with the Simula language, in which OOP ideas were first used for simulation applications. But the biggest boost to OOP's growth came from the Smalltalk language, developed in the 1970s and 1980s at Xerox's Palo Alto Research Center.

Apple computer got into OOP through both an interest in Smalltalk and an early attempt at bringing objects to Pascal. On the Lisa computer, Apple developed a version of Object Pascal called Clascal ("Pascal with classes"). Then, in the 1980s, in consultation with Niklaus Wirth, the author of Pascal, Apple's software engineers developed a version of Object Pascal for the Macintosh, which became Macintosh Programmer's Workshop (MPW) Pascal.

Object Pascal itself—the language specification—is in the public domain, and two other companies have since produced Object Pascal compilers for the Macintosh. TML Systems turned its TML Pascal into an Object Pascal with release 2.0, which runs in Apple's MPW environment. THINK Technologies, acquired a couple of years ago by Symantec, developed release 2.0 of its Lightspeed Pascal (now known as THINK Pascal) to be an Object Pascal running in its own freestanding environment.

Object Pascal has also spread to the MS-DOS world, in Microsoft's QuickPascal, which follows the Apple Object Pascal model closely, and in Borland's Turbo Pascal, which has been object oriented since version 5.5. With version 6.0, Turbo Pascal supports a MacApp-like object-oriented application framework called Turbo Vision. Turbo Pascal changes the Object Pascal model considerably, incorporating many features from the C++ language.

Turbo Pascal might well influence the direction of Object Pascal both on the PC and on the Mac. Apple is already committed to its new C++ as a development language, and a Pascal++ is in the works, eventually to succeed today's Object Pascal. (As you might guess, Pascal++, if it's called that, will adopt many C++-like features while retaining a more Pascal-like syntax. You can follow the debate over Apple's Pascal++ design in the MacApp Developers Association newsletter, *Frameworks*, starting with the January 1991, issue—see MADA in Appendix C.)

You can use MacApp with C++ now, but most MacApp programming is still done with Object Pascal. Object Pascal has its limitations, so it may never be the Great American Software Engineering Language. But because it packs lots of power into a simple syntax and because there are lots of Pascal programmers on the Mac, expect Object Pascal to be around and in use for quite some time.

Will You Have to Give Up the Pascal You Know and Love?

Unlike "pure" OOP languages such as Smalltalk, in which performing any task requires objects, Object Pascal is a hybrid language. Object Pascal merely adds OOP features to standard Pascal through a handful of extensions. The new syntax you

have to learn in order to use objects is minimal, although the opening up of possibility you get with objects isn't.

You can ease into OOP if you like, incorporating a few objects here and there and putting the rest of your code in ordinary Pascal. Or you can write highly object-oriented programs. You'll see some of each kind of programming in this book.

Summary

Object-oriented programming (OOP) is a technique for designing and implementing programs by modeling the structures and capabilities of real-world objects. Object Pascal is an extended version of standard Pascal that supports the basic features of object-oriented programming: classes, objects, methods, inheritance, subclassing, overriding, polymorphism, and runtime binding. Essentially, these object-oriented features are implemented in Object Pascal through the class, a new Pascal type that resembles a Pascal record type.

The rest of this book will lead you gently into OOP and Object Pascal, help you understand the concepts underlying application frameworks like MacApp and the techniques you'll need to work with application frameworks, and explore the powers of polymorphism and the value of OOP for designing reusable software components.

In Chapter 2, we'll go on a more deliberate, detailed tour of the fundamental OOP concepts.

2

OOP OVERVIEW

Object-oriented programming and Object Pascal rest on these fundamental concepts:

- Objects
- Methods
- Classes
- Inheritance
- Subclassing
- Overriding
- Polymorphism
- Runtime binding

Most of this book is an exploration of these fundamentals. Each concept is quite simple, and most are only logical extensions of the Pascal you already know. But simple as they are, they are rich in the ways you can use them and in the programming power you can wring out of them.

In this chapter, we'll begin a slow spiral through the concepts, touching on them first at a general level and in uncomplicated contexts for the benefit of readers new to OOP or not too experienced at programming. Then, in several stages in the rest of Part 1, we'll pass through the concepts again, introducing more detail and making more connections among them.

In the rest of the book, we'll apply these concepts to ever more ambitious projects, including several fair-sized applications, a simple application framework related to MacApp, a complex "reusable software component," and a host of tools, utility objects, and data structures.

Let's begin with a brief capsule discussion of each concept, one that omits the variations and complications for now.

Objects

An object is a small module of data and code that can be combined with other objects to produce larger software systems. An object contains data fields just as a Pascal record variable does. But an object also contains procedures and functions called "methods" that operate on the data. A Pascal object type declaration—an object class declaration—takes this form:

```
type
  objectClassName = object

    { Data fields--also known as "instance variables" }

    { Procedure and function headings--also known as "method headings" }

  end;
```

Collectively, an object's data fields (its instance variables) and procedures and functions (methods) are called its "components," just as a record's data fields are its components. Figure 2-1 shows a sample Object Pascal object type declaration.

```
type
  TRectangle = object(TObject)

    { Data fields = instance variables }
    fRect: Rect;              { Rectangle in Macintosh terms; a QuickDraw type }
    fColor: Colors;           { Type defined elsewhere }
    fRotated: Boolean;        { Rotated or not? }

    { Procedures and functions = methods }
    procedure IRectangle(Rectangle: Rect; color: Colors);
    procedure Move(newTop, newLeft: Integer);
    procedure Resize(height, width: Integer);
    procedure SetColor(color: Colors);
    procedure Display;
    procedure Hide;
    function LocationOf: Point;
    function SizeOf: Point;
    function ColorOf: Colors;
    procedure Rotate(degrees: Integer);
  end;  { Declaration of object class TRectangle }
```

Figure 2-1.
A sample object type declaration, the declaration of a TRectangle *class.*

Objects and Records

You'll notice that the object class declaration is a type declaration (*T* for "type") and that the declaration is very similar to a record type declaration. Technically, an

object is a variable of an object type. Like a record type, an object type includes a field list (*f* for "field"). A record type declaration, of course, doesn't list procedure and function headings as components—but an object type declaration does. This equation captures the essence of objects:

Records + Procedures = Objects

An object is much like a record to which procedures have been added. Adding procedures "encapsulates" a set of data and the operations on the data in one module.

In *Object-Oriented Programming for the Macintosh* (Schmucker, 1986a), Kurt J. Schmucker characterizes objects as "little autonomous computers that pass messages back and forth as part of a larger system." As such, objects are a new way of handling data in a Pascal program—although they're also simply an extension to the data-handling facilities long available in Pascal, as we'll soon see.

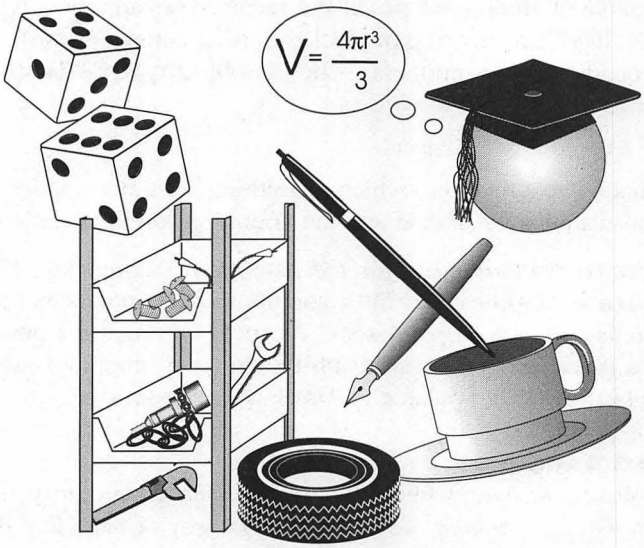
A few object examples

Objects, of course, are everywhere—your computer, your pen, your coffee cup, your geranium. Object-oriented programming is a way of modeling the real-world objects around you in software.

We'll see a wide variety of objects in this book: employee objects that can calculate their pay; spheres that can calculate their own volumes; buttons that can tell when they've been clicked with the mouse and know what to do as a result; auto parts bins that can hold multiple parts types; dice games that can create the players and dice and start the dice rolling; players that know the rules of the game and how to throw the dice; dice that know how to roll and return a pseudorandom number in a certain range; "views" that know how to display all or part of a window's contents; linked lists that can contain any kinds of objects you can imagine (even all at once); and even "application objects" that can run an application—putting up and operating its menus, handling its windows, creating its other objects, and responding to requests from those objects.

Almost any programming problem will present opportunities for using objects.

> **NOTE:** *Speaking of objects as "knowing" this or that might seem to suggest that object-oriented programming is some kind of artificial intelligence programming. Although AI programmers are indeed using OOP these days, OOP is not AI. To speak of objects as knowing is merely to indulge in a handy figure of speech. OOP people often anthropomorphize their objects. You can take that or leave it, as you like.*
>
> *And Kurt Schmucker's description of objects as "little autonomous computers that pass messages back and forth" might seem to suggest that objects function concurrently. They don't at all. Their functioning is strictly sequential, although often the sequence can appear less than straightforward because of polymorphism and runtime binding.*



Object Types and Object Variables

We've noted that an object is a variable of some object type—an instance of a class.

An object variable (instance) declaration

The format of an object variable declaration looks quite familiar:

```
var
  anObject: ItsType;      { Format of a declaration }
```

The next example shows what an object type declaration looks like. We'll go into the syntax in more detail later.

An example object type (class) declaration

In Chapter 3 we'll develop a sphere object. A sphere object contains data fields for its radius and perhaps for other features (its location, for instance). A sphere object also contains the procedures and functions to compute its own circumference, area, and volume and to display itself on the Mac's screen in some way. We simply declare the sphere type, create a sphere object (as a Pascal variable), initialize the sphere object with its radius (and its location and other features), and "ask" it to compute and display its vital statistics. The sphere object contains those capabilities, so in a sense the object "knows" how to do as "asked."

Here's what a declaration for the sphere object type looks like:

```
type
  TSphere = object

      { The object type's data fields--like a record type's }
      fRadius: Real;
      fCenter: Point;

      { The object type's methods--operations }
      { This part of declaration is totally unlike a record type's }
      procedure ISphere(radius: Real; centerV, centerH: Integer);
      function CircumferenceOf: Real;
      function AreaOf: Real;
      function VolumeOf: Real;
      procedure Display;
    end;  { Class TSphere }
```

An object type can be declared only at the global level of a program or unit. Unlike other types, an object type can't be declared inside a nested scope—inside a procedure or function. An object variable can be declared anywhere you can declare an ordinary variable.

What an object in the *TSphere* class does

After you create a sphere object (see "Creating and Using Objects" later in this chapter), you call its *ISphere* method (*I* for "initialization") to initialize its three data fields. You can call the sphere object's three computation methods directly and have them compute values based on the sphere object's radius. More typically, though, you call the sphere object's *Display* method to print out all of the sphere's vital statistics—which the *Display* method accomplishes by calling the three computation methods. Thus, the sphere object in our example is designed to manipulate its own data, using its own code. That's really the essence of objects.

Some Naming Conventions

As you will have noticed by now, names of object types (classes) are preceded by *T* for "type" and names of object data fields (instance variables) are preceded by *f* for "field." MacApp programmers follow these conventions, and because they're also the prevailing conventions in the literature on Object Pascal (at least in the Macintosh world), we'll follow these and most of the other MacApp naming conventions, too. (The THINK Class Library, TCL, uses *C* for "class" for object type names and doesn't precede field names with *f*.) Prefixing the class name with *I* for "initialization," as in the initialization routine *ISphere*, is also a MacApp convention. We'll sometimes deviate from this convention, calling the initialization routine *Init* instead, but the MacApp convention does have its uses, as we'll see later.

Data Fields (Instance Variables)

An object type declaration can contain as many data fields as it needs to. Each can be of any type—even, recursively, of the object type whose declaration the data field is in. Within the declaration of type *TSphere,* for example, we could have a data field of type *TSphere.*

So far we've used the term "data field" because it's familiar from usage for Pascal records, but most of the literature on OOP calls data fields "instance variables," so we will too. An "instance variable" is a variable (data field) belonging to an instance (object) of some object class. Each instance of a class has its own copies of its class's instance variables because each object might need to store different values in the instance variables. For example, class *TSphere* has two instance variables, *fRadius* and *fCenter.* One of those objects could have a radius of *100* and a center at point *(20, 20)*, and another could have a radius of *5* and a center at point *(5, 23)*. An object's instance variables are typed, just as a Pascal record's fields are.

In an object class declaration, all instance variables must precede any method headings.

Procedure and Function Declarations (Method Headings)

The object class declaration can contain as many method headings as it needs to. Each heading is a complete procedure or function declaration, the same as if it were declared in the interface part of a Pascal unit or in a Pascal forward declaration (without the keyword *forward*) in a program.

Variants?

Object types do resemble record types, with the keyword *object* replacing the keyword *record* in the declaration and with the addition of method headings listed inside the object type (class) declaration. But don't get the idea that an object is a record. In particular, as Figure 2-2 shows with its examples of correct and incorrect declarations, an object cannot have a variant part. You'll see that subclassing provides a much better mechanism for variation than variant records do.

```
type
  A = record
        field1: Integer;
        case Boolean of
          false: (ch: Char);
          true: (int: Integer);
      end;
```

```
type
  A = record
        field1: Integer;
        case Boolean of
          false: (ch: Char);
          true: (int: Integer);
      end;
```

Figure 2-2.
No variant part in an object declaration.

Procedure and Function Implementations (Method Bodies)

The implementations of the procedures and functions (method bodies) whose headings appear in the object type declaration appear elsewhere in the program or in a unit, below the declaration.

Creating and Using Objects

We can create and use an object much as we do any other entity in Pascal—by declaring a type, then declaring a variable of that type, and then using the variable. We've looked at the type declaration. Here's the rest of the process:

```
var
  aSphere: TSphere;              { Declaring object variable }
⋮
  New(aSphere);                  { Allocating memory for object }
  aSphere.ISphere(3, 50, 50);    { Initializing object }
  ⋮
  aSphere.Display;               { Calling object's display method }
```

Note that an object is a dynamic variable that uses memory allocated from the heap at runtime. Like a pointer, an object must be allocated with Pascal's *New* procedure. We'll examine the rest of the example's rather strange-looking syntax shortly.

Methods

We've observed that an object "contains" procedures and functions to work on its data, and we've looked at some procedure and function declarations. Borrowing from the Smalltalk language, we call these procedures and functions "methods." What the object actually contains, as we'll see, is a way to locate the code of its methods. Rather than have each object in a class contain its own copy of each method's code, an object-oriented program stores that code in one place when the program is compiled so that the same code can be called by any number of individual objects in that class.

Calling Methods ("Sending Messages")

Methods look like ordinary Pascal procedures and functions, and calling methods resembles calling procedures and functions, although the syntax is a bit different, as we'll see shortly. Calling an object's methods is often referred to as "sending the object messages." In Figure 2-3 on the next page, messages are sent to the *rect* object's corresponding methods.

"Sending messages" is a more accurate way to put it in some other OOP languages, such as Smalltalk, than it is in Object Pascal, but it's a fair analogy for what happens, and it "feels like" what we do. We'll usually say "sending a message" because the OOP literature does, but sometimes we'll say "calling a method."

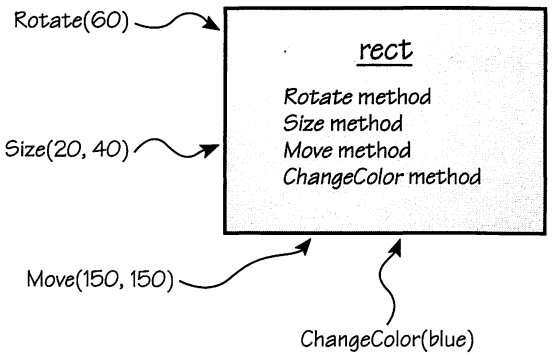


Figure 2-3.
Messages are sent to the object— the object's methods are called.

Coding Methods

A method can contain any kind of code you can imagine. A method might create a Mac window, handle a mouse click in a menu item, compute the circumference of a sphere, or simply set or get one of its object's instance variables. Most methods are small, often only a line or two of code, but we'll see some large methods later in the book. If you tend to write big procedures, you might write big methods, although using objects has a way of forcing you to modularize code more than usual. In most respects, methods really are simply procedures and functions, the kind you're already comfortable with. But in an important sense a method "belongs" to a particular object, and its actions take place in the context of that object.

A sphere method declaration and implementation

Our example sphere object class contains an instance variable to hold the value of a sphere's radius. When we create a sphere object, we have to initialize its radius before the sphere object can compute its own surface area, for instance, or display the value of its volume. Here's how the *ISphere* method is declared inside the *TSphere* object class declaration:

```
procedure ISphere(radius: Real; centerV, centerH: Integer);
```

And here's the full body of the method—its implementation:

```
procedure TSphere.ISphere;
begin
  fRadius := radius;   { fRadius is the radius field }
  fCenter.v := centerV;
  fCenter.h := centerH;
end;  { ISphere method }
```

Notice two things about the *ISphere* method's code:

- The qualifying prefix in *ISphere*'s name, *TSphere.ISphere*. The prefix was unnecessary inside the object class declaration for *TSphere* because the method heading was contained by the declaration—*ISphere* was clearly associated there with class *TSphere*.

  But the method body is given separately, outside the declaration, so the prefix is necessary to associate the method body with the declaration's object class. You might want to develop the habit of using the prefix in both the method declaration and the method body.

- The omission of *ISphere*'s parameter list when we write the method's body. The list is mandatory when we declare the method inside the object class declaration, but it's optional when we give the method body. We'll usually provide the parameter list as we work with the examples in this book, but you might prefer not to in your own programs.

Which sphere?

If there are several sphere objects around, how can we tell which sphere object we are initializing when we send an ISphere message? In ordinary Pascal, we'd include a parameter of the appropriate type and pass the name of the sphere we mean to the procedure, like so:

```
ISphere(thisSphere, 3, 50, 50);
```

In Object Pascal, we do something like this instead—prefixing the method name with the object variable name:

```
{ Send ISphere message to the object aSphere }
aSphere.ISphere(3, 50, 50);
```

where *aSphere* is the name of a particular sphere object, already allocated by means of the *New* procedure.

The syntax here resembles the dot notation we use to access the fields of a record in ordinary Pascal, and, in fact, we use the same notation to access the instance variables of an object:

```
aSphere.fRadius := 3;   { Similar to Pascal record data field access }
aSphere.fCenter.v := 50;
aSphere.fCenter.h := 50;
```

Encapsulation

Although you are free to access an object's instance variables directly, as we have just done, it's considered good OOP form to access instance variables only indirectly, through methods. Hence, part of good object class design is to provide any methods necessary to set or get the class's instance variables without resorting to direct access. You should program to treat an object as a capsule, accessible only by means of methods. You can violate an object's encapsulation for special purposes, but you shouldn't do that lightly. Figure 2-4 on the next page depicts outside and inside views of an object.

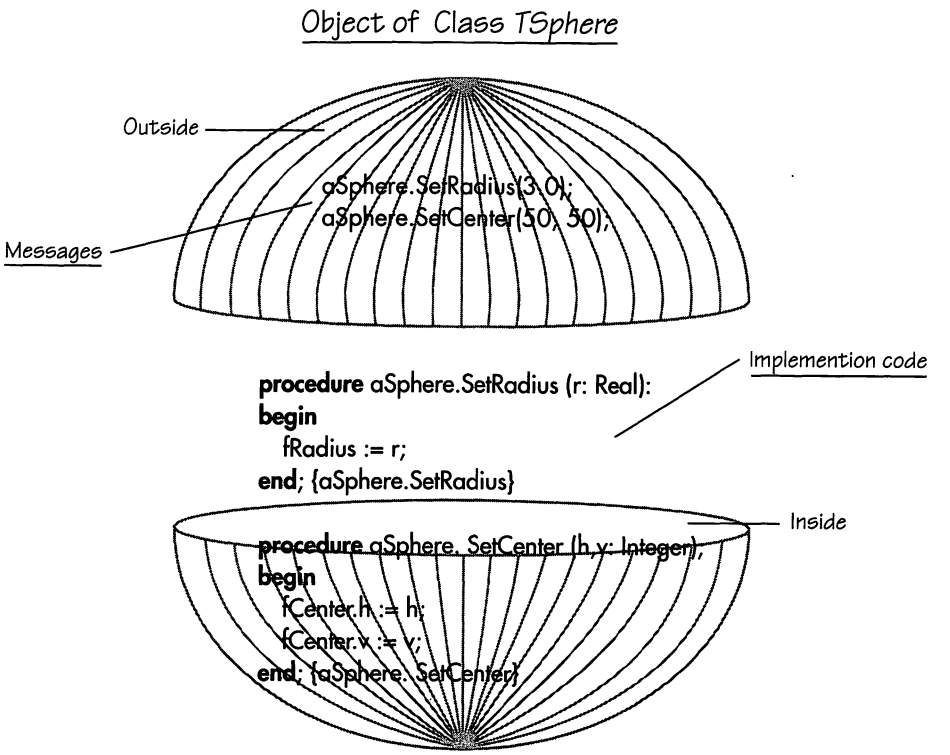


Figure 2-4.
Encapsulation—avoiding direct access to the inside.

Dereferencing?

It might seem odd to use record-like dot notation for a procedure call, but that's how it's done. By the way, *aSphere* is actually something like a pointer or handle to the actual object in the heap. But we don't have to—and must not!—use pointer-type or handle-type dereferencing to get at the object. Here's how *not* to do it—neither

```
aSphere^.fRadius;  { Not how it's done }
```

nor

```
aSphere^^.fRadius;  { Not how it's done }
```

We must remember that *aSphere* is an object, not a pointer. Although objects are actually implemented with handles on the Mac, objects do not use pointer dereferencing syntax. The dereferencing is done for us, implicitly, by the compiler.

(Turbo Pascal for MS-DOS uses explicit dereferencing for some objects, but no Object Pascal on the Mac—at this writing—does it that way.)

The *self* pseudovariable

When we're writing the method code for an object class, it's sometimes useful to be able to reference the current object; that is, when a particular object's method is executing, we might want the object to refer to itself.

We have the object do this by using the word *self* as illustrated in Figure 2-5. Every time a method is called, in addition to the parameters in its parameter list, an extra, implicit parameter is passed to the method: a reference to the current object. We don't pass the extra parameter explicitly. Inside the method code, we can access the reference to the current object by means of the "pseudovariable" *self*. For example,

```
{ Inside Init method code of object anObject }

self.Display;              { Call one of self's methods }
self.fRect := theRect;     { Access one of self's instance variables}
aVar := self.fLength;
```

The pseudovariable is accessible, of course, only from inside the object's methods. We use *self* to make it clear in our method code that the instance variables and methods we're accessing are the current object's own. Such usage is entirely optional. We'll do it as a convention, but we don't have to. *self* does have more serious uses: For example, sometimes it's useful for one object to pass a reference to itself to some other object. Then the other object can use the reference to access the first object's instance variables or methods. We'll explore the use of *self* further in Chapter 7.

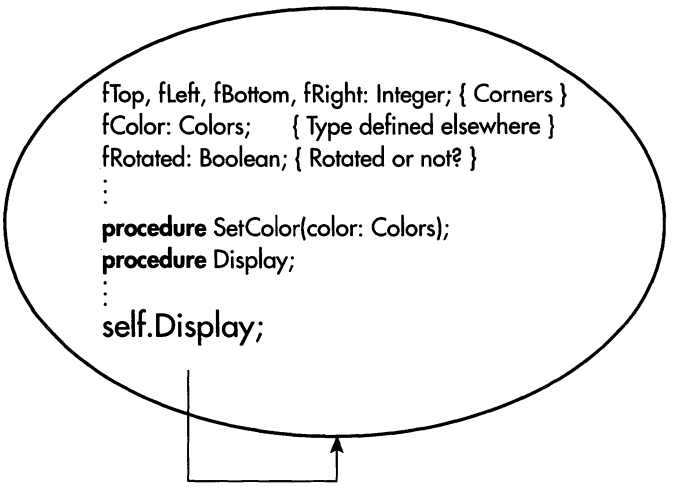


Figure 2-5.
The self *pseudovariable.*

Where's the body?

Method bodies are given either

- Somewhere below the object type declaration in the same declaration section:

```
program Something;

  { Type declarations }
  {  including object types }

  { Procedure and function declarations }
  {  including method bodies }

begin { Main program }
:
```

- Or, if the object type was declared in a Pascal unit, in the unit's implementation part:

```
unit Something;

interface

  { Object type and other type declarations }
  { Other declarations }

implementation

  { Method bodies }
  { Other procedure and function bodies }

end.
```

Regardless of where, when the body is given, the method's procedure or function heading must be repeated with the method name qualified by the class name, as we've seen earlier:

```
procedure TSphere.ISphere;
```

You don't have to repeat the parameter list or the function result type, but your programs will be more readable if you do. If the method has been overridden, the keyword *override* should appear with the method's heading in the declaration, and you can repeat the *override* keyword with the method body, although it's optional there. Either of these forms is acceptable:

```
procedure TSphere.ISphere(radius: Real; centerV, centerH: Integer);
begin
:
```

or

```
procedure TSphere.ISphere;
begin
:
```

Figure 2-6 shows an outline of a standard Pascal unit, in case you need a reminder of the unit construct.

```
unit UName;                                  Unit name
interface                                    Interface part begins
  uses                                       uses clause—other units used
    UOther, UOther2;
  type                                       Declarations: types, consts, vars,
    SomeType = (one, two, three);              procedure and function headings
  procedure DoIt(aParameter: SomeType);
  function ValueOf: SomeType;

                                             End of interface part
implementation                               Implementation part begins
  procedure DoIt;                            Bodies of procedures and functions
    var                                        declared in the interface
      x: SomeType;
  begin                                      Other procedures and functions
    x := aParameter;                           needed for private use inside the
  end; { Body of procedure DoIt }              implementation part
  function ValueOf;
  begin                                      And any types, consts, vars, and
    ValueOf := 23;                             even private objects needed
  end; { function ValueOf }                    inside the implementation
end. { Unit UName }                          End of unit
```

Figure 2-6.
Parts and syntax of a standard Pascal unit.

Classes

Here and there I've been carrying the word "class" in parentheses as I've talked about object types. As we go on in this book, I'll speak more often of object classes than of object types. Individual objects, such as our spheres, are simply Pascal variables—similar to pointer variables in that we must allocate memory for them in the heap with Pascal's *New* procedure. These dynamic variables are declared, as we've seen, the way other variables are:

```
var
  aSphere: TSphere;
```

TSphere is the object variable's type. Object types are a new kind of Pascal type, similar in appearance and declaration, as we've observed, to record types.

Another way to think and talk about objects and their types—one you'll commonly find in the OOP literature—is by means of the concept of "class." Figure 2-7 uses classes of foods to illustrate the notion of class.

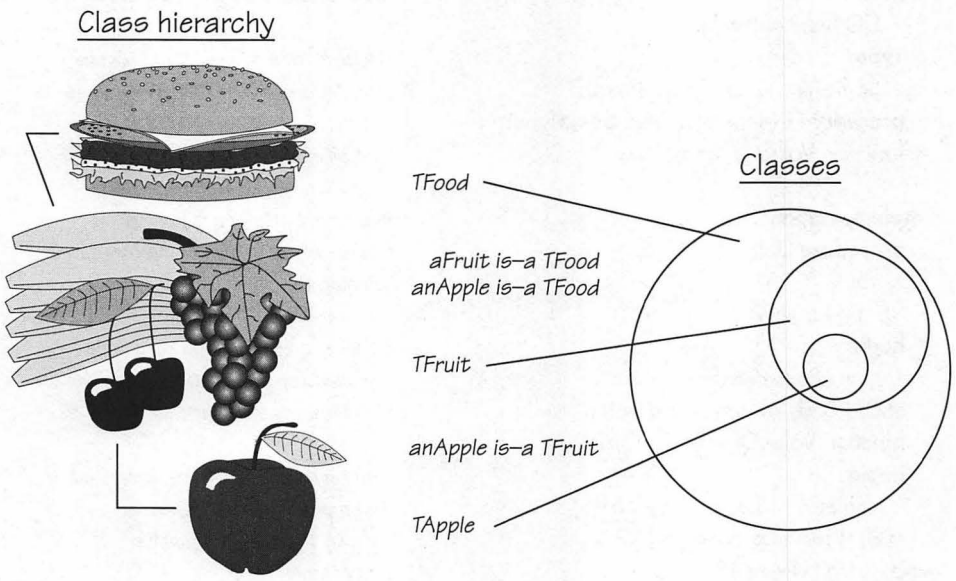


Figure 2-7.
The food hierarchy.

Class (or classification) is a common logical tool familiar to us from any number of fields. Biologists classify life into phyla, genera, orders, species, and so forth. Geometers classify various shapes in a plane in a similar hierarchical way: A square is a member of the rectangle class, which is a member of the parallelogram class, which is a member of the quadrilateral class, which is a member of the polygon class, and so on, as shown in Figure 2-8.

Just as an apple *is-a TApple,* and a *TFruit,* and so on, a given object instance such as *aSquare is-a TSquare,* and a *TRectangle,* and so on. The relationship of the object to its hierarchy of classes is often called the "*is-a* link."

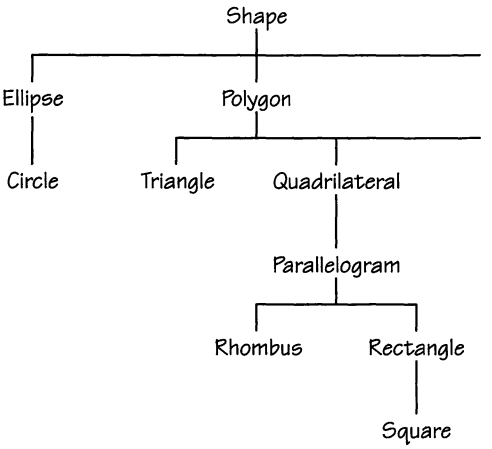


Figure 2-8.
A hierarchy of related geometric shape classes.

Instances

The object itself, *aSquare,* is called an "instance" of its class. A class (or our familiar "object type"), you'll remember from Chapter 1, is a template from which instances such as *aSquare* are created. Our programs declare the object classes as types and the object instances as variables. The instances are actually created dynamically at runtime by means of *New.*

Subclassing, Inheritance, and Overriding

One of the interesting—exciting, powerful, stupendous, neat—things about object classes is that we can derive an object "subclass" from a preexisting class by means of "inheritance." Much as a child inherits many of its parent's features and abilities, a subclass can inherit its instance variables and methods from an ancestor class. Imagine a Pascal type that can inherit the features of another type and then add to what it inherits or modify the inherited features. We'll really put this capability to work in Chapter 5, "Programming by Differences."

An example of subclassing, inheritance, and overriding

Here's an example to illustrate the related principles of subclassing, inheritance, and overriding. Suppose that we already have an object class called *TMicrocomputer* as it's declared on the next page.

```
type
  TMicrocomputer = object

      fProcessor: String;
      fRAMSize: Integer;
      fWordSize: Integer;  { 8-bit, 16-bit, etc. }
      ⋮
      procedure Install;
      procedure BootUp;
      procedure InsertDisk;
      function GetRAMSize: Integer;
      ⋮
    end;  { Class TMicrocomputer }
```

Now suppose that we want to derive a more specialized class from *TMicrocomputer*:

```
type
  TMacintosh = object(TMicrocomputer)

      { Inherits TMicrocomputer instance variables }
      {  and adds some of its own }
      fMouseType: String;
      fModel: String;
      ⋮

      { Inherits TMicrocomputer methods }
      {  and adds some of its own }
      procedure SetStartupApp;
      procedure ChoosePrinter;
      ⋮

      { "Overrides" some of TMicrocomputer's methods }
      procedure Install;
      override;
      procedure BootUp;
      override;
    end;  { Class TMacintosh }
```

The example is whimsical, of course, but it faithfully shows how we'd declare the derived subclass and then differentiate it from its ancestor class. See Figure 2-9 for another example of inheritance and differentiation.

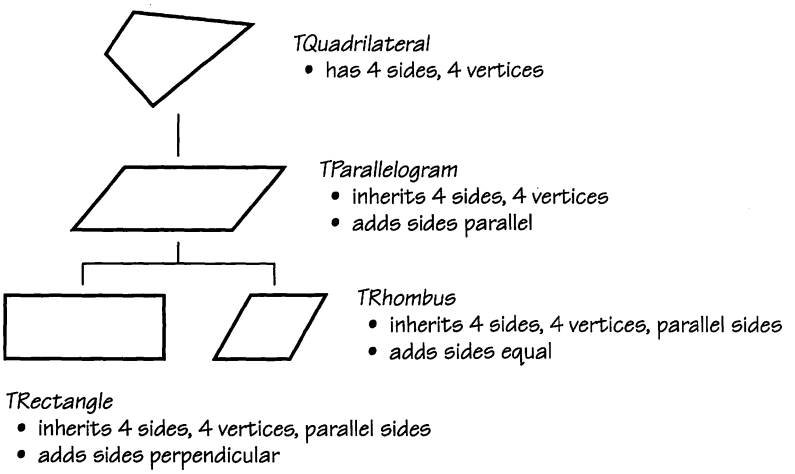


Figure 2-9.
Inheritance and differentiation.

Subclassing and Inheritance

The class from which a subclass is derived is its "immediate" ancestor. If the immediate ancestor was derived from some other class, our new subclass has more distant ancestors, too; hence, the qualifier "immediate." If we derive class *C* from class *B*, which was originally derived from class *A*, the result is an object hierarchy. The hierarchy can be infinitely long—but in Object Pascal no descendant class can have more than one immediate ancestor. Inheritance is single in Object Pascal, but each ancestor up the line can have ancestors. The linkage stops when an ancestor does not have any ancestors of its own. The hierarchy shown in Figure 2-8 on page 29 illustrates the derivation of a square by means of single inheritance. Figure 2-10 illustrates the derivation of a square by means of multiple inheritance.

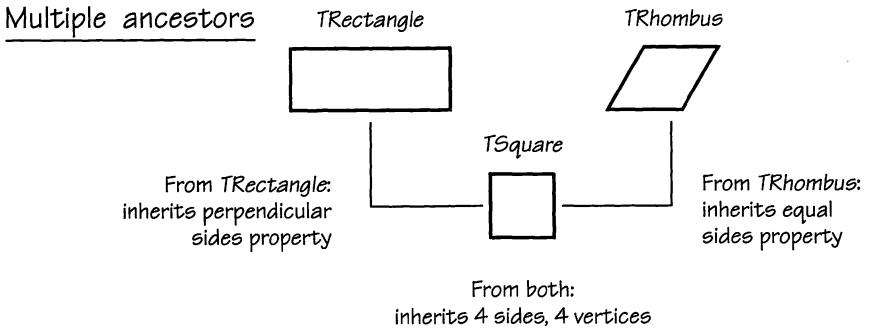


Figure 2-10.
Multiple inheritance, a feature of some OOP languages but not of Object Pascal.

What's inherited?

A descendant inherits everything—every instance variable and every method—from its immediate ancestor, plus everything from all its immediate ancestor's ancestors. So class *C* in our example inherits not only the instance variables and methods of class *B* but also the instance variables and methods of class *A*. The one limitation on inheritance is that, if any ancestor "overrode" any of its inherited methods, its descendants inherit the overridden versions of the methods, not the originals. A subclass can't override instance variables, though. The result is that class *C* is a composite of all its ancestors. Figure 2-11 illustrates inherited and overridden object class components passed down through an object class hierarchy.

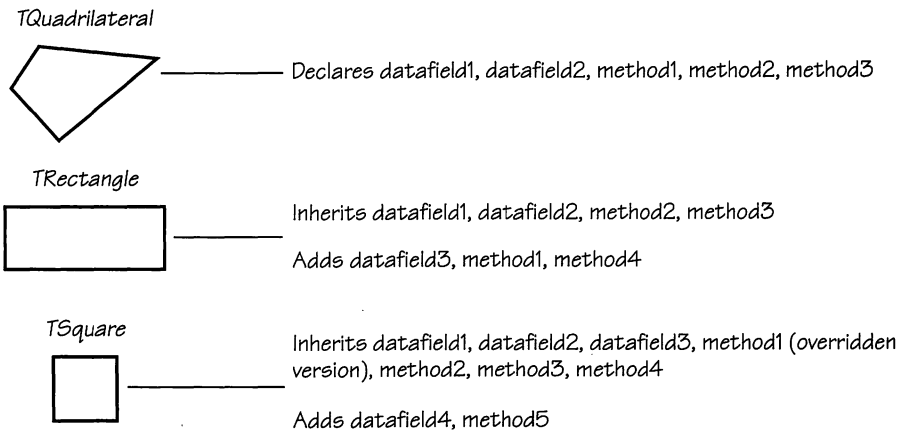


Figure 2-11.
What class TSquare *inherits.*

An example of declarations for a class hierarchy

Let's declare several small classes in a hierarchy to demonstrate inheritance. Figure 2-12 shows how we can create a food class hierarchy.

```
type
  Courses = (appetizer, salad, main, dessert);

  TFood = object(TObject)  { Another type declaration }
      fCourse: Courses;

      procedure Init(forNumberOfGuests: Integer);
      procedure Prepare;
      procedure Serve;
    end;  { Class TFood }
```

Figure 2-12. *(continued)*
Class hierarchy declarations.

Figure 2-12. *continued*

```
TFruit = object(TFood)
    { Inherits fCourse instance variable }

    procedure Peel;          { Adds a new method }
    { Overrides all of TFood's methods }
    procedure Init(forNumberOfGuests: Integer);
    override;
    procedure Prepare;
    override;
    procedure Serve;
    override;
  end;  { Class TFruit }

AppleKinds = (GrannySmith, McIntosh, Delicious);

TApple = object(TFruit)
    { Inherits fCourse from TFood through TFruit }
    fKind: AppleKinds;       { and adds a new instance variable }

    { Inherits TFruit's Init method }
    {  but overrides the others }
    procedure Prepare;
    override;
    procedure Serve;
    override;
  end;  { Class TApple }
```

Notice that other type declarations are intermixed with the object class declarations—an object class declaration is simply another type declaration. The class hierarchy we've just created is shown abstractly in Figure 2-13 on the next page.

TObject is a special class in THINK Pascal that programmers often use to provide a common ancestor for other classes. *TObject* contains four methods that it makes available to its descendants: *Clone, ShallowClone, Free,* and *ShallowFree.* We'll discuss most of those methods in detail in Chapter 7. *Free* is the most often used—to dispose of an object's storage after using the object. Any class descended from *TObject* inherits its four methods, so any *TObject* descendant knows how to clone and free itself.

The chain of descent can be followed down the hierarchy because each class names its immediate ancestor in the "heritage spot"—in the parentheses immediately following the keyword *object*:

```
type
  TClassName = object(ancestorName)  { Heritage spot }
```

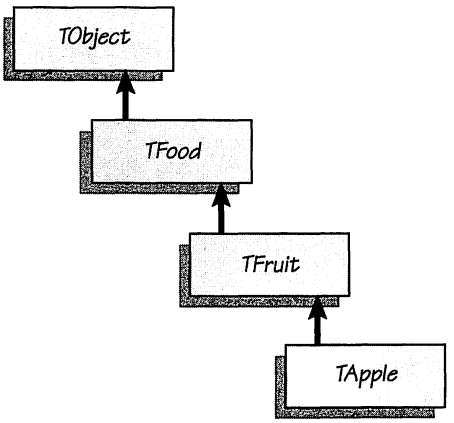


Figure 2-13.
The food hierarchy.

The food example also demonstrates the use of instance variables and methods. We'll look at overriding methods in the next section. *TFood* does most of the work in the hierarchy, defining most of the instance variables and methods the other classes need. *TFruit* inherits everything from *TFood*—except what it overrides. *TApple* inherits everything from both *TFood* and *TFruit*—except what it and *TFruit* override.

> **NOTE:** *A descendant class doesn't list anything that it inherits. If the descendant class did list its inherited instance variables and methods, that would cause an error: The compiler would think that the descendant class was trying to override the instance variables and methods incorrectly.*

Each subclass in the food hierarchy adds something it hasn't inherited, thus *extending* what the ancestor(s) provides. And each subclass overrides one or more of its ancestor's methods, thus becoming more specialized. *TFruit* objects probably need to be initialized, prepared, and served differently than other kinds of food, so *TFruit* overrides those *TFood* methods. And *TApple* objects can require special treatment beyond, or different from, the treatment of other fruits. In fact, the class *TFood* is so abstract that some or all of its methods might actually be stubs—procedures with no code. A stub means that we can't fully specify the method at this level; instead, we defer specifying the method's details until a more specialized subclass specifies them. Usually, the abstract class *TFood*—in our example—requires its subclasses to override its methods. Creating an abstract ancestor class with stubbed methods is a common practice. We'll go into abstract classes again and in more detail in Chapter 7.

Subclassing and Overriding

Let's look again at part of our example of subclassing (Figure 2-12 on pages 32–33):

```
type
  TFruit = object(TFood)

      procedure Peel;
      procedure Init(forNumberOfGuests: Integer);
      override;
      procedure Prepare;
      override;
      procedure Serve;
      override;
    end;  { Class TFruit }

  AppleKinds = (GrannySmith, McIntosh, Delicious);

  TApple = object(TFruit)

      { No instance variable or method declarations }

    end;  { Class TApple }
```

So far, the *TApple* class would be a duplicate of *TFruit* in every way except in its name. Even though it lists no instance variables and methods, it inherits everything from *TFruit.* To differentiate the descendant from the ancestor, as we did earlier with *TApple*, you can either extend the instance variables and methods of the ancestor class or make the descendant class more specialized:

```
type
  TApple = object(TFruit)

      fKind: AppleKinds;        { Adds a new instance variable }

      procedure Core;           { Adds a new method }
      { Overrides two methods }
      procedure Prepare;
      override;
      procedure Serve;
      override;
    end;  { Class TApple }
```

Now it's evident that you can make a descendant class different from its immediate ancestor in three ways:

- You can *add new instance variables* to those the descendant class inherits and thus extend the ancestor class.
- You can *add new methods* to those the descendant class inherits and thus extend the ancestor class.
- You can *override the methods* inherited from the ancestor class and thus make the descendant class more specialized.

Note these two limitations:

- You can't override instance variables.
- You can add or override methods, but you can't delete them. Nor can you delete instance variables.

How to override a method

To override a method, you redeclare it in the descendant class and follow the declaration with the keyword *override*:

```
{ In the descendant class }
procedure SetRadius(radius: Real);
override;
function GetRadius: Real;
override;
```

Figure 2-14 illustrates the consequences of overriding the *Display* method of the ancestor class.

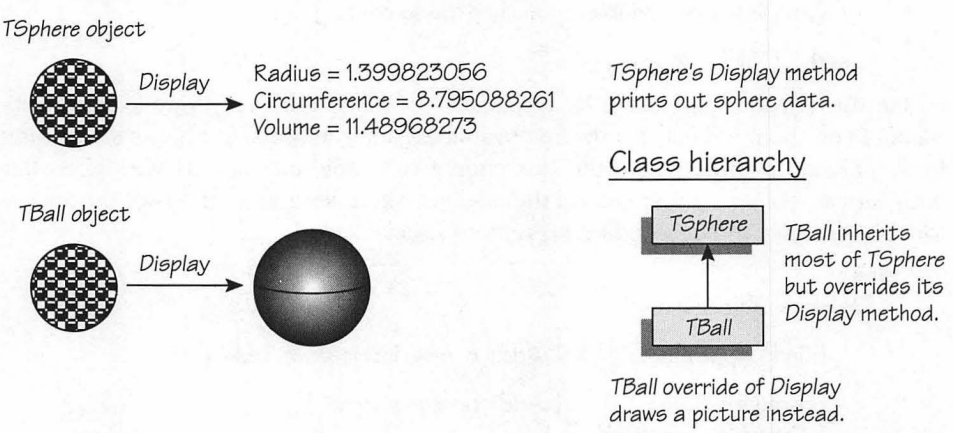


Figure 2-14.
The effect of overriding.

Extensible Objects

The subclassing mechanism makes objects highly extensible, and OOP programmers often choose to remold an existing class to a particular need—not by altering the class itself but by using a new subclass to either extend or customize the existing class. The ability to create an extended or specialized version of an existing class makes classes highly reusable—you can easily fix up any shortcomings of a class you want to reuse.

An extended subclass does more than its ancestor(s) did—mainly by adding instance variables and methods. A specialized subclass does things differently than its ancestor(s) did—mainly by overriding its ancestor's methods.

Some Caveats

- When you override a method, you can't change anything about the method's heading in the original declaration—not the method's name; nor the number, order, and types of its parameters; nor the type of a function method's return. This can present problems, as we'll see later on.
- When class *C* overrides a method from class *B*, it no longer inherits that method, but an object of class *C* can still call the method of class *B* that class *C* has overridden.
- You can't override instance variables (fields). The subclass inherits its ancestor's instance variables whether you want it to or not.

Polymorphism

If you've begun to get a glimmer of the rich potential the inheritance-overriding combination offers the programmer, wait until you see polymorphism and runtime binding at work. Objects become "polymorphic" (Greek for "many shapes") when they are part of a class hierarchy. It's possible for various unrelated objects (not in the same hierarchy) to respond to the same message. This isn't quite what we mean by full polymorphism, though, because the objects aren't assignment compatible.

An example of polymorphism

We'll reuse our food example to understand polymorphism. Take another look at the class hierarchy we declared in Figure 2-12 on pages 32–33.

Class *TApple* inherits all the instance variables of classes *TFruit* and *TFood* (in this case, only one—*fCourse*). It also inherits all their methods—except those it overrides. Three of *TApple*'s four method names are found in all its ancestor classes. Later, we'll see the importance of this sharing of method names—in this instance, of *Init*, *Prepare*, and *Serve*—for sending messages to polymorphic objects.

Now, suppose that we declare an object variable of each class:

```
var
  aFood: TFood;
  aFruit: TFruit;
  anApple: TApple;
```

Here's where polymorphism comes in. We allocate and initialize each of our variables:

```
:
New(aFood);  { Allocate memory for an object instance from the heap }
aFood.Init(8);  { Set object's instance variables, etc. }
New(aFruit);
aFruit.Init(8);
New(anApple);
anApple.Init(8);
```

Then we can make the following legal variable assignments:

```
aFood := aFruit;      { Fruit descends from food }
aFood := anApple;     { Apple descends from food }
aFruit := anApple;    { Apple descends from fruit }
```

but not these:

```
anApple := aFruit;    { Illegal--fruit doesn't descend from apple }
anApple := aFood;     { Illegal--food doesn't descend from apple }
aFruit := aFood;      { Illegal--food doesn't descend from fruit }
```

You can assign objects of a descendant class to variables of any of its ancestor classes, but you can't assign ancestor objects to descendant variables.

> **NOTE:** *"Object" and "variable" are equivalent terms in this context.* anApple *is a variable of class* TApple*;* anApple *is thus an "object reference" to a class* TApple *object, although we commonly think of the variable as the object itself.*

What the Compiler Doesn't Know

When a program compiles, the compiler has no way of telling what class of object will be "contained" by a particular object variable at any point. At runtime, for example, many different objects, of differing classes, might be assigned to a *TFood* variable. This ability of a variable to contain objects from many different classes makes the *TFood* variable polymorphic—it can take on different "shapes."

Sending Messages to Polymorphic Objects

Suppose that, as our program runs, the object variable *anApple* gets assigned to *aFood*. And suppose we send *aFood* a *Prepare* message because we want the item of food currently in *aFood* to be washed, peeled, diced, or whatever might constitute preparing it for consumption:

```
aFood.Prepare;
```

Because the compiler has no way of telling that *anApple* will get assigned to *aFood* at runtime, it can't "bind" the method call to *aFood.Prepare* at compile time—the compiler can't specify exactly which method code will be called when *Prepare* gets executed at runtime. Will it be *TFood*'s *Prepare* method? *TFruit*'s? Or *TApple*'s? Fortunately, all three classes contain a method called *Prepare*—never mind that each *Prepare* probably does something different. At runtime, the correct *Prepare* method will be called—the one belonging to the actual object in *aFood* at that point, no matter what class the object belongs to. Figure 2-15 illustrates how the same Draw message sent to different objects evokes object-peculiar *Draw* methods.

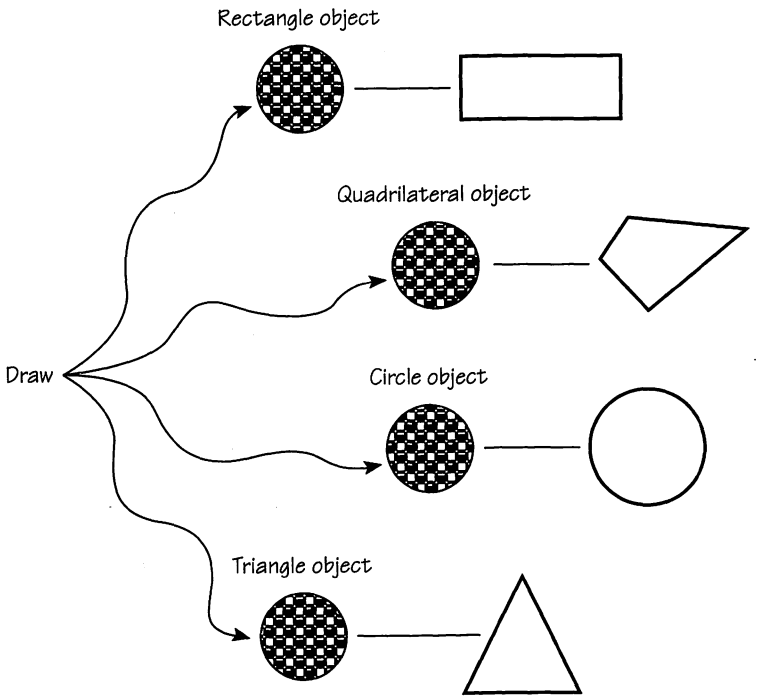


Figure 2-15.
Polymorphism. Sending the same message (Draw) to different shapes produces different results.

Still another example of polymorphism

An extension of our food example further illustrates the value of polymorphism. Suppose we declare an array of *TFood*, just as we can declare an array of any other Pascal type:

```
var
  dinner:  array[1..3] of TFood;
```

Now, if *TFood* has the descendants *TPear*, *THamburger*, and *TApple*, we can store objects of those types in the array:

```
var
  aPear: TPear;                { Descendant of TFruit }
  aHamburger: THamburger;      { Descendant of TFood }
  anApple: TApple;             { Descendant of TFruit }
:
dinner[1] := aPear;
dinner[2] := aHamburger;
dinner[3] := anApple;
```

Figure 2-16 on the next page shows the array of foods.

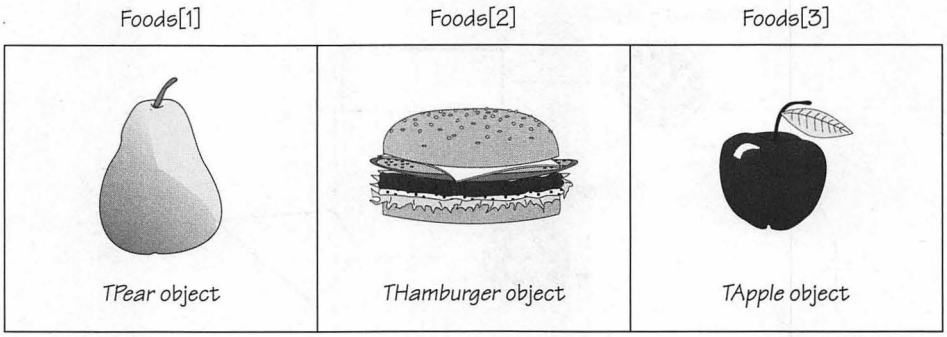


Figure 2-16.
A polymorphic array of various food objects.

Storing these objects of different types in the array is possible because we can assign objects of descendant classes to variables of their ancestor classes.

And when it's time to prepare dinner, we can simply move through the array, preparing each food item:

```
for i := 1 to 3 do
  dinner[i].Prepare;  { Send Prepare message to object in dinner[i] }
```

Each object stored in the array will receive the Prepare message and invoke its own *Prepare* method. Each food will be prepared to perfection, in the way a food of its type should be.

The Power of Polymorphism

Because of polymorphism, you can

- Store multiple, different (but related) types in a variable, an array, or another data structure
- Send the same message (call the same method) in a way that ensures the appropriate action for each item

We'll explore polymorphism more in Chapter 6 and again in Part 3. It's a powerful tool for programming.

Runtime Binding

If the compiler can't bind a method call "early," at compile time, the method call has to be bound "late," at runtime, when enough information is available.

Early Binding

When the compiler encounters an ordinary procedure or function call in your code, it already knows where that code is stored because it has already compiled—and

reserved space for—the procedure or function. Pascal's declaration rules ensure this. Encountering the procedure or function call causes the compiler to generate code that puts the parameters on the stack. The compiler then looks up the address of the procedure's code and generates a jump to that address, followed by code to retrieve any *var* parameters from the stack after the call. (That's the general idea, anyway.) This is called "compile time," or "early," binding. The compiler can bind the procedure call to the code's address at compile time.

Binding on the Fly—Late Binding

But we've seen that early binding won't work with polymorphic objects. In their case, binding—based on what object is actually "in" the object variable at the time—has to take place at runtime, not compile time. When *aFood* really refers to ("contains") *anApple*, the code generated by the compiler has to look at the apple object and find the apple object's *Prepare* method code. At runtime the program uses the actual object to locate the code for the method call.

Late Binding and Polymorphism

Runtime binding is actually what makes polymorphism possible. Other than having a general idea of what it is, though, you probably don't need detailed knowledge of what your particular compiler does to make runtime binding work. If you see how polymorphism works in general and what it can be used for, you're in business. We will look at one hypothetical mechanism for implementing runtime binding in Chapter 9.

Object-Oriented Design (OOD)

Objects are likely to change the way you design programs. Object-oriented design really breaks down into two kinds of design: designing object classes and designing applications that use objects. Object Pascal lets you use objects in your programs as little or as much as you want to. If you want to ease into objects, you can begin by using them sparingly, in small ways. The example programs in the next few chapters are relatively straightforward, simply using objects in otherwise more or less standard programs. Later in the book, our object-oriented programs will become very object oriented indeed.

OOP Class Design

Designing an object class depends to a considerable degree on how you plan to use it. Some classes are custom designs, to be used once and thrown away. We'll see some classes of this kind in the book. But many classes have the potential to be reused in other programs. Examples of reusable classes include the button class we develop in Chapter 4 and the dice class we develop in Part 2. Many programs need mouse-clickable buttons and dice, so we'll design those classes with generality

and reusability in mind. We can plug such classes into new programs with little or no change. If any change is needed, we'll typically accomplish it by subclassing rather than by altering the original class. Sometimes we might not even have access to the source code for the original class, but, as we'll see, we can still subclass the original class for our own purposes.

Object class design includes decisions about what instance variables are needed, what methods are needed, how those methods are to perform, and what parameters the methods will need. Class design also includes decisions about whether to subclass an existing class or start from scratch.

Abstract classes

Some classes, such as *TObject* in our example earlier in this chapter, are designed primarily to be subclassed, not to have object instances themselves. Called "abstract classes," they serve as the top of an object hierarchy, passing along all the instance variables and methods they have to potential descendants. We'll see a number of other uses for such classes. In Chapter 6 we'll look at some ways to use abstract classes with polymorphic class design. And some of our example programs in Part 2 use abstract classes to help overcome some Pascal scoping problems. We'll see abstract classes throughout the book.

OOP Application Design

When you use classes in an application design, your biggest decision is how object-oriented the application is to be. The game example in Part 2 and the linked-list software component in Part 3 are highly object-oriented applications. In Apple's MacApp, object-oriented design makes the application framework extremely extensible. OOD in MacApp provides you with almost all of the Macintosh user interface. You add in only the individually functional parts of your application, the particular things you want the application to do over and above its Mac interface functions. You can also override parts of MacApp to alter the interface by adding your own features.

As you examine the requirements for a program, you need to decide what objects you need to design. In Chapter 10, we'll use a simple approach to finding the likely objects in our problem.

Once we find the objects, we'll need to ask ourselves several questions. What characteristics should be represented as instance variables? What behaviors or operations on the data should we implement as methods? Which objects can send each other messages, and should this communication be one way or two way? How can we use inheritance to tie objects together, reduce code bulk, build on top of tested code, and make objects reusable? How are the objects in the system to be integrated into a whole? How can we test not only the individual objects but also the system? These are some of the most important questions to answer as we design an OOP application.

Object-oriented design tends to be data-centered. In traditional programs, data exists only to be processed by your algorithms. In OOP applications, the data can be the main source of the objects in a program. For example, from traditional data processing applications such as payroll and inventory programs, employees, departments, accounts, and inventory items are all likely candidates to become objects. An employee object, for instance, would carry the usual employee information in its instance variables and would be able to compute salary and benefits by means of its methods. Such an approach isn't far from using employee records—but now the "records" know how to process their own data. Or, to put it another way, the data knows how to process itself.

Object-oriented design and program construction also tend to be done from the bottom up. Traditional programs are more likely to be done from the top down, at least primarily. Object-oriented designers often begin with the objects they need, then consider the architecture required to integrate the objects, and then develop the objects one or two at a time, testing them with driver programs as they go. That's precisely the approach we'll take with the sample application in Part 2. In Chapter 10 we'll take a detailed look at the bottom-up approach.

Summary

Objects are software models, usually of real-world objects. Typically, an OOP programmer discovers a program's objects in its data, although there are other sources. In an important sense, the notion of objects is really only an extension of the structured programming concepts that have become dominant in the last 20 years.

An object is declared in the form of a class, which specifies the object's heritage (ancestry), its instance variables (data fields), and its method headings (procedure and function headings). An actual object is "instantiated" from the class by declaring a variable of the class (type), calling Pascal's *New* procedure, and initializing the variable. The resulting object can then send and receive messages, that is, call the methods of other objects and have its own methods called. When an object receives a message corresponding to one of its methods, it responds by invoking the method.

We can derive a new object class from another class by subclassing. By naming its immediate ancestor, a new class inherits all the instance variables and methods of the ancestor—and of the ancestor's ancestors (unless an intervening ancestor has overridden ancestor methods). An object hierarchy consists of a chain of ancestor classes ending in a final descendant object class. Descendants can always be added to the hierarchy. A new descendant class can extend its inheritance by adding new instance variables and methods. It can customize its inheritance so that it becomes a more specialized subclass by overriding methods it inherits. These subclassing techniques make objects highly extensible and reusable, which is their chief appeal to programmers and software development managers.

Polymorphism is the principle that the same message can be sent to different objects, and that, as long as all of those objects have corresponding methods, we can count on the objects' behaving correctly, meaningfully, and individually. If all the objects in a list have *Draw* methods, for example, we can send them Draw messages. Each object will respond correctly by drawing itself according to its own drawing rules. Thus, "Draw" means different things to different objects. Polymorphism is made possible by runtime, or "late," binding of procedure calls to the addresses of theprocedure code and by our ability to assign descendant objects to variables of their ancestor classes.

So far, of course, we haven't put an object into action in a real program. It's time for several examples. The first, in Chapter 3, illustrates basic object syntax and usage. The second, in Chapters 4 and 5, illustrates subclassing, inheritance, and overriding. And the last, in Chapter 6, illustrates polymorphism and runtime binding. Advanced readers might prefer to scan the next few chapters and move on to Chapters 7 through 9. You'll find the complete syntax of Object Pascal in Chapter 9.

3

RECORDS PLUS PROCEDURES EQUALS OBJECTS

The little play on Niklaus Wirth's *Algorithms Plus Data Structures Equals Programs* (Wirth 1975) in the title of this chapter exploits the similarity of Object Pascal object types (classes) to standard Pascal record types. Much of the focus in this chapter is on the similarities between conventional Pascal and Object Pascal. We'll shift the focus toward their differences in later chapters.

In this chapter, we'll put some of the concepts we looked at in Chapter 2 to work in some simple object-oriented programs. We'll see

- A full object class declaration and implementations of the class's methods
- The declaration, creation, initialization, and use of objects
- One use of the *self* pseudovariable
- Class declarations in a main program and in a separate unit
- Use of a class as an abstract data type
- Subclassing a new class from an existing one

We're going to contrast an object-oriented example with a standard Pascal example that does the same job. In all, we'll look at three programs:

- Spheres I, a standard Pascal program that computes and prints the circumferences, surface areas, and volumes of three separate spheres.
- Spheres II, an Object Pascal program that does the same thing. The class *TSphere* is declared in the declaration part of the program.
- Spheres III, an Object Pascal program in which the *TSphere* class declaration has been moved to a separately compiled Pascal unit.

Key differences among the three Spheres programs include different ways of declaring the *Sphere* data type and different ways of handling the operations on spheres.

Standard Pascal: Program Spheres I

Spheres I, shown in Listing 3-1, is an extremely simple standard Pascal program so that you can focus on what's happening syntactically rather than on what the program does. In the program, we define an *Init* procedure, three computation functions, and a *Display* procedure. The procedures and functions work with data stored in a variable of the record type *Sphere*. The program initializes three *Sphere* variables, computes the spheres' circumferences, areas, and volumes, prints out these values, and quits.

In its simplicity, Spheres I points out the syntactic and functional differences between ordinary Pascal programming and object-oriented Pascal programming as we'll practice it in the Spheres II and Spheres III programs.

```
program SpheresI;                { File is SpheresI.p }

  const
    pi = 3.1415926535;           { A constant value of type Real }

  type
    Sphere = record              { Record defining a sphere }
        radius: Real;            {  data type }
        center: Point;
      end;  { Type Sphere--a record type, not a class }

  var
    sphere1, sphere2, sphere3: Sphere;  { Three spheres }
    window: Rect;                { THINK Pascal Text window rectangle }

    { Note:  The spheres are declared statically, not dynamically }
    { These are ordinary record-type variables, unlike the handles }
    {  that reference dynamically allocated objects }

  procedure Init (var theSphere: Sphere; r: Real; ctrH, ctrV: Integer);
  begin
    theSphere.radius := r;
    theSphere.center.h := ctrH;
    theSphere.center.v := ctrV;
  end;  { Init }
```

Listing 3-1. *(continued)*
Spheres I, a standard Pascal program.

Listing 3-1. *continued*

```
function Circumference (ofSphere: Sphere): Real;
begin
  Circumference := 2 * pi * ofSphere.radius;
end;  { Circumference }

function Area (ofSphere: Sphere): Real;
  var
    r: Real;
begin
  r := ofSphere.radius;
  Area := 4.0 * pi * r * r;
end;  { Area }

function Volume (ofSphere: Sphere): Real;
  var
    r: Real;
begin
  r := ofSphere.radius;
  Volume := (4.0 * pi * r * r * r) / 3.0;
end;  { Volume }

procedure Display (theSphere: Sphere);
begin
  Writeln('  Circumference is    ', Circumference(theSphere));
  Writeln('  Area is             ', Area(theSphere));
  Writeln('  Volume is           ', Volume(theSphere));
  Writeln;
end;  { Display }

begin { Spheres1 }
  { Set up THINK Pascal Text window for display }
  SetRect(window, 5, 40, 400, 300);
  SetTextRect(window);
  ShowText;
  InitCursor;

  { Set all the fields of sphere1's data record }
  Init(sphere1, 1.0, 100, 100);          { Using Init procedure }

  { Display sphere1's information }
  Writeln('Sphere 1:  ');
  Display(sphere1);
```

(continued)

Listing 3-1. *continued*

```
    { Set all the fields of sphere2's data record }
    sphere2.radius := 2.0;              { Using dot-notation field access }
    sphere2.center.h := 150;
    sphere2.center.v := 150;

    { Display sphere2's information }
    Writeln('Sphere 2:  ');
    Display(sphere2);

    { Set all the fields of sphere3's data record }
    Init(sphere3, 3.0, 50, 50);

    { Display sphere3's information }
    Writeln('Sphere 3:  ');
    Display(sphere3);

    Writeln('End of SpheresI Demo');

    { Let user read the output }
    repeat
    until Button;
end.  { SpheresI }
```

Let's note a few things about program Spheres I before we go on to its OOP counterparts.

- All of Spheres I has to "know" how the circumferences, areas, and volumes of spheres are computed. In Spheres I the data knowledge is right up front, in functions known at the global level.
- Again, *Sphere* is a record type in Spheres I.
- The three *Sphere* variables are ordinary static variables, not dynamically allocated pointers (or object references).
- Each procedure or function requires a parameter identifying which sphere it will access.

Object Pascal: Programs Spheres II and Spheres III

Using objects, Spheres II and Spheres III will accomplish the same results as Spheres I. Figure 3-1 shows the difference between a record type declaration and an object class declaration for a sphere data type.

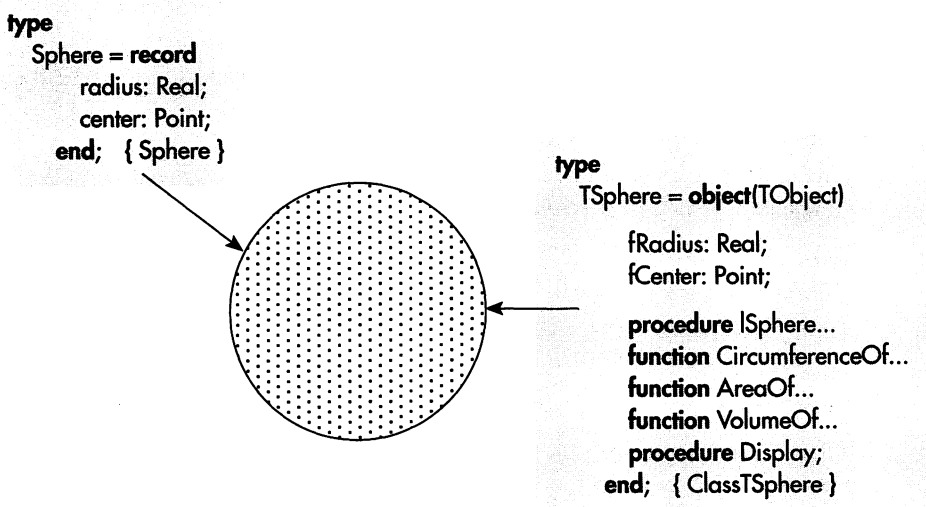


Figure 3-1.
A record type declaration and an object class declaration—sphere type declarations without and with objects.

The only difference between Spheres II and Spheres III lies in where the *TSphere* object class is declared. In Spheres II, class *TSphere* is declared in the program, with its method bodies declared just below it in the same declaration part:

```
program SpheresII;

  uses
    ObjIntf;                    { Provides class TObject }

  type
    TSphere = object(TObject)   { The class declaration }
        :
      end;  { Class TSphere }

  { The method bodies: }
  procedure TSphere.ISphere;    { Note the prefix TSphere }
  begin
    :
  end;  { TSphere.ISphere }

  :

  { Other method bodies }

begin { Main program }

  :

end.  { SpheresII }
```

Spheres III moves the declaration and the implementation of *TSphere*'s methods into a Pascal unit:

```
unit USphere;

interface                          { The unit's interface part }

  uses
    ObjIntf;                       { Provides the class TObject }

  type
    TSphere = object(TObject)      { The class declaration }
        ⋮
      end;  { Class TSphere }

implementation                     { The unit's implementation part }

  { The method bodies: }
  procedure TSphere.ISphere;       { Note the prefix TSphere }
  begin
    ⋮
  end;  { TSphere.ISphere }

  ⋮

  { Other method bodies }

end.  { USpheres }
```

Because Spheres II and Spheres III are so small and simple, determining what objects to design is easy. All we need is a class of sphere objects. In larger programs, of course, we usually need many objects. In Chapter 10, we'll look at how to determine what objects we need.

Designing a Sphere Class

To design an object class, we need to consider the objects' characteristics and behaviors. In the class declaration we represent characteristics as instance variables and behaviors as methods.

What are the characteristics of spheres? When we use the term "sphere" here, we mean a mathematical sphere. Spheres in the real world—planets, basketballs, and ball bearings—have additional characteristics, but we're interested here in the mathematical characteristics of abstract spheres. All we plan to do with spheres is compute the characteristics we don't want to define outright as instance variables—the characteristics that can be derived from the basic data.

As the diagram in Figure 3-2 illustrates, a mathematical sphere has two essential characteristics: a radius and a location in three-dimensional space. We'll define the characteristics in relation to a plane rather than in three-dimensional space, using

Macintosh screen coordinates to specify their locations. We'll use two instance variables for those characteristics: a type *Real* for the radius and a type *Point* (a QuickDraw type) for the sphere's center point.

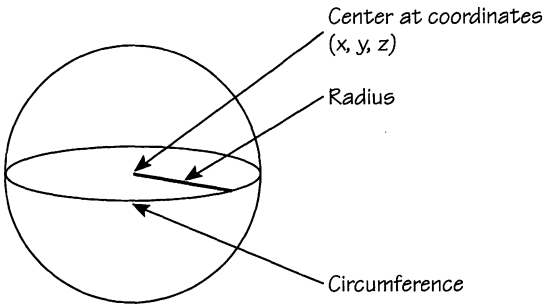


Figure 3-2.
Essential sphere characteristics— radius and location.

The *TSphere* Class Declaration

For Spheres II and Spheres III, we declare our *TSphere* data type as an object type rather than as a record type. Let's make a direct comparison of the two declarations:

```
type
  Sphere = record              { Record defining a sphere }
      radius: Real;            {  data type }
      center: Point;
    end; { Type Sphere--a record type, not a class }
```

and

```
type
  TSphere = object(TObject)  { Note heritage of this class }

      fRadius: Real;           { Stores sphere's radius }
      fCenter: Point;          { Stores sphere's location }

      procedure ISphere (radius: Real; h, v: Integer);
        { Initializes a sphere object }
      function CircumferenceOf: Real;
        { Returns the current circumference of a sphere object }
      function AreaOf: Real;
        { Returns the current area of a sphere object }
      function VolumeOf: Real;
        { Returns the current volume of a sphere object }
      procedure Display;
        { Prints the current radius, location, and function results for a sphere object }
    end;  { Class TSphere }
```

The *TSphere* object type declaration is for Spheres II. Note that because the class is declared in the Spheres II program instead of in a unit, it's not reusable by other programs (short of cutting and pasting the code), so we include only the methods this program will use directly.

In Spheres III, by declaring the *TSphere* class in a separate unit, we make the class reusable. Anticipating future reuse, we want to make it more complete than we actually need to for direct use in this one program. In the *TSphere* declaration for Spheres III, we add two methods for completeness:

```
function RadiusOf: Real;
function CenterOf: Point;
```

Before we go on, let's note a few salient points about our record type declaration and our object class declarations:

- *TSphere*'s declarations use the keyword *object* rather than *record*.
- *TSphere* declares exactly the same two data fields as type *Sphere* does except that we've prefixed the field names with *f*, according to our Object Pascal naming convention.
- *TSphere* declares exactly the same five procedures and functions that *Sphere* does except that they are declared inside the *TSphere* declaration rather than below it, as they are with *Sphere*.
- In Spheres III, *TSphere* declares two new, immediately unnecessary methods for getting the values of a sphere object's instance variables. It does this for completeness and to lend itself to encapsulation.
- *TSphere*'s methods are declared in the same way that the procedures and functions associated with *Sphere* are except that they have one fewer parameter—they don't include a parameter for identifying the sphere that the procedure or function is to operate on.

As we'll see, the method bodies of *TSphere* are quite similar to the implementation of *Sphere*'s associated procedures and functions. *Sphere*'s procedure implementations aren't prefixed with a type name because the association between type *Sphere* and its operations is a matter of tacit convention, not syntax.

Inside *TSphere*'s method bodies we can refer to instance variables directly, without qualifying them as we must references in procedure and function implementations to *Sphere*'s data fields. (Our *Sphere* record type could accomplish this qualification by dot notation or by using *with*.)

Now let's look at Spheres II and Spheres III in more detail. Instead of dealing with the full source code for Spheres II, we'll look at the class declaration only. The method bodies are implemented identically to those in Spheres III, which we will show completely, except that Spheres II puts the method bodies in the program's

declaration part instead of in a unit's implementation part and that we've added a couple of extra methods to Spheres III for completeness. You can see the full source code for Spheres II in the folder Spheres, Ch 3 on the code disk.

Program Spheres II

Here's the class declaration for *TSphere* as it appears in Spheres II:

```
type
  TSphere = object(TObject)  { Note heritage of this class }

      fRadius: Real;          { Stores sphere's radius }
      fCenter: Point;         { Stores sphere's location }

      procedure ISphere (radius: Real; h, v: Integer);
        { Initializes a sphere object }
      function CircumferenceOf: Real;
        { Returns the current circumference of a sphere object }
      function AreaOf: Real;
        { Returns the current area of a sphere object }
      function VolumeOf: Real;
        { Returns the current volume of a sphere object }
      procedure Display;
        { Prints the current radius, location, and function results }
        {  for a sphere object }
    end;  { Class TSphere }

    { Method bodies here ... }

begin { Main }
  :
```

The structure of Spheres II is a legitimate approach, but the unit structuring of Spheres III is preferable because we can import a class defined in a unit into any number of programs by means of their *uses* clauses. Classes defined in units are reusable; the same classes defined in main programs aren't.

Program Spheres III

Listing 3-2 on the next page shows the main program for Spheres III. It's followed immediately by Listing 3-3, the reusable *USphere* unit that defines the *TSphere* object type.

Spheres III main program

Listing 3-2 on the following pages is the main program for Spheres III.

```
program SpheresIII;      { File is SpheresIII.p }

  uses
    ObjIntf, USphere;  { Definitions of classes TObject and TSphere }

  var
    sphere1, sphere2, sphere3: TSphere;  { Type defined in unit USphere }
    h: Point;
    window: Rect;

begin { SpheresIII }
  { Set up THINK Pascal Text window for display }
  SetRect(window, 5, 40, 400, 300);
  SetTextRect(window);
  ShowText;
  InitCursor;

  New(sphere1);
  sphere1.ISphere(1.0, 100, 100);

  { View sphere1's information using the Display method }
  Writeln('Display sphere1: ');
  sphere1.Display;

  New(sphere2);
  sphere2.ISphere(2.0, 150, 150);

  { View sphere2's information using the Display method }
  Writeln('Display sphere2:  ');
  sphere2.Display;

  New(sphere3);
  sphere3.ISphere(3.0, 50, 50);

  { View sphere3's information using the Display method }
  Writeln('Display sphere3:  ');
  sphere3.Display;

  Writeln('End of SpheresIII Demo');
```

Listing 3-2. *(continued)*
The main program for Spheres III. Object types are declared and implemented separately from the main program.

Listing 3-2. *continued*

```
    { Free the three sphere objects }
    sphere1.Free;
    sphere2.Free;
    sphere3.Free;

    { Wait for user to read output }
    repeat
    until Button;
  end.  { SpheresIII }
```

Spheres III unit *USphere*

Listing 3-3 shows the full *USphere* unit in Spheres III that declares and implements object type *TSphere*.

```
unit USphere;   { File is USphere.p }

interface

  uses
    ObjIntf;      { Defines TObject }

{ Here we declare the TSphere class; its method bodies are postponed }
{  until the implementation part below }
{ We could also declare other classes, constants, types, variables, }
{  procedures, and functions in this interface part }

  type
    TSphere = object(TObject)     { Note heritage of this class }

        { Instance variables }
        fRadius: Real;            { Stores sphere's radius }
        fCenter: Point;           { Stores sphere's location }

        { Methods--the method names include the optional class name prefix, }
        {  as in "procedure TSphere.Display;" }
        procedure TSphere.ISphere (radius: Real; h, v: Integer);
          { Initializes a sphere object }
```

Listing 3-3. *(continued)*

The unit USphere *for Spheres III. Declared and implemented in a unit separate from the main program, the object type* TSphere *is reusable.*

Listing 3-3. *continued*

```
    function TSphere.RadiusOf: Real;
      { Returns the current radius of a sphere object }
    function TSphere.CenterOf: Point;
      { Returns the current location of a sphere object }
    function TSphere.CircumferenceOf: Real;
      { Returns the current circumference of a sphere object }
    function TSphere.AreaOf: Real;
      { Returns the current area of a sphere object }
    function TSphere.VolumeOf: Real;
      { Returns the current volume of a sphere object }
    procedure TSphere.Display;
      { Prints the current radius, location, and function results for a sphere
        object }

  { Note:  The class also inherits all of class TObject's methods }
  end;  { Class TSphere }

implementation

{ Here we implement TSphere's method bodies, plus any constants, types, }
{  variables, or supporting procedures or functions used by the methods }

  const
    pi = 3.1415926535;        { No need to declare this publicly }

{ The method names include the optional class name, parameter }
{  lists, and function return types }
{ This is good practice and makes preparing files easier, although }
{  you are free to omit all but the keyword procedure or function and }
{  the method name }

  procedure TSphere.ISphere (radius: Real; h, v: Integer);
    { Note the name ISphere instead of Init }
  begin
    self.fRadius := radius;       { Assign parameters to instance variables }
    self.fCenter.h := h;
    self.fCenter.v := v;
  end;  { TSphere.ISphere }

  function TSphere.RadiusOf: Real;
  begin
    RadiusOf := self.fRadius;
  end;  { TSphere.RadiusOf }
```

(continued)

Listing 3-3. *continued*

```
function TSphere.CenterOf: Point;
begin
  CenterOf := self.fCenter;
end;  { TSphere.CenterOf }

function TSphere.CircumferenceOf: Real;
begin
  CircumferenceOf := 2.0 * pi * self.fRadius;
end;  { TSphere.CircumferenceOf }

function TSphere.AreaOf: Real;
  var
    r: Real;
begin
  r := self.fRadius;
  AreaOf := 4.0 * pi * r * r;
end;  { TSphere.AreaOf }

function TSphere.VolumeOf: Real;
  var
    r: Real;
begin
  r := self.fRadius;      { Get the value of this object's instance variable }
  VolumeOf := (4.0 * pi * r * r * r) / 3.0;
end;  { TSphere.VolumeOf }

procedure TSphere.Display;
begin
  Writeln('  Center:          ', self.fCenter.v, ' , ', self.fCenter.h);
  Writeln('  Radius:          ', self.fRadius);
  Writeln('  Circumference:   ', self.CircumferenceOf);
  Writeln('  Area:            ', self.AreaOf);
  Writeln('  Volume:          ', self.VolumeOf);
  Writeln;
end;  { TSphere.Display }
end.  { Unit USphere II }
```

Again, the only real difference between Spheres II and Spheres III is that in Spheres III *TSphere* has been declared in a unit, for reusability. Any client program, such as our Spheres III, can import it with a *uses* clause.

Self in the method code

Note that, inside *TSphere*'s methods, each time we referred to an instance variable or a method of *TSphere*, we prefixed its name with *self*. That usage is utterly optional but useful because it helps to distinguish the object's own components, especially method names, from other variables. Using *self* improves the clarity of code. We'll see other uses for *self* later.

Comparison of Spheres I and Spheres III

On the face of it, Spheres I and Spheres III look and work much alike. (The similarity is even more striking if you compare Spheres I and Spheres II, where the *TSphere* declaration is in the main program, much as the *Sphere* type declaration and procedures and functions in Spheres I are in the main program. We could make Spheres I more like Spheres III by putting its type declaration and operations in a unit.)

For a program as small as Spheres I, the conventional code might be preferable. Code bulk for the compiled program is a little less than for Spheres III, and Spheres I executes fewer instructions than Spheres III. Spheres III might have the extra overhead of looking up methods in a dispatch table (as we'll see in Chapter 9). And you're used to coding in standard Pascal, so why change?

Spheres III, the OOP approach, does offer some advantages:

- The sphere type in Spheres III is much more reusable. You can import the type and its support into any program. You can also subclass *TSphere* to get spheres with more or different capabilities; for instance, you might use it as the basis for classes of balls, planetary models, or soap bubbles.
- Spheres III is superior at information hiding. Data and operations are encapsulated in the same structure instead of spread out all over the place. (That's less a virtue in a small program like this one than in a large one, but as soon as a program begins to grow, you'll be grateful for encapsulation.) Spheres are easy to create and use, and the outside consumer doesn't have to know anything about their innards. The consumer can simply "ask" a sphere to do its tricks.
- Spheres III is a more natural way (once you get used to the new syntax) to model real-world spheres in software. OOP really does make programming easier (and more fun).

Syntax Notes

The biggest differences between standard Pascal syntax and Object Pascal syntax are in the way we declare the sphere data type and the way we make calls that affect our spheres. We'll cover the *TSphere* declaration syntax here and the syntax of calling object methods in the next section.

Syntax for OOP type declaration

Superficially, type *Sphere* in Spheres I and type *TSphere* in Spheres III look much alike. Type *Sphere* is a standard Pascal record structure of the form

```
type
  Name = record
      dataField1: ItsType;
      dataField2: ItsType;
      ⋮
    end;
```

Of course, a Pascal record can contain only data fields of any type or types, including nested record types, pointers, arrays, sets, packed types, enumerated types, and so on; a record can also have a fixed part and a variant part. A Pascal record can't contain methods (procedures and functions).

Type *TSphere*, on the other hand, contains both instance variables and method headers. The instance variables follow the same rules as data fields for records do. The object class declaration takes the form

```
type
  Name = object(optionalAncestorName)

      instVar1: ItsType;
      instVar2: ItsType;
      ⋮

      MethodHeader1 (optionalParameters);
      MethodHeader2 (optionalParameters);
      ⋮
    end;
```

The optional ancestor name following the *object* keyword (in parentheses) lets the class inherit the instance variables and methods of the ancestor. We'll look at inheritance again in Chapter 4. Remember that this is an object class, not the object itself.

The method headers can be headers of procedures or functions. Of course, function methods must give the return type after the (optional) parameters and a colon. Although we call the operations "methods" here, they follow the normal rules for Pascal procedures and functions.

We might note also that the *TSphere* class declaration amounts to the interface part of an abstract data type (ADT) definition. It lists the data type and the methods or operations that can be performed on the type. The implementation part of the ADT is somewhere else; in the case of Spheres II, the method implementations (bodies) are just below the class declaration. The method bodies are presented exactly as nested procedures and functions in the declaration part of a standard Pascal program or subprogram are.

One important syntactic point that sets the declaration of Object Pascal methods apart from the declaration of ordinary Pascal procedures and functions is that inside the object class declaration, we can list the methods in either of two ways:

```
procedure Method1(parameters);
```

or

```
procedure TSphere.Method2(parameters);
```

The form for *Method2* puts the class name, *TSphere*, in front of the method name, with a dot to separate them. This is entirely optional inside the class declaration, as the form of *Method1* shows. The dot notation qualifies the method name with its associated class. Qualifying the method name is, you'll recall, required when you implement the method body—to associate the method with its class.

Syntax for OOP method calling

In Smalltalk-style OOP parlance, we "send messages" to objects in order to have them invoke their methods. This is like saying to a sphere, "tell me your circumference, please." The object knows what the circumference message means and invokes its circumference method to respond to the message.

Spheres II sends messages (calls methods) in two places: inside the *Display* method, where we send the CircumferenceOf, AreaOf, and VolumeOf messages to *self*; and in the body of the main program, where we send initialization and display messages to the sphere objects after we create them with *New*.

The syntax of a method call (or message send) resembles the syntax of a record component access in standard Pascal:

```
recordName.ComponentFieldName;              { Record component access }
```

and

```
objectName.Message(parameters);             { Procedure method call }
```

or

```
variable := objectName.Message(parameters); { Function method call }
```

We follow the object name with a dot and then the method name. The method name is simply the name of a procedure or function in the method list of the object class declaration. If the method requires any parameters, we have to supply them in the call. A function method works just as an ordinary Pascal function does—it returns a value of some specified type. We can use that value in an assignment:

```
Answer1 := sphere1.VolumeOf;
```

or in an expression:

```
Answer1 := 1.0 + (3.0 * pi) - sphere1.VolumeOf;
```

For a summary of the procedures for declaring, initializing, using, assigning, and disposing of objects, see Chapter 7.

Why Isn't There a *New* Method?

Why don't we simply declare a *New* method, as some languages let us do, and call it to create an object? In Object Pascal, that would be trying to send a message to an object that didn't exist yet. There would be no link yet to the method's code. In Object Pascal we have to create the object with Pascal's *New* procedure, as if the object were a pointer. Then we can start sending the object messages—initialization, display, and data-related messages, for instance. The newly created object is then ready to respond with its methods.

We could write a *NewObjectType* function or procedure *outside* the object class declaration and then call it to create and initialize the object for us in one neat call. The call might look like this one:

```
mySphere := NewSphere(3.0, 100, 100); { A function--not a method }
```

which internally calls *New* to create a temporary *TSphere* variable, initializes it, and passes back the object reference to it as the function result:

```
function NewSphere(radius : Real; centerV, centerH : Integer): TSphere;
  var
    tempSphere: TSphere;
begin
  New(tempSphere);                { Creates a new sphere and initializes it }
  tempSphere.ISphere(radius, centerV, centerH);
  NewSphere := tempSphere;  { Passes it back as function result }
end;  { function NewSphere--not a method }
```

This ordinary function must be declared outside the class declaration, not inside it. It is not a method. We'll look again at this tidy approach to object creation and initialization later.

New Classes from Old—Subclassing *TSphere*

We'll develop the subclassing topic much more completely in Chapter 5, but now that we have class *TSphere*, let's look briefly at how to subclass it to make descendants of *TSphere*. Let's sketch out a class *TBall*.

In the real world, a ball *is-a* sphere. So, too, in our OOP world. Balls have the general properties of spheres plus additional properties of their own.

Our class *TBall* is marked (for the compiler) as a descendant of *TSphere* by our declaring *TSphere* as *TBall*'s immediate ancestor in the heritage spot in *TBall*'s declaration:

```
type
  TBall = object(TSphere)
```

Because of this declaration, *TBall* inherits all of *TSphere*'s instance variables and methods. Of course, *TBall* can also override any of *TSphere*'s methods that aren't quite right, but it can't override *TSphere*'s instance variables or eliminate inherited instance variables and methods that it doesn't want.

A ball is a sphere, so of course we can talk about its radius, location, circumference, and so forth. But balls have additional properties. For example, balls can bounce—so we need some measurement of how "bouncy" a given ball object is. Balls can also have colors and patterns on their surfaces. And they are different sizes, textures, weights, and so forth, depending on what they're to be used for. And, of course, some balls, such as footballs, aren't spheres, so our classification isn't complete.

Here's a rough class declaration for *TBall*:

```
type
  TBall = object(TSphere)

      fColor: Colors;
      fPattern: Patterns;
      fBounciness: Real;
      { Also inherits fRadius and fCenter }

      procedure IBall(radius: Real; v, h: Integer; color: Colors; pat:
         Patterns; bounciness: Real);
      function ColorOf: Colors;
      function PatternOf: Patterns;
      function BouncinessOf: Real;
      procedure Display;
      override;
      procedure Drop;
      procedure Throw;
      procedure Kick;
      { Also inherits CircumferenceOf, AreaOf, }
      {  and VolumeOf }
    end;  { Class TBall }
```

Notice all the characteristics and methods that *TBall* inherits from *TSphere*. But notice, too, the override of *TSphere*'s *Display* method. Spheres are abstract, mathematical objects, but balls are real-world physical objects. Displaying a ball with pattern and color is going to be different from displaying a sphere. As illustrated in Figure 3-3, *TBall* inherits from, overrides, and extends its ancestor class *TSphere*. Besides, in *TSphere* we chose to display a sphere by printing out its characteristics rather than drawing it. So we have *TBall* override *Display* so that *TBall* can display differently.

We'll see many examples of subclassing and overriding in the chapters to come.

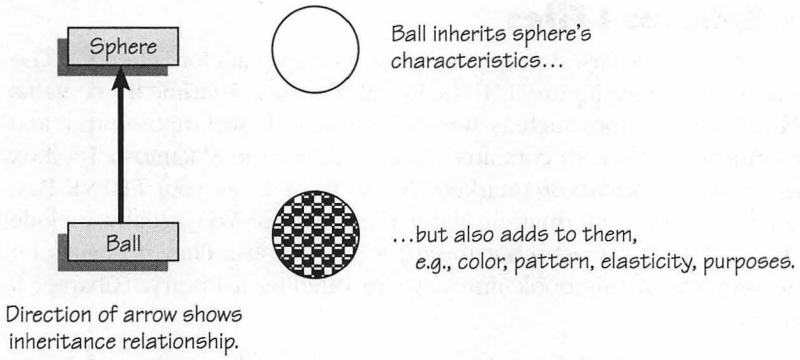


Figure 3-3.
TBall's *inheritance from and extension of* TSphere.

Compiling the Examples

Essentially, compiling object-oriented programs is no different from compiling ordinary Pascal programs, especially those that use units.

To compile the example programs in this chapter and throughout the book, see the discussion of compiling in Chapter 7 and consult the documentation for your Object Pascal system. In MPW Pascal or TML Pascal, you use a line command something like

Pascal [*options*] *filename(s)*

where *options* stands for compiler switch options. In Chapter 7 and Appendix B you'll find more information about compiling the programs in this book in MPW.

In THINK Pascal, you create a "project" (whose name by convention ends in the pi symbol), arrange the files in the correct order in the project window, as illustrated by the Spheres I files shown in Figure 3-4, pull down the Run menu, and choose Go.

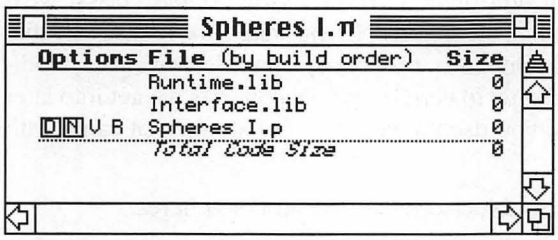


Figure 3-4.
THINK Pascal project window for Spheres I.

About the Spheres I Files

Figure 3-4 shows the THINK Pascal project window as it appears for Spheres I. The files Runtime.lib and Interface.lib are THINK Pascal libraries. Runtime.lib contains code for standard Pascal routines such as *Writeln* and other Pascal input/output and file-handling routines. Interface.lib contains "glue" code for the Macintosh Toolbox routines defined in *Inside Macintosh* (marked "Not in ROM"). See your THINK Pascal documentation for more information about the libraries. You usually include these files in your THINK projects, but there are exceptions. They'll always be included in the projects in this book unless we're building a library. (Chapter 9 discusses libraries.)

The file SpheresI.p contains the Spheres I program's source code.

About the Spheres II Files

The THINK Pascal project window shown in Figure 3-5 contains some of the same files as the window shown in Figure 3-1, with one addition and one change.

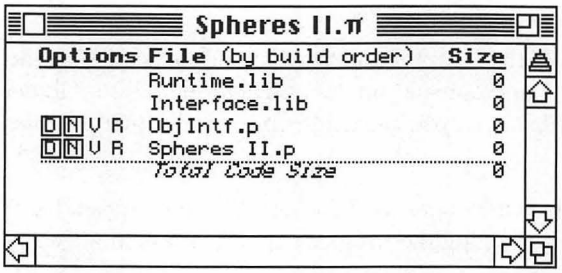


Figure 3-5.
THINK Pascal project window for Spheres II.

The file ObjIntf.p contains a Pascal unit defining the reusable *TObject* class. We'll look at *TObject* again in Chapter 8. For Spheres II, be sure *not* to use the file ObjIntf.px on the code disk that accompanies this book. That file contains a modified version of ObjIntf.p and is used only in certain special contexts we get into later in the book. For Spheres II, be sure you use the version of ObjIntf.p that came with THINK Pascal.

The file SpheresII.p contains our first object-oriented version of Spheres.

About the Spheres III Files

Figure 3-6, which shows the project window for Spheres III, also lists ObjIntf.p. Use the version that came with THINK Pascal, not our special version.

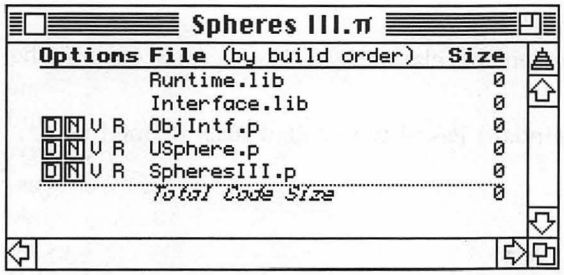


Figure 3-6.
THINK Pascal project window for Spheres III.

The file USphere.p contains a Pascal unit defining the *TSphere* class.

The file SpheresIII.p contains a Pascal program that lists unit *USphere* in its *uses* clause so that it can reference the *TSphere* instance variables and call class *TSphere*'s methods.

Summary

In this chapter, we wrote the same program in standard Pascal and in Object Pascal. The Object Pascal code in program Spheres III doesn't draw on everything we've learned so far about objects—we'll see more in Chapters 4 through 6—but we did declare and implement the object class *TSphere*. We also saw the sense in which this class is abstract when compared to the more concrete *TBall* class.

We saw one optional way to use the word *self* for clarity—here, to make it clear that the instance variables we access are in the object in which the access occurs.

We saw that object classes can be declared and implemented in the declaration part of a main program or in a Pascal unit and that the *USpheres* (unit) way of packaging the class *TSphere* is the preferred way to do it because that makes the class reusable.

We didn't consider it overtly as such, but we saw the class instantiated in an outsider program; that is, the outsider program declared several object variables, or instances, of type *TSphere*, created and initialized the objects, and sent them some messages. The objects "knew" how to respond by executing the methods corresponding to the messages they received.

Finally, we took a quick look at subclassing *TSphere* to create *TBall*, a new kind of object class that still had spherelike properties in addition to those it added to the *TSphere* ancestor.

Projects

- Write a unit that declares and implements class *TCube*. Write a small program like Spheres III to test *TCube*.
- Starting with the QuickDraw standard Pascal type called *Rect*, defined as a record,

```
type
  Rect = record

      case Integer of
        0: (top, left, bottom, right: Integer);
        1: (topLeft, botRight: Point);
  end;  { Rect }
```

  write a unit that declares and implements class *TRectangle* as a complete ADT (abstract data type), including an instance variable, *fRect*, of type *Rect*. Include methods to set and return *fRect*'s value. This unit will be useful in graphics programs in which you sometimes need a rectangle object and sometimes need a plain rectangle. Write a small program to test class *TRectangle*.

4

MACOBJECTS

Now that we've seen a simple example of object-oriented programming with Object Pascal, let's do something more interesting and useful. Let's develop the first of the book's many MacObjects. Typical MacObjects are buttons, windows, documents, and even "application" objects. Our first MacObject is an abstract class of Macintosh buttons from which we can derive classes for the kinds of buttons shown in Figure 4-1 on page 69. We'll see

- A good-sized object class
- Abstract classes
- The Macintosh interface put into classes
- Basic event handling

Before We Get Lost in the Details

TButton will be our first really useful and nontrivial example of an object class. The button class is a fair-sized, fairly complicated example—developing it will take some time and space. The big payoff comes in the next chapter, when we derive a number of useful classes from *TButton*.

TButton will also be our first serious example of subclassing an existing class to produce a new class and of the subclass's overriding some of its ancestor's methods. *TButton*'s immediate ancestor is an abstract class called *TEvtHandler*, which knows how to "handle" Macintosh events such as mouse clicks. Because it inherits from *TEvtHandler*, a *TButton* instance *is-a TEvtHandler*—an object that knows how to handle events. We'll look at *TEvtHandler* briefly here, just enough to get the idea, and develop it fully in Part 2.

Our Design Procedure

The button design we do in this chapter and the next is preliminary. It will work, but later we'll reconsider some of the decisions we make here in light of a larger library of other MacObjects. We'll look at that library in Part 2. In this chapter, we'll

- Survey the characteristics and behaviors we want in our buttons
- Look at the place of buttons in the larger object hierarchy
- Decide what instance variables (of what types) are needed to capture a button's characteristics
- Decide what methods are needed to implement a button's behaviors
- Decide how to package the button source code (in units)
- Think through the process of having a button "know" when the user has clicked it with the mouse
- Develop the ways buttons respond to mouse clicks and certain keypresses
- Look at how suitable our design will be as a base for creating other button subclasses

We'll deliberately keep the button class simple here because we're still far from ready to discuss all the intricacies of the other objects in the library of MacObjects.

General Design of the Button Class

In Macintosh terms, a button is a thing on the screen that the user activates by clicking with the mouse (or sometimes by pressing a key), usually to initiate some action. Buttons come in a variety of shapes and styles—some are shown in Figure 4-1. The standard Mac buttons include the familiar rounded-corner rectangle style, the radio button, and the check box. Other interesting button styles such as those available in HyperCard have evolved—rectangular buttons, transparent buttons, and iconic buttons. (A transparent button acts as an invisible button; it usually overlays a graphic. An iconic button displays an icon that the user can click.)

Standard Macintosh buttons are created and operated by means of Control Manager routines, but to keep the discussion a bit simpler, we'll design our buttons without recourse to the Control Manager. To see a button implemented by means of the Control Manager, look at class *CButton* and its subclasses *CCheckBox* and *CRadioButton* in the THINK Class Library and take a look at TCL's *CRadioGroup* class.

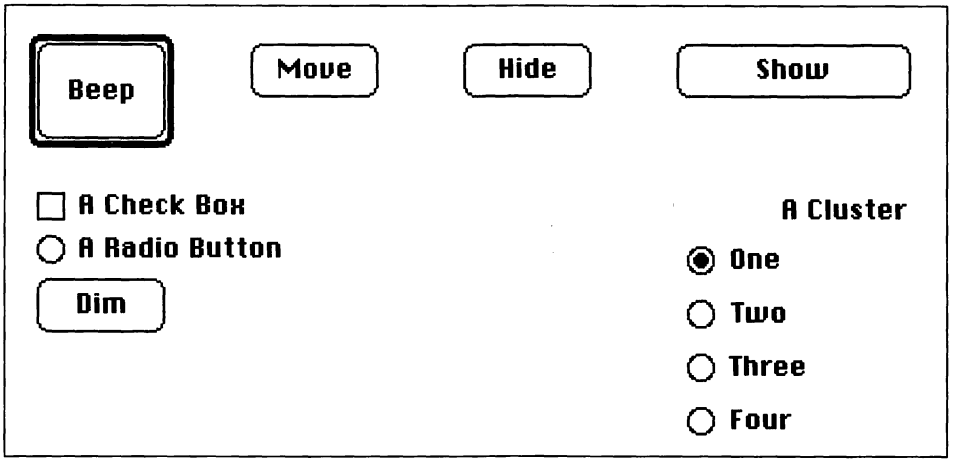


Figure 4-1.
Sample Macintosh buttons.

TButton's Characteristics

Buttons do two main things: They display themselves, and they respond characteristically when they detect a click. Let's describe the behavior of all buttons in a general way that anticipates the methods we'll create in the *TButton* class declaration and in its subclass declarations.

A Mac button should draw itself in the appropriate way. *TButton* displays the standard rounded rectangle, but some of its subclasses override *TButton*'s *Display* method to draw a different shape or style. A button can also hide itself, move itself to a new location, and "dim" itself.

An active button can respond to mouse clicks, performing the button's programmed action. An inactive button can't—and is dimmed so that the user can see its inactive status. An inactive button, one that can't be used in the current context, typically has its title dimmed in gray. We'll give all buttons the ability to dim, although dimming looks a little different for different button styles. *TButton* subclasses will dim in their own ways by overriding *TButton*'s dimming method.

If several buttons are present—as in a dialog box—we usually mark one as the default, or preset, button by having the program put a dark outline around it. The default button is the one the user should choose if in doubt—the Cancel button is often the default button. Our button class will make it possible to declare a button the default, although having a default button will be optional. A default button usually responds to a press of the Return or the Enter key as if the button had been clicked. This meaning of "default" won't apply to check box and radio button classes, which will override the default-setting methods of *TButton*.

The "Macintosh User Interface Guidelines" (*Inside Macintosh,* I-23) say that when a button is clicked, it should give the user visual feedback, usually highlighting in which every pixel in the button's display is inverted and then reinverted to normal. Note that check boxes and radio buttons are highlighted differently. If the mouse button is pressed while the mouse pointer is in a button, the button is highlighted; if the mouse pointer moves out of the button while the mouse button is still down, the button loses its highlight. If the mouse pointer moves back into the button while the mouse button is still down, the button is highlighted again. A button click is complete when the mouse button is released while the mouse pointer is in the button. If the mouse button is released when the mouse pointer is out of the button, the click doesn't count. *TButton*'s *Clicked* method tracks the mouse to provide this behavior.

When a button is clicked, it should itself be able to detect the click and respond by performing some characteristic action, such as beeping, hiding itself, computing the Gross National Product, or rolling some dice. A button receives an event; tests whether the event belongs to it and whether it's an event it should respond to; and then performs an action specified in a *DoClick* method. *DoClick*, of course, means something different for a check box, say, than for a radio button, so both the *TCheckBox* and the *TRadioButton* subclasses will, as we'll see, need to override *TButton*'s *DoClick* method.

TButton's Subclasses

TButton is an abstract class in that it defines most button characteristics and methods but is not meant to be instantiated with real objects. To use *TButton*, we'll subclass it and then declare instances of its subclasses.

The one thing we can't design into the general *TButton* class is what happens when a button gets clicked. Buttons respond in a wide variety of ways. Our strategy is to put into *TButton* all the general capabilities of buttons—instance variables and methods for showing, hiding, moving, detecting clicks, and so on—but to have specialized subclasses fill in the details of what action a button performs when clicked. In *TButton*, the *DoClick* method will remain a "stub," an empty, abstract method that descendants must override. *TButton* descendants might also override other *TButton* methods, especially those that have to do with display.

TButton actually has two kinds of descendants: those that override *DoClick* to make standard rounded-corner, rectangle-style Macintosh buttons perform particular actions and those that create new kinds of buttons such as check boxes, radio buttons, transparent buttons, and icon buttons. The *TCheckBox* and *TRadioButton* classes are ready to be instantiated with real objects. Others, such as *TTransparentButton* and *TIconButton*, must be further subclassed to override *DoClick*.

The TRadioCluster subclass

Because radio buttons are almost always grouped in "clusters," we'll develop a subclass *TRadioCluster*. Radio buttons in a cluster act the way selection buttons on a car radio do. If one button is on, the others must be off. Choosing a new button turns it

on and turns the rest off. Managing a cluster of radio buttons has always been a task that programmers had to do from scratch. We'll automate that task with a subclass that produces and manages cluster objects. We'll get to these subclasses in the next chapter.

TButton's Place in the Scheme of Things

Because *TButton* has ancestors and descendants, it's part of an object hierarchy. (See Chapter 8.) We want *TButton* to be a descendant of the abstract class *TObject*, of course, so that buttons can inherit *TObject*'s *Free* method in order to dispose of themselves when finished. And we're going to design another abstract class called *TEvtHandler*. *TEvtHandler*, whose name and concept we borrow from MacApp, is the class of all objects that know how to handle Macintosh events such as mouse-downs, mouse-ups, key-downs, and so forth. Any *TEvtHandler* descendant, including *TButton*, inherits instance variables and methods for processing events, although typically subclasses must override most of *TEvtHandler*'s methods.

In the interest of thoroughness, we could also create a class called *TControl*, the class of all control objects, which, in addition to buttons, might include scrollbars and perhaps menus. Our version of *TButton* doesn't use the Control Manager, however, so we'll skip *TControl*.

TButton's object class hierarchy

The object class hierarchy in which *TButton* participates is shown in Figure 4-2 as it appears in THINK Pascal's Class Browser.

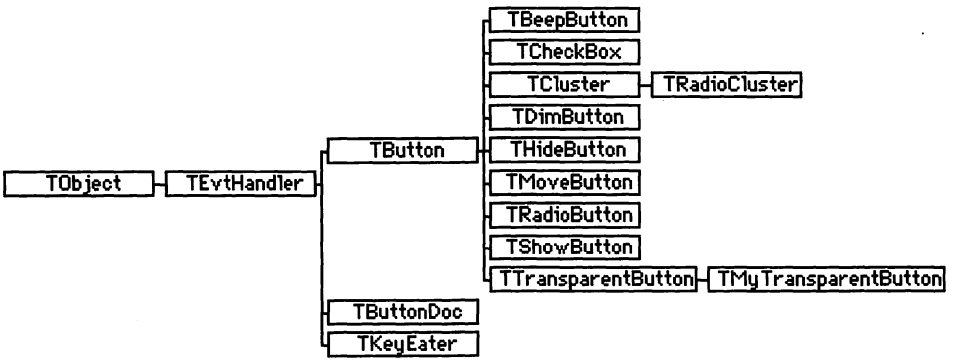


Figure 4-2.
TButton *in its object class hierarchy.*

TButton's is a bigger hierarchy than any we've seen so far. The hierarchy begins with *TObject* and *TEvtHandler*, abstract classes that exist only to pass capabilities down to their descendants. *TButton* itself has numerous descendants, some of which, such as *TRadioButton*, are special button types—nonabstract (instantiable)

button classes that override *DoClick*. (We'll worry about the classes *TButtonDoc* and *TKeyEater* a bit later.)

TEvtHandler declares *TObject* as its immediate ancestor for now (we'll change that later, as our class library grows):

```
type
  TEvtHandler = object(TObject)
```

Likewise, *TButton* declares *TEvtHandler* as its immediate ancestor (again, this ancestry will change later on), and each *TButton* subclass declares *TButton* as its immediate ancestor:

```
type
  TButton = object(TEvtHandler)
  :
  TBeepButton = object(TButton)
  :
  TRadioButton = object(TButton)
```

Because a class inherits from all of its ancestors, each *TButton* subclass is not only a *TButton* but also a *TEvtHandler* and a *TObject*, inheriting *TObject*'s *Clone* and *Free* methods and *TEvtHandler*'s methods as well as those of *TButton*.

Thinking about *TButton* in this context reminds us that *TButton* itself shares features with other object classes (with objects that can free themselves and with event-handler objects, for instance) and that, as an abstract class, *TButton* is itself designed solely to pass on its capabilities to its more specialized descendants.

TButton's Definition

Here's the *TButton* class declaration, including instance variables and methods for the capabilities of a button object that we've already looked at.

```
type
  TButton = object(TEvtHandler)

    fRect: Rect;                    { Button's display rectangle }
    fTitle: Str255;                 { Button's title }
    fActive: Boolean;               { Button clickable if true }
    fDefaultButton: Boolean;        { Button highlighted as default choice }
    fMustFrameDefault: Boolean;     { By default, button should be framed }
    fCornerWidth: Integer;          { Button corners rounded by these }
    fCornerHeight: Integer;         {   two values }

    { Initialization methods }
    procedure IButton (title: Str255; active: Boolean; r: Rect);
    procedure SetCornerRounding (width, height: Integer);
    procedure SetDisplayRect (r: Rect);
    procedure SetTitle (title: Str255);
```

```
    { Inquiry methods }
    function DisplayRectOf: Rect;
    function TitleOf: Str255;
    function IsActive: Boolean;
    function IsDefault: Boolean;
    function MustOutline: Boolean;

    { Display methods }
    procedure Display;
    procedure Hide (stayActive: Boolean);
    procedure Dim;
    procedure Hilite;
    procedure Activate (doActivate: Boolean);
    procedure MakeDefault (doMakeDefault: Boolean; frameIt: Boolean);
    procedure Move (toRect: Rect);

    { Event-handling methods }
    function Clicked (event: EventRecord): Boolean;
      { Clicked is a private method, not to be called by clients }
    function DoMouseCommand (event: EventRecord): Boolean;
    override;
    function DoKeyCommand (event: EventRecord): Boolean;
    override;
    procedure DoClick;              { Stub method--must override }
  end; { Class TButton }
```

TButton's Instance Variables

TButton's instance variables have mainly to do with display:

- So that it knows where to display itself on the Mac screen, we give a button the instance variable *fRect*, which holds the coordinates of a rectangle.
- Because most buttons display a title of some kind, we provide the instance variable *fTitle*.
- In *fActive*, we track a button's active status, meaning that it can respond to mouse clicks, or its inactive status, meaning that it can't respond and is therefore dimmed.
- In *fDefaultButton*, we record a button's default status. We might want to highlight the default button by drawing a heavy outline around it; if we do, we'll specify an outline with *fMustFrameDefault*.
- The standard button is drawn as a rounded-corner rectangle, so we provide for specifying how round to make the corners with the *fCornerWidth* and *fCornerHeight* instance variables.

Many of these fields are Boolean status flags.

TButton's Methods

We'll look at the design of *TButton*'s initialization, inquiry, and display methods first. Then we'll look at the event-handling methods *TButton* inherits from its ancestor *TEvtHandler* and round out *TButton*'s own event-handling methods.

Initialization methods

When you create a new button object you have to specify its title (used by most buttons), whether it will be active initially, and where it is to display itself (by default, in a rectangle occupying certain screen coordinates).

The initialization method, *TButton.IButton*, takes care of those details by means of parameters and sets the other instance variables to their default values. Here's the code for our *IButton* method:

```
procedure TButton.IButton (title: Str255; active: Boolean; r: Rect);
begin
  self.fTitle := title;
  self.fActive := active;
  self.fRect := r;
  self.fDefaultButton := false;
  self.fMustFrameDefault := false;
  self.fCornerWidth := 12;              { Default--can change with }
  self.fCornerHeight := 12;             {   the method SetCornerRounding }
  self.IEvtHandler(self);               { Initialize ancestor's instance variables }
end; { TButton.IButton }
```

Recall that the use of *self* is optional here. As most initialization methods do, this one simply sets the values of the instance variables, to either values passed as parameters or default values. The initialization method of a class that's in a hierarchy should also initialize its immediate ancestor's instance variables. In this case, *TEvtHandler* has an *IEvtHandler* method, so we simply send an IEvtHandler message.

By default, *IEvtHandler* sets *fNextHandler* to *nil*. If *fNextHandler* later acquires a successor in the event-handler chain (which we'll explain shortly), we can call *SetNextHandler* to make this button point to that next handler object. But it's important to initialize the instance variable now. The button can end up as the last object in the event-handler chain, and we want it to be marked as such with an *fNextHandler* instance variable set to *nil*.

We've also provided methods for setting (or resetting) most of those initialized instance variables independently:

```
procedure SetCornerRounding (width, height: Integer);
procedure SetDisplayRect (r: Rect);
procedure SetTitle(title: Str255);
{ And the inherited SetNextHandler, of course }
```

We'll skip the code for these methods, which is similar to *IButton*'s but sets only one or two instance variables. These methods let you reset the button's characteristics on the fly. A button's title could change at some point, for example, so you want a method to pass the new title to the button object. Then you have to send the Display message to the button again. That's true of *IButton* and all of these methods; they don't display the button for you.

Inquiry methods

Anticipating that a programmer might want to ask a button what its current title is or where it's displaying itself or about some other detail, *TButton* supplies several inquiry methods:

```
function DisplayRectOf: Rect;
function TitleOf: Str255;
function IsActive: Boolean;
function IsDefault: Boolean;
function MustOutline: Boolean;
{ And the inherited NextHandlerOf }
```

Most of these methods won't be used much, but providing them improves the chances that other programmers will honor the button object's encapsulation by not accessing its instance variables directly. All the inquiry methods are quite similar, so we'll look at one method to get the idea:

```
function DisplayRectOf: Rect;
begin
  DisplayRectOf := self.fRect; { Returns the display rectangle }
end; { TButton.DisplayRectOf }
```

Display methods

TButton's display methods either draw the button in some way or affect how it will be drawn the next time. The main method is *Display*, which defaults to drawing the button as a rounded rectangle with its title in the center. *Hide* erases the button (and its outline if it's the default button). *Activate(false)* and *Dim* both cause the button title to be redrawn in gray. *Hilite* inverts the basic display, or restores it. *Activate(true)* sends a Display message again. *MakeDefault(true, true)* causes the button to be outlined with a larger rounded rectangle drawn with a three-pixel pen and sends Display to do the outlining based on changed instance variables. *Move* erases the button in one place, resets its rectangle, and sends Display.

```
procedure Display;
procedure Hide (stayActive: Boolean);
procedure Dim;
procedure Hilite;
procedure Activate (doActivate: Boolean);
procedure MakeDefault (doMakeDefault: Boolean; frameIt: Boolean);
procedure Move (toRect: Rect);
```

Let's look at several of these important methods, starting with *Display*.

Display draws the button's shape, conditionally draws a dark outline around the button if it's been declared the default button, draws the button's title in its center (in black on white, not in gray), and sets *fActive* to *true*. (If we didn't want the button to be active, we would send *Dim* or *Activate(false)* instead.) Before drawing, the *Display* method saves the old drawing environment, and it restores the saved environment at the end of the method. That's good practice for any graphics routine—it should leave the system environment as it found it.

```
procedure TButton.Display;
  var
    oldPenState: PenState;
    buttonRect: Rect;
begin
  GetPenState(oldPenState);
  PenNormal;
  FrameRoundRect(self.fRect, self.fCornerWidth, self.fCornerHeight);

  { Outline button to show that it is default }
  if self.fDefaultButton and self.fMustFrameDefault then
    begin
      buttonRect := self.fRect;
      InsetRect(buttonRect, -4, -4);
      PenSize(3, 3);
      FrameRoundRect(buttonRect, self.fCornerWidth, self.fCornerHeight);
      PenNormal;
    end;

  { Use DrawTextInRect from unit UTextDrawing }
  DrawTextInRect(self.fTitle, self.fRect, false, false);
  self.fActive := true;
  SetPenState(oldPenState);
end; { TButton.Display }
```

Hide is straightforward. It erases the entire rectangle containing the button. If the button is the default button, *Hide* also erases the outline. We can specify in the parameter *stayActive* whether the hidden button is to be active.

```
procedure TButton.Hide (stayActive: Boolean);
  var
    buttonRect: Rect;
```

```
begin
  if self.fDefaultButton then              { Erase default highlighting and button }
    begin
      buttonRect := self.fRect;
      InsetRect(buttonRect, -4, -4);
      EraseRoundRect(buttonRect, self.fCornerWidth, self.fCornerHeight);
    end
  else                                     { Simply erase button }
    EraseRoundRect(self.fRect, self.fCornerWidth, self.fCornerHeight);

  if not stayActive then
    self.fActive := false;                 { Hidden buttons usually not clickable }
end;  { TButton.Hide }
```

Dim is only slightly more complicated than *Hide*. *Dim* erases the old button, including its outline if it's the default button, redraws the button, and redraws the title in gray. *Dim* doesn't bother to redraw the outline because a dimmed button can no longer be considered the default button. In fact, *Dim* resets *fDefaultButton* to *false*.

```
procedure TButton.Dim;
  var
    oldPenState: PenState;
    buttonRect: Rect;
begin
  if self.fActive then                { If inactive, don't bother dimming--already dim }
    begin
      GetPenState(oldPenState);
      EraseRoundRect(self.fRect, self.fCornerWidth, self.fCornerHeight);
      if self.fDefaultButton and self.fMustFrameDefault then
        begin
          buttonRect := self.fRect;
          InsetRect(buttonRect, -4, -4);
          EraseRoundRect(buttonRect, self.fCornerWidth, self.fCornerHeight);
        end;
      self.fDefaultButton := false;  { Not default anymore }

      FrameRoundRect(self.fRect, self.fCornerWidth, self.fCornerHeight);
      DrawTextInRect(self.fTitle, self.fRect, true, false);
      self.fActive := false;
      SetPenState(oldPenState);
    end;
 { Else do nothing }
end;  { TButton.Dim }
```

Hilite simply inverts the pixels of the button's display rectangle, leaving them inverted until *Hilite* is called again to restore the rectangle's pixels. The method ignores a default outline if one is present, highlighting only the actual button.

```
procedure TButton.Hilite;
begin
  InvertRoundRect(self.fRect, self.fCornerWidth, self.fCornerHeight);
end;  { TButton.Hilite }
```

The mouse-tracking code in the *Clicked* method, which we'll develop later in this chapter, sends the Hilite message while the mouse button is down to turn highlighting on or off depending on whether the mouse pointer is in the button. Different button styles show highlights in different ways, and a button responds in its own way, thanks to polymorphism and runtime binding, which we'll return to in Chapter 6.

Activate isn't much more complicated. Depending on whether we pass *true* or *false* to it, it calls either *Display* or *Dim*. We can always send the Display or Dim message ourselves, but *Activate* is a more descriptive word to put in our code. *Activate* doesn't affect the instance variable *fActive*; both *Display* and *Dim* reset that field.

```
procedure TButton.Activate (doActivate: Boolean);
begin
  if doActivate then
    self.Display  { Sets fActive to true }
  else
    self.Dim;     { Sets fActive to false }
end;  { TButton.Activate }
```

MakeDefault is simple, too. It resets the *fDefaultButton* and *fMustFrameDefault* instance variables and then calls *Display* if the button needs to be redrawn with an outline. The *MakeDefault* method has a side effect: By calling *Display*, it makes the default button active if it wasn't. If *Display* isn't called because no outline is needed, *MakeDefault* sets the button to active because it makes little sense for the button to be inactive when it's the default.

```
procedure TButton.MakeDefault (doMakeDefault: Boolean; frameIt: Boolean);
begin
  self.fDefaultButton := doMakeDefault;   { True if making button the default }
                                          { False if "undefaulting" button }
  { Frame default button if necessary }
  self.fMustFrameDefault := frameIt;
  if self.fMustFrameDefault then
    self.Display                          { Also makes button active }
  else
    self.fActive := true;
end;  { TButton.MakeDefault }
```

Finally, *Move* has only a tangential effect on the button's appearance. The button gets displayed in a new location. We might be moving either an active button or a dimmed one. (We could also implement a drag method, using the QuickDraw function *DragGrayRgn*, but I'll leave that to you.)

```
procedure TButton.Move (toRect: Rect);
begin
  self.Hide(true);              { Hide button at old location }
                                { Include default outline if fDefaultButton is true }
  self.fRect := toRect;         { Set new drawing rectangle }
  if self.fActive then
    self.Display                { Display button there }
  else
    self.Dim;                   { Display dimmed button there }
end;  { TButton.Move }
```

TButton's Location

We'll put *TButton* into its own unit, *UButton*. Figure 4-3 sketches in *UButton*'s structure.

```
unit UButton;

interface
  uses
    ObjIntf, UEvtHandler, UTextDrawing;

  type
    TButton = object(TEvtHandler)      { Ancestor will change to }
                                       {  TPicoView in Part 2 }
        { Field list }
        { Method list }
      end;  { Class TButton }

    { Other standard button classes--TRadioButton, TCheckBox, }
    {  TCluster, and TRadioCluster--declared here }

implementation

  { TButton method code }
  { Method code for other standard button subclasses }

end.  { UButton }
```

Figure 4-3.
Unit UButton*'s structure.*

Having developed most of *TButton*'s methods, we'll detour now to take a look at *TButton*'s relationship to *TEvtHandler*. Then we'll return to the remaining *TButton* methods.

TButton and Class TEvtHandler

On the Macintosh, a program is event driven: When the user clicks a button, pulls down a menu, or presses a key, the Mac's operating system puts that event into an event queue. The application calls a Toolbox function—either *GetNextEvent* for Finder or, for MultiFinder, *WaitNextEvent*—to get the event from the queue and process it. Each event is dispatched to the piece of code appropriate for handling it. The program continually cycles in its event loop, getting and handling events or waiting for the user to do something.

Events include mouse-downs, mouse-ups, key-downs, autokeys, and more. See *Inside Macintosh*, Volume I, for details. For *TButton*, we'll be directly interested in only two event types: mouse-down and key-down.

TEvtHandler as an Abstract Class

Many objects need to be able to process events: buttons, scrollbars, text fields, and so on. Processing an event means doing something in response to it. In particular, we want a button object to do something when a mouse-down event occurs in its rectangle. We'll call any object that inherits from *TEvtHandler*—and that thereby knows how to handle events—an "event-handler object."

Abstract Classes

Classes like *TEvtHandler* and *TObject* are abstract in that they exist simply to pass functionality to a variety of descendant classes. They float at the top of the hierarchy, above the real world, available for subclassing by all sorts of "real" classes that need what they have to offer.

The idea is to write a new class—say, *TButton*—that declares *TEvtHandler* as its immediate ancestor. Then a *TButton* object automatically *is-a TEvtHandler* object as well as whatever else it is. This is our first serious example of subclassing.

We'll never declare an instance of an abstract class such as *TEvtHandler* or *TObject*. After all, what's the use of a generic object or a generic event handler? Only the specific descendants of the abstract classes make sense. They inherit from their abstract ancestors and then add new functionality. *TButton* has—by inheritance—instance variables for event handling and methods for handling events and freeing itself. And *TButton* itself is abstract—for it to be of any use, we must override its *DoClick* method in a subclass.

TEvtHandler is even more abstract than *TButton*. Not only do we not declare any instances of it, but it also doesn't even represent a real-world object, and its methods do very little. *TEvtHandler* does serve to give many kinds of objects a shared capacity, and we can link *TEvtHandler* objects in a list to facilitate processing events. We'll refer to this list of *TEvtHandler* objects (some of them buttons, others scrollbars, others pop-up menus, text fields, and so on) as our "event-handler chain." We have to do some initial work to set up the chain, but then *TEvtHandler*'s methods manage it for us. Figure 4-4 shows the chain.

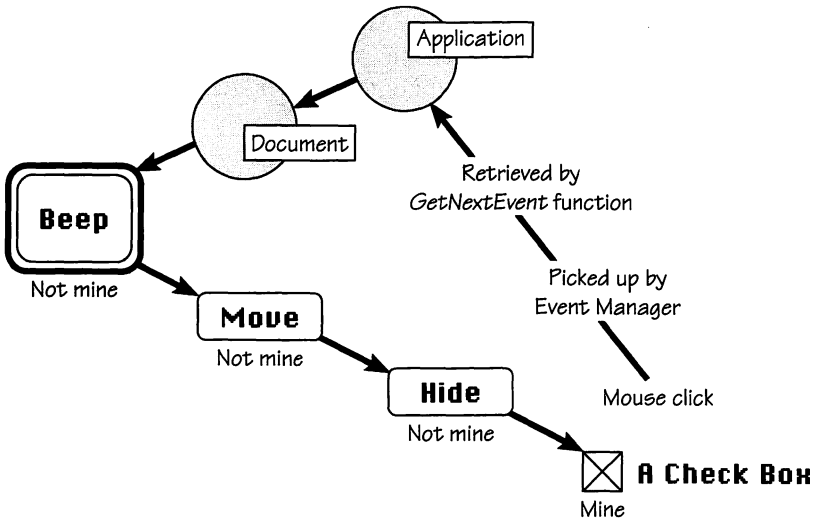


Figure 4-4.
The event-handler chain—a mouse click passed along the chain to a button object.

The name and concept for *TEvtHandler* is borrowed from Apple's MacApp, where it's the basis of an elaborate scheme for passing events to objects that need to handle them. Our version is much simpler, suited more to small, custom-built applications. MacApp, TCL, and other general application frameworks need a more elaborate event-handling system because they must be able to accommodate unforeseeable objects and events.

Any object that inherits from *TEvtHandler* acquires several methods—methods for handling mouse clicks, keypresses, menu selections, and more. For *TButton*, we'll be interested in only a couple of *TEvtHandler*'s methods.

TButton as a TEvtHandler

Making *TButton* a descendant of *TEvtHandler* is an example of "programming by differences," although a rather strange one. After all, we're deriving *TButton* from a class with which it has little in common. Later, when we derive some button subclasses from *TButton*, we'll see more typical examples of programming by differences.

We begin with the assumption that our button objects will be linked into an event-handler chain. As a *TEvtHandler*, a *TButton* object can link itself to another object that will be the *TButton* object's "next handler." When our button can't handle an event itself, it has a way to pass the event on to some other handler. It doesn't need to know who or what that handler is. The code inherited from *TEvtHandler* takes care of finding out about those details.

TButton inherits a number of methods from *TEvtHandler*, but we're interested mainly in two *TEvtHandler* methods that *TButton* overrides in order to do things its own way:

```
function DoMouseCommand (event: EventRecord): Boolean;
```

and

```
function DoKeyCommand (event: EventRecord): Boolean;
```

What TButton Does with Its TEvtHandler Methods

The *DoMouseCommand* method lets objects respond to a mouse click (a mouse-down event followed by a mouse-up event). When it isn't overridden by a subclass, *DoMouseCommand* simply forwards an event to the next event-handler object in the list. When it receives an event, the application initially intercepts and handles the event—a mouse-down event, for instance—itself. The application will most often check to see whether the mouse-down event occurred in its menu bar and will handle the event itself if it did. If the mouse-down event did not occur in the menu bar, the application doesn't handle the event; instead, it passes the event along the event-handler chain. Objects down the line can override *TEvtHandler.DoMouseCommand* to handle the event themselves. If they don't override it, the version of *DoMouseCommand* they inherit simply passes the event along to the next object.

A button object overrides *TEvtHandler.DoMouseCommand* in order to test whether the mouse-down was inside the button's display rectangle. If it was, it handles the mouse-down by sending itself the DoClick message and returning *true* for *DoMouseCommand*. If the mouse-down event wasn't inside the button's display rectangle, the event passes along the chain from object to object to object. Eventually all those calls to *DoMouseCommand* functions will return a result. And sooner or later, the first object in the list will pass the result back as its own result. Figure 4-5 shows the process each button in the chain goes through when it receives a mouse-down event.

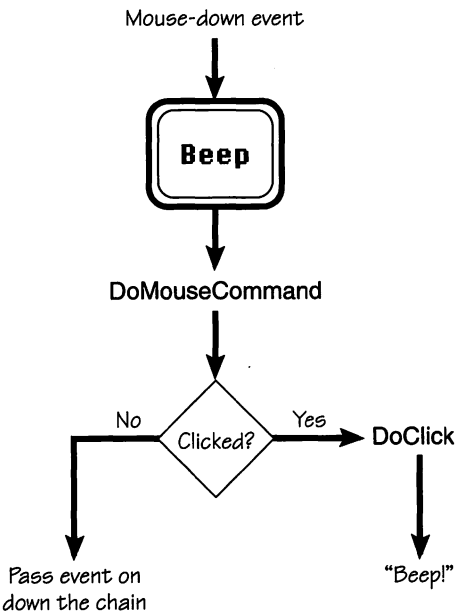


Figure 4-5.
Mouse-down sequence—method calls resulting from mouse-down event in button.

Here's the code for *TButton*'s *DoMouseCommand* event handler that overrides *TEvtHandler.DoMouseCommand*:

```
function TButton.DoMouseCommand (event: EventRecord): Boolean;
  override;
begin
  if self.Clicked(event) then              { Event belongs to this button }
    begin
      self.DoClick;                        { Clicked, so respond }
      DoMouseCommand := true;
    end
  else                                     { Not a click in this button }
    { Pass to next event handler }
    DoMouseCommand := inherited DoMouseCommand(event);
end;  { TButton.DoMouseCommand }
```

The *DoKeyCommand* method lets objects respond to key-down events. The application first checks a key-down event to see whether it's a command-key equivalent for a menu choice and handles it if it is. Otherwise, the application passes the key event along the event-handler chain so that other objects can get a chance to handle the event. Figure 4-6 on the next page illustrates the sequence.

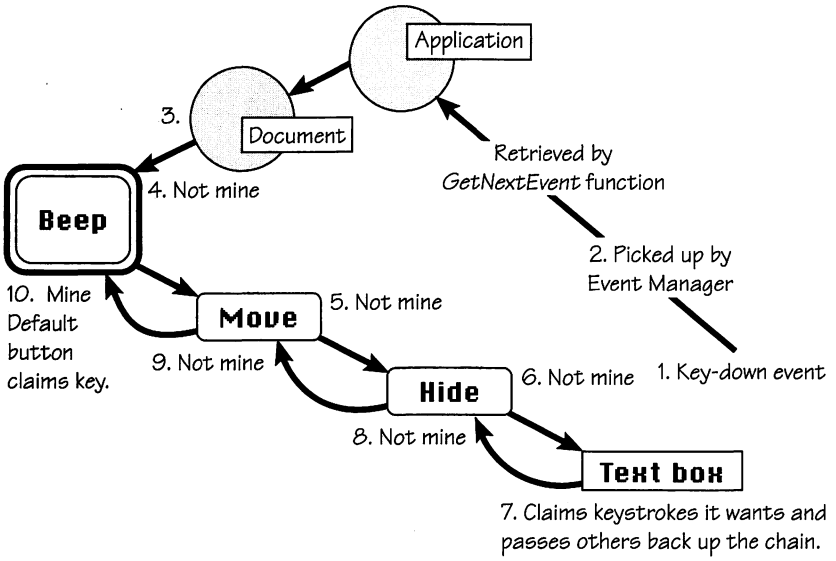


Figure 4-6.
Key-down sequence—key-down event passing through buttons and text box in the event-handler chain.

Technically, a button's *DoKeyCommand* method could be overridden to accept any keystroke as a "hot key." But a *TButton* object will respond to certain key-down events only if it happens to be the default button. The default button normally treats a press of the Return key or the Enter key as if the button had been clicked (as in Macintosh dialog boxes). There's one small hitch, though: A key-down event has no location, so we can't test whether the key-down event was "in our button" as we can with a mouse click. To be able to use a key-down event, the button must meet certain conditions. First, it has to be the default button. Second, no other event handler can have handled or can be able to handle the event. A text field, for example, might need the key-in, so we don't want a button to preempt the key-in unless we're sure no other object can claim it. The key-down event has to be passed all the way down the chain before the default button gets a crack at it. Only on the key-down event's way back up the chain, as the calls to *DoKeyCommand* unwind, each returning *false*, does the default button check the key-down event. Here's the code for the *DoKeyCommand* handler for buttons:

```
function TButton.DoKeyCommand (event: EventRecord): Boolean;
  const
    Return = 13;                    { Return key }
    Enter = 3;                      { Enter key  }
```

```
  var
    wasHandled: Boolean;
    chCode: Integer;
begin
  { Nobody else has handled the event--can any following }
  {  handler do it? }
  { Go on down the chain }

  wasHandled := inherited DoKeyCommand(event);

  { None can handle, before or after this button, so }
  {  can this button? }
  if not wasHandled then
    begin
      with event do
        begin

          { Is this the default button, }
          {  and is key Return or Enter? }
          chCode := BitAnd(message, charCodeMask);
          if self.fDefaultButton and ((chCode = Return) or (chCode = Enter)) then
            begin
              self.DoClick;          { Same as a click, so respond }
              wasHandled := true;
            end;
        end;  { with }
    end;  { If wasHandled is false }

  DoKeyCommand := wasHandled;
end;  { TButton.DoKeyCommand }
```

How a Button Responds to a Click

When a button object receives a mouse-down event that has been passed to it along the event-handler chain, it has to determine whether the mouse-down event "belongs" to it. If the mouse-down event belongs to it and the mouse button is released (a mouse-up event) while the mouse pointer is still in the button, the button carries out the task it was created for.

The button's *Clicked* method determines whether the mouse-down event belongs to the button and whether the mouse-down is followed by a mouse-up while the mouse pointer is in the button. The button's *DoClick* method carries out the task the button was created for.

The *Clicked* method

We saw that when a button's *DoMouseCommand* method is called, it first sends the button (*self*) a Clicked message:

```
⋮
if self.Clicked(event) then                { Event belongs to this button }
  begin
    self.DoClick;                          { Clicked, so respond }
    DoMouseCommand := true;
  end
else
⋮
```

We call *Clicked* a "private" method. We need it to get the button's job done, but we don't want outsiders to call it themselves. The only way we can avoid that right now is to warn outsiders off. Later, we'll look at a way to make *Clicked* an ordinary function, which we can hide in the implementation part of the unit and not make public at all. We're not making *Clicked* a hidden function now because it needs direct access to the button's instance variables and methods. Ordinary procedures and functions don't have direct access. See the discussion "Utility Procedures" under "Writing Methods" in Chapter 7.

What does *Clicked* have to do? It must convert the mouse-down's location (a point) from global to local window coordinates, using QuickDraw's *GlobalToLocal* function. It must see whether the point is within the button's display rectangle, using QuickDraw's *PtInRect* function. It must track the mouse pointer as long as the mouse button is down, highlighting and unhighlighting the button as the mouse pointer moves in and out of it, and return *true* if, when the mouse button is released, the mouse pointer is in the button or *false* if it isn't. Here's *Clicked*:

```
function TButton.Clicked (event: EventRecord): Boolean;
  var
    thePoint: Point;
    outOfButton: Boolean;
begin
  if not self.fActive then                  { Not responding to clicks now }
    Clicked := false
  else if event.what = mouseDown then { Active and open for business }
    begin
      { Convert the hit point to local coordinates }
      thePoint := event.where;
      GlobalToLocal(thePoint);

      { See whether mouse pointer was inside button at mouse-down }
      outOfButton := not PtInRect(thePoint, self.fRect);
```

```
        if not OutOfButton then
          begin
            self.Hilite;                    { Highlight button initially }

            { While mouse button still down, track pointer in and out of button }
            while stillDown do
              begin
                GetMouse(thePoint);         { See where mouse pointer is now }

                { Mouse pointer is in the button but moves out }
                if not outOfButton and (not PtInRect(thePoint, self.fRect)) then
                  begin
                    self.Hilite;            { Turn off highlighting }
                    outOfButton := true;
                  end

                { Or it's out of the button but comes back in }
                else if outOfButton and PtInRect(thePoint, self.fRect) then
                begin
                  self.Hilite;              { Turn on highlighting }
                  outOfButton := false;
                  end;
              end;  { while }

            { Once mouse button is released, see where mouse pointer is }
            if not outOfButton then
              begin
                { Released inside, so real click }
                Clicked := true;
                self.Hilite;                { Turn final highlighting off }
              end
            else
              { Released outside, so doesn't count }
              Clicked := false;

            InitCursor;

          end  { If mouse-down is not out of button }
        else                                { Mouse-down is out of button }
          Clicked := false;
      end
    else
      Clicked := false;                     { Event wasn't a mouse-down }
  end;  { TButton.Clicked }
```

The *Clicked* method makes one preliminary test, to see whether the button is currently active. An inactive button wouldn't respond to clicks, so we check.

The location of the mouse-down is given by the *where* field in the event record. (See *Inside Macintosh*, I-249.) We convert the location to local coordinates.

Then, if the mouse pointer was initially inside the button's display rectangle, we loop until the mouse button is released. While looping, we continually check for a change of status—mouse pointer moving out of the button or mouse pointer moving back into the button—and highlight or unhighlight accordingly. Finally, when the mouse button is released, we see whether the pointer ended up inside the button and return *true* if it did.

We've made good use of OOP here. All Macintosh buttons highlight when the mouse pointer is in them and unhighlight when it leaves. But the highlighting process differs from one kind of button to another. Ordinary buttons simply invert, but check boxes and radio buttons highlight by drawing the button's square or circle in bolder lines, drawing and redrawing as the mouse pointer moves in and out of the button. We could have made the *Clicked* code slightly faster by directly coding the highlighting rather than sending a Hilite message, but each button-style subclass would then have had to override the *Clicked* method so that it could do the highlighting part differently. As it stands, all buttons can use this version of *Clicked* by means of inheritance. Our *Clicked* method sends the Hilite message, and the button in question invokes the appropriate *Hilite* method—the button having overridden what *TButton* does for highlighting. This works because each button style's *Clicked* method, inherited from *TButton*, knows who *self* is—the button it belongs to.

The *DoClick* method

What does a clicked button do? Whatever you want it to. It can beep, or flash, or save a file, or contact Detroit via your modem, or quit the program. The button does whatever you program it to do, just as HyperCard buttons do whatever you script them to do.

The way we've designed our buttons, we need to subclass *TButton* to get each specific functional kind of button. Let's look at an example. Suppose we want an ordinary button labeled "Beep" that will beep the Mac's speaker the number of times we specify.

To create the class of this button, we'd subclass *TButton*, adding an instance variable and an initialization method and overriding *TButton*'s *DoClick* method:

```
type
  TBeepButton = object(TButton)

      fBeeps: Integer;        { Number of times to beep }

      procedure TBeepButton.IBeepButton (title: Str255; active: Boolean;
          numBeeps: Integer; r: Rect);
      procedure TBeepButton.DoClick;
      override;
    end;  { Class TBeepButton }
```

Now we can instantiate as many actual beep buttons as we need from the class:

```
var
  aBeeper: TBeepButton;
  anotherBeeper: TBeepButton;
  :
```

And here are the implementations of the *IBeepButton* and *DoClick* methods for the *TBeepButton* class:

```
procedure TBeepButton.IBeepButton (title: Str255; active: Boolean; numBeeps: Integer;
    r: Rect);
begin
  self.IButton(title, active, r);     { Call ancestor's initialization method }
  self.fBeeps := numBeeps;
end;  { TBeepButton.IBeepButton }

procedure TBeepButton.DoClick;
  override;
  var
    i: Integer;
begin
  for i := 1 to self.fBeeps do
    Sysbeep(1);
end;  { TBeepButton.DoClick }
```

Each new kind of button will subclass *TButton*, providing a *DoClick* method that knows what to do when that kind of button is clicked.

When a particular beep button, say *aBeeper*, is clicked, the *Clicked* method reports *true*, and the *DoClick* method associated with that button subclass is executed.

Summary

We've developed a serious object class, including design considerations, the class declaration, and implementations of the class's methods. *TButton* is also an example of how objects can encapsulate parts of the Macintosh user interface. Moreover, the *TButton* class is an excellent platform for creating new, specialized button classes, as we'll show fully in the next chapter.

Some of *TButton*'s functionality comes from its ancestors, by means of inheritance. By making it a subclass of *TEvtHandler*, which itself is a subclass of *TObject*, we arrange for *TButton* to inherit what those classes can do. As a *TObject*, a button can free itself (release its own storage in the heap) when we finish with it. As a *TEvtHandler*, a button can handle events such as mouse-downs and key-downs. If it can't handle a particular event, it knows how to pass it on to some other object.

Of course, our button class is not complete: It can't possibly specify what every conceivable button instance might actually do when handling events. And *TButton*

itself, like its ancestors *TEvtHandler* and *TObject*, is not instantiable. To create real button instances, we must write a subclass of *TButton* and override its *DoClick* method. The subclass is extremely simple: All it has to provide is the *DoClick* override; everything else it needs to be a button is available by means of inheritance. The subclass can be instantiated—that is, we can declare variables of its type and use them as objects.

To round things out and see some real examples of "programming by differences," we'll look in the next chapter at three subclasses of buttons that we'll derive from *TButton*, just as we did the beep button. We'll look at *TCheckBox*, *TRadioButton*, and *TTransparentButton*.

Projects

- Write a variety of *TButton* subclasses along the lines of the beep button we looked at in this chapter. Write buttons with *DoClick* methods that flash the buttons, move them, hide them, and so on. Be creative. Then write a simple conventional Mac program that creates some of your buttons, links them in an event chain so that events can be sent to them, and uses an event loop to send mouse-down events to your buttons. We'll write just such a program in the next chapter, so feel free to peek and to borrow code from it as you need to.
- Add *Drag* and *Resize* methods to *TButton*. *Drag* should use the QuickDraw routine *DragGrayRgn* (*Inside Macintosh*, I-294) to drag a dotted outline of the button around, resetting its display rectangle and redisplaying it at the new location. *Resize* should increase or decrease the size of the display rectangle and redisplay the button. Keep in mind the amount of space the button title needs for display.

5

PROGRAMMING BY DIFFERENCES

The key idea we'll take up in this chapter, and one of the main tenets of OOP, is that new program structures can be derived by means of subclassing that takes advantage of inheritance. We looked at brief examples of subclassing and overriding in Chapters 3 and 4. In this chapter, we'll explore the techniques in detail.

The button classes we design in this chapter will inherit their common features from *TButton*. But each derived class will differentiate itself from *TButton* by overriding some of *TButton*'s methods (and in some cases by adding new instance variables).

We'll look at

- *TCheckBox*
- *TRadioButton*
- *TRadioCluster*
- *TTransparentButton*
- Several interesting adjunct classes
- An example button program

Some Rules First: Inheritance and Subclassing

We derive one class from another by writing a subclass. The subclass makes its descendant relationship clear by specifying its immediate ancestor in the heritage spot of its class declaration:

```
type
  TSubclass = object(TImmediateAncestor)
```

The subclass inherits all instance variables and methods from its ancestor including anything the ancestor inherited from its ancestor(s). Subclassing can create a whole hierarchy of related classes.

A subclass can have new instance variables and methods that its ancestor didn't have. And it can override any of the ancestor's methods that don't suit its needs. A subclass can't override its ancestor's instance variables, and it can't delete or rename instance variables or methods. It can't change the parameter list or the function return type of a method it overrides, either.

A subclass actually inherits the most recent version of a method, its immediate ancestor's version. If ancestor *B* overrides method *M* of ancestor *A*, subclass *C* gets *B*'s version of method *M*, not *A*'s; but a method of *B* can still call *A*'s version of method *M* by means of the *inherited* keyword. We'll look at uses of the *inherited* keyword in some detail in Chapter 8.

Class TCheckBox

Check boxes are the little squares with text to their right that you see in Macintosh dialog boxes. They're used mainly to set Boolean parameters—they toggle between two states. A check box can have an X in it, or it can be empty. Usually, an application uses a Control Manager routine, *GetCtlValue*, to read the state of the box. When it detects a click in the box, the application can put an X in the box (or erase one) with *SetCtlValue*. Figure 5-1 shows check boxes in several different states.

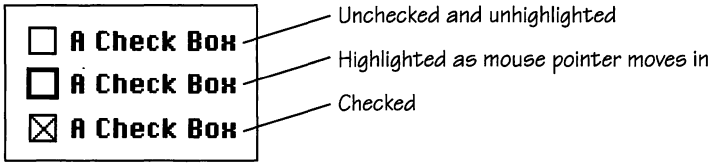


Figure 5-1.
Macintosh check boxes in various states.

Deriving TCheckBox

Let's derive *TCheckBox* from *TButton*. *TButton* isn't based on the Control Manager, so we'll provide *TCheckBox* with methods for causing a box to be checked or unchecked and for reading its current state. We can use both the *Display* method and the *Hilite* method for checking and unchecking. We'll need a Boolean instance variable *fChecked* and a function method *IsChecked* to return the value of *fChecked*. We must create a new initialization method because a check box always has some initial state—checked or unchecked.

What methods of *TButton* will have to change? The main one, of course, is *Display*—a check box doesn't look anything like a rounded-rectangle button. Likewise, the methods that draw or erase the button or write its title will need to change.

What *TButton* methods stay the same? *SetDisplayRect*, *SetTitle*, *DisplayRectOf*, *TitleOf*, *IsActive*, *Activate*, *Move*, *Clicked*; the *TEvtHandler* methods *NextHandlerOf* and *SetNextHandler*; and *TObject*'s *Free* and *Clone* methods.

In a check boxes context, some methods no longer make sense, so we'll "stub out" *SetCornerRounding*, *IsDefault*, *MustOutline*, and *MakeDefault*. Because check boxes can't be default buttons, they shouldn't respond to keypresses, so we'll also stub out *DoKeyCommand*. To stub out a method, we override it and leave the body of the overridden version empty. If an object calls the overridden method, the method does nothing. If any stubbed-out method is a function method, it returns *false*, *0*, or some other empty value corresponding to the function's return type.

Here's the declaration for *TCheckBox*, which we'll put into the same unit that defines *TButton* and the other standard Macintosh button types:

```
type
  TCheckBox = object(TButton)

      fChecked: Boolean;      { True if box is checked }
      fTitleRect: Rect;       { Coordinates of title }
      fHilited: Boolean;      { True if box is highlighted }

      { New methods }
      procedure TCheckBox.ICheckBox (title: Str255; active: Boolean;
          checked: Boolean; r: Rect);
      function TCheckBox.IsChecked: Boolean
        { True if check box is checked }

      { Overridden methods }
      procedure TCheckBox.DoClick;
      override;
      procedure TCheckBox.Display;
      override;
      procedure TCheckBox.Hide (stayActive: Boolean);
      override;
      procedure TCheckBox.Dim;
      override;
      procedure TCheckBox.Hilite;
      override;

      { "Stubbed-out" methods--for button characteristics not }
      {  found in check boxes }
      procedure TCheckBox.SetCornerRounding (width, height: Integer);
      override;
      function TCheckBox.IsDefault: Boolean;
      override;
      function TCheckBox.MustOutline: Boolean;
      override;
      procedure TCheckBox.MakeDefault (doMakeDefault: Boolean;
          frameIt: Boolean);
      override;
```

```
    function TCheckBox.DoKeyCommand (event: EventRecord): Boolean;
    override;

    { Remaining TButton methods inherited as is }
  end;  { Class TCheckBox }
```

Check Box Initialization

The only unusual task that the initialization method for check boxes must do is to use the button's display rectangle (the check box) to construct a second rectangle for the button's title—offset to the right of the check box. This rectangle is stored in *fTitleRect*.

```
procedure TCheckBox.ICheckBox (title: Str255; active: Boolean; checked: Boolean;
    r: Rect);
begin
  with r do                    { Ensure that r is right size for check box }
    begin
      right := left + 13;
      bottom := top + 13;
    end;  { with }
  self.IButton(title, active, r);  { Call ancestor's initialization method }
  self.fChecked := checked;  { Set initial checked state of box }
  self.fHilited := false;

  { Offset title rectangle to right }
  with self.fTitleRect do
    begin
      top := self.fRect.top - 3;
      bottom := self.fRect.bottom + 3;
      left := self.fRect.right + 4;
      right := left + StringWidth(self.fTitle) + 8;
    end;  { with }
end;  { TCheckBox.ICheckBox }
```

Check Box Display

TCheckBox's *Display* method is quite a bit simpler than *TButton*'s—mainly because we push off some of the work onto a non-method procedure called *DrawX*, which actually draws in the X if the box is checked. Here's *Display*:

```
procedure TCheckBox.Display;
  override;
  var
    oldPenState: PenState;
    titleRect: Rect;
```

```
begin
  GetPenState(oldPenState);   { Save current pen state }
  PenNormal;
  FrameRect(self.fRect);      { Draw button's box }

 { Note:  We pass self as a parameter to the DrawX procedure }
 {  so that it can access instance variables of the current check box object }
 { Alternatively, we could pass those variables as parameters }
  if self.fChecked then
    DrawX(self);
  DrawTextInRect(self.fTitle, self.fTitleRect, false, false);
  self.fActive := true;
  SetPenState(oldPenState);   { Restore original pen state }
end;  { TCheckBox.Display }
```

If the button needs to be checked, *Display* calls *DrawX*, which is simply a utility routine, not a method:

```
procedure DrawX (btn: TCheckBox);  { Not a method--draws X in a checked }
                                   {  check box }
  var
    oldPen: PenState;
begin
  GetPenState(oldPen);        { Save pen state }
  PenNormal;
  with btn.fRect do           { Draw X }
    begin
      MoveTo(left, top);
      LineTo(right - 1, bottom - 1);
      MoveTo(left, bottom - 1);
      LineTo(right - 1, top + 1);
    end;  { with }
  SetPenState(oldPen);        { Restore pen state }
end;  { DrawX--not a method }
```

We pass the *DrawX* utility procedure a copy of the check box's display rectangle, *fRect*, and it uses QuickDraw calls to draw the X in the box. The last check box in Figure 5-1 on page 92 contains an X.

Check Box Highlighting

Earlier we noted that check boxes and radio buttons are highlighted differently than rounded-rectangle buttons. If you click a check box and hold the mouse button down, the check box outline is redrawn in bold. If you then move the mouse pointer out of the check box without letting the mouse button up, the check box is redrawn again in single-pixel lines. The X inside, if any, stays the same. The first two check boxes in Figure 5-1 on page 92 show the unhighlighted and highlighted states, and the *Hilite* method is on the next page.

```
procedure TCheckBox.Hilite;
  override;
  var
    oldPen: PenState;
begin
  EraseRect(self.fRect);
  GetPenState(oldPen);

  { Draw normal or bold depending on current highlight state }
  if self.fHilited then
    PenNormal
  else
    PenSize(2, 2);
  FrameRect(self.fRect);              { Make check box's outline bold }
  if self.fChecked then
    DrawX(self);                      { Redraw X in check box }
  SetPenState(oldPen);
  self.fHilited := not fHilited;      { Toggle highlight state }
end;  { TCheckBox.Hilite }
```

Other Check Box Methods

The *TCheckBox* methods *Hide* and *Dim* are straightforward overrides of *TButton*'s methods. *Hide* erases the button and its title (now in the rectangle specified by *fTitleRect*) and sets the button to inactive. *Dim* erases the X in the box, if any, and redraws the title in dimmed gray. Their main difference from *TButton*'s methods is that *TCheckBox*'s methods have to handle the title differently. You can see the code for *TCheckBox*'s *Hide* and *Dim* methods in the folder Buttons, Ch 5 on the code disk, in the file UFirstButton.p. We won't concern ourselves further here with the methods *TCheckBox* overrides in order to stub them out. You can see those methods on the code disk, too.

Check Box *DoClick*

Finally, we must write *TCheckBox.DoClick*. Clicking is one area in which check boxes and radio buttons differ from rounded-rectangle buttons—clicking a check box or a radio button does not trigger an action. Instead, clicking sets its state to on or off. The application then reads its state to see what the user wanted. Usually, a state sets a variable that controls further processing in some way. A game program, for instance, might use the states of a cluster of radio buttons to determine the number of players.

So what does the *TCheckBox.DoClick* method do? We call it from the method *TCheckBox.DoMouseCommand* (inherited from *TButton*) to toggle the check box on or off and redisplay it accordingly—checked or not.

```
procedure TCheckBox.DoClick;  { For a check box, DoClick either }
                              {  checks or unchecks the box }
  override;
begin
  self.fChecked := not self.fChecked;  { Toggle value of on/off }
  self.Hide(true);                     {  and then redisplay }
  self.Display;
end;  { TCheckBox.DoClick }
```

Hide(true) means "hide but stay active."

Classes *TRadioButton* and *TRadioCluster*

TRadioButton is very similar to *TCheckBox*, so we won't go into much of its code here. You can see all of it on the code disk.

Like a check box, a radio button toggles on and off when clicked. Instead of an X in a box, a radio button that is on shows a dark dot in the center of its circle. A radio button highlights much as a check box does. The Boolean method *IsOn* reads the radio button. Figure 5-2 shows radio buttons in several different states.

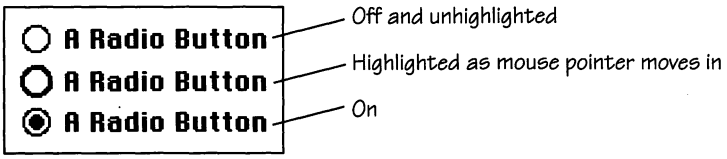


Figure 5-2.
Macintosh radio buttons in various states.

Here's the *TRadioButton* class declaration:

```
type
  TRadioButton = object(TButton)

      fOn: Boolean;
      fTitleRect: Rect;
      fCluster: TCluster;       { Nil if button not in a cluster }
      fHilited: Boolean;

      { New methods }
      procedure TRadioButton.IRadioButton (title: Str255; active: Boolean;
          onOff: Boolean; r: Rect; cluster: TCluster);
      procedure TRadioButton.SetCluster (cluster: TCluster);
      procedure TRadioButton.TurnOn (doTurnOn: Boolean);
      function TRadioButton.IsOn: Boolean

      { Overridden methods }
      procedure TRadioButton.DoClick;
      override;
```

```
    procedure TRadioButton.Display;
    override;
    procedure TRadioButton.Hide (stayActive: Boolean);
    override;
    procedure TRadioButton.Dim;
    override;
    procedure TRadioButton.Hilite;
    override;

    { "Stubbed-out" methods--relate to button characteristics not found }
    {  in radio buttons }
    procedure TRadioButton.SetCornerRounding (width, height: Integer);
    override;
    function TRadioButton.IsDefault: Boolean;
    override; { Returns false if called }
    function TRadioButton.MustOutline: Boolean;
    override; { Returns false if called }
    procedure TRadioButton.MakeDefault (doMakeDefault: Boolean;
        frameIt: Boolean);
    override;
    function TRadioButton.DoKeyCommand (event: EventRecord): Boolean;
    override;

    { Remaining TButton methods inherited as is }
  end;  { Class TRadioButton }
```

What's different? Primarily, the fact that a radio button seldom stands alone. A radio button is almost always part of a cluster, in which several radio buttons represent a number of mutually exclusive states—turning some program feature on or off, choosing from a number of options, or doing something else. Within a cluster, only one button at a time can be on. Initially, one button—the default—must be on. If the user selects another, the first is turned off as the second goes on. They're like the cluster of buttons you use to select stations on your car radio. Figure 5-3 shows a cluster of radio buttons with their titles.

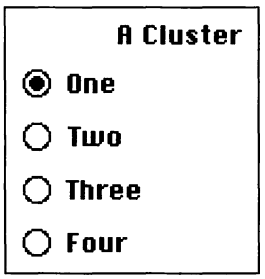


Figure 5-3.
A Macintosh radio button cluster with titles.

Class *TRadioButton* makes an individual radio button keep track of whether it's in a cluster, and if so, which cluster. The instance variable *fCluster* is an object reference to a *TRadioCluster* object. If a radio button gets clicked, it toggles itself on (unless it's already on, in which case it ignores the event) and then sends a message to the cluster object to tell it to turn off whatever other button is currently on. This important action takes place in *TRadioButton.DoClick*:

```
procedure TRadioButton.DoClick;
  override;
begin
  { Just clicked, so turn on or off, depending on current state }
  if (self.fCluster <> nil) then  { Button is in a cluster }
    begin
      if (not self.fOn) then
        self.fOn := true;         { Can turn a cluster button on but not off }
    end
  else                            { Button not in cluster, so click can }
    self.fOn := not self.fOn;     {  toggle the button's state }

  self.Display;

  { Just turned on, so notify button's cluster, if any }
  if self.fOn and (self.fCluster <> nil) then
    fCluster.SetCurrent(self);
end;  { TRadioButton.DoClick }
```

This code lets radio buttons either stand alone or be used in clusters. The *SetCluster* method can be called to install a reference to a button's cluster object in *fCluster*—or not. If no cluster is installed, the reference in *fCluster* is *nil*. Then, in *DoClick*, we check to see whether the button (*self*) is in a cluster by checking to see whether *fCluster* is *nil*. If *fCluster* isn't *nil*, the radio button behaves in one way; if it is, the radio button behaves in another.

If a radio button is in a cluster, the user can't click it to turn it off—one button must always be on. In other words, a radio button in a cluster can't simply be toggled on and off. If it's not a member of a cluster, a radio button can be toggled the way a checkbox is. *DoClick* redisplays a button regardless of whether it's in a cluster.

The key part of the *TRadioButton.DoClick* method code is the last *if* statement. If a button has been turned on and it is in a cluster, it has to notify the cluster. Calling

```
fCluster.SetCurrent(self)
```

is an ordinary method call through an object reference that happens to be stored as an instance variable in an object. If there really is a cluster object (if *fCluster* doesn't equal *nil*), the program sends it a SetCurrent message. What's interesting is that we pass *fCluster* a reference to *self*, the current button. The cluster can then use the reference to keep track of which button in the cluster is currently on.

SetCurrent also tells the previously on button to turn itself off. Keep in mind that *SetCurrent* is a method of *TRadioCluster*, not of *TRadioButton*.

```
procedure TRadioCluster.SetCurrent (newSelection: TButton);
  var
    i, oldHilite: Integer;
    rad: TRadioButton;
begin
  rad := TRadioButton(newSelection);      { Must typecast }
  if rad <> nil then
    begin
      oldHilite := self.fCurrentlySelected; { Record which button now on }
      i := 1;
      while rad <> self.fButtons[i] do    { Check current button against the }
                                          {  reference just passed in }
        i := i + 1;
      if i <= kClusterSize then
        self.fCurrentlySelected := i;
      { Else we should do something about the error }

      { Turn off old highlighted button }
      if (oldHilite > 0) and (oldHilite <> self.fCurrentlySelected) then
        self.fButtons[oldHilite].TurnOn(false);
    end;
  { Else fCurrentlySelected doesn't change }
  { Need error handling here }
end;  { TRadioCluster.SetCurrent }
```

This code brings up a number of issues, some of which we aren't quite ready for yet.

A Pascal scoping problem forces us to pass to *SetCurrent* a parameter of type *TButton*, which we then have to typecast to *TRadioButton*. The scoping problem occurs because *TRadioButton* objects and *TRadioCluster* objects need to refer to each other in order to communicate. So we declare the classes in this order, inventing a special, abstract *TCluster* class for the occasion:

TButton
TCluster—is an abstract class
TRadioButton—has a field of type *TCluster*
TRadioCluster—refers to *TRadioButtons*

The trick is fully described in Chapter 8, where we'll look at using abstract superclasses to solve Pascal scoping difficulties.

SetCurrent saves a copy of the index number of the currently highlighted radio button, then checks to be sure the button reference passed in is correct, and then makes the update official by resetting *fCurrentlySelected*. Finally, *SetCurrent* turns off the previously selected radio button by sending it a TurnOn message with the parameter

false. Thus, the primary responsibility of a radio button cluster object is carried out when one of its buttons sends the object a message to tell it what to do.

The rest of *TRadioButton*'s methods are very similar to *TCheckBox*'s. See those methods on the code disk.

A Little More About Radio Button Clusters

Although we won't go into the code for *TRadioCluster* here, let's briefly look at the class and the functions of its objects. (Figure 5-3 on page 98 shows a typical radio button cluster.)

A cluster manages a group of radio buttons. We've arbitrarily set an upper bound of ten buttons per cluster, using an array of *TRadioButton* as an instance variable of *TRadioCluster*:

```
fButtons: array[1..kClusterSize] of TRadioButton;
```

where *kClusterSize* equals 10. The currently selected button in the cluster is kept by storing an index into *fButtons* in *fCurrentlySelected* (an integer).

Here is the declaration for *TRadioCluster*:

```
type
  TRadioCluster = object(TCluster)

      { A cluster is treated as a button, making it easy to put a }
      {  whole cluster in a list of buttons }
      { The cluster can receive events as if it were a button and }
      {  pass them on to its subordinate radio buttons }
      fButtons: array[1..kClusterSize] of TRadioButton;
      fCurrentlySelected: Integer;
      fDefaultRadioButton: Integer;
      fClusterTitle: ButtonTitle;
      fTitleRect: Rect;

      { New methods }
      procedure TRadioCluster.IRadioCluster (clusterTitle: ButtonTitle; titleRect: Rect;
          numButtons: Integer; titles: TitleArray; rects: RectArray;
          defaultButton: Integer);
      function TRadioCluster.CurrentButtonOf: Integer;

      { Overridden methods }
      procedure TRadioCluster.SetCurrent (newSelection: TButton);
      override;
      procedure TRadioCluster.Display;
      override;
      procedure TRadioCluster.Hide (stayActive: Boolean);
      override;
      procedure TRadioCluster.Dim;
      override;
```

```
    procedure TRadioCluster.Hilite;
    override;
    function TRadioCluster.DoMouseCommand (event: EventRecord): Boolean;
    override;

    { Remaining TButton methods are inherited as is }
  end;  { Class TRadioCluster }
```

TRadioCluster is declared as a descendant of *TCluster*, and *TCluster* is a subclass of *TButton*—making *TRadioCluster* indirectly a subclass of *TButton*, too! By making the cluster itself a "button," we make it easy to put whole clusters into the event-handling chain and otherwise make the class convenient to use. Of course, a cluster isn't your ordinary button. It doesn't *do* something in the sense that a button does something; so, we don't override *TButton*'s empty *DoClick* method. But *TRadioCluster* does override many of *TButton*'s methods. When the application calls *TRadioCluster*'s *Display* method, it simply sends Display messages to every one of the cluster's subordinate radio buttons. *TRadioCluster*'s *Hide* hides them all. Its *Dim* dims all of their titles as well as its own overall cluster title. Its *DoMouseCommand* sends a DoMouseCommand message to each subordinate radio button in turn. Because highlighting a cluster can have no real meaning, its *Hilite* method simply sends a Display message. We've already looked at *TRadioCluster.SetCurrent*.

The application can ask the cluster which button is on by sending a CurrentButtonOf message, which returns the button number. It's up to the application to know what that particular button stands for in application terms—a player number, the state of some program feature, or something else.

Initializing *TRadioCluster* Objects

TRadioCluster initialization is the remaining method to look at:

```
procedure TRadioCluster.IRadioCluster (clusterTitle: Str255; titleRect: Rect;
    numButtons: Integer; titles: TitleArray; rects: RectArray; defaultButton: Integer);
  const
    kActiveYes = true;        { Values to pass to radio buttons }
    kOnNo = false;
  var
    aRadBtn: TRadioButton;
    i, last: Integer;
begin
  self.fClusterTitle := clusterTitle;
  self.fTitleRect := titleRect;
  { Should check whether numButtons is greater than }
  {  kClusterSize and handle the error if it is }
  for i := 1 to numButtons do
    begin
      New(aRadBtn);
```

```
        { Note:  We pass self as a parameter so that the new radio button }
        {  can identify the cluster object it belongs to }
        aRadBtn.IRadioButton(titles[i], kActiveYes, kOnNo, rects[i], self);
        aRadBtn.SetNextHandler(nil);
        self.fButtons[i] := aRadBtn;
        last := i;
      end;  { for }
    for i := last + 1 to kClusterSize do  { Assign nil to any unused array elements }
      self.fButtons[i] := nil;
    self.fDefaultRadioButton := defaultButton;
    self.fCurrentlySelected := self.fDefaultRadioButton;
    if (self.fDefaultRadioButton >= 1) and (self.fDefaultRadioButton
        <= numButtons) then self.fButtons[self.fDefaultRadioButton].TurnOn(true);

    { Else we should do something about the error }
  end;  { TRadioCluster.IRadioCluster }
```

Part of the initialization code merely assigns parameters to the cluster's instance variables. The important pieces of code create and initialize the radio buttons in the cluster, mark any unused array positions as *nil*, and turn on the default radio button. The default button itself takes care of displaying its state.

Radio buttons in the cluster are created as active buttons but are turned off so that we can then turn one of them on. They're each assigned a title and a display rectangle. And each is assigned an object reference pointing to the cluster object so that the radio buttons can send the cluster a SetCurrent message. The cluster ultimately gets linked into the event-handler chain, but its dependent radio buttons don't. *TRadioCluster.DoMouseCommand* sends a DoMouseCommand message to each button, moving down the array. If none of the buttons can handle the event, the cluster passes the message on to whatever the next event handler might be.

Using *TRadioCluster*

Creating a new radio cluster involves setting the number of buttons in the cluster and the default button, setting up a title array and a rectangle array, calling *New(aCluster)* where *aCluster* is a variable of type *TRadioCluster*, and sending *aCluster.IRadioCluster* to pass in all the initializing parameters.

Then the application can pass mouse-down events to the cluster object, which in turn passes them to the radio buttons. After the user has finished with the cluster, the application uses *CurrentButtonOf* to find out which radio button in the cluster is on.

Class *TTransparentButton*

To round out our examples of *TButton* subclasses—and our examples of programming by differences—let's take a quick look at a class modeled on the transparent buttons in HyperCard.

Transparent buttons act just as other buttons do, but they're normally invisible. (You can opt to display a transparent button's title.) In HyperCard, a transparent button typically overlays a graphic of some kind—a picture, a part of a picture, a word in some text. For instance, one of the HyperCard demonstration stacks shows a horse. If you click the horse's eye, you see another card depicting another creature that has an eye. The trick that makes this work is that each clickable object in the picture is overlaid by a transparent button. Conceptually, the button is in a layer above the graphic, so it intercepts mouse clicks and handles them with its script.

In most ways, a *TTransparentButton* object is merely a *TButton* object. The main differences are in how the button is displayed, hidden, dimmed, and highlighted.

Display is easy. There's nothing to draw—unless we choose to show the button's title. Our version shows the title in the center of the invisible button. (We could alter that with a subclass to display the title below the button, say.)

```
procedure TTransparentButton.Display;
  override;
begin
  { A complete implementation should save the bits under the }
  {  button before "drawing" it }
  { Not implemented here }

  { Nothing to show except possibly the title }
  if self.fShowTitle then
    DrawTextInRect(self.fTitle, fRect, false, false)
  else
    EraseRect(self.fRect);
  self.fDrawnAlready := true;
end;  { TTransparentButton.Display }
```

For a complete implementation of *TTransparentButton*, we would have to deal with one catch. Suppose that initially we display the button's title on top of a graphic—a typical use—but that later we want to redisplay the button without the title. Unfortunately, we've already obliterated part of the graphic beneath the title. One solution would be to save the bits composing the graphic under the button in an off-screen bitmap. We would do that at initialization and again if the button moves. Then we could always restore those bits, no matter what our so-called invisible button has done to them. This isn't an especially object-oriented concern, so you won't find the code to save and restore the underlying bits in an off-screen bitmap on the code disk. But see the projects at the end of the chapter.

Hide does nothing if the button's title wasn't showing; otherwise, it restores the underlying bits, covering up the title.

Dim does nothing except deactivate the button unless the title was showing, in which case it redraws the title in gray.

Hilite works as in *TButton*, inverting the pixels within the display rectangle. The inherited version of *Hilite* will work fine.

One more operation that might be useful for transparent buttons is a different, and additional, sort of highlighting. In HyperCard, if you choose the Button Tool all buttons on the screen are outlined with a thin, solid line so that you can see where they are. We'll provide a procedure *Outline* that takes care of that. *Outline(true)* does the outlining. *Outline(false)* removes it.

Here's the class declaration for *TTransparentButton*:

```
type
  TTransparentButton = object(TButton)

      fShowTitle: Boolean;              { True if you want the title to show }
      { fTheBits: BitMap;--not implemented in this version }
      fDrawnAlready: Boolean;           { True if already displayed }

      procedure TTransparentButton.ITransparentButton (title: Str255; active: Boolean;
          r: Rect; showTitle: Boolean);
        { Initialize a transparent button object, active or inactive }
        {  and with title showing or not }
      procedure TTransparentButton.Display;
      override;
        { "Display" the transparent button }
        { Does nothing unless fShowTitle is true }
      procedure TTransparentButton.Hide (stayActive: Boolean);
      override;
        { "Hide" the transparent button }
        { Remains clickable if stayActive is true }
        { Does nothing unless title is showing }
      procedure TTransparentButton.Dim;
      override;
        { "Dim" the transparent button and make it inactive }
        { If its title is showing, dim it }
      procedure TTransparentButton.Outline (doOutline: Boolean);
        { Draw a rectangular outline around the button (or erase one) }
      { procedure TTransparentButton.Free; }
      { override; }
        { Free the button and its bitmap--not implemented here }
    end;  { Class TTransparentButton }
```

You can see the code on the code disk. Notice that *TTransparentButton*, like its ancestor, *TButton*, is an abstract class. To make a real *TTransparentButton* instance, we need a subclass of *TTransparentButton* to override—and fill in—its *DoClick* method. As it stands, *TTransparentButton* inherits *TButton*'s empty *DoClick*.

The Example Program TestButtons

To see how buttons are used and how they function on screen, we'll develop a program that

- Creates a "button document" object to manage the selection of various kinds of buttons and a window
- Creates Beep, Move, Hide, Show, and Dim buttons; a check box; a lone radio button; a radio button cluster; and a transparent button
- Displays the buttons and sends events to them as the user clicks them or presses keys
- Frees all objects and quits when the user clicks in the window's close box

The Application

The application this time around is simply a Pascal main program. (Later, in Part 2, we'll look at an "application object.") The program declares a variable of type *TButtonDoc*, creates the button document object with *New*, and has the object initialize itself. The application has a *MainEventLoop* procedure to get events and procedures to handle window activate events and close the output window when the user clicks the close box. The application does not put up any menus or do anything fancy. Listing 5-1 shows an outline of the body of our main program.

```
program ButtonDemo;

  uses
    ObjIntf, UEvtHandler, UButton, UHyperButtons, UButtonTypes, UButtonDoc;

  var
    aButtonDoc: TButtonDoc;{ "Document" that displays buttons }
    quitting: Boolean;

  procedure DoClose;
  :
  procedure DoActivate (event: EventRecord);
  :
```

Listing 5-1. *(continued)*
A Pascal main program that demonstrates buttons.

Listing 5-1. *continued*

```
    procedure DoActivate (event: EventRecord);
    ⋮
    procedure DoUpdate (event: EventRecord);
    ⋮
    procedure MainEventLoop;
    ⋮
  begin { ButtonDemo }
    New(aButtonDoc);{ Create the button document object }
    aButtonDoc.Init;{ Initialize it--draws window and creates buttons }
    quitting := false;
    MainEventLoop;{ Loop until user quits }
    aButtonDoc.Free;{ Release storage for button document and its buttons }
  end.  { ButtonDemo }
```

Figure 4-1, in Chapter 4, shows the output of Button Demo.

Class *TButtonDoc*

The "button document" is a custom object that declares variables of various button types and creates the actual buttons, stringing them together in an event-handler chain. Here's its declaration:

```
type
  TButtonDoc = object(TEvtHandler)

      fTheWindow: WindowPtr;     { Window to display buttons in }

      procedure TButtonDoc.Init;
      procedure TButtonDoc.Display;
      function TButtonDoc.DoMouseCommand (event: EventRecord): Boolean;
      override;
      function TButtonDoc.DoKeyCommand (event: EventRecord): Boolean;
      override;
      procedure TButtonDoc.Free;  { Needs to free its buttons, too }
      override;
    end;  { Class TButtonDoc }
```

Note that because *TButtonDoc* is declared as an event handler, its *DoMouseCommand* and *DoKeyCommand* methods can forward events to the chain of buttons. *Init* creates the buttons and forms the event chain. *Display* sends a Display message to each button in the chain. *DoMouseCommand* and *DoKeyCommand* send events along the chain. *Free* sends a Free message to each button in the chain and then frees the *TButtonDoc* object.

The Buttons

Here's the list of button variables declared in *TButtonDoc*:

```
var
  { A variety of button objects, plus a key-eater }
  beepBtn: TBeepButton;                { Beeps }
  moveBtn: TMoveButton;                { Moves to a new location }
  hideBtn: THideButton;                { Hides itself }
  showBtn: TShowButton;                { Shows the hidden button }
  dimBtn: TDimButton;                  { Dims itself }
  checkBx: TCheckBox;                  { Gets checked or unchecked }
  radBtn: TRadioButton;                { Turns on or off }
  cluster: TRadioCluster;              { Shows several related radio buttons }
  transBtn: TMyTransparentButton;      { Flashes and shows or hides its title }
  keyEater: TKeyEater;                 { Claims every other key-down }
```

The "key-eater" object simulates a text field or a file—some kind of data sink—that has priority for key-ins. It takes the key-down events that it wants (in this case, every other key-in) and lets the default button (Beep) have a crack at anything left. The Beep button, as the default, responds to Return or Enter as if it has been clicked but only if the key-eater has not eaten the key-in. The key-eater lets us show how the default button can coexist with other objects that take key-down events. We examined the general mechanism in Chapter 4.

Compiling the Example

Full source code for program Button Demo and its supporting units is on the code disk. To compile Button Demo, your files should be in the order in which they appear in the THINK Pascal project window shown in Figure 5-4.

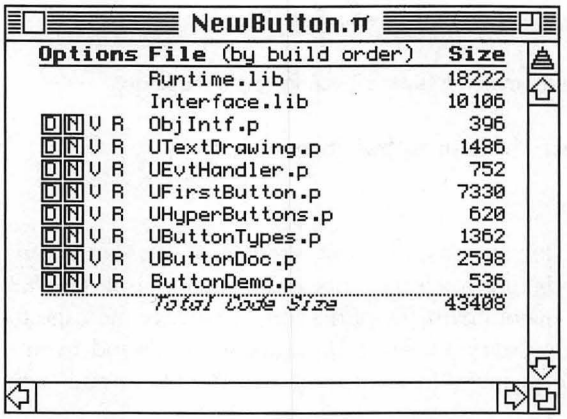


Figure 5-4.
Compilation order for Button Demo program.

About the ButtonDemo Files

The first files in the project window are the usual ones in an object-oriented THINK Pascal project: Runtime.lib, Interface.lib, and ObjIntf.p (the version that came with your THINK Pascal). See the discussion for Figures 3-1 through 3-3 in Chapter 3 and your THINK Pascal documentation. The other files are peculiar to this project.

The file UTextDrawing.p was written to support and simplify drawing titles in buttons. You can see it on the code disk, in the folder Buttons, Ch 5.

The file UEvtHandler.p contains the definition of the *TEvtHandler* class, which we examined in part in Chapter 4.

The files UFirstButton.p and UHyperButtons.p contain class declarations for all our button classes. UFirstButton.p contains classes *TButton*, *TCheckBox*, *TRadioButton*, and *TRadioCluster*. UHyperButtons.p currently contains class *TTRansparentButton*. Someday it might contain other classes, such as *TIconButton*, *TRectButton*, and other HyperCard-like button classes. We name the file UHyperButtons.p because its contents have been modeled on HyperCard buttons.

The file UButtonTypes.p contains application-specific subclasses of the abstract button classes in UFirstButton.p and UHyperButtons.p. For example, there's a class declaration for class *TBeepButton*, the class of buttons that beep when clicked.

The file UButtonDoc.p contains the definition of class *TButtonDoc*, which creates the program's button objects and manages them. In particular, *TButtonDoc* serves as a conduit for events passed to the buttons from the main program.

The file ButtonDemo.p contains the main program. It creates a *TButtonDoc* object and runs a Macintosh event loop to pass events to the buttons.

Summary

In this chapter we saw how programming by differences can easily generate a variety of useful classes from the abstract *TButton* class by means of subclassing, inheritance, and overriding. We also designed classes and wrote some methods.

Projects

- Develop the class *TRectButton*, which, as in HyperCard, displays a button as a rectangle with the title centered. (This is not the same as our transparent button; it's not transparent.) Make the lines of the rectangle one pixel thick.
- Develop the class *TIconButton*, which, given the resource ID of an ICON resource, uses the icon for display. You can use *GetIcon* and *PlotIcon* (see *Inside Macintosh,* I-473). As in HyperCard, the button title should appear in 9-point Geneva below the icon.
- Can you think of any other capabilities our *TButton* buttons should have?

- Implement a solution to save and restore the bits underlying a *TTransparentButton*. You'll need to save the bits to an off-screen bitmap before displaying the button if the title is showing and then restore the bits if the button is redisplayed without its title. Hint: Read about bitmaps in *Inside Macintosh*, I-144, consult *Macintosh Revealed*, Volume 1 (Chernicoff 1985), and take a look at class *CBitMap* in the THINK Class Library.

6

MANY SHAPES

In this chapter, we'll focus on polymorphism—what it is and how to use it. We'll write an example program that creates and uses a small bin of auto parts, simulating inventory activities in an auto parts store. Along the way, we'll look at

- The role of object hierarchies in polymorphism
- The assignment of objects
- Calling the methods of a "contained" object
- Polymorphic arrays
- Container, or collection, classes
- More about writing methods
- Extending an existing hierarchy by adding new object types
- "Promoting" objects by typecasting
- Manager objects

Polymorphism

Thanks to the mechanism of runtime binding, which we'll go into in detail in Chapter 9, polymorphic objects can appear to take on a variety of different shapes. An ancestor object can "contain" or reference an object of any of its descendant classes. The ancestor object has the corollary ability to call a method of ("send a message to") a contained object and have the appropriate code be executed at runtime.

Recall Chapter 2's array of foods. Each element of the array was of type *TFood*, a broad class of objects that have a family resemblance. All are foods of one kind or another, so all share the general properties of food.

Any array element can be assigned an instance of any class descended from *TFood*, such as *TApple*, *TOrange*, or *THamburger*. This kind of assignment is what we mean by "containing."

Recall, too, that we were able to loop through the food array, sending each contained object a common message, such as Prepare or Serve. Of course, each different kind of food object implements its method corresponding to such a method differently.

Assigning Objects

Let's examine the rules for objects assignment. Given a class hierarchy like the one shown in Figure 6-1,

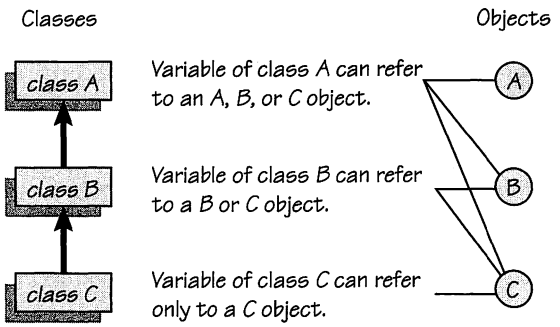


Figure 6-1.
Polymorphism in a hierarchy.

we can assign objects of class *B* or class *C* to a variable of class *A* (and class *C* objects to a class *B* variable). Given these variable declarations,

```
var
  anA, anotherA: A;
  aB: B;
  aC: C;
```

we can make assignments such as

```
anA := anotherA;
anA := aB;
anA := aC;
aB := aC;
```

although not

```
aC := anA;          { Illegal assignment }
aC := aB;           { Illegal assignment }
aB := anA;          { Illegal assignment }
```

Any object can be assigned to a variable of one of its ancestor classes, but an ancestor object can't be assigned to a variable of any of its descendants.

Runtime binding makes the ascending assignments possible and gives great flexibility to OOP code.

When a class *C* object is assigned to a class *A* variable,

```
var
  anA: A;
  aC: C;
⋮
New(aC);
aC.Init;
⋮
anA := aC;
```

we can think of variable *anA* as "containing" the object *aC*. In reality, *anA* references *aC* through an object reference (implemented with a Macintosh handle), but it's convenient to think of *anA* as containing *aC* in the same way that a variable of type Integer contains an integer value.

Calling Methods of Polymorphic Objects

Suppose that class *A* and class *C* both have a method called *DoX*. Given the assignment of *aC* to *anA*, we can write something like this:

```
anA.DoX;        { Polymorphic method call }
```

The actual method executed will be the one belonging to the class *C* object that is referenced by *anA* (after the assignment). In other words, even though *anA* is of class *A*, the program won't execute *anA.DoX*—it correctly executes *aC.DoX*. Likewise, if we assign a class *B* object to *anA*, the program executes the *B.DoX* method. At runtime, the actual object referenced by *anA* is used to locate and call the appropriate piece of method code. This is OOP's famous runtime binding.

Multiple References

Don't mistake an object reference variable—*anA*, for instance—for the object itself. Remember that in reality the object is a block of storage in the heap and that *anA* is a handle to the block of storage.

And note that in assigning *aC* to *anA* we set up a situation in which the actual object has two separate references to it: *anA* and *aC*. This is what Grady Booch, in his *Object-Oriented Design: With Applications* (Booch 1990), calls "structural sharing," and there's some danger in it. For example, if we free the object by means of the message

```
aC.Free;        { Calls inherited Free method }
```

the block of storage no longer exists and *anA* is an invalid handle. If we subsequently try to access the object through *anA*, say to send it a message, we'll produce a runtime error.

We'll have more to say about this topic in Part 3.

Different Strokes

The standard example of polymorphism in the OOP literature uses a hierarchy of geometric shapes, as shown in Figure 6-2.

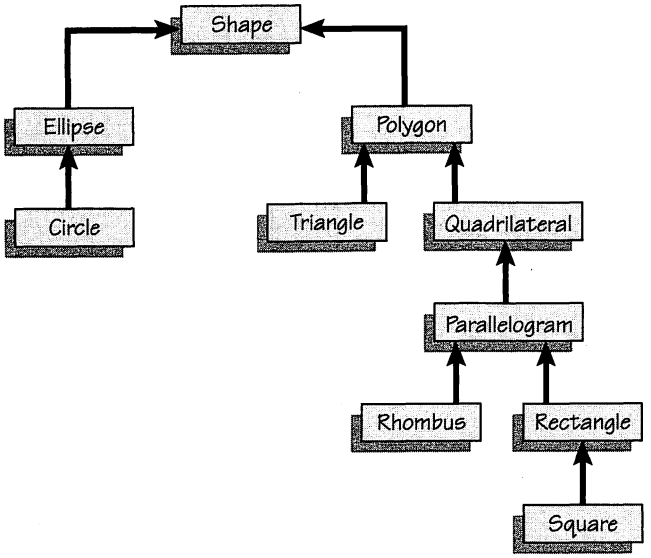


Figure 6-2.
A hierarchy of shapes.

We've already seen, in the food hierarchy example in Chapter 2, that we can declare an array of an abstract class such as *TFood* or *TShape*:

```
var
  shapes: array[1..numShapes] of TShape;
```

Then, if we can assign *aC* to its ancestor *anA*, we can assign any descendant shape object to an element of the array:

```
shapes[3] := aSquare;
shapes[4] := aRectangle;
{ And so on... }
```

Once the array is filled with a variety of shape objects, we can loop through the array, sending each object a Draw message:

```
for i := 1 to numShapes do
  shapes[i].Draw;
```

This results in calling the *Draw* method for each shape object in the array. Of course, the *Draw* method of a *TSquare* object will be different in some respects from that of a *TRectangle* or a *TParallelogram* object. *TSquare.Draw* could override

TRectangle.Draw to take advantage of the fact that squares have sides of equal length, which might simplify the drawing a bit. Polymorphism lets you ask each object to draw itself—and each object responds in the appropriate way.

Containers

Polymorphic objects are often called "containers" or "container classes." Unlike in standard Pascal, in Object Pascal we can assign objects of different types to an object variable.

A container can be a single variable, a structured variable like an array, or some other structure, such as a linked list or a queue. An alternative term is "collection." We'll usually say "container" when we mean a single variable and "collection" when we mean something more complex. The term "collection" can accommodate both a group of unordered items, often of multiple types—although some collections contain only one type—and a group of items ordered in some way.

In Part 3, we'll look more closely at containers and collections.

The Example Program Parts Bin

Now we'll use some real code to explore the ins and outs of using polymorphism. We'll develop a rudimentary auto parts inventory system—with the focus on polymorphism more than on the realistic conduct of an inventory program.

The program Parts Bin uses a polymorphic array, much like the array of shapes we've already looked at. It first defines the array type *Bin*:

```
type
  Bin: array[1..kBinSize] of TMiscPart;
```

and then declares the variable *aBin*:

```
var
  aBin: Bin        { A miscellaneous parts bin }
```

Imagine a small auto parts store with its parts mostly categorized—but with a few small, miscellaneous parts just heaped in a common bin. The variable *bin* can hold *kBinSize* of these miscellaneous parts. Note that the base class of the array is *TMiscPart*: All objects in the bin must be descendants of *TMiscPart* in order to be assignable to the array's elements.

> **NOTE:** *By convention, we prefix constants used by a class with the letter* k, *although we'll occassionally use a* c *prefix instead. Later in this chapter, we'll use the* c *prefix with a special array of "auto parts type codes." We can't declare constants inside a class declaration, but we must declare them somewhere before their use in method bodies.*

The Auto Parts Hierarchy

In order to make a polymorphic array capable of holding multiple kinds of auto parts, we need a hierarchy of auto parts classes. (We could develop the array by using standard Pascal's variant records, but the object solution is more elegant and easier to work with. And polymorphism is much better implemented with class hierarchies than with variant records.)

The class hierarchy is fundamental to polymorphism. A freestanding object class, with no ancestors and no descendants, doesn't allow the assignment of other types to its variables (although you can send a common message to several freestanding object classes). If class *A* is not part of a hierarchy, you can assign class *A* objects to class *A* variables, of course, but that's not polymorphism. Polymorphism rests on the ability to assign descendant objects to ancestor variables.

Now that we appreciate the importance for our purposes of an auto parts hierarchy, let's look at Figure 6-3 to see how we might set ours up.

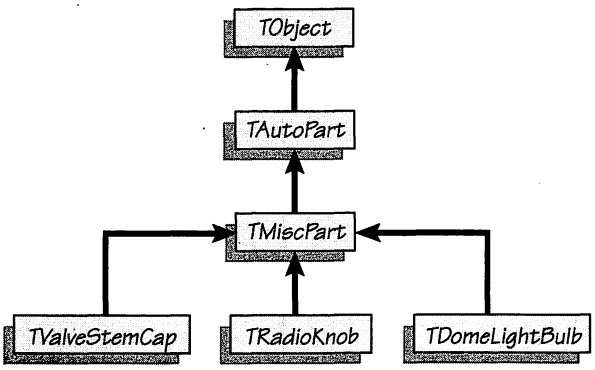


Figure 6-3.
The auto parts class hierarchy.

TMiscPart is ancestor to a group of classes representing trivial auto parts, the kinds of things we might decide to call "miscellaneous" if we ran a pretty relaxed shop. A full-scale auto parts inventory system would have many subclasses of *TAutoPart* besides *TMiscPart*—such as *TBattery*, *TTire*, *TRearBumper*, and so on—but we won't be concerned with them here.

Designing the hierarchy

Because we want to take advantage of polymorphism, we'll design not a single class at a time but a whole hierarchy of classes all at once. This approach to class design makes sense not only for objects that will take advantage of polymorphism but also for objects that we expect to create through subclassing later on. Moving from general classes at the top to more specific classes toward the bottom, a hierarchy provides lots of places for subclasses to "hook in." A class that's pretty similar to one of

the specialized classes in the hierarchy might be hooked in at a low level, and a class that will differ a great deal from most of the existing classes might hook in at a much higher, more general, level. Figure 6-4 suggests how such an elaboration of a class hierarchy might proceed.

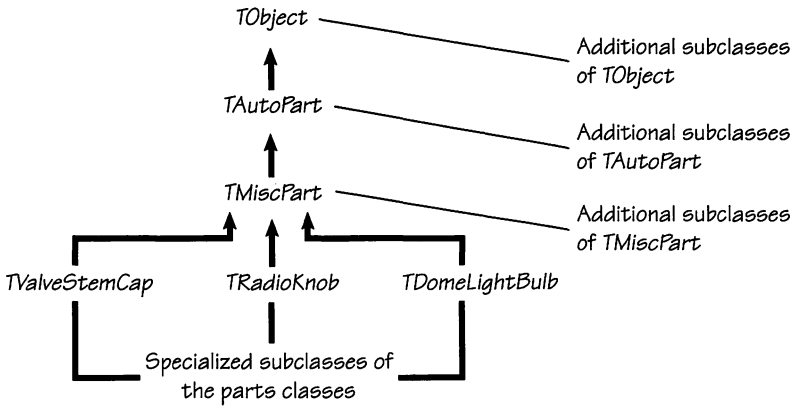


Figure 6-4.
Hooks in the class hierarchy.

All auto parts are bound to have some characteristics in common, so we'll embody those characteristics as far up the object hierarchy as possible—in class *TAutoPart*. *TAutoPart* is itself a descendant of the abstract class *TObject*, but that's merely to bequeath *TObject*'s *Free* method to the whole auto parts hierarchy so that any auto parts object can free itself when we finish with it.

By building common features into *TAutoPart*, we allow all subclasses to inherit them. Of course, some subclasses might need more instance variables and more or different methods, so we'll probably need to do some extending and overriding.

Why aren't classes *TValveStemCap*, *TRadioKnob*, and *TDomeLightBulb* descended directly from *TAutoPart*? Why do we need an intervening *TMiscPart* class? Because we're writing the portion of the inventory system that manages a bin of miscellaneous parts. The base type of the bin array is *TMiscPart* rather than *TAutoPart*—that precludes assigning nonmiscellaneous parts to the miscellaneous bin. For instance, a *TRearBumper* object won't be assigned to the bin unless it's a descendant of *TMiscPart*. That gives us some control over polymorphic assignment so that it won't get out of hand. (We could be utterly general, using an array of *TObject* if we wanted to—but then any object could be assigned to the array. We'll have more to say about this in Part 3.)

Both *TAutoPart* and *TMiscPart* are abstract classes: We'll never declare an instance of either. They exist to pass on abilities to their descendants, and to "abstract out"

the common properties of numerous specific objects. Abstract classes serve well as the base types of polymorphic arrays. We'll discover more uses for abstract classes in Chapter 8.

But we mean the classes *TValveStemCap*, *TRadioKnob*, and *TDomeLightBulb* to be instantiated. We'll be filling the parts bin with instances of these classes.

What can we do with the bin?

Among the inventory operations we'll perform on the bin are

- Putting parts objects in
- Taking parts objects out
- Looping through the bin to prepare an inventory report
- Searching the bin for a certain object, or for an object of a given kind

Putting parts into the bin is a matter of assigning objects to the array's elements, just as we'd assign integers to the elements of an integer array using array-access notation:

```
aBin[3] := anObject;
```

Taking parts out of the bin is a little more complex because each array element in this case is an object reference (in effect, a handle). We access each as usual, but then we need to remove the object from the array by assigning the array element the value *nil*, which marks the object's place as "empty":

```
currentObject := aBin[i];
aBin[i] := nil;
```

Preparing the inventory report is where polymorphism really comes into play. Because *aBin*'s array elements can contain any one of three different parts classes, we simply rely on the fact that every auto parts object has a characteristic *Report* method. As we loop through the array, we call each object's *Report* method, and the method responds by reporting (displaying) the object's data:

```
for i := 1 to kBinSize do
  if aBin[i] <> nil then
    aBin[i].Report;
```

Note that we have to test each element for *nil* before calling its *Report* method. Calling a method of an object that doesn't exist results in a runtime error. First, we need to see whether the array really contains the object.

Searching for items in the bin also involves looping through the array, but we're likely to use different methods. If we're searching for any item of class *TRadioKnob*, for instance, we'll call the *PartTypeOf* method for each object in the bin. *PartTypeOf* reports the part type of the item we're currently checking as in the code on the next page.

```
var
  chosenItem: TMiscPart;
  i, itsSlot: Integer;
⋮
i := 1;
while i <= kBinSize do
  begin
    if aBin[i] <> nil then     { Something in element? }
      begin
        { Does element have right type? }
        if aBin[i].PartTypeOf = desiredType then
          begin
            chosenItem := aBin[i];
            itsSlot := i;
            Leave;
          end;  { if }
      end;  { if }
    i := i + 1;                { Continue searching }
  end;  { while }
```

Alternatively, this will work:

```
⋮
if Member(aBin[i], TRadioKnob) then
  begin
    chosenItem := aBin[i];
    itsSlot := i;
    Leave;
  end;
```

Notice, though, that it's hard to encapsulate the alternative code in a general procedure or method. We can't pass a name like *TRadioKnob* as a parameter to use in the *Member* function call.

To search for a particular object, we'd need to add more information to the objects, such as a serial number or an inventory number. That would entail using methods to set and get the unique ID information. Then we could use the method that gets the unique ID in a loop like those above. (We'll look at a general solution soon.)

In any case, except for assignment, accessing the bin's contents is always done by methods such as *PartTypeOf*.

Developing the *TAutoPart* Class

TAutoPart embodies most of the auto parts class hierarchy's functionality. This is in accord with the principle of putting as much functionality as we can as high in the object hierarchy as we can so that more classes can inherit it.

No doubt any auto part in the store has at least these common characteristics:

- A part number, using some national or local convention
- A description, such as "Knob, Auto Radio"
- A vendor—the manufacturer or seller of the part
- A unit price

And for the operations we might want to perform on that *TAutoPart* data, we can include

- Methods to set and get each instance variable, as usual
- An initialization method
- A method to tell the type of a part (radio knob, valve stem cap, rear bumper)
- A method to report the part's data, for inventory
- Perhaps a method to count parts of a type and determine when to reorder (We'll skip this one here.)

Just a brief comment on methods to set and get the values of instance variables: To maintain the integrity of our parts data, we might not want to make certain methods available. For example, an item's part number is fixed; we don't want to provide a method to set it. Once it's initialized, it's untouchable. Similarly, some kinds of data might require security limitations. For instance, in a personnel database, salary information might be confidential, so a method to retrieve it would probably include security processing to establish legal access.

Here's the declaration of *TAutoPart*:

```
type
  TAutoPart = object(TObject)

      fPartNumber: Integer;          { 5 digits }
      fDescription: Str255;          { 15 characters }
      fVendor: Str255;               { 15 characters }
      fUnitPrice: Real;              { 2 decimal places }

      procedure Init (partNum: Integer; desc, vendor: Str255; unitPrice: Real);
        { Initialize an auto part object with its part information }
      procedure SetDescription (desc: Str255);
        { Correct or change the part's description }
      procedure SetVendor (vendor: Str255);
        { Correct or change the part's vendor information }
```

```
    procedure SetUnitPrice (unitPrice: Real);
      { Correct or change the part's unit price }

    function PartTypeOf: Integer;
      { An abstract method }
    function PartNameOf: Str255;
      { An abstract method }
    function PartNumberOf: Integer;
      { Return the part's part number }
    function DescriptionOf: Str255;
      { Return the part's description }
    function VendorOf: Str255;
      { Return the part's vendor name }
    function UnitPriceOf: Real;
      { Return the part's unit price }

    procedure Report;
      { Report the part's information as one formatted line of output }
  end;  { Class TAutoPart }
```

TAutoPart's Methods

The *Init* method and all the methods that set and get instance variables (except two) should seem straightforward to you by this time. They simply set or return the values of a *TAutoPart* object's instance variables. We'll focus now on only three *TAutoPart* methods: *PartTypeOf* and *PartNameOf* (the two methods we've called "abstract") and *Report.*

The *PartTypeOf* and *PartNameOf* methods

How should we implement a *PartTypeOf* method? It should return, in some form or other, the class/type of the object. Turbo Pascal, for MS-DOS machines, has a built-in *TypeOf* function that returns the actual type of a variable. *TypeOf* can be used with objects, too. But in Macintosh Pascals so far, we're limited in this respect. We can't directly recover a variable's type.

Our first thought might be to add an *fPartTypeOf* instance variable to the *TAutoPart* class. We'd make it an integer rather than, say, a string containing the name of the class, and we'd create a set of simple codes to represent the kinds of auto parts in our system:

```
const
  cValveStemCap = 1;
  cRadioKnob = 2;
  cDomeLightBulb = 3;
```

Unfortunately, adding another instance variable to a class is not a step to take lightly. Every single object instance—every auto part in our system—would have to have storage space for that integer code. With thousands of objects, we'd be talking about

twice that many bytes of storage—a potential memory-allocation problem of significant proportions.

Instead, we recognize two things:

- Every kind of part needs a different code.
- We'd like to store that code only once per kind, not once per part.

The solution is to leave the *PartTypeOf* method undefined at the *TAutoPart* level of the hierarchy. We mark the method at that level as an abstract method, which means that subclasses must override it to fill in the body. In the *TAutoPart* implementation, the *PartTypeOf* method is a stub:

```
function TAutoPart.PartTypeOf: Integer;
begin
  { Abstract--descendant part classes must override this method }
end;  { TAutoPart.PartTypeOf }
```

Later in this chapter, we'll see how the *TAutoPart* descendants implement the method. Essentially, each descendant class will provide a method that "knows" the part's type code rather than storing the type code in a variable.

The *PartNameOf* method presents a similar problem, with a similar solution. This method should return a string with the name of the type. We make it an abstract method at the *TAutoPart* level and require descendants to override it.

The *Report* method

The *Report* method is simple. It uses Pascal's *Write* statement to display the object's characteristics as one line of instance variables, just as we'd display a record. All auto parts will do this the same way, so we can define—both declare and implement—*Report* at the *TAutoPart* level and simply let descendants inherit the ability to report themselves.

Abstract Methods vs. Abstract Classes

Don't confuse abstract *methods* with abstract *classes*. An abstract class is one we don't declare any instances of. An abstract method is one we don't— or can't—implement at the current level of the hierarchy. Its implementation is deferred for subclasses to provide by overriding the abstract method.

Some abstract methods *must* be overridden by subclasses. For others, overriding is optional. It's a good idea to label which methods can be overridden optionally and which must be overridden. Object Pascal doesn't provide any kind of mechanism to force an outside programmer to provide the override, so you have to make your requirements known via your documentation.

When we loop through the parts bin sending every object a Report message, the resulting output is a line of information for each object, as many lines as there are objects. We might want to pretty up the report as a whole by first displaying some headings, the date, and so on, then calling the objects' *Report* methods for the body of the report, and finally adding some totals and summary information at the bottom. But that overarching concern is a function of the whole report, not of the individual objects, so we'd write it elsewhere (perhaps in the bin manager class we cover later), not in the objects' *Report* method.

Here's the implementation of *TAutoPart.Report*:

```
procedure TAutoPart.Report;
  const
    kSpace = Chr(32);
begin
  Write(PadRight(self.PartNameOf, 15, kSpace));
  Write('  ', self.fPartNumber :5);
  Write('  ', PadRight(self.fDescription, 20, kSpace));
  Write('  ', PadRight(self.fVendor, 20, kSpace));
  Writeln('  $', self.fUnitPrice :6 :2);
end;  { TAutoPart.Report }
```

The utility function *PadRight* is declared earlier in the Parts Bin program. It left-justifies a string by padding the end of the string with blanks up to the width of the field we want to print it in. The code for *PadRight* and for the rest of the Parts Bin program is provided in full on the code disk.

Report needs to write one piece of information not stored in an instance variable: the name of the part type in string form. It calls *self*'s *PartNameOf* method, which returns the correct name string for whatever kind of part *self* happens to be. (We're careful, though, to directly access data kept in instance variables rather than calling the methods to get it. From outside an object, good practice calls for accessing the data only by means of a method, but from inside the object's own methods, direct access to the instance variables is more efficient. Calling the object's own methods—from inside another of its methods—has a noticeable impact on speed.)

Developing *TAutoPart*'s Descendants

Now let's take a look at *TMiscPart* and then at the classes that represent actual parts.

The *TMiscPart* Class

The *TMiscPart* class exists in the hierarchy only to screen out auto parts objects of types that aren't supposed to be stored in the miscellaneous parts bin. It adds nothing new, and it changes nothing. In fact, its declaration on the next page is a good example of using an "empty" class declaration.

```
type
  TMiscPart = object(TAutoPart)
    end;
```

The class has no new instance variables and no new methods. It inherits everything from its ancestor, *TAutoPart*, and overrides nothing.

But *TMiscPart* also serves as the base type for our polymorphic bin array so that we can assign only objects of its descendant types to the array.

The *TValveStemCap*, *TRadioKnob*, and *TDomeLightBulb* Classes

The descendants of *TMiscPart* are not abstract classes. We intend to create instances of them to store in the bin array. Because we do plan to instantiate these classes, they are required to provide implementations of the abstract methods that *TAutoPart* couldn't implement: *PartTypeOf* and *PartNameOf*.

The descendant types must override these methods that they inherited from *TAutoPart* by means of *TMiscPart*.

The class declarations for *TValveStemCap*, *TRadioKnob*, and *TDomeLightBulb* are so similar that we'll look at only one here. (You can see all three on the code disk.)

```
type
  TValveStemCap = object(TMiscPart)  { One kind of miscellaneous part }

    function PartTypeOf: Integer;    { Return the value of cValveStemCap }
    override;
    function PartNameOf: Str255;     { Return the value of cValveStemCapStr }
    override;
  end;  { Class TValveStemCap }
```

The *TValveStemCap* class inherits everything—except the two methods it overrides—from its ancestors; thus, it has four instance variables and thirteen other methods (including those it inherits from *TAutoPart* and *TObject*).

Implementing *PartTypeOf* and *PartNameOf*

We had to override the two methods because each object must supply its own "type code" (indicating the kind of part it is) and a string version of the type's name. We didn't want to take up valuable storage space in every single object to store that information: Ideally, we'd like to store it only once for the whole class, not once for every instance of the class. The solution is to provide method implementations at this descendant level that "know" the required information. Now we'll develop codes for the part types, representing each kind as an integer value and defining the codes in constant declarations. Using constants works because the information is fixed.

Here are the bodies of *TValveStemCap.PartTypeOf* and *TValveStemCap.PartNameOf*:

```
function TValveStemCap.PartTypeOf: Integer;
  override;
begin
  PartTypeOf := cValveStemCap;        { Part type as integer code }
end;  { TValveStemCap.PartTypeOf }

function TValveStemCap.PartNameOf: Str255;
  override;
begin
  PartNameOf := cValveStemCapStr;  { Part name as string constant }
end;  { TValveStemCap.PartNameOf }
```

The string values reported by the *PartNameOf* methods in the three classes are defined as constants:

```
const
  cValveStemCapStr = 'Valve Stem Cap';
  cRadioKnobStr = 'Radio Knob';
  cDomeLightBulbStr = 'Domelight Bulb';
```

The part type and part name information is "hard-wired" into each class by means of the methods. Each part class "knows" that it's a *TValveStemCap* object (or whatever) and can report its type name as the string *Valve Stem Cap* (or whatever). Every instance of the class can report this information by virtue of its having the appropriate method. No instance has to store the information directly. In general, adding methods to a class is cheaper than adding instance variables.

The folder Parts Bin, Ch 6 on the code disk provides full source code for the auto parts class hierarchy.

Common Method Names

Polymorphic objects are pretty easy and safe to use, but you'll get the best results if you know a few of the ins and outs. Every class up and down the auto parts hierarchy has the same list of methods. *TAutoPart*, *TMiscPart*, *TValveStemCap*, and so on all have these names in their method lists (thanks mainly to inheritance):

- *Init*
- *SetDescription*, *SetVendor*, *SetUnitPrice*
- *PartTypeOf*, *PartNameOf*
- *PartNumberOf*, *DescriptionOf*, *VendorOf*, *UnitPriceOf*
- *Report*
- *Free*, *ShallowFree*, *Clone*, *ShallowClone* (from *TObject*)

These are the methods we can call polymorphically.

At runtime, we can't tell, at any given moment, what object type is actually going to be assigned to ("contained" by), say, array element *aBin[4]*. As Figure 6-5 suggests, the object in the array element *aBin[4]* could be of type *TValveStemCap*, *TRadioKnob*, or *TDomeLightBulb*.

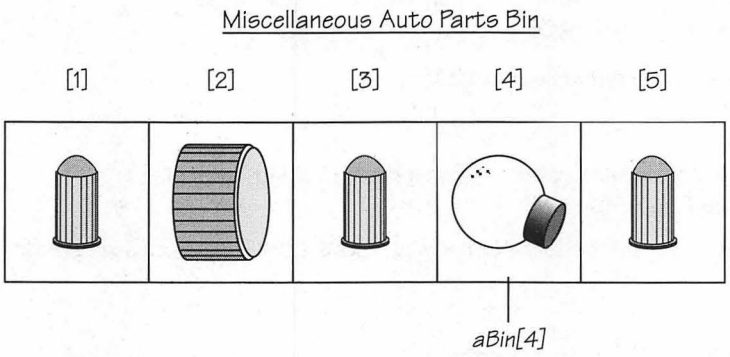


Figure 6-5.
What's in the array element?

However, all the classes in the hierarchy have the same methods, so we can send these rather anonymous messages:

```
aBin[4].Report;                    { Report method of whatever object }
⋮
itsType := aBin[4].PartTypeOf;     { PartTypeOf method of whatever object }
```

Because the array is polymorphic, the method code of the actual object is called, not the corresponding method of *TMiscPart*, the array's base type. (Of course, the call *aBin[4].Report* really calls *TAutoPart*'s *Report* method no matter what object is in *aBin[4]* because the *Report* method is inherited all the way down the hierarchy. But *aBin[4].PartTypeOf* calls different code depending on whether *aBin[4]* contains a valve stem cap, a radio knob, or a domelight bulb. That's because those objects override the *PartTypeOf* method, providing their own code for it.)

Nothing in common

Fine, so far. But suppose we have this hierarchy:

```
type
  A = object                  { No ancestors }

    procedure X;              { A's X method }
    procedure Y;              { A's Y method }
  end;  { Class A }
```

```
B = object(A)                { Ancestor is A }
    { B inherits A's X }
    procedure Y;             { B's Y method }
    override;
    procedure Z;             { New method in B }
  end;  { Class B }
```

Here, class *B* shares *A*'s *X* method name (by inheritance) and *A*'s *Y* method name (although *B* overrides *Y*). We can legally send messages like

```
var
  anAObj: A;
  aBObj: B;
⋮
anAObj.X;                    { Any object in anAObj has an X method }
anAObj.Y;                    { So does any B object }
aBObj.X;
aBObj.Y;
aBObj.Z;
```

but *not*

```
anAObj.Z;                    { Compiler error }
```

To the compiler, that last message looks like a call to *A*'s *Z* method—but *A* has no *Z*. The compiler can't tell that, at runtime, *anAObj* will actually reference a class *B* object. It could make the call if it knew so. (It's hard to tell what might be assigned to *anAObj* at runtime—it might really be a class *A* object. The compiler is right to complain.)

Hence, classes must share method names if polymorphic method calls are to work. Of course, some of the identically named methods might be overrides that do entirely different things. That doesn't matter, and, in fact, that's what makes polymorphism so powerful.

The wrong base type

Part of our task in designing an object class hierarchy to take advantage of polymorphism is to ensure that all classes in the hierarchy share any method that might be called in a polymorphic situation.

But let's look again at the auto parts hierarchy. Class *TObject* appears to violate what we've just said. It's part of our hierarchy, yet it lacks methods—such as *Report*—that are sure to be called polymorphically. Is that a problem?

That lack would cause trouble only if *TObject* were used as the base type of our bin array. If we had declared

```
type
  Bin: array[1..kBinSize] of TObject;
```

the compiler would reject messages such as

```
aBin[4].Report;
```

Element *aBin[4]* is of type *TObject*, but *TObject* has no *Report* method. (Never mind what's actually in *aBin[4]*.)

Only the class that actually gets used as the base type for polymorphism and its descendants must share method names. (Later, in Part 3, we'll turn around and use *TObject* and classes like it as a base type for polymorphic structures such as linked lists, to make them as general as possible. But there will be a cost, and we'll have to deal with the problem of calling methods of the unknown contained objects.)

Completely the same?

Must all classes in the hierarchy—from the base type on down, that is—share exactly the same method lists? No. They need to share only any method names that will be called in polymorphic situations—methods, that is, of objects contained polymorphically in ancestral-type variables, as in our bin array. Some classes might have other methods that will never be called that way.

Accessing the Data in Polymorphic Objects

When we can't predict what kind of object will actually occupy a polymorphic variable, how can we reliably access the object in the variable to use its data and to do things with the object?

Common Methods Again

We've already seen that, if all objects that can be stored in the variable have common methods, we can simply call those methods polymorphically to extract or update the object's data—no matter what the object is.

Using the *Member* Function

If we absolutely must know what type an object is, we can use the *Member* function:

```
if Member(X, Class1) then
  :
else if Member(X, Class2) then
  :
else
  :
```

This answers the question "Is *X* of type *Class1*?" but it can't answer the question "What type is *X*?" To decide what *X*'s class is, we have to run *X* through a sequence of *if* statements that cover *all* the possible types. If it will do the job, calling common methods is usually simpler and more direct.

We'll look at the *Member* function in more detail in Chapter 9.

Promoting the Object

If we know the object's type but the containing object variable can't respond to a message (because its class doesn't have that method), we can "promote" the object by typecasting and then send it the message:

```
var
  anAObj: A;
  aBObj, anotherBObj: B;
⋮
New(aBObj);                          { Create an object }
⋮
anAObj := aBObj;                     { Assign B object to A-type variable }
⋮
anotherBObj := B(anAObj);            { Promote to B-type by typecasting }
```

where *B(anAObj)* is a typecast, converting an *A* class variable to a *B* class.

We can also use typecasting in more shorthand ways. For example, we can typecast on the fly, while sending a message:

```
B(anAObj).DoMethod;
result := B(anAObj).DoFunctionMethod(a, b, c);
```

Of course, before typecasting we need to know what type the variable actually contains. The *Member* function can help here, by determining the contained type before we make the typecast:

```
if Member(anAObj, B) then            { The test }
  anotherBObj := B(anAObj);          { The typecast }
```

Extending the Bin

Suppose that we already have a unit, *UPartsBin*, defining *TAutoPart*, *TMiscPart*, and the three descendant classes of *TMiscPart* and that we've used the classes in a working program. Now suppose that we want to extend the program by extending the parts bin to contain another type of miscellaneous object, a door lock knob type. Figure 6-6 on the next page shows the parts hierarchy with the new class added. We didn't foresee a need for this object originally, but, wonder of wonders, the requirements have now changed. How do we add the new descendant class?

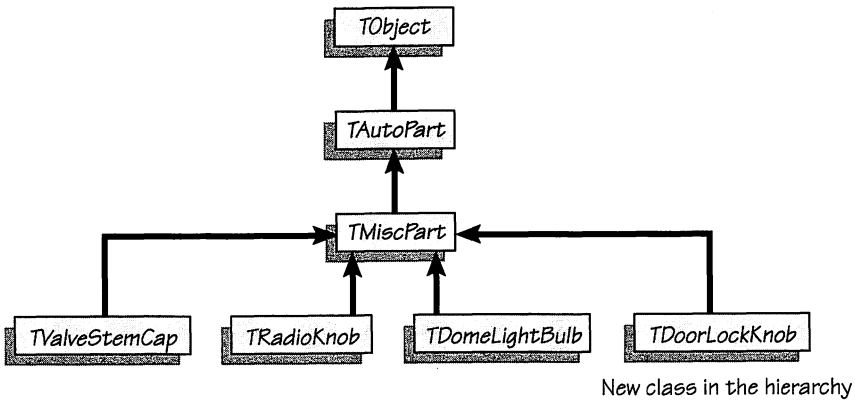


Figure 6-6.
Extending the bin to contain TDoorLockKnob.

Defining a New Subclass of *TMiscPart*

We'll need to write a new subclass, *TDoorLockKnob*. It can be in its own unit, listing *UPartsBin* in its *uses* clause. The class's declaration will look much like those of *TValveStemCap* and its siblings:

```
type
  TDoorLockKnob = object(TMiscPart)

    function PartTypeOf: Integer;
    override;
    function PartNameOf: Str255;
    override;
  end;  { Class TDoorLockKnob }
```

The class overrides the same two methods and inherits all the characteristics of its earlier siblings.

The implementations for the *PartTypeOf* and *PartNameOf* methods are just like those in *TValveStemCap* and the other types, although we'll need some new constants—we'll look at them next.

Adding Some New Type Constants

In order to develop the overridden methods in *TDoorLockKnob*, we need two new constants, one for its part type and one for the string version of the part type:

```
const
  cDoorLockKnob = 4;          { Following our earlier code sequence }
  cDoorLockKnobStr = 'Door Lock Knob';
```

We can simply insert these constants at the beginning of our new *UDoorLockKnob* unit. With the other constants imported from *UPartsBin* by means of the *uses* clause, the new constants fit right into the scheme of things.

Adding *TDoorLockKnob* to the Calling Programs

Our program that manipulates the parts bin now needs a small modification or two so that the new part type will be included in its repertoire. First, we modify the program to add *UDoorLockKnob* to the *uses* clause. Everything declared in that unit's interface part will be imported into the calling program.

Second, we need to add some code to create and use *TDoorLockKnob* objects. We also need to change how the main program initializes the utility variable *LastPartType* (declared in the *NewDomeLightBulb* function in *UPartsBin*) to *cDoorLockKnob* instead of *cDomeLightBulb*. (That's just a handy item used in *for* loops that cycle through all possible bin content types.)

By using constants for our parts types instead of an enumerated type, we make extension much easier, too. We can always add another constant to a list, and we can do it in another unit. But we wouldn't be able to extend an enumerated type without starting from scratch.

Although we do have to recompile the calling program, we don't have to change anything about the parts bin array, and we don't have to modify 1,000 *case* statements scattered all over the place. Our classes, with their methods, and our list of constants as codes for the parts types will handle as many extensions as we like.

Encapsulating the Parts Bin with a Manager Object

As it stands now, the parts bin is merely an array. To work with it, we write our own *if* statements, *for* loops, and so on. We create objects ourselves and assign them directly to the bin. We remove objects directly, too, and it's up to us to remember to set removed elements to *nil* in the array.

A better—and safer—approach would be to create a parts bin object, as shown in Figure 6-7 on the next page. Such an object would be a manager for the bin. The object would contain the bin array as one of its instance variables, and it would have methods to add parts to the bin, remove parts, generate reports of various kinds, and search for parts.

A bin manager object would encapsulate the entire bin mechanism with a higher-level abstraction. Instead of just a naked array, a miscellaneous parts bin would now be an object in its own right.

Our demonstration program in this chapter doesn't use a parts bin object, but see the projects at the end of the chapter.

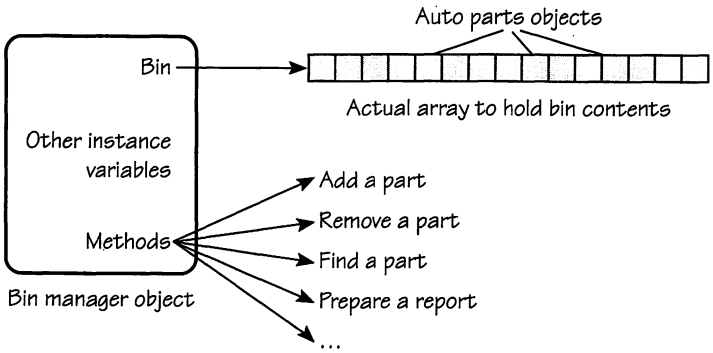


Figure 6-7.
A bin manager object and a bin array with its contents.

Here's the declaration for a hypothetical bin object:

```
type
  TMiscBin = object(TObject)

      fBin: Bin;                { The bin array }
      fNextEmptySlot: Integer;
      fTheError: BinErrorTypes;

      procedure Init;
        { Creates an empty bin array with nil in slots }
      procedure AddPart(p: TMiscPart);
        { Puts part in next available slot }
      function GetPartByType(partType: Integer): TMiscPart;
        { Gets first part of given type }
      function GetPartByDesc(desc: Str255): TMiscPart;
        { Gets first part with given description }
      function GetPartByVendor(vendor: Str255): TMiscPart;
        { Gets first part with given vendor }
      function IsFull: Boolean;
        { Returns true if the bin is full }
      procedure Reorder(partType: Integer);
        { Generates paperwork to reorder part based on }
        {  current in-stock level }
      procedure Report;
        { Produces a report of all parts in the bin }
      procedure ReportByType;
        { Produces a report sorted by type }
      function BinError: BinErrorTypes;
        { Checks after other calls to see if there was }
        {  an error }
```

```
    procedure Free;
    override;
      { Frees bin contents, and then frees bin object }
  end;  { Class TMiscBin }
```

The bin array is now declared inside *TMiscBin* as

```
fBin: Bin;        { A TMiscBin instance variable }
```

The Methods

TMiscBin has methods to create and initialize an empty bin array, add parts objects to the bin, remove parts, do reporting for the whole bin, and so on.

The reporting methods work by looping through the array, sending each object a Report message, just as we showed earlier in our main program. But now the loop is encapsulated in an object method.

TMiscBin overrides *TObject.Free,* so it can send *Free* to each parts object currently in the bin and then free the bin object.

In all of the methods, we should protect ourselves from errors such as trying to add to a full bin, trying to find an object in an empty bin, trying to send a message to an empty array element, and so on. We could add a simple error-handling mechanism to facilitate these checks. If an error is found, the variable *fTheError* would be set with an enumerated type (not shown). The calling program would then check the results of its messages to the bin object by calling the *BinError* method.

We'll show many more such manager objects throughout the book—anywhere we need to encapsulate some complex activity or data structure with an easy, clean interface. The principle of encapsulation is valuable.

Using the Parts Bin Object

To use the bin object in a program, we would first create and initialize a new parts bin object. Then we would add parts to it, remove parts from it, and so on. We would prepare reports of all items in the bin or of all items sorted by type. Because the bin is a subclass of *TObject,* we could dispose of the bin and its contents by sending *aBin.Free.*

```
var
  aBin: TMiscBin;
  p: TRadioKnob;
  q: TValveStemCap;
  r: TDomeLightBulb;
  aType: Integer;
  :
New(aBin);                    { Create and initialize the bin itself }
```

```
aBin.Init;
:
New(p);                                  { Create an object and add it to the bin }
aBin.AddPart(p);
New(q);
aBin.AddPart(q);
New(r);
aBin.AddPart(r);
aBin.Report;                             { Produce a report of all items in order }
aBin.ReportByType;                       { Produce a report sorted by type }
aType := cValveStemCap;
thePart := aBin.GetPartByType(aType);    { Remove a part }
aBin.Free;                               { Free the bin and all of its contents }
```

Compiling the Example

To compile the Parts Bin example in this chapter, compile the files in the order in which they appear in the project window shown in Figure 6-8.

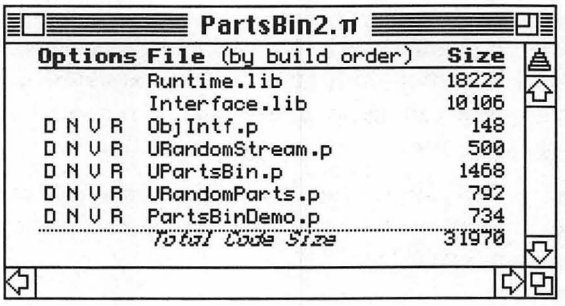


Figure 6-8.
THINK Pascal project window showing compilation order for Parts Bin example program.

The file ObjIntf.p comes with your THINK Pascal package. ObjIntf.p defines the class *TObject*, from which other classes in the program inherit.

The other files are on the code disk. The file URandomStream.p provides a class of objects that can generate several kinds of random numbers. It's used here to generate "random" auto parts for the demo, and you'll see it again in Part 2, where it underlies a class of dice. We'll go into class *TRandom* later.

The file UPartsBin.p defines the *TAutoPart* class and its subclasses.

The file URandomParts.p provides a number of ordinary functions that use a random number–generating object to generate random auto parts for the bin.

The file PartsBinDemo.p contains the main program, which creates a parts bin object and exercises it in various ways.

Summary

In this chapter we saw how we can use polymorphism to store objects of multiple classes in an object class variable. We also saw how the same message sent to different object classes invokes the correct method code for each object class.

We've looked at designing and using the object class hierarchies that are essential to polymorphism. In Chapter 8 and Part 3 we'll look further at object class hierarchies.

Through the auto parts bin example, we've seen the important techniques and potential pitfalls in handling polymorphic objects.

Finally, we've looked at the concepts of containers and manager objects.

The example program is too large to show in its entirety in the chapter, so you can see the code for *UPartsBin* and a small demo program (which, by the way, also shows a random number–generating object) on the code disk.

Projects

- Implement the parts bin manager object, class *TMiscBin.*
- Try out the techniques for extending an existing program with new classes by writing and testing the addition of class *TDoorLockKnob* to the parts bin. You can base your work on the discussion in "Extending the Bin," which begins on page 129. What modifications to UPartsBin are required? What modifications to the main (demo) program on the code disk are required?

7

OOP COOKBOOK

Now let's pull together some loose ends and summarize the important practical facts. In this chapter, we'll assemble a collection of OOP techniques and do-it-yourself guidelines in a format that's handy for reference. We'll look at

- Designing classes
- Creating, initializing, using, assigning, and disposing of objects
- Using units
- Compiling, debugging, and optimizing objects
- Writing methods

Designing Classes: Some Guidelines

Let's start with several brief guidelines for designing classes. We'll look at application design in Chapter 10.

Design Hierarchies Rather Than Standalone Classes

Most of the benefits of OOP arise from capitalizing on inheritance and programming by differences and from using polymorphism. Inheritance pays off through subclassing—which creates hierarchies. And polymorphism is a natural property of hierarchies. Standalone classes can be useful, but they don't benefit from inheritance and they're not normally polymorphic.

Take a class of button objects like the ones we developed in Chapter 4. Looking down the button object class hierarchy, we can project various subclasses of buttons: radio buttons, check boxes, iconic buttons, transparent buttons, and so on. A *TButton* class should make building those varieties of buttons as easy as possible, abstracting out their "buttonness" and letting each subclass differentiate itself through details accomplished by adding new capabilities and overriding existing ones. Figure 7-1 on the next page shows such an elaboration of our old button object class hierarchy.

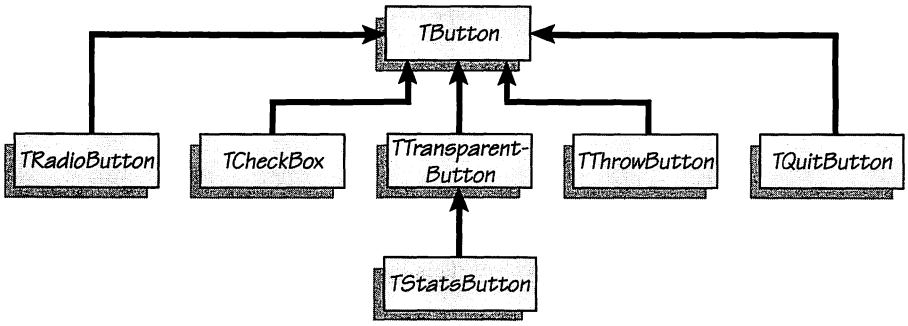


Figure 7-1.
Button descendants—from TButton *on down.*

Looking back up the hierarchy from *TButton*, on the other hand, we might see an ultimate *TControl* class. Other eventual descendants of *TControl* besides *TButton* might include *TScrollbar*, *TPopup*, and perhaps *TMenu*. Figure 7-2 shows *TButton*'s ancestor hierarchy, giving you a picture that's close to the actual MacApp control hierarchy. (In MacApp, radio buttons and check boxes aren't subclasses of *TButton*.)

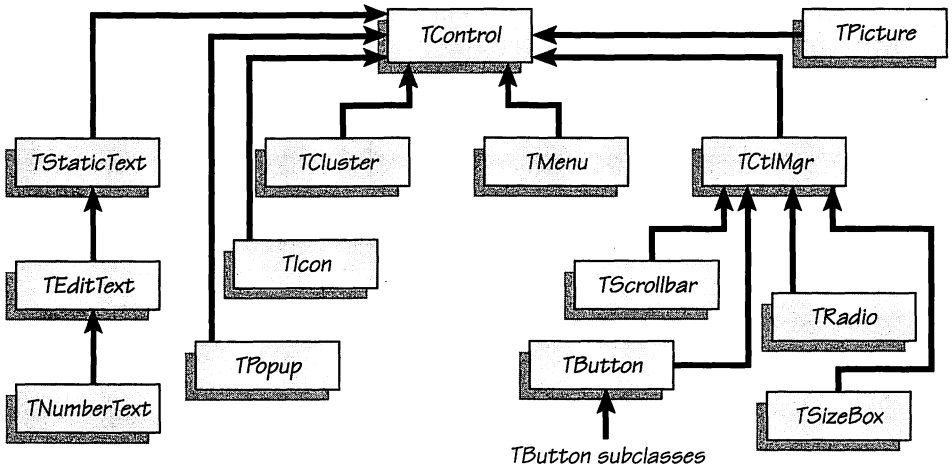


Figure 7-2.
Button ancestors—TControl *and its descendants, including* TButton.

What about classes higher than *TControl*? One function of a control object must be to display itself on the screen, so *TControl* might be a descendant of a *TView* class—a class of objects that display themselves in windows. Higher still? All controls, and

probably all views, have to respond to mouse clicks and perhaps other events. Thus, *TView* might be one descendant of a *TEvtHandler* class—the class of objects that have methods for responding to mouse clicks, keypresses, and menu selections. Above that? *TObject*—the class of objects that know how to "free" and "clone" objects. Any *TObject* descendant can dispose of itself or make a copy of itself—a useful ability for buttons or any other objects.

Figure 7-3 shows this partial hierarchy. It corresponds to some of the most important components of typical Macintosh programs; we'll make extensive use of some of these classes later in the book.

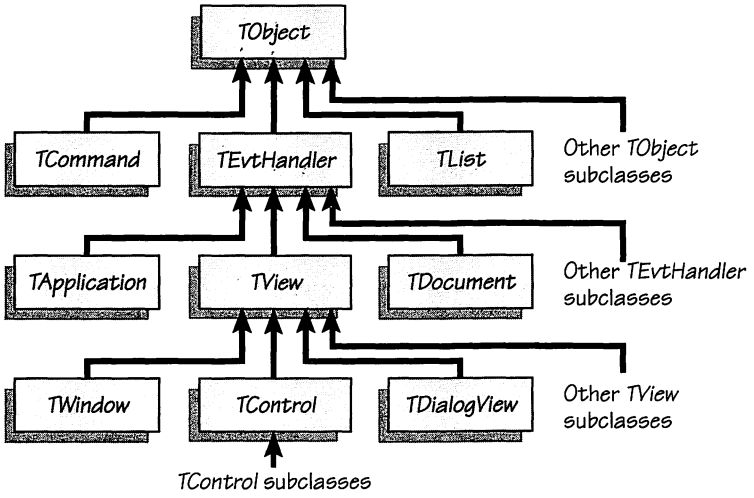


Figure 7-3.
More ancestors—from TControl *up.*

As we go down the hierarchy, the classes grow more specific. And each lower class inherits more and more from its many ancestors: the ability to free and clone itself, the ability to respond to events, the ability to display itself, the ability to scroll data when clicked with the mouse. Each new class might add only a little, but, by a process of accretion, inherited powers are built up. As a result, a particular button class *is-a* control, *is-a* view, *is-an* event handler, and *is-a TObject.*

Instead of thinking only of a button class, we've looked at buttons in their hierarchical context and seen how to make them more powerful in and of themselves and more interrelated with other objects. This is the most fruitful approach to class design. When you design a linked list class, for instance, try to design a general one—even if its functionality is incomplete—as you create the specific one you need. The extra work will pay off in reusability later.

Design for Reusability

Many object classes are one of a kind, custom-built to solve a particular programming problem. But many more are potential "reusable software components," as Brad Cox calls them in *Object Oriented Programming: An Evolutionary Approach* (Cox 1986a). When possible, design with reusability in mind, as we'll do when we create a list component in Part 3.

A good reusable class (and hierarchy, because we reuse hierarchies, not only classes) is

- Independent of other classes (except those in its own hierarchy) and requires no knowledge of other classes.
- Complete—with methods to meet all foreseeable needs.
- Separate—in a unit, either by itself or with a few other closely related classes. (Remember, you can't reuse a class you've declared in the declaration part of a program.)
- Useful—provides some commonly needed facility.
- Easy and clear—is designed so that it invites reuse. (This implies good, but brief, documentation.)
- Clean—represents a single abstraction, is not a catchall for unrelated real-world objects.
- Extensible—is easy to subclass to add new capabilities or alter unsuitable capabilities.

Encapsulate

When you put an instance variable into a class, include methods to set and get its value unless the instance variable is essentially a private field, to be manipulated by the class's methods only and not by what Cox calls "consumers" of encapsulated objects.

If you provide a clean enough and complete enough interface to your class, consumers will want to abide by the principle of encapsulation.

Use Polymorphism When It Makes Sense To

Watch for situations in which you need to manipulate numerous similar-but-different kinds of objects: data-intensive applications such as auto parts inventory, data structures such as arrays or lists that can contain different types, applications that perform interobject communication. As you design, keep the idea of using polymorphism for these situations uppermost.

Look to the Data

The biggest single source of objects in an OOP program is likely to be the data the program processes: auto parts, employee records, chunks of text, numbers, graphical objects, data in spreadsheet cells, and the like.

Model the real-world objects represented by your data as software objects. But remember that many objects come from a world of abstract ideas rather than the physical world. Later in the book, we'll look at some interesting object classes that have nothing to do with the mundane world.

Delegate

Rather than having one object that does everything, think in terms of groups or communities of objects that divide up the tasks and that work by communicating with each other. A complicated object might create and use several other objects to implement parts of its functionality. A complex graphic, for example, might be drawn component by component by a number of objects, each of which knows how to draw one simple part of the whole graphic. Such a complex object is called a composite object.

Working with Objects

In this section, let's summarize what we have to do to create and use objects. We've covered most of the advice in this section in earlier chapters, but here we pull it all together in one place.

Declaring and Creating Objects

Because objects are dynamic, we declare them as variables of specific object types and then create them (allocate memory for them from the heap) with *New*:

```
var
  aSphere: TSphere;          { Declare an instance }
  :
New(aSphere);                { Create it }
```

Of course, class *TSphere* must already have been declared.

Initializing Objects

Once an object instance has been declared and memory for it has been allocated from the heap, we need to initialize its instance variables (and perhaps call some of its methods to accomplish other parts of the initialization):

```
aSphere.ISphere(3, 50, 50);
```

After we create the object, the first message we send is to its initialization method, but until *New* has been called to create the object, we can't send it any messages.

Using Objects

After initialization, we use an object by sending it messages (calling its methods). Technically, we can also directly access the object's instance variables to assign values to them or read their values, but such direct access is not considered good OOP practice. Instead, we should send messages—if the methods are provided—to set and get instance variables. We send messages by prefixing the method's name with the object's name and supplying any required parameters:

```
aSphere.ISphere(4, 100, 150);
aSphere.Display;
⋮
result := anObject.FunctionMethod;
```

Assigning Objects

For starters, of course, we can assign an object of type *X* to any class *X* variable:

```
var
  anXObject, anotherXObject: X;
⋮
anXObject := anotherXObject;
```

But we can also assign an object of a descendant type to any variable of an ancestor class. This is part of what makes polymorphism possible. Suppose we have these declarations:

```
type
  X = object
      procedure M;
    end;  { Class X }

  W = object(X)
      procedure M;
      override;
    end;  { Class W }

var
  aWObject, anotherWObject: W;
  anXObject: X;
```

In this example, class *W* is a descendant of class *X*. Then, polymorphically, we can make the assignment

```
anXObject := aWObject;      { Assign descendant object to ancestor variable }
```

When the assignment is polymorphic, it can also affect message sending. Suppose that, after the assignment we've just made, we send this message:

```
anXObject.M;      { Call method M }
```

The method actually invoked will be class *W*'s method *M*. At runtime, *anXObject*, despite its type, is actually a reference to an object of class *W*, so the correct method (*W*'s) gets called by means of the runtime binding mechanism. This works only because both the *X* and the *W* classes have an *M* method, even though their respective *M* methods undoubtedly do different things.

If class *X* didn't have a method *M*, we'd get a compiler error: The compiler wouldn't be able to tell that the object actually referred to at runtime (*W*) does have an *M* method. All the compiler would know is that the variable *anXObject* is of type *X* and that *X* has no *M* method. The fact that both classes do have a method named *M* satisfies the compiler—and the correct method gets executed at runtime.

The alternative—if *X* didn't have an *M* method—would be to promote the object referred to by *anXObject* by means of typecasting. Assuming that we can be sure that *anXObject* really does refer to an object of class *W*, we can typecast *anXObject* to class *W*:

```
anotherWObject := W(anXObject);          { "Promotion" by typecasting }
```

We can use the *Member* function to check the object's class before typecasting:

```
if Member(anXObject, W) then
  anotherWObject := W(anXObject);
```

Errors with Objects

Attempting to access instance variables of or to call methods of an object that hasn't been created with the *New* procedure is equivalent to dereferencing an undefined pointer.

If an object reference variable is undefined, we don't know where its underlying handle might point. Trying to access an instance variable or call a method will result in a runtime error. (This is the case before we call *New* to allocate memory for the object and after we send it a Free message to dispose of it.)

Also, if an object reference variable has the value *nil*, trying to access its instance variables or call its methods results in a *nil* dereference error, just as with a pointer.

And if we send a message to an object that invokes a method that the object doesn't have, the result is a compiler error. (See the discussion earlier in this chapter on assigning objects polymorphically.)

If we create two objects, *anA* and *aB*, with *New* and then assign *aB* to *anA*,

```
New(anA);
New(aB);
:
anA := aB;          { anA's original object is lost }
```

the memory originally referred to by *anA* is lost in the heap, with no reference to it.

Object references are implicitly dereferenced for us by the compiler, so we need to be cautious about using them in situations in which the heap might be compacted. For instance, as Kurt Schmucker points out in *Object-Oriented Programming for the Macintosh* (Schmucker 1986c), if you use an object in a *with* statement like this one:

```
with anA.fAnInstanceVar do
  begin
    :
    SomeProc;    { Causes heap scrambling }
    :
  end;
```

the address the compiler calculates for *fAnInstanceVar* can become invalid if anything in the *begin...end* block, such as *SomeProc*, causes the heap to be compacted and thus causes object *anA* to move.

Something similar can happen if we pass an instance variable of an object as a *var* parameter to a procedure or function:

```
SomeProc(anA.fAnInstanceVar);
```

We'd be safer in these cases to use a local auxiliary variable:

```
aux := anA.fAnInstanceVar;
with aux do
  begin
    :
  end
SomeProc(aux);
```

Of course, we can lock object references (which are really handles), including *self*, with calls to *HLock* and *HUnlock*. When we do, we must typecast the object reference to a *Handle*:

```
HLock(Handle(self));
```

Disposing of Objects

Disposing of a dynamically allocated object when we finish with it depends on how we declared the object's class. If we made the class a descendant, even indirectly, of class *TObject*, we can call the *Free* method the class inherits from *TObject*. This frees up the storage for the object in the heap and leaves its reference variable in an undefined state:

```
aSphere.Free;    { Dispose of the object aSphere }
```

It's common practice to use *TObject* as the top of an object class hierarchy so that all of the objects inherit *TObject*'s *Free* method (and its other methods—see Chapter 8).

If the object doesn't inherit a *Free* method, the alternative is to do to the object what *Free* does. *Free* actually calls the method *ShallowFree*, which calls *DisposHandle* to do the dirty work:

```
DisposHandle(Handle(self));    { Note spelling of DisposHandle }
```

The object reference to *self* is implemented as a handle, so we can typecast *self* to the generic type *Handle* and then call the Toolbox routine *DisposHandle* to free the memory allocated for *self*.

> **NOTE:** *Once we free an object, any reference to it (there could be more than one) is undefined. Trying to access an instance variable or call a method through an undefined object reference results in a runtime error. To be safe, it's a good idea to set a freed reference to* nil *and check its status before trying to use it again.*
>
> *A corollary: Before freeing an object, we should consider whether multiple object reference variables might point to it. We can free the object through any of the object reference variables that point to the object. But we need to be careful afterwards about trying to access an object through reference* A *that we earlier freed through reference* B. *We'll go into this a little more in Part 3.*

Using Units

Although we can declare object classes in the outermost scope of a main program, it makes more sense to declare them in Pascal units so that they can be used again. Every Object Pascal implementation for the Macintosh uses the unit construct. In a unit, the object class is normally declared in the unit's interface part. The corresponding method bodies are declared in the unit's implementation part. Any program (or any other unit) can then use the class's unit to gain access to the class.

Private Objects

We can also declare and use private objects. If we declare an object class in the implementation part of a unit instead of in the interface part, the object can be used in the implementation part of the unit but not by consumer units or programs that declare the unit in their *uses* clauses. This parallels our ability to declare private constants, types, variables, procedures, and functions in the implementation part of a unit. We might declare a utility object class in a unit's implementation part for our private use rather than making it publicly available to the whole world.

Unit Dependencies

Object class dependencies (inheritance) can result in unit dependencies that affect compilation order. For instance, if object class *C* (in unit *UC*) inherits from object class *B* (in unit *UB*), unit *UC* needs to list unit *UB* in its *uses* clause. Also, in general,

if unit *UC* uses unit *UB* and unit *UB* uses something defined in unit *UA*, not only must *UB* list *UA* in its *uses* clause, but *UC* must list both *UB* and *UA*. The exception occurs if *UB* uses the item defined in *UA* only in its implementation part (privately), not in its interface part. Then *UC* won't have to list *UA*, just *UB*. In a *uses* clause, the units are listed in order of dependency. In our example, that order follows the hierarchy down: *UA*, *UB*, *UC*.

Figure 7-4, for example, shows a class *TBouncingBall* as a descendant of class *TSphere*, which in turn is a descendant of class *TObject*.

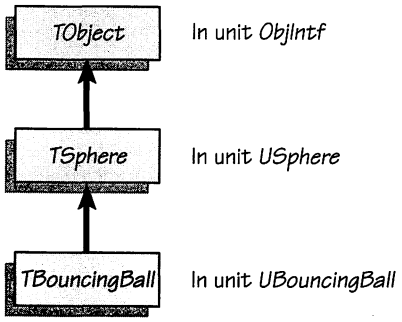


Figure 7-4.
Bouncing ball hierarchy—with ball descended from sphere.

We would use this *uses* clause in unit *UBouncingBall*:

```
uses ObjIntf, USphere;          { In UBouncingBall }
```

In the main program, which uses the unit *UBouncingBall*, the *uses* clause would look like this:

```
uses
  ObjIntf, USphere, UBouncingBall;          { Main program }
```

Putting different object classes in different units is a good idea from the standpoint of modularization. But the practice can introduce some complications when we try to make objects of classes in those different units communicate mutually. We can avoid these complications by using abstract superclasses, as we'll do in Chapter 8. Alternatively, we can declare and implement more than one object class in a single unit. Grady Booch suggests in *Object-Oriented Design: With Applications* (Booch 1991) that the optimum approach lies somewhere between putting each class in its own unit and putting all classes in one unit.

What we're really talking about here is physical modularization—how we divide up our classes into separate units. The idea Booch is advancing is that we should arrange class declarations so that dependencies that force recompilation of lots of code when we make a small change are minimized.

Spreading a Large Class Over Multiple Units

A really large class might have dozens, or even hundreds, of methods. For example, the *TPicoApp* class (a Macintosh application class) that we develop in Part 2 has at least 40 methods.

When we compile such a class, the files can become too large to be used effectively in small amounts of memory. Suppose, for instance, that we face the same limitations I was confronted with as I prepared the code for this book. Using THINK Pascal in 1 megabyte of RAM, and especially using THINK's debugging tools on large projects with numerous files, I sometimes ran out of memory and couldn't open a large source file to let the compiler point out where an error had occurred.

There are ways to break up the code so that file sizes are reduced. In MPW Pascal, we can use the *Include* directive (*$I*) to put the interface part of a definition in one file and its implementation in another:

```
unit Something;          { This is an MPW example }
interface
  uses
  :

  { Declarations of classes and other program elements }

implementation

  {ISomething.Incl.p}    { Causes other file to be included here at compile time }
end.
```

Because THINK Pascal doesn't have an include directive, we have to do something like this:

```
unit Something;
interface
  uses
:

  { Declarations of classes and other program elements }

implementation
  { Bodies of ordinary procedures and functions }
  {  but not of methods }
  { Some or all method bodies are in unit }
  {  Something2 below }
end;

unit Something2;
interface
  uses Something;
  { Empty declaration section }
  { Declarations are in unit Something }
```

```
implementation
  { Method bodies for classes in unit Something }
end.
```

Figure 7-5 illustrates a schema for organizing unit code into interface and implementation files.

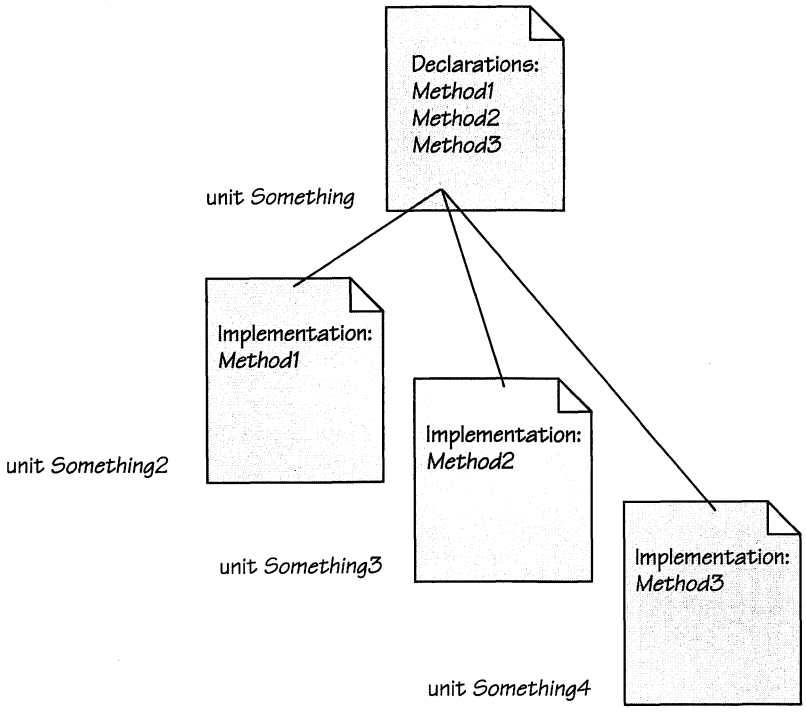


Figure 7-5.
Multiple units—declarations in one unit and bodies in others.

We need to observe one limit in using this technique: We can defer method bodies to another unit this way because the name of the method is always prefixed with the class name as a qualifier. But we can't defer the implementations of ordinary procedures and functions this way because they have to be implemented in the unit in which we declare them.

Compiling Objects

In this section we'll look at how to compile OOP programs and at some of the pitfalls to watch out for when compiling.

How to Compile Your Object Pascal Program

Object Pascal programs are really no different from standard Pascal programs as far as compiling is concerned. We do have to list units and the main program in the correct order to ensure that we take care of interunit dependencies.

A typical Object Pascal program consists of a main program (often quite short) plus one or more units containing object class declarations and other support code. We've looked at several small programs so far, and we'll look at a couple of larger ones in Part 2.

In MPW Pascal and TML Pascal, we compile with a command something like this:

```
Pascal Sample.p
```

or possibly

```
Pascal unit1.p unit2.p Sample.p
```

with the units in the right order and the main program last. *Pascal* is the command-line command to invoke the Pascal compiler. The arguments are file names, and the line might also contain some compiler option switches.

After compiling in MPW or TML, we must link the compiled program with any libraries it uses to create an executable file. To do so, we run the Link program. Here's a sample link:

```
Link -o Sample -t SMPL -c 'SMP ' Sample.p.o δ
"{Libraries}Runtime.o" "{Libraries}Interface.o" δ
"{PLibraries}Paslib.o"
```

This sample shows the command (*Link*), some option switches (*-o*, *-t*, *-c*), libraries (*Runtime.o*, *Interface.o*), and how the linker is told where to look for the libraries (*{DirectoryName}libraryFilename.o*). The symbol δ (Option-D) is a continuation character. See the documentation for MPW, MPW Pascal, and TML Pascal for details. A good reference is Joel West's *Programming with Macintosh Programmer's Workshop* (West 1987), which covers MPW and MPW Pascal—MPW through version 2.0 although MPW's latest release at this writing is 3.0.

All the compilers that run under MPW—including TML Pascal as well as C, Modula-2, and FORTRAN compilers—let you create standalone, double-clickable applications, desk accessories, and code resources such as INIT, defproc, and PACK resources. In addition, MPW's linker lets you mix compiled modules from any language in your program. THINK is more limited in this last regard.

THINK Pascal provides an interactive menu-driven programming environment organized around "projects." A project is a document containing compiled modules, information about where to locate source files, and information used in debugging the program. A project organizes the "make" order of files in a window—you can use the mouse to drag file names around in the window to change file order and to show which files go into which code segments. In THINK Pascal you don't have to master command-line or Unix-like syntax. You can use the Go, Step Over, Step Into, and Step Out menu commands in the Run menu to run a program with THINK's interpreter, using the built-in, source-level debugging tools and letting the interpreter find and mark error locations for you. Once the code checks out, you can use the Set Project Type command in the Project menu to specify the kind of file that the compiler will create ("build"). THINK can then compile native code to create a standalone, double-clickable application, a desk accessory, a device driver, or a code resource. Of course, each of those kinds of files has its own structure and requirements. The project manages linking as an integral part of compiling. One of THINK's additional strengths is the built-in LightsBug source-level debugger, which takes good advantage of the Macintosh user interface.

Compiling Pitfalls

As Pascal types, object classes are subject to the same scoping rules (more or less) that other types are, but an object class won't compile correctly if you try to declare it in a nested scope—so much for declaring a utility object on the fly in a procedure, although you can declare the class outside the procedure and allocate instances of it inside.

We'll also run into compilation or linking problems if we

- Declare a method and forget to implement it
- Implement a method without having declared it in our object class declaration
- Mismatch the name, parameter list, or function return type of a method between its declaration and its implementation

Errors will occur, too, if we

- Forget to qualify method names in the implementation part with the class name (*procedure ClassName.MethodName*)
- Try to use the keyword *inherited* or *self* outside a method body

Incorrectly using polymorphism can lead to compiling problems, too. The most likely error occurs in this situation:

We have a polymorphic variable, *A*, which might have a descendant object, say, *B*, assigned to it at runtime. If we call one of *B*'s methods using *A* but *A* doesn't have a method of that name, the compiler will complain even though at runtime the message might make sense. See the discussion "Assigning Objects" earlier in this chapter.

Debugging Objects

Debugging objects is like debugging other dynamic variables, with the added complication of polymorphism. Fortunately, all the Object Pascal programming environments on the Macintosh have good source-level debuggers. MPW Pascal and TML Pascal can use a debugger such as Apple's SADE debugger. THINK Pascal has its own built-in LightsBug debugger. All of these environments also support low-level debuggers like MacsBug, Icom Simulations's TMON, or Jasik's The Debugger.

To determine what's happening to an object variable at runtime, you need to trace the object reference to the object's block of memory in the heap. Object references are implemented using Macintosh handles, so when debugging you have to make a double dereference (^^) to get to the block—the only time you dereference object references yourself. You can trace the double dereference using tools and techniques supplied by the debugger.

And, because of polymorphism, some of your object variables might refer to actual objects of descendant classes. The blocks of memory for objects of these classes might be different lengths, depending mainly—for most implementations of Object Pascal—on how many instance variables an object has and what their sizes are. THINK's LightsBug, for instance, requires that you typecast polymorphic variables to see their actual contents. Fortunately, as Figure 7-6 shows, LightsBug makes such typecasting easy.

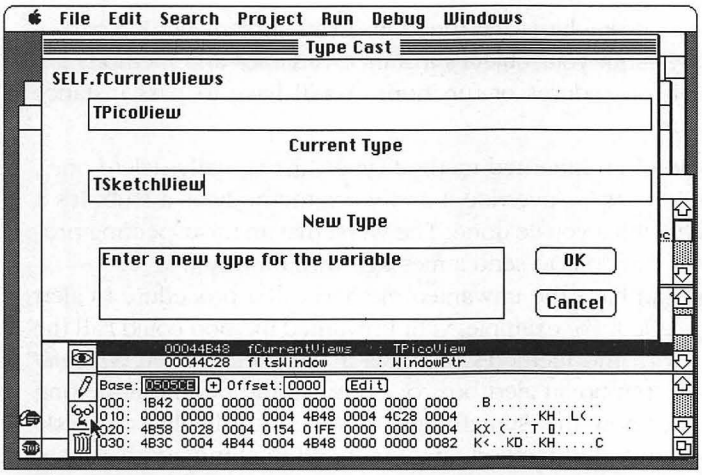


Figure 7-6.
Typecasting a polymorphic variable—here, in THINK Pascal's LightsBug window.

For most OOP debugging, the facilities provided by the programming environment and the general information in Chapter 9 (see "On the Heap") should be enough.

Writing Methods

We've already looked at how methods are declared. Now we'll consider a few tips on writing methods.

You can use methods for anything you can use procedures or functions for—executing I/O operations, displaying graphics, manipulating the Mac via the Toolbox, performing computations, setting and getting the values of instance variables, and just about anything else you can think of. In this book we'll write object methods that run a Macintosh event loop, set up menus, detect whether a button has been clicked, compute the circumference of a sphere, simulate a roll of the dice by generating a pseudorandom number, draw the contents of a window, operate a Macintosh dialog box, insert an item in a linked list, write data to a file, and more.

Here are some tips on writing methods:

- Feel free to declare and use object variables in your methods as local variables. Just don't try to declare any object classes inside a method.
- During development, implement and debug a method or two at a time and either omit some method declarations from the class declaration until later or write the implementations of methods you aren't ready for yet as stubs (empty bodies).
- You can use an object's instance variables as global variables for your methods. The instance variables are visible from inside any of the object's methods.
- You can write ordinary procedures and functions in an implementation part—to use as utilities, say—but your object's instance variables and methods are unknown inside such procedures or functions. You'll have to pass instance variables as parameters.
- If you'd like to get rid of an inherited method (you can't actually delete one), you can at least "stub it out"—override it and write the body as a stub. It's a good idea to document what you've done. The worst that an unsuspecting program using your class can do is to send a message with no result.

  Alternatively, you can have the unwanted method call a procedure to alert you if the method is called. For example, your unwanted method could call the procedure *Abstract* with the method's name as a parameter. Then *Abstract* could print a message, put up an alert box, or even halt the program—letting you, the programmer, know that something is wrong. Most likely, your code mistakenly calls the unwanted method, and you need to eliminate that inappropriate call.
- You can call a method recursively—like any Pascal procedure or function.
- Inside the method code of an object, avoid unnecessarily calling the object's other methods. Where possible, directly access instance variables instead of using methods. This approach improves the performance of your code considerably.

Special Methods—Private, Basic, and Abstract

A private method is not intended to be used by programmers using its class. A basic method is not intended to be overridden by programmers subclassing its class. An abstract method is one that performs a task only if a subclass overrides it.

Private methods

The designer of a class might need a few methods for his or her own use—to support the rest of the class design. Such private methods are not intended for public use by programmers using the class. Unfortunately, in order to be methods of the class, they must be listed alongside public methods in the class declaration. This makes them visible to and usable by outsiders, and Pascal gives us no way to prevent their use by outsiders. All we can do is comment and rely on the programmers using the class to behave themselves. This isn't very safe programming practice, but it's all that Pascal provides for. *TButton.Clicked* is an example of a private method.

The only real alternative to private methods is to use ordinary procedures and functions, which you can declare privately in the implementation part of a unit without listing them in the class declaration. Be sure you can distinguish between private methods and private procedures. Ordinary procedures and functions don't have access to the class's instance variables. You have to pass the instance variables to them as parameters, which can make this approach somewhat cumbersome. In this book, we'll take both the private method and the ordinary procedure approaches, and we'll sometimes use an alternative to passing instance variables as parameters to ordinary procedures. (See the discussion of passing a reference to the object whose method is calling the utility under "Utility Procedures and Functions Associated with Objects" later in this chapter.)

Basic methods

Sometimes the class designer includes a method that should never (or hardly ever) be overridden. Such methods are called basic methods. For example, of the four *TObject* class methods—*Clone*, *ShallowClone*, *Free*, and *ShallowFree*—you should never override the basic methods *ShallowClone* and *ShallowFree*. The methods *Clone* and *ShallowClone* perform exactly the same task—they copy an object. A subclass writer can safely override *Clone* to work in special ways, assured that the fundamental cloning ability of *ShallowClone* has not been lost in the process. By never being overridden, *ShallowClone* is always available anywhere in the hierarchy. *Free* and *ShallowFree* work in the same way.

Abstract methods

Many abstract classes have abstract methods that we are required to override in order to flesh them out in subclasses. Often, there isn't enough information at the level of the abstract class to define the method in detail but subclass writers need to know that such a method is needed. Such must-override methods are often written as stubs—with empty bodies. The class designer thus divides up responsibility: He or she furnishes only so much of the class's functionality; subclass writers will have to furnish the details at their own levels.

Utility Procedures and Functions Associated with Objects

When we developed the example class in Chapter 6, we included utility functions called something like *New*[ClassName] (*NewValveStemCap*, for example). Such a utility encapsulates creating an instance of the class and calling the class's initialization method for the new instance. The function thus returns a fully initialized instance of the class. This saves an outside programmer from making several calls just to create an object.

Because a *New*[ClassName] function actually creates an instance, it can't be a method. Instead, you declare it in the unit interface part outside the class declaration it's associated with. When you give the body of such a utility function in the unit's implementation part,

- Don't qualify the function name with the class name because the function is not a method.
- Don't use the code inside the function body to access instance variables or methods of the class directly—it can't.
- You can't defer the body of the function to a different unit the way you can split up a large class across multiple units.

These facts are true of any ordinary procedure or function (non-method) in a unit. Sometimes it's handy to write utility procedures or functions in the unit implementation part to be called by the methods. But because nonmethod procedures and functions can't access the class's instance variables, you have to pass the fields as parameters. You can pass all needed fields as individual parameters, or you can pass a reference to the object (*self*) whose method is calling the utility. In the second case, the utility procedure can send messages and access instance variables through the reference passed to it. Here's a short example:

Utility procedure *Q* is called by method *X* of class *C*. Procedure *Q* needs to access instance variables of the class *C* object whose method calls it. The procedure heading of *Q* looks like this:

```
procedure Q (whichObject: C);
```

Inside method *X*, we call procedure *Q* this way:

```
Q(self);              { Pass reference to self to Q }
```

and inside procedure *Q*, we can then do things like this:

```
whichObject.fSomeField := 3;     { whichObject is self }
whichObject.SomeMethod;
```

The C++ language lets you declare functions as "friends" of a class. Even though friend functions are outsiders to the class syntactically, they are granted access to the class's inner works. Our utilities aren't equivalent to C++ friend functions, but the mechanism we've just outlined gives our utility procedures the access to the class that they need.

What should be a method, and what shouldn't?

The consideration of utility procedures and objects leads to the question How do you decide which procedures and functions to declare as methods, and which as utilities?

In part, the answer has to do with visibility. Procedures and functions that are to be visible to (and callable by) outside consumers of a class must be declared in the interface part of the unit. Our "*New*[ClassName]" functions are an example.

But many procedures and functions are merely utilities, intended for use only inside the implementation part of a unit. To keep them private, we must not declare them in the interface part of the unit.

If such utilities need direct access to the innards of a class, we might be tempted to make them methods simply to grant them the access they need, as we did *TButton.Clicked*. Keep in mind, though, that making the utilities methods makes them public. And we don't really want outside programmers who use our class calling our private utilities.

To avoid that danger, we might prefer to pass a reference to *self* to the utilities we call from our methods. The result is slightly messier to implement, but it's also safer in terms of the encapsulation of our objects. *TButton.Clicked* would thus become an ordinary function with two parameters instead of one:

```
function Clicked(event: EventRecord; whichButton: TButton): Boolean;
```

As a private utility function, *Clicked* is declared only in the implementation part of the unit. To give it access to the button's innards, we pass it a reference to the calling button.

Methods vs. procedures

We should take advantage of OOP and use methods when it's appropriate, though. Suppose we have a class hierarchy like the food example we've looked at several times already. In Figure 7-7 we show the hierarchy with more descendants of *TFruit* than we've shown before.

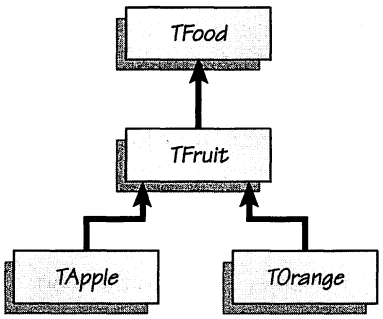


Figure 7-7.
The food hierarchy again—with more descendants.

Each entity in the hierarchy can be "prepared"—in some fashion appropriate to the particular class. Apples might simply be washed, and oranges be peeled and sectioned.

Now let's take a pre-OOP approach to preparing foods. The following ordinary procedure takes an object as its parameter:

```
procedure Prepare(food: TFood);  { Not a method }
begin
  if Member(food, TApple) then
    Wash(food)
  else if Member(food, TOrange) then
    begin
      Peel(food);
      Section(food);
    end
  else if Member(food, TFruit) then
    begin
      { Prepare generic fruit item somehow }
    end
  else if Member(food, TFood) then
    begin
      { Prepare generic food item somehow }
    end;
end;  { procedure Prepare }
```

This approach would work, of course. It's the kind of thing that Pascal programmers do all the time. But it's not OOP, and it's not easily extensible code if we ever want to add another type of food.

What's better? Because this is an object hierarchy, we can count on methods declared high in the hierarchy to be inherited by classes lower down. So, if *TFood* has a method called *Prepare* (instead of the nonmethod procedure we just wrote), *TFruit* inherits that method. So do *TApple*, *TOrange*, and any other subclasses we might choose to add later.

But if *TApple*, say, has its own way of preparing (different from the *Prepare* method it inherits from *TFruit*), it can override *TFruit*'s *Prepare* method to handle preparing in its own way. So can *TOrange* or any other class.

It's immensely more OOP-like to give each class its own *Prepare* method than to create one complicated *Prepare* procedure that is used with many different objects. Any time we are about to pass an object as a parameter, we should stop and ask ourselves whether a method would be more appropriate.

The principle is this: Whenever it makes sense, have the object itself do the work.

Projects

- Implement and test your library hierarchy of shapes. Include methods to draw, erase, and move shapes. For each shape class, add a utility function, *New*[Shape], to create instances of the shape easily.
- If your Mac Toolbox skills are up to it, add methods to drag and resize your shape objects. You might also include a method to select a drawn shape with the mouse and add "handles" to it like those used for resizing and adjusting shapes in most drawing programs. Hint: You might want to read ahead to see how events are handled in PicoApp in Part 2. You could treat a shape as a "view" (a kind of "button") that can tell whether it's been clicked, so look at the *Clicked* method of class *TButton*. And see the *TEvtHandler* class in Chapter 4 and Part 2. If your shapes are descendants of *TEvtHandler*, they can handle mouse clicks.

8

THE FAMILY TREE

In this chapter, we'll explore some practical consequences of the hierarchical structuring of object classes. As we go, we'll (unfortunately) uncover a few weaknesses of OOP—weaknesses, at least, as far as its use in large software projects is concerned.

We'll consider a number of other topics related to using classes in hierarchies:

- Using *inherited*: calling an ancestor's method that we've overridden
- Using abstract superclasses—*TObject*
- Scope and communication among objects
- Outlets: a corollary technique for managing communication
- Inserting superclasses to propagate capabilities
- Composite objects
- Multiple inheritance

The Structure of Object Hierarchies

Object Pascal allows single inheritance only, not the multiple inheritance some languages allow; that is, each class has one and only one immediate ancestor, although it can have many ancestors up the hierarchy.

Because of single inheritance, an Object Pascal hierarchy has a distinct structure:

- The hierarchy can contain any number of classes.
- The topmost (root, or base) class has no ancestors.
- Any class in the hierarchy can have any number of descendants.
- Descendant classes inherit all instance variables and all methods of all of their ancestors (except those methods that are overridden).

We can have several or many hierarchies in a single application or class library, but all hierarchies share the same structure. An object hierarchy has the shape of a directed acyclic graph, or DAG, a tree in which each node has only one parent but can have many children or none. Figure 8-1 on the next page shows such a structure, with multiple children at each node.

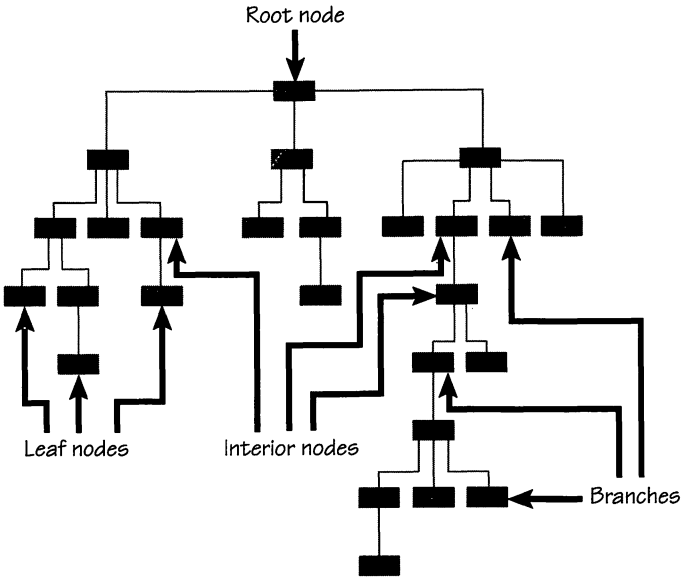


Figure 8-1.
Object hierarchy structure.

If you're familiar with the Macintosh's hierarchical file system (HFS), you have already seen such a structure. HFS is a tree-structured hierarchy of directories and files. An HFS directory can contain many subdirectories but can have only one parent directory. And a file or subdirectory can be in only one directory.

Another HFS concept might help to clarify object hierarchy structure: A path in the HFS descends through the file tree from directory to subdirectory to subdirectory...and eventually to files. Any directory, subdirectory, or file can be reached by one and only one path from the root directory. Two subdirectories or files might or might not be in the same path from the root. Figure 8-2 shows two paths from the same root class. This concept has a bearing on what happens when multiple programmers work on the same object hierarchy at the same time. We'll look at that problem in more detail later.

Let's take a look at classes *A*, *B*, *C*, *D*, and *E*, as shown in Figure 8-3. Class *A* is an ancestor to *B*, *C*, *D*, and *E*; that is, class *A* is in a path from the root to each of these classes. Class *B* is an ancestor of *D*, and class *C* is an ancestor of *E*. Classes *B*, *C*, *D*, and *E* inherit from *A*. *D* inherits from *B*, and *E* inherits from *C*. But class *D* does not inherit from class *C*—*C* is not in a path from the root to *D*. Likewise, class *E* doesn't inherit from *B*.

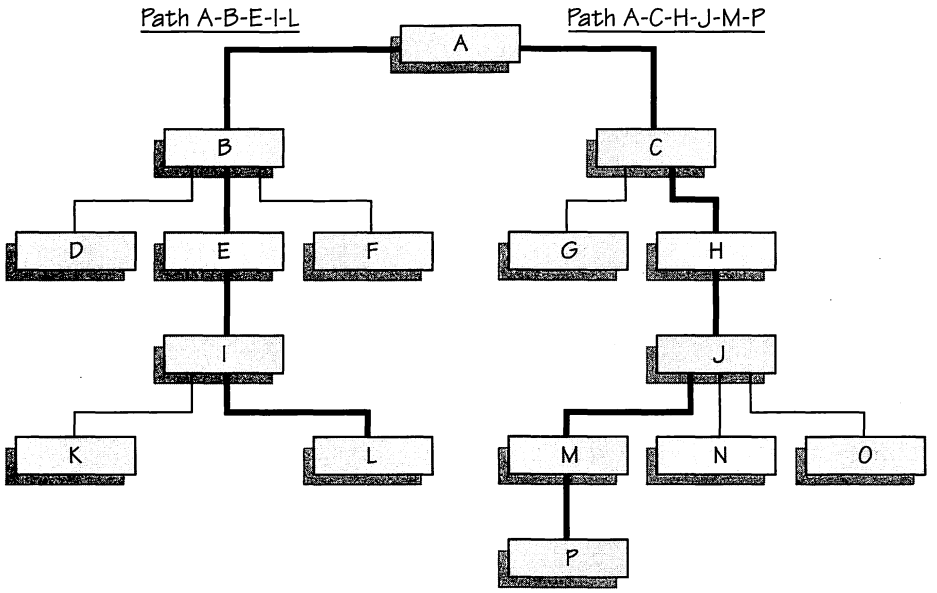


Figure 8-2.
Paths in an object hierarchy.

The whole hierarchy is descended from one or more common ancestors. But as the tree branches more and more, some classes become cousins of others and do not share all of the same inheritance.

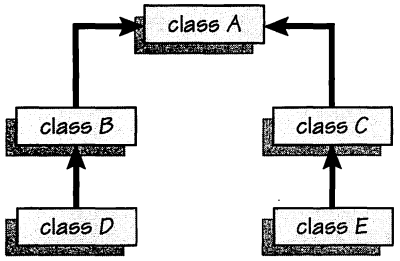


Figure 8-3.
A sample object class hierarchy.

Of course, many hierarchies are even simpler, and many are even linear: One line of descendant classes might come straight down from a root class at the top—the tree wouldn't branch. For an example of a linear hierarchy, see Figure 8-4 on the next page.

Figure 8-4.
A nonbranching hierarchy. Each class has only one subclass.

Extending Object Hierarchies

Within the treelike shape of a class hierarchy, we can make changes:

- We can subclass the lowest class(es)—the tree's "leaves."
- We can subclass "interior" classes, starting new branches.
- We can insert a new class somewhere in the middle of the tree, possibly even above the current root class.
- We can rewrite a class somewhere in the middle of the tree—an "interior node."

Figure 8-5 shows these points of change.

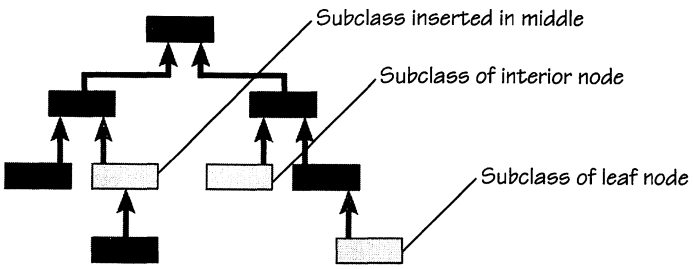


Figure 8-5.
Points of change in a hierarchy—subclassing at a leaf node and at an interior node and inserting a new subclass in the middle of the tree.

Subclassing the Leaves

The most common place to change an existing hierarchy is at the bottom. By subclassing a "leaf" class, we extend the hierarchy downward. The new class names its ancestor but doesn't actually alter anything in the existing hierarchy. Figure 8-6 shows this simplest kind of hierarchy extension.

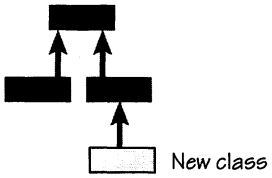


Figure 8-6.
Subclassing a leaf—has no effect on other classes in the hierarchy.

Because this kind of subclassing is unintrusive, we can do it even when the existing hierarchy has already been compiled. We can do it even if we don't have the source code for the hierarchy's classes—provided we have information about the classes' instance variables and methods.

Subclassing an Interior Node

A class doesn't have to be a leaf in order for us to subclass it. As Figure 8-7 shows, we can also subclass a class that is an "interior node" in the hierarchy tree.

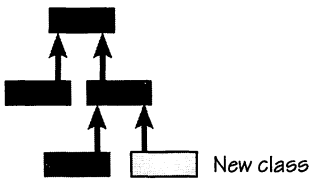


Figure 8-7.
Subclassing an interior node—has no effect on other classes in the hierarchy.

When we subclass an interior node, we create a new branch of the tree. The new class inherits from all classes above it, but it has no real relationship to classes that are already beside or below it. The new class is not in the same path as those classes.

As with subclassing at a leaf, interior subclassing doesn't require that we have source code for the ancestors—provided we have information about the ancestor classes' instance variables and methods.

Inserting a Class into the Hierarchy

We can slip a new class into a hierarchy above some other class or classes. To do that, we have to alter the heritage of any class directly below the insertion point that used to refer to the class above the insertion point. Figure 8-8 on the next page shows the altered relationships. To make these changes, we must have the source code for the classes that need to be changed.

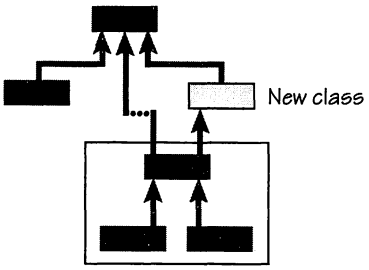


Figure 8-8.
Inserting a class into a hierarchy—changes the heritage of the class below and affects all classes below the inserted class.

Inserting classes is a way to add a new capability to a lot of classes at the same time. Ideally, we'd design our hierarchy with the capability already in place, but we all know that the world—and our planning—doesn't necessarily work that way.

We'll see an example of this kind of alteration later in the book when we invent a class called *TLinkable*. Any descendant of *TLinkable* can be linked up as a node in a linked list. We could also work the other way around—making *TLinkable* a descendant of the class that needs to be linkable, but *TLinkable* can be a subclass of only one other class. If numerous classes in a hierarchy need linkability, we might insert *TLinkable* into the hierarchy above those classes.

Inserting a new class affects all classes below it, of course, so doing that is potentially dangerous if we don't keep those effects in mind.

Which classes does the insertion of the new class affect? In terms of the whole hierarchy, this depends on where the insertion occurs. If the insertion is at the very top of the hierarchy, it will affect every class. But if it's lower, it will affect only those classes below it in the same path—actually, all paths of which it's an element. Classes in paths that don't pass through the changed class aren't affected.

Rewriting an Interior Class

If we have the source code for it, we can alter an existing class in the hierarchy. We might, for instance, add, remove, or rename instance variables; add, remove, or rename methods; change method parameter lists; rewrite method code; and so on. We might change the class's immediate ancestor by altering the heritage spot after the keyword *object*.

Sometimes we might alter a class's heritage to insert another class above it or to move it (and its descendants) to another part of the tree.

Of course, we must have the source code. This kind of change to a hierarchy is probably less common—and more dangerous—than simply subclassing its leaf classes. The principal danger is that we might change something that a subclass depends on. We'll discuss the dangers in more detail later.

Hierarchies in Applications and Libraries

An object-oriented application that uses many object classes might contain them all in one hierarchy. Or the application might contain many smaller, independent hierarchies.

Likewise, a compiled library of classes can consist of one big hierarchy or numerous hierarchies of various sizes. Some of the hierarchies might contain variations of classes that also exist in other hierarchies.

In "The Forest for the Trees" (Vernon, 1989), Vaughn Vernon uses the term "tree" to describe one large hierarchy in which all classes in the application or library descend from a common ancestor. The Smalltalk programming environment and a big part of Apple's MacApp extensible application framework are examples of the tree-structuring of class libraries. The term "forest" describes many separate trees—many hierarchies of various sizes. If we were to ignore our practice of making every new class a descendant (directly or indirectly) of *TObject*, the classes in this book would constitute a small forest.

Libraries

A library is a collection of object classes (much like a traditional subroutine library) we can use in our object-oriented applications. The nice thing about an OOP class library, of course, is that we can extend it by subclassing its existing classes. If a library class isn't quite what we need, we can make a subclass that alters the library class to suit us.

MacApp and the THINK Class Library are two class libraries for the Macintosh. They provide mainly classes that help us create a consistent Macintosh user interface quickly and easily. In the MS-DOS world, Turbo Power has recently released Object Professional, a library of classes for Turbo Pascal programs that is useful for producing a window and menu interface for PC programs. Its general thrust, although not its specific design or implementation, resembles MacApp's and TCL's. Borland's even newer library, Turbo Vision, just released with Turbo Pascal 6.0, resembles MacApp and TCL in its emphasis on the interface, too.

Forest or tree?

Is it better to have a library that's all one big tree? Or one that consists of many smaller trees?

A single-hierarchy library would be more elegant, but practically speaking, there are some good reasons for treating a library as a forest. In Smalltalk, in which the entire programming environment is object-oriented and everything is an object, the class library is considerably treelike. But in Pascal, in which programs are compiled, we're more likely to tailor hierarchies to the application, developing libraries that are more forestlike—lots of small, independent hierarchies and even repetitions of many classes, some optimized for one kind of use, some for another. Some applications

might use data structures such as lists and trees extensively, putting all sorts of data objects into those structures. The data objects, as we'll see in Part 3, would work best if optimized for their roles as nodes in structures. Other applications might not use data structures heavily, so we'd want versions of the same kinds of data objects not weighed down by the baggage needed to make them nodes in structures.

Fitting new classes neatly into an existing hierarchy can be difficult, especially in Object Pascal. Finding just the right niche can be tricky, and any changes to a hierarchy propagate down the hierarchy with side effects that can cause problems in the lower classes.

The most important consideration has to do with the effect that library structure can have on software engineering and practical matters of software development.

Hierarchies and Software Engineering

Logically, the basic unit of modularity in an OOP application (or library) is the hierarchy, not the single class. This has some important implications for software engineers and for shops doing large software projects with OOP. (We'll consider a distinction between logical modularity and physical modularity later in this chapter.)

The class itself is a nice improvement on traditional structured programming techniques—even in Object Pascal, in which encapsulation is very weak. Encapsulating the data with the methods that work on it localizes functionality, hides information, and promotes data abstraction. The ability to subclass makes classes easy to reuse, and software reusability is one of the main benefits claimed for OOP.

All of this might make you think at first glance that the class is the unit of modularity. If that were the case, two programmers—let's call them Aaron and Robin—could work on two different classes at the same time without problems. That's clearly not the case, though, in the frequent situation in which one of the classes descends from the other.

Because a class almost always participates in a hierarchy with other classes, some of which it is derived from and some of which are derived from it, there are lots of interclass dependencies. Let's look at some of them.

Instance Variable Names and Method Names

When class *B* inherits from class *A*, it inherits all of class *A*'s instance variable names and method names. We can't reuse any of those names in a lower class (unless we override a method). If Aaron uses an instance variable named *fRect* in class *A* and Robin then tries to create another instance variable with that name in class *B*, the compiler complains.

Method Functionality

Class *B* inherits all of class *A*'s methods. It might override some of them, but those that it doesn't override it relies on "as is." When Aaron makes changes to a method's semantics in class *A*, the change can have big consequences for class *B*. If Robin can anticipate the change to *A*, she might design *B* quite differently.

Furthermore, a method can send messages to its own object (call other methods of *self*). If class *B*'s method *M* calls method *N*, inherited from class *A*, and Aaron has changed method *A.N*, class *B* objects might not behave as Robin wants *B* objects to.

The *inherited* Mechanism

Similarly, if Robin overrides an inherited method in class *B* that, at some point, calls the inherited version of the method (the one in class *A*) and Aaron later changes that method in class *A*, class *B* might no longer perform as Robin expects it to.

Logical Modularity vs. Physical Modularity

So far, we've been looking at the logical modularity of OOP programs. Because lower classes in a hierarchy inherit from upper classes, changes in an upper class propagate downward to the lower ones. This is why we say the whole hierarchy (or at least a whole path or subtree within a hierarchy) is the logical unit of modularity.

But in Pascal, there's also a physical unit of modularity: the Pascal unit. You can spread classes across units in a variety of ways: one class per unit, multiple classes per unit, even multiple units per class (at least, the implementation part of a class divided up into more than one unit, as we saw in Chapter 7). The unit is a separately compilable piece of code, but it, too, can have dependencies on other physical modules (other units).

We'll see some of the consequences of working with both kinds of modularity.

Consequences for Teamwork

Hierarchies can become quite large, and, although we'd like to think that the upper classes would be rather stable by the time the lower ones got written, all of us know from experience that that's not a realistic expectation. Things can change, even at the upper levels of the hierarchy.

If Aaron, Robin, Bill, and Toni are all working on the same application whose centerpiece is a large hierarchy, some of them on higher-level classes, others on lower ones, they have plenty of potential for conflicts. Four programmers are actually working on the same logical module at once.

Of course, if Aaron, Robin, Bill, and Toni work on classes in different paths in the hierarchy, we can regard the work of one programmer as insulated from that of another. But in many cases, that isn't possible.

We can use physical modularity to our advantage in these situations. Suppose that we put all the classes in a particular part of the hierarchy into one Pascal unit (one physical module). Then we can safely assign one programmer to work on that entire module. Still, we have to be careful. Keep in mind that hierarchies tend to "fan out" as we move down through them, as illustrated by Figure 8-9.

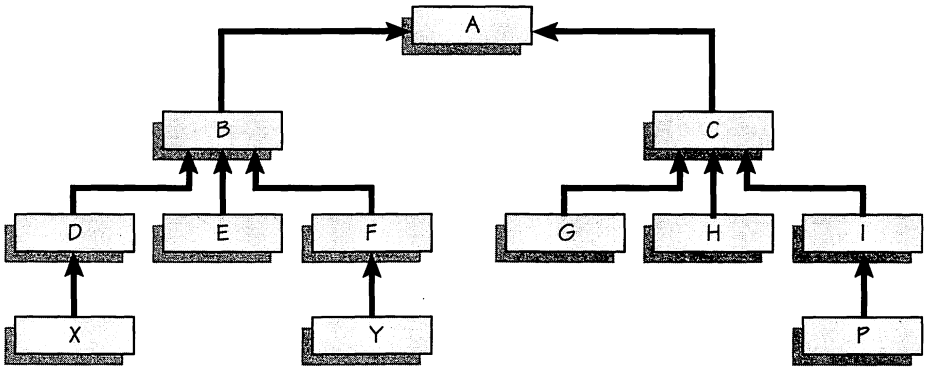


Figure 8-9.
Hierarchy "fan-out."

We can isolate the path *B-D-X* in one unit and assign Aaron to work on it. But if Aaron alters class *B*, he can still affect Robin as she works on classes *F* and *Y* because those classes also descend from *B*.

Even so, this is the direction in which we'd have to go in order to insulate one programmer's work from that of another. One approach might be to assign Aaron to class *B* and all of its descendants, and Robin to class *C* and all of its descendants, with neither of them allowed to work on class *A*. We could still break Aaron's branch of the tree up into units in any way we wanted. He could work on any of those units but not on any of the units containing the classes Robin is responsible for.

Consequences for Software Engineering

The potential difficulties and the need for high levels of communication among programmers working simultaneously in the same path or subtree (or the need for ways to divide up the hierarchy safely) suggest that Object Pascal has some important weaknesses as a development language for large, multiprogrammer software projects. (Of course, standard Pascal has its weaknesses from a software engineering point of view, so this news is not earthshaking.)

Are other OOP languages better in this regard? So far, not very much. Most, such as Smalltalk, are really one-programmer environments. Even languages like Eiffel, which was designed with software engineering principles in mind, suffer from the "hierarchy problem." Bertrand Meyer's otherwise excellent *Object-Oriented Software Construction* (Meyer 1988) doesn't address this issue at all. For a good—and

sobering—discussion of some of these issues, see Scott Guthery's "Are the Emperor's New Clothes Object Oriented?" in *Dr. Dobb's Journal* (Guthery 1989). And see Grady Booch on the same subject in *Object-Oriented Design: With Applications* (Booch 1991).

The moral is that OOP promises a lot in terms of encapsulation, information hiding, extensibility, and reusability, but it's still a fairly immature software engineering technology. Where the project is small enough or focused enough to be done by only one or two programmers, Object Pascal looks very bright. But project managers looking for the best approach for large projects might want to bypass Object Pascal in favor of some other OOP language. If they do use OOP, they will need to devise schemes for controlling code changes so that one programmer doesn't step on another.

Software Components

In *Object Oriented Programming: An Evolutionary Approach* (Cox 1986a), Brad Cox envisioned an industry in which software suppliers built self-contained software components (linked lists, search modules, interfaces, databases, editors, buttons, spelling checkers, and so forth) using OOP techniques. Then software consumers would acquire these components and plug them into the applications they were building. Because of OOP, a component could have a public interface over the hidden internals of the component's implementation and be readily incorporated into a new project. Theoretically, a component would be thoroughly tested and certified for reliability—and in any case it would be self-contained sufficiently to minimize side effects in the rest of an application. The concept of software components parallels developments in the computer hardware industry in which engineers can rely on a great variety of off-the-shelf chips and other components that they can plug into the circuit boards they're designing.

Of course, the software component technology is in its infancy. There are many OOP languages and very few ways to meld the code from one with code from another. Right now, every language needs its own set of components—it will be a while before all will be able to draw on good components from any source, in any language. Distribution of components is another problem, although that could probably be handled, in the long run, in much the same way it is in the hardware component industry—through catalogs, trade shows, and the experience of software engineers trained to use a wide array of standard components.

Calling an Ancestor's Overridden Method

When we override an ancestor's method in a new subclass, the subclass no longer inherits the method. Instead, the subclass supplies a new version of the method that, presumably, works differently.

Suppose, however, that we find a need to call a method we've overridden. Can it be done? Yes. Object Pascal introduces the keyword *inherited*, which lets us "override an override" to call the method as it would have been inherited without the override. Figure 8-10 shows what gets called when a method uses the keyword *inherited*.

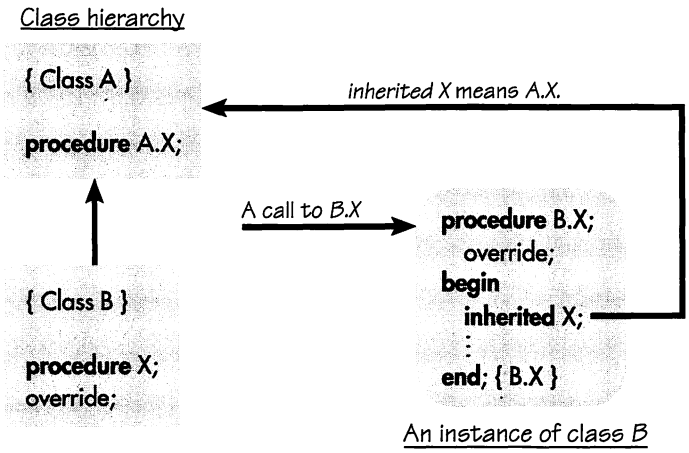


Figure 8-10.
The inherited *mechanism. A call to procedure* B.X *results in a call to* A.X *when* inherited X *is invoked inside* B.X.

Why would we want to do this? Well, for example, the ancestor's method might do some drawing to which we only want to add. We'd like to retain the drawing code of the ancestor's version of the method and supplement what it does in our overriding method. Or we might call the ancestor's initialization method to supplement our own. There are lots of uses for the *inherited* mechanism. To illustrate, let's consider these three *inherited* scenarios:

1. We want to do what the overridden method did, and then do more. Suppose we have a ball object to be used in an animated simulation. We want to first print information about the ball's circumference, area, and volume, which we do in the ancestor class's *Display* method. Then we want to display the information graphically as well. The overriding *Display* method in our ball subclass would be structured like this:

```
inherited Display;      { Print information about the ball }
{ Then give the drawing code }
```

2. We want to do some work in the overriding method and then do what the overridden method did (the reverse of example 1). This time we want to draw the ball and then print the volume information:

```
{ Drawing code }
inherited Display;
```

3. We want to do some preliminary work in the overriding method, then do what the overridden method did, and then do more. Maybe we'd like to draw the ball, print its circumference and so forth, and then print its coefficients of bounce and velocity. Here's that structure:

```
{Drawing code }
inherited Display;
{ Extra printing code }
```

Limits on the Use of *inherited*

The *inherited* keyword can be used only inside the method bodies of the class whose ancestor's method we want to invoke. We can't call the ancestor's method from our main program, say. The keyword *inherited* is meaningful only in the context of an object hierarchy.

We can, however, use *inherited* inside the body of another method in the class besides the overridden method. Suppose we have two classes, *A* and *B*. *B* inherits from *A* but overrides *A*'s method *X*. We've shown uses for *inherited* inside the body of *B.X*. But we could also call *inherited X* from inside, say, *B.Y*, another method of *B*. Methods call each other all the time; for example, in *TButton.Clicked*, we call the *Hilite* method of whatever button is actually being used.

So *inherited* can be put to work inside an overriding method to augment what the overriding method does, as we saw in examples 1–3, or elsewhere (although not outside the methods of the class) to substitute the overridden method for the overriding one in special circumstances.

Calling Overridden Methods Further Up the Hierarchy

Keep in mind that we can use *inherited* to call a method overridden from an immediate ancestor class only. But suppose we have a declaration like this:

```
type
  A = object
      procedure Init;
    end;

  B = object(A)
      procedure Init;
      override;
    end;

  C = object(B)
      procedure Init;
      override;
    end;
```

Suppose also that from inside *C.Init* we'd like to call, not *B.Init*, which *C* has overridden, but *A.Init*, which *B* overrode. *A.Init* is simply too far up the hierarchy for *inherited* to be useful.

This is where we can make a case for the MacApp convention of naming *A*'s initialization method *IA*, *B*'s method *IB*, and *C*'s method *IC* (and *TSphere*'s method *ISphere*)—simply prefixing the class name with *I* for "initialization." If we did that, we wouldn't need the override, and we'd inherit *A.IA* and *B.IB* all the way down the line. We could call any of them we wanted.

This is one area where the object-oriented version of Turbo Pascal (for MS-DOS) beats out Object Pascals on the Mac. Turbo Pascal, like C++, lets you call all the way back up the hierarchy simply by prefixing the method name with the name of the class whose version of it you want to call:

```
A.Init;      { Called from a method of type C in Turbo Pascal for PCs }
             { 'A' is a class name, not an instance name }
```

Abstract Superclasses and *TObject*

An abstract class is one that is not meant to have objects instantiated from it. Its purpose, besides providing as much functionality as possible to its subclasses (serving as a solid base from which they can inherit), is to

- "Abstract out" the common essence of a group of related classes and put that commonality into a single class from which all of them can inherit
- Provide a common ancestor for several subclasses—for polymorphism, to help resolve scoping problems, and so on

Often, abstract classes are very abstract indeed—an abstract class might be able to define only some parts of the necessary instance variables and methods, leaving it to its descendants to fill in the details. For example, some of an abstract class's methods might have to be implemented as stubs because there isn't enough information to flesh them out. Writers of subclasses are expected to override such an abstract method. This amounts to an agreement between the class designer and his or her consumers—the designer can do only so much of the job; the consumers who subclass must handle the details.

We should document such "must-override" methods clearly for future subclassers. It's also a good idea to offer the subclassers guidelines and constraints on how to do the overriding.

Abstract Superclasses

Sometimes, several classes need a common ancestor to tie them together. We've already seen the use of such an umbrella class to provide an array that can hold several types at once (a polymorphic array). If we have classes *TApple*, *TOrange*, and

TPersimmon and want to store them all in the same array, we might invent a class *TFruit*. Each class would list *TFruit* as its ancestor. Then we could declare an array

```
var
  fruitBasket: array [1..12] of TFruit;
```

and put all the descendant fruit objects we wanted to into one basket.

> **NOTE:** *Remember that at runtime there's no way to know which particular type of fruit object is in an array element. To ensure that we can call the methods of a contained object, we should give* TFruit *the same list of methods as its descendants and implement each method as a stub. Then each descendant should override the methods. See the discussion "Scope and Communication Among Objects," later in this chapter.*

Abstract superclasses and scope

We can also use abstract superclasses to resolve problems of Pascal scope when objects of two classes must send each other messages. We'll look at how to use abstract superclasses this way later, under "Scope and Communication Among Objects" and under "Outlets: A Corollary Technique for Handling Scope Problems."

The *TObject* Class

The most commonly used abstract superclass is called *TObject*. All the Object Pascals on the Macintosh provide *TObject* in one form or another. (The THINK Class Library uses its own version of *TObject*, called *CObject*.) *TObject*'s declaration in THINK Pascal is in the interface unit *ObjIntf*. It's provided in a unit rather than built into the compiler.

Class *TObject* represents the notion of "objectness." Recalling the idea of an *is-a* link among classes (Chapter 2), we can say that any descendant of *TObject is-an* object (or a *TObject*). That's the descendant's most generic description.

TObject carries no instance variables and exists only to bequeath its four methods—*Clone*, *ShallowClone*, *Free*, and *ShallowFree*—to any descendant.

- *Clone* uses the *HandToHand* Toolbox call to make a duplicate copy of the object under a new object reference (handle). *Clone* can be overridden.
- *ShallowClone* does exactly the same thing as *Clone*, but whereas *Clone* is intended to be overridable, *ShallowClone* should never be overridden. That way, if we do override *Clone* somewhere down the hierarchy, we still inherit *ShallowClone* for bare-bones cloning. (*ShallowClone* is a basic method—we shouldn't override it.)
- *Free* uses the *DisposHandle* Toolbox call to dispose of the heap storage of the object calling it. *Free* can be overridden.
- *ShallowFree* bears the same relationship to *Free* that *ShallowClone* bears to *Clone*. *ShallowFree* should never be overridden.

Using *TObject* is completely optional—an object class does not have to declare an immediate ancestor in the heritage spot. But putting *TObject* at or near the top of our class hierarchies is good practice because we do need to manage the freeing of objects and might occasionally need to clone them. Using *TObject* is an easy way to get that capability. And it costs very little because *TObject* bequeaths no instance variables to expand the size of our objects.

Overriding *Free*

If an object somewhere down the hierarchy from *TObject* allocates its own memory from the heap (for its own data structures), or if it creates subordinate objects, it needs to dispose of that storage when it finishes with it. In particular, when we free an object that has subordinate data structures, it needs a chance to clean up before freeing itself. The simplest way to give it this opportunity is to override *TObject.Free* and add to what the overridden version does. The new version of *Free* can first free up the local storage for data structures using Pascal's *Dispose* statement, the Toolbox's *DisposHandle* call, and so forth. It can also send any subordinate objects a Free message to free them before freeing itself. Then it can call *TObject.Free* by using the keyword *inherited*—if *Free* didn't get overridden somewhere higher in the hierarchy—or call *ShallowFree* if necessary.

Here's a sample override of *Free*:

```
procedure TMyClass.Free;
  override;
begin
  DisposHandle(Handle(myStack));
  Dispose(myPtr);
  fASubordinateObj.Free;
  inherited Free;        { Call TObject's Free—don't call self.Free }
end;  { TMyClass.Free }
```

NOTE: *If you call* Self.Free *here, you'll be calling the same method again. Call* inherited Free *instead.*

Augmenting *TObject*

Can we add things to *TObject*, given that it "comes with the territory," so to speak? Yes, because *TObject* is simply a class, declared the way any other class in a Pascal unit is. We don't want to add instance variables and methods to *TObject* lightly—it's the ultimate ancestor of almost every other class we write. In particular, we need to be cautious about adding instance variables; every single instance of every *TObject* subclass will inherit those fields, adding considerably to the overhead of the application. Methods, however, incur little extra overhead—just the method code, which is stored only once in an application. That's pretty acceptable.

One useful addition to class *TObject* is a method called *TypeOf*, which returns the class name of an object as a string.

TypeOf can be useful in several situations to ask an object what type it is, but Object Pascal on the Mac doesn't provide a standard way to do that. (Turbo Pascal for MS-DOS does provide a built-in *TypeOf* function.) We'll develop our own makeshift *TypeOf* function and go into some of its uses.

Recall from Chapter 6 that we gave *TMiscPart* objects the ability to report their types, both as an integer code and as a string such as 'Valve Stem Cap'; this time, we'll make a similar ability more widespread by adding a method for the ability to class *TObject*. This is the first time we've modified *TObject*.

In class *TObject*, we add the method

```
function TObject.TypeOf: Str30;
begin
  TypeOf := 'TObject';
end;  { TObject.TypeOf }
```

and we also declare, in unit *ObjIntf*, where *TObject* is declared:

```
type
  Str30 = string[30];
```

Now every descendant of *TObject* inherits the ability to report its type. Of course, unless they override *TypeOf*, *TObject*'s descendants can report themselves to be of type *TObject* only, which isn't very useful. Each instantiable object class needs to override *TypeOf* to report its own type as a string.

How efficient is this technique? Not bad. We didn't add any fields to *TObject*, so we didn't increase the size of its descendants, either. By adding a method we did add a few bytes of compiled code to the overhead of our program, but not much. And what we've gained makes it worthwhile. Objects that don't need to report their types don't have to override *TypeOf*. But *TypeOf* will be very useful in approaches to some special problems.

In Part 3, we'll be dealing with the problem of searching linked lists that, because of polymorphism, can contain many different object types simultaneously. In our list searches—for example, in lists of mail—we can answer this question: Are you of type *TLetter*? We simply use the *Member* function:

```
if Member(you, TLetter) then ...
```

But we can't yet answer this question for some arbitrary object in a list: What type are you? With *TypeOf*, we can. The code in our "search key object" contains a line something like

```
condition := (target.TypeOf = fTheType);
```

where *fTheType* contains a string of type *Str30*. Now we can pass any arbitrary type-name string as a variable to a more general search key object. Provided that *target* is an object whose class overrides *TypeOf*, we're in business. We might not be able to see into an object inside a list, but we can recover its identity with *TypeOf*.

Another key use for such a feature is in a different domain entirely: writing arbitrary objects to a file and reading them again (for something like a "file of ClassName," in Pascal terms). For instance, we might want to write our list of various objects to a file. The difficulty arises when we try to read an object from the file and reconstitute it as an object. How can we tell what kind of object it was if we've stored objects of the types *TLetter*, *TMemo*, and *TReport* in the same file?

The basic solution, suggested by Wilson, Rosenstein, and Shafer in *Programming with MacApp* (Wilson 1990), is to write the object's type to the file with the object and then use that to decipher what type of object to recreate when we're reading objects back in. But what to write? We could write some kind of code, such as an integer. But suppose we wanted several different programs to read and write the same kinds of data. In that case, a better, more universal way to identify objects is by name: *TLetter*, and so on. We can use *TypeOf* to get an object's type as we're about to write it to the file. We have the object write its own type first and then its data. When it's time to read objects from the file, we first read a string denoting the type of the next object, use that to create an object of the appropriate type, and then have that new object read its own data. See Chapter 25 for more information on getting the object to do the work itself.

Scope and Communication Among Objects

The "scope" of a class's component identifiers (instance variable and method names) extends over the object's type declaration and the type's method bodies even if the bodies are textually separate. This means that all instance variables and method names are visible from inside method bodies. It also means that object component names have to be unique within a class hierarchy.

Method bodies are visible to the class declaration and vice versa. And the instance variable and method identifiers are visible to the outside world.

Also, if class *A* is declared before class *B* in a unit (or program),

```
type
  A = object              { Declared first }
      ⋮
    end;  { Class A }
```

and

```
  B = object              { Declared second }
      ⋮
    end;  { Class B }
```

the names in *A* are visible to *B* and can be used in *B*'s declaration. However, the names in *B* are not visible to *A* and cannot be used in *A*'s declaration. In other words, *A*'s declaration can't mention *B*—can't use *B* as an instance variable type, a parameter type, or a function method's return type. See Figure 8-11 with its notes on what's visible to what. This is in keeping with the general rules of scope in Pascal, although, as we'll see, objects add some new wrinkles.

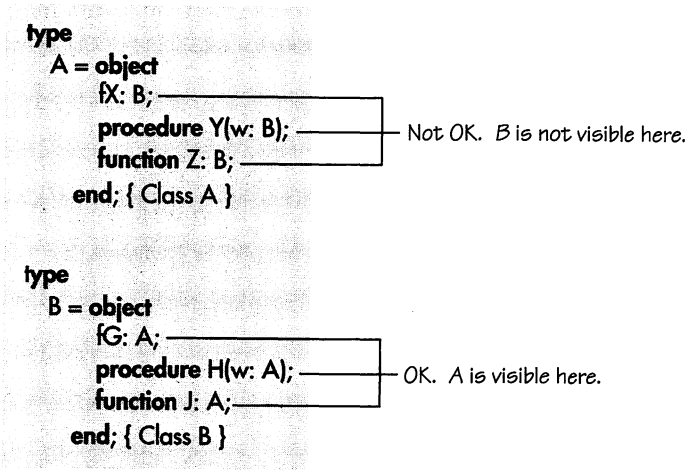


Figure 8-11.
Visibility of names in classes.

Two-Way Communication Between Objects

Suppose we have objects of classes *A* and *B*, where *A* is declared before *B*. We'd like *A*-type objects and *B*-type objects to be able to send each other messages (call each other's methods). Despite Pascal's rules of scope, this turns out to be easy.

Fortunately, object class structure mimics the structure of Pascal units in a handy way. A Pascal unit has two parts: a public interface part and a private implementation part. Program elements declared in the interface part are visible to the outside world and can be used by consumer programs that list the unit in their *uses* clauses. The details of procedure and function bodies are postponed until the implementation part, where they are hidden from the world. Only the implementing programmer is supposed to have access to the implementation.

Likewise, an object class definition has two parts. The class declaration mimics a unit's interface part by listing what's public—the instance variables and only the headers for procedures and functions (methods). The bodies of the class's methods are not given in the class declaration. They're postponed until later.

The declaration of method headers is similar to the *forward* mechanism in standard Pascal. By putting the keyword *forward* at the end of a procedure or function heading in standard Pascal, we make the procedure or function name visible at that point. Then we're free to postpone the implementation until later in the program. The useful effect of this mechanism is that the later implementation can reference program elements defined after the *forward* declaration.

Similarly, method bodies can reference anything defined after the class declaration but before the bodies. This allows us to make two classes visible to each other by declaring them in the same unit (or program declaration part).

Here's an example—we'll suppose that two classes are declared in a program declaration part:

```
program Something;
  type
    A = object

        { Instance variables }

        procedure M;
        function N: Integer;
      end;  { A }

    B = object

        { Instance variables }

        procedure X;
      end;  { B }
  ⋮
  { Method bodies for A and B }
  { All given after both class declarations }
  ⋮
```

Then, in a method body of *A*, we'd like to do things like this:

```
var
  aB: B;    { Declare a B-type object variable }
⋮
aB.X;       { Call one of its methods }
```

Likewise, in the method code for *B*, we'd like to declare *A*-type objects and call their methods.

The second case is no problem. By normal rules of scope, *A* is visible to *B*, so it's OK to declare *A*-type variables and call their methods.

The first case, though, would be a problem without the unitlike structure of class definitions. In their method bodies, *A* and *B* are mutually visible. *A*-type objects can declare *B*-type objects and call their methods, and vice versa. Objects of the two types can communicate.

Two Problems

That's not the end of the scope story, though. There are two potential obstacles to two-way communication between objects.

- One of the classes might be declared (and implemented) in another unit.
- Both classes might need to use each other's names in instance variable types, method parameter lists, or function method return types (elements visible to the outside world).

Let's consider each problem in turn.

Classes not declared in the same unit

Suppose that class *A* is declared in unit *UA* and that class *B* is declared in unit *UB*. The *uses* clause in *UB* looks like this:

```
uses
  UA;
```

This makes *A* visible to *B*, but not vice versa. Objects of the two types can't communicate mutually.

One solution would be to put both classes in a single unit. That might not be practical, though. We'll look at a solution to both this and the next problem. The solution uses abstract superclasses.

Instance variables, parameters, and return types

Not only do we have a problem of scope and visibility when class *A* is declared in one unit and class *B* in another, but there are other difficulties as well. For the moment, we'll ignore the problem of multiple units and focus on classes declared in the same unit.

If the first-declared class (*A*) must use the second-declared class (*B*) in an instance variable, a method parameter, or a function method's return type, unit structure won't solve the problem because class *A* has to know about class *B* in the interface part. Here are some sample declarations to illustrate:

```
unit Something;

interface

  uses
    X, Y, Z;

  type
    A = object  { A needs to know about B, but can't }
        fDataField: B;  { Instance variable of type B }

        procedure M(aVariable: B);{ Parameter type }
        function N: B;  { Function return type }
      end;  { Class A }

    B = object
        fField: A;  { B also references A }
        ⋮
      end;  { Class B }

implementation
  ⋮
```

One approach with this code, of course, would be simply to declare *B* before *A*, if possible. But in some situations—such as here—that's not possible. Suppose, for instance, as in our example, that *B*-type objects need to reference class *A* in instance variables, parameter lists, and the like at the same time that *A*-type objects need to use *B* in instance variables, parameter lists, and so on. This sets up a mutual dependency between the two types that is basically a problem of declaration order, and the implementation part can't help. *B* is unknown to the declaration of *A*. So how can we manage the mutuality we want?

Polymorphism to the rescue

The answer lies in polymorphism, the ability to call the same method of different types of objects. By creating an abstract superclass, *C*, over classes *A* and *B*, we set up a hierarchy and then rely on the polymorphism that goes with hierarchies. Figure 8-12 shows how a superclass can relate two classes.

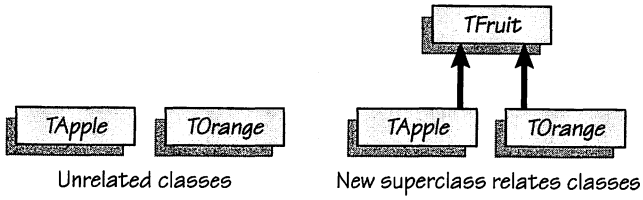


Figure 8-12.
The superclass solution.

Here's the idea: Instead of referring to class *B* inside *A*'s declaration, we refer to class *C*, where *C* is a class known to both *A* and *B*. *C* can be an abstract class with no real substance, if we like. All we need is to declare both *A* and *B* as *C*'s descendants.

Then we can use the fact that it's possible to assign an object of a descendant class to a variable of any of the class's ancestors. That is, we can freely assign objects of type *A* or *B* to a class *C* variable.

Here's some code that creates the hierarchy:

```
unit Something;

interface

  uses
    X, Y, Z;

  type
    C = object
      end;  { Class C }
```

```
  A = object(C)              { Ancestor is C }
      fDataField: C;         { We use C-type variables... }
      procedure M(aVariable: C);              { and parameters... }
      function N: C;         { and function return types }
      procedure SetDataField(value: C);
    end;  { Class A }

  B = object(C)              { Ancestor is C }
      fField: A;
      procedure Q;
    end;  { Class B }

implementation
  ⋮
```

In the implementation, because descendants can be assigned to ancestors, we can assign *B*-type objects to the *C*-type instance variables and parameters in an *A*-type object. *A*'s methods can send messages to a *B*-type object without ever knowing of the existence of such objects:

```
var
  obj1: A;
  obj2: B;
  obj3: C;

⋮
obj1.SetDataField(obj2);      { Assign a B-type object to A's fDataField }
obj1.M(obj2);                 { Use a B-type object as a parameter
                                to A's method M }
obj3 := obj1.N;               { Remember, return type of N is C }
```

and, inside the implementation of *A*—within procedure *A.M*, say—we could send a *B*-type object message, such as

```
procedure A.M;
begin
  ⋮
  self.fDataField.Q;     { Actually, a message to a B-type object }
  ⋮
end;
```

This sends the Q message to whatever object happens to be occupying the variable *fDataField*. At this point in the program, it's the object we declared as *obj2*. I can't emphasize the usefulness of this enough. Even though the *A*-type object has no way of knowing what kind of object is going to be in the *fDataField* variable at runtime, it can still call the *Q* method of that mystery object. (For more on this technique, see "Outlets: A Corollary Technique for Handling Scope Problems," later in this chapter.)

We're not quite through with class *C*, however.

One last wrinkle

When we looked at assigning objects in Chapter 7, we noted that calling the method of an object currently assigned to a variable of one of its ancestor types can be tricky. We need to impose one more requirement on the abstract superclass *C* that we defined to solve our scope problems.

The call to an ancestor's method in this method is just the kind of polymorphic message we're referring to:

```
procedure A.M;
begin
  :
  self.fDataField.Q;      { Actually, a message to a B-type object }
  :
end;
```

At runtime, all that matters is that the actual object referenced by *fDataField* has a method named *Q*. The correct method will be called.

But at compile time, we'll get an error at the message if class *C* doesn't also have a method called *Q*. The compiler can't tell that some descendant object will be assigned to *fDataField* at runtime. All it can tell is that we're trying to call the *Q* method of a *C*-type object. By the way, this limitation also precludes using a class like *TObject* as our abstract superclass because *TObject* has no *Q* method. We could always add one to *TObject*, but it's preferable to invent a new class for the job.

So here's the extra requirement for our communications solution. When we declare abstract superclass *C*, we give it an empty *Q* method. Then, if descendant class *A* or *B* has a *Q* method, it must be an override of the *Q* inherited from class *C*. That will clear up the last obstacle. Here's what the new class declarations should look like:

```
type
  C = object
      procedure Q;    { Implemented as a stub }
    end;  { Class C }

A = object(C)           { Ancestor is C }
      fDataField: C;    { We use C-type variables... }
      procedure M(aVariable: C);                    { and parameters... }
      function N: C;    { and function return types }
    end;  { Class A }

  B = object(C)         { Ancestor is C }
      procedure Q;      { Must override C's Q }
      override;
    end;  { Class B }
```

One word of practical advice is in order: During development, you might find yourself adding methods to classes *A* and *B*. If those methods are ever to be called

polymorphically, you must remember to add the new methods also to class *C* (probably as stubs). Then you have to make the versions in classes *A* and *B* overrides. You'll get "Method not found" compiler error messages each time you forget to add the method in both places or to add the *override* keyword. And don't forget to provide bodies somewhere for these dummy methods. The linker will complain if you don't.

Multiple units again

The final question is how we implement the solution we've just explained if the object classes are in different units.

The answer is to put class *C* in a separate unit too, one that the other classes' units list in their *uses* clauses. Then class *C* will be visible in class *A*'s unit and in class *B*'s unit. Figure 8-13 illustrates the superclass solution applied across multiple units.

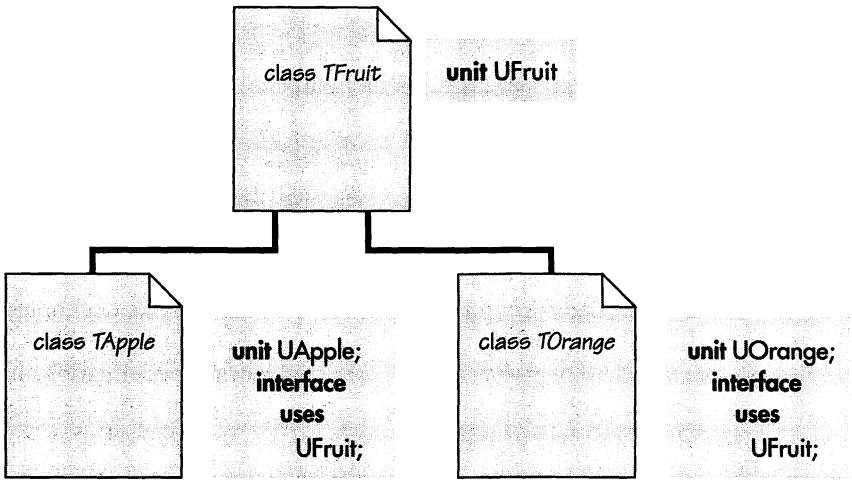


Figure 8-13.
Scope and units—superclass C *in its own unit.*

Most Macintosh programmers use a special unit for their global variables, types, constants, and so forth. This is a good place for class *C*'s declaration and its implementation. It's really just a utility item.

The one limitation on this solution, though, is that if either class *A*'s unit or class *B*'s unit has already been compiled as a library file, we won't be able to change the class's ancestry to *C*. And we won't be able to add *C*'s unit to the appropriate *uses* clause. The solution works only on complete source code that we can make slight modifications to.

In Part 2, we'll look at an example of precisely the communications solution we've laid out here: an abstract superclass called *TDocument*, which is ancestor to another,

more specific document class that must be declared after various classes that need to reference documents. We also saw a small example with buttons and radio button clusters in Chapter 5.

In a moment, we'll look at a corollary to the abstract superclass approach. Objects of different types can communicate mutually using "outlet" variables, a term borrowed from the NextStep operating system on the NeXT computer.

Coupling

A last word on the subject of communications among objects. The rather involved "abstract superclass solution" we've developed implies high coupling among object classes. The more the various classes used in a program have to know about each other, the more "tightly coupled" they are.

As a general rule of software engineering, it's a good idea to avoid tight coupling. Tight coupling increases program complexity significantly, and that increases development time, the potential for errors, and the cost of maintenance.

So programs with less two-way communication are generally preferred. The other side of the argument is that in object-oriented programs the very nature of things is for objects to communicate with each other. There will be many cases where two-way communication—and the superclass solution—are unavoidable, or even preferable.

When we design classes intended for high reusability, they should be as decoupled as possible from all other classes. Ideally, a reusable object doesn't have to call the methods of any other class. Keeping it that way ensures a wider range of situations in which the class can be used. (And if we do end up using a decoupled reusable class but need to ask it to call other objects' methods for purposes undreamed of when the class was created, we can always subclass it to add the extra baggage we need.)

Outlets: A Corollary Technique for Handling Scope Problems

We've looked at the effects of Pascal's scoping rules on our desire to have objects of different types communicate by calling each other's methods. In this section, we present a corollary technique for managing mutual communication.

The scope problem occurred with class declarations like these, which are in different units:

```
{ In unit UButton }

type
  Button = object

      { Instance variables }
```

```
    procedure M;
    function N: Integer;
  end;  { Class Button }

{ In unit UA, which uses unit UButton }

type
  A = object

    { Instance variables }

    procedure X;
  end;  { Class A }
```

where *Button* is visible to *A* (because of declaration order and the *uses* clause) but *A* is not visible to *Button*.

Suppose a method of class *Button* needs to call methods of class *A* despite the invisibility of *A* to *Button*. Here's a neat solution, based again on the abstract superclass approach, with a little help from subclassing.

We subclass *Button* with class *MyButton*, adding an instance variable called something like *fOutlet*, of type *C* (where *C* is a superclass of both *A* and *Button*):

```
type
  MyButton = object(Button)
    fOutlet: C;        { An "outlet" instance variable }
    :
```

(The *MyButton* subclass is necessary so that we can add the instance variable and then, as we do below, see to its proper initialization.)

Of course, class *C* must be visible to class *MyButton*. In *MyButton*'s initialization method, we use as a parameter the object that will be assigned to *MyButton*'s *fOutlet* instance variable:

```
procedure MyButton.IMyButton(outlet: C);
begin
  self.fOutlet := outlet; { "Set" the outlet }
  :
end;
```

Next, when we create an object of class *MyButton* and send it an initialization message, we pass a reference to our class *A* object:

```
var
  aButton: MyButton;
  anAObj: A;
:
New(anAObj);          { The A-type object we want to call }
```

```
:
New(aButton);
aButton.IMyButton(anAObj);          { Pass an A-type object to the button }
```

This passes the button object a reference to an object of class *A*. Note that if an *A*-type object itself creates and initializes the button, it can pass *self* as the parameter.

The button object can then use the outlet to send messages to an object of class *A*:

```
{ In a method of MyButton }

fOutlet.M;                  { Call M method of object in outlet }
```

The "outlet" term is borrowed from Steve Jobs's NeXT computer, where many objects in the NextStep interface have outlets. An outlet is a place to "plug in" a reference to another object. It's also an outlet for messages to the object stored in *fOutlet*.

Inserting Superclasses to Propagate Capability

We've seen that abstract superclasses exist primarily to hand down capabilities to their descendants.

A good example is class *TObject*. It's abstract—we never instantiate any objects from it. It exists only to bequeath the *Clone* and *Free* methods to its descendants. Every descendant of *TObject*, no matter how many intervening classes there are in the hierarchy, inherits methods to clone itself and to free itself.

Most class hierarchies put *TObject* at the top of the hierarchy just to get the benefit of its *Free* and *Clone* methods. This is done by having the next highest class declare *TObject* as its immediate ancestor in the heritage spot.

In effect, *TObject* is inserted at or near the top of the hierarchy, and its capabilities then propagate down the hierarchy to all descendant classes. This is a good way to ensure that all classes in a hierarchy inherit some generally useful trait. Figure 8-14 shows how the inserted class propagates its abilities downward.

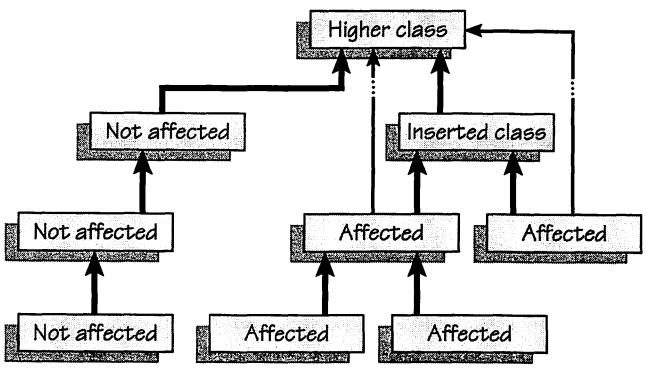


Figure 8-14.
Propagating abilities—relationship of an inserted class to the classes around it.

TErrorHandler is a simple example of a similar useful abstract superclass that we can insert into our hierarchies. We'll insert this one just below *TObject*. The next class down the hierarchy then declares *TErrorHandler* as its immediate ancestor. Then the whole hierarchy inherits *Clone* and *Free* from *TObject*—passed through *TErrorHandler*—plus a simple error-handling mechanism. Every object instantiated from a class in this hierarchy will inherit error-handling ability.

Class TErrorHandler

Here's the class:

```
type
  TErrorHandler = object(TObject)

    fTheError: Integer;      { Most recent error state }
    fLastError: Integer;     { Previous error state }

    procedure IErrorHandler;
    procedure SetError (code: Integer);
    function Error: Integer;
    function LastError: Integer;
    procedure DoHandleError (code: Integer);
  end;  { Class TErrorHandler }
```

By inheritance, any descendant of *TErrorHandler* is—in addition to whatever else it is—a *TErrorHandler* object. It inherits the fields and methods needed to perform its error-handling functions.

By default, a *TErrorHandler* object sets the *fTheError* field with an error code appropriate to the object. A method called *Error* returns the value of that field and resets the field to *kNoErr* (which equals *0*). Consumers can either set *fTheError* directly or, preferably, call the *SetError* method to do it.

We use integers for the error codes to make the class extensible for any error type the consumer might recognize but which we can't now foresee. We provide the following basic codes at the top of *TErrorHandler*'s unit:

```
const
  kNoErr = 0;          { Everything's OK }
  kMiscErr = 1;        { Error, but of no set type }
  kOverflow = 2;       { Overflow of some data structure }
  kUnderflow = 3;      { Underflow, e.g., popping empty stack }
  kNotFound = 4;       { Unsuccessful search }
  kOutofRange = 5;     { Some value was out of acceptable bounds }
```

Consumers can readily add to this list as they need to. They should start numbering beyond 5, and they can't reuse any of these identifiers.

The *DoHandleError* method simply calls *SetError*, but *DoHandleError* can be overridden to do anything the consumer might want. A consumer could use an overridden version to undo damage, back out of a situation, put up an alert box, halt the

program, and so on. A method that allocates storage could simply post an "overflow" error code with *SetError* if it fails to complete the allocation—or it could call its overridden version of *DoHandleError* to free up enough space from a private memory reserve.

TErrorHandler methods

Here are example method implementations for *TErrorHandler*:

```
procedure TErrorHandler.IErrorHandler;
  { Call this from initialization method of any subclass }
begin
  self.fTheError := kNoErr;
  self.fLastError := kNoErr;
end;  { TErrorHandler.IErrorHandler }

procedure TErrorHandler.SetError (code: Integer);
begin
  self.fLastError := self.fTheError;   { Save previous error }
  self.fTheError := code;              { Set new one }
end;  { TErrorHandler.SetError }

function TErrorHandler.Error: Integer;
begin
  Error := self.fTheError;             { Value to return  }
  self.fLastError := self.fTheError;   { Save for undoing }
  self.fTheError := kNoErr;            { Reset current code }
end;  { TErrorHandler.Error }

function TErrorHandler.LastError: Integer;
begin
  LastError := self.fLastError;
end;  { TErrorHandler.LastError }

procedure TErrorHandler.DoHandleError (code: Integer);
begin
  self.SetError(code);                 { Just call SetError }
    { Consumers are invited to override this method as they need to }
end;  { TErrorHandler.DoHandleError }
```

The *fLastError* instance variable contains the value of the previous error in case it's needed. We do this because the *fTheError* variable gets reset each time we read it with *Error*.

Of course, we can't use this mechanism to trap Macintosh system errors, but there will be lots of error conditions in our own code for which this is a built-in—yet extensible—mechanism for handling errors. We'll use it extensively (extended a bit) in Parts 2 and 3. We'll also look at an alternative way to use *TErrorHandler* later in this chapter.

Other Inserted Superclasses

Later we'll see other examples of inserting abstract superclasses into a hierarchy in order to propagate their capabilities down through the whole hierarchy.

In particular, we'll further develop class *TEvtHandler*, a class of objects that can handle Macintosh mouse, key, and menu events; class *TRandom*, whose descendants can generate pseudorandom numbers; and several classes centered around *TLinkable*—classes of objects that can link themselves into a linked list, a binary tree, and so on. *TEvtHandler* provides a mechanism for sending events such as mouse-downs to the correct target objects. *TLinkable* helps us with lists of documents, lists of windows, lists of views, and so on because those kinds of objects "know" how to be part of a linked list. A subclass, *TDoublyLinkable*, provides doubly linked list capability. *TTreeable* allows an object to be inserted in a binary tree. And a more general class, *TNode*, allows one object to be inserted in either a list or a tree.

Composite Objects

A composite object is one that contains instance variables that refer to other objects. We've already seen examples of composites, but let's look at one explicitly.

Suppose we have an object that needs to handle errors. We've worked with class *TErrorHandler* already in this chapter, but we made it the ancestor of any class needing to handle errors. As in Figure 8-15, those classes acquired their abilities from *TErrorHandler* by inheritance.

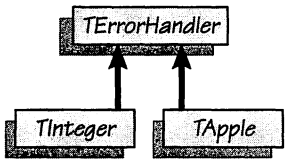


Figure 8-15.
Acquiring an ability by inheritance. TInteger *and* TApple *are error handlers, inheriting all of* TErrorHandler*'s instance variables and methods.*

In Part 3 we'll develop a linked list object that needs to handle errors if bad nodes are added, if operations go awry, and so on. One approach is to make our *TList* class a subclass of *TErrorHandler*. Then the list acquires the ability by inheriting it.

The composite approach works this way:

```
type
  TList = object(TObject)  { Not TErrorHandler }
      fItsErrHdlr: TErrorHandler;
      :
    end;  { Class TList }
```

In *TList*'s initialization method, we create and install an error-handler object:

```
New(self.fItsErrHdlr);
self.fItsErrHdlr.IErrorHandler;
```

Now the list object "contains" an instance of an error-handler object. As Figure 8-16 suggests, we can say that the list has acquired a capability by using another class rather than by inheriting from it. To use the error handler, the list object's methods send messages to the error handler this way:

```
self.fItsErrHdlr.SetError(theErr);
```

If we made *TList* a subclass of *TErrorHandler*, *TList* would inherit the error-handling methods. We wouldn't prefix the method names with something like *fItsErrHdlr*. Instead, we could say *self.SetError(theErr)*.

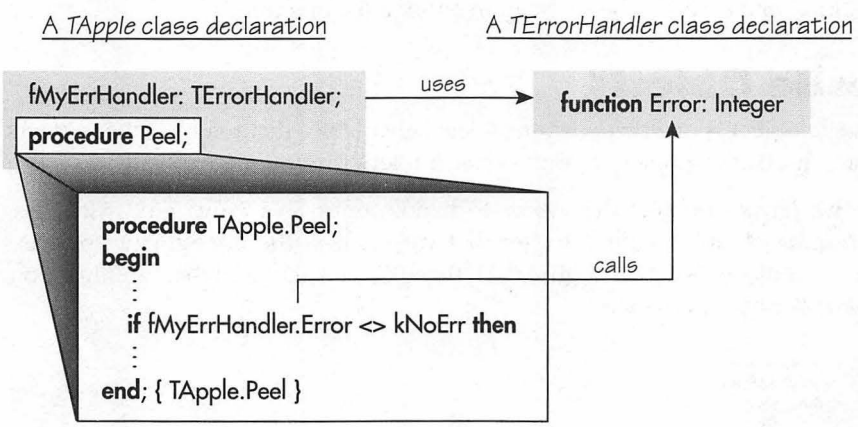


Figure 8-16.
Acquiring an ability by use.

Which approach is better? Neither, really. In some situations, one might be better than the other. Inserting a class like *TErrorHandler* into an existing hierarchy might force us to make some classes error handlers—with the extra overhead that entails—that don't need to be. In that case, the composite approach would be better.

Multiple Inheritance

We've pointed out that in Object Pascal a class can inherit from one and only one immediate ancestor. Some languages provide the more powerful multiple inheritance mechanism. With multiple inheritance, a class can inherit properties and behaviors from more than one immediate ancestor at a time. As Figure 8-17 illustrates, under multiple inheritance, class *WolfMan* could inherit some of its nature from class *Man* and some from class *Wolf*.

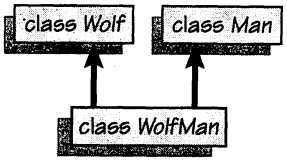


Figure 8-17.
Multiple inheritance.

With multiple inheritance a hierarchy can exhibit fan-in as well as fan-out tendencies. The tree branches can reconverge.

Multiple inheritance leads to some complications that languages using it must somehow resolve. In particular, what if classes *Man* and *Wolf* both have an instance variable called *fSpecies*? Which *fSpecies* does *WolfMan* inherit? Or, more troublesome, if *Man* and *Wolf* both have a method *MakeALiving*, which version of *MakeALiving* does *WolfMan* inherit? Undoubtedly, wolves and men make a living in very different ways.

Faking Multiple Inheritance in Object Pascal

In the more limited Object Pascal, can we at least fake multiple inheritance? In a sense, yes, although such fakery can have some undesirable side effects.

Let's look at class *WolfMan* in Object Pascal. Because a class can't have more than one immediate ancestor, we have to string things out in one of the ways shown in Figure 8-18.

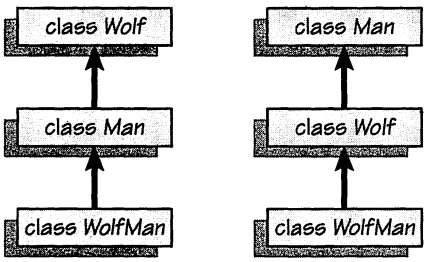


Figure 8-18.
Two ways to fake multiple inheritance.

Either *Wolf* must be ancestor to *Man*, or *Man* to *Wolf*. *WolfMan* inherits the characteristics of both ancestors, just as we want it to. The side effect is that either class *Man* has to carry a lot of often-unnecessary *Wolf* baggage along, or vice versa. Class *Man* might end up with a method *RunInPack*, for instance. (Come to think of it...)

A more realistic example would be a tree data structure that can be characterized as a list of its subtrees. Each tree node is a list. It would be useful if tree nodes could

inherit their node-ishness from a class *TNode* and their list-ishness from a class *TList*, both at the same time. In some languages, that's just how you'd create such a structure.

In some cases, where the effect of multiple inheritance is critical to our needs, the price we pay for this solution (the extra baggage) might be worth it.

Summary

In this chapter, we looked at the intricacies of the object class hierarchy—its structure, what we can do with that structure, and some potential difficulties hierarchies can create when Object Pascal is used for large software projects involving many programmers.

We also explored the *inherited* mechanism, abstract superclasses, class *TObject*, Pascal scope, two-way communication between objects, composite objects, and multiple inheritance.

Projects

- Write a small program to try out the scoping solutions we've discussed. Declare two mutually dependent classes that reference each other in their class declarations. Then use an abstract superclass to bridge the two and make them visible to each other.
- Write a small program to experiment with *TObject*'s *Free* and *Clone* methods. In so doing, take a look at the interface file for *TObject* (called ObjIntf.p) for your system. Write down your own explanation for why *TObject* defines both *Clone* and *ShallowClone* methods. Can you think of other situations in which *X* and *ShallowX* methods might be useful?

9

INSIDE OBJECT PASCAL

Now we'll get into Object Pascal in considerably more technical detail. This chapter can be a useful reference as you read the rest of the book and as you take off on your own object-oriented projects. We'll cover

- Object Pascal's syntax
- What objects are like in the heap
- A hypothetical runtime binding mechanism
- Efficiency
- Optimizing
- Compiling libraries containing objects

Object Pascal Syntax

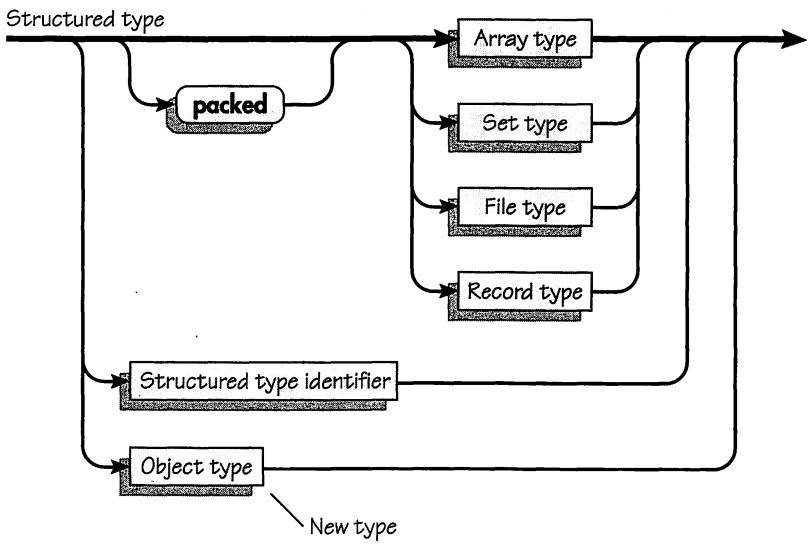
Object Pascal is an extension of standard Pascal and, no doubt under the influence of Niklaus Wirth's parsimony, a small extension at that. To support objects, Object Pascal's designers at Apple added only four new keywords: *object*, *inherited*, *override*, and *self*. For the full Object Pascal technical specifications, get a copy of Larry Tesler's "Object Pascal Report" (Tesler 1985), an extension of Jensen and Wirth's *Pascal User Manual and Report*, Third Edition (Jensen 1984), through the Apple Programmers and Developers Association (APDA).

The Apple designers' syntax additions relate to

- The definition of structured types
- The declaration of an object type
- The "heritage" identifier
- The declaration of methods
- The implementation of method body blocks
- The syntax of method calls

Definition of Structured Types

To the standard Pascal structured types—arrays, sets, files, and records—Object Pascal adds the object type. The object type resembles the record type but can't be packed and can't have a variant part. Unlike other types, an object type can be declared only in the type declaration part of a main program or a unit. An object type can't be declared in a nested scope (a procedure or function declaration block), in a variable declaration part, or in a formal parameter list. Figure 9-1 shows the addition of an object type to the railroad diagram for structured types.



Adapted, by permission, from Symantec's *THINK Pascal Manual*

Figure 9-1.
Structured types in Pascal—with the addition of an object type.

Declaration of an Object Type

Figure 9-2 diagrams the syntax for an object type declaration, including the declaration's heritage spot.

Heritage declaration

To indicate that it inherits from an ancestor object type (class), an object type declaration includes a "heritage" identifier in parentheses after the keyword *object*. For example,

```
type
  TButton = object(TEvtHandler)
```

where the heritage identifier (*TEvtHandler* in this case) is the name of the immediate ancestor class. A new object type is not required to inherit from another class, so

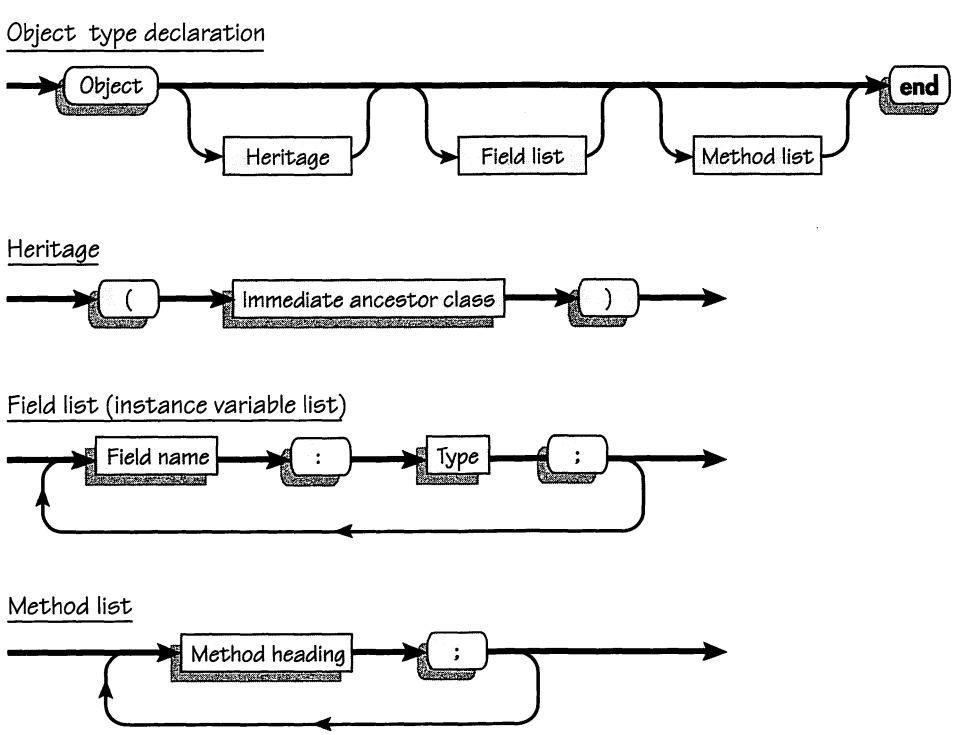
the heritage element in the object type declaration is optional. Most object types do inherit, though, if only from class *TObject*. This gives objects of the inheriting type the ability to free their storage in the heap after use. We looked at *TObject* and its *Free* method in Chapter 8. Figure 9-2 shows the syntax for the heritage identifier.

Field list (instance variable list) declaration

Like a record type, an object type can have data fields (often called instance variables) as components. The fields can be of any type or types, including, recursively, the type of the object within whose declaration they appear. In an object type declaration, all instance variables must be listed before any method headings are. Figure 9-2 shows the syntax for the field list.

Method list (method headings) declaration

Unlike a record type, an object type can also have method components. In an object type declaration, the method headings are complete procedure or function headings. The syntax for a method list is also shown in Figure 9-2.

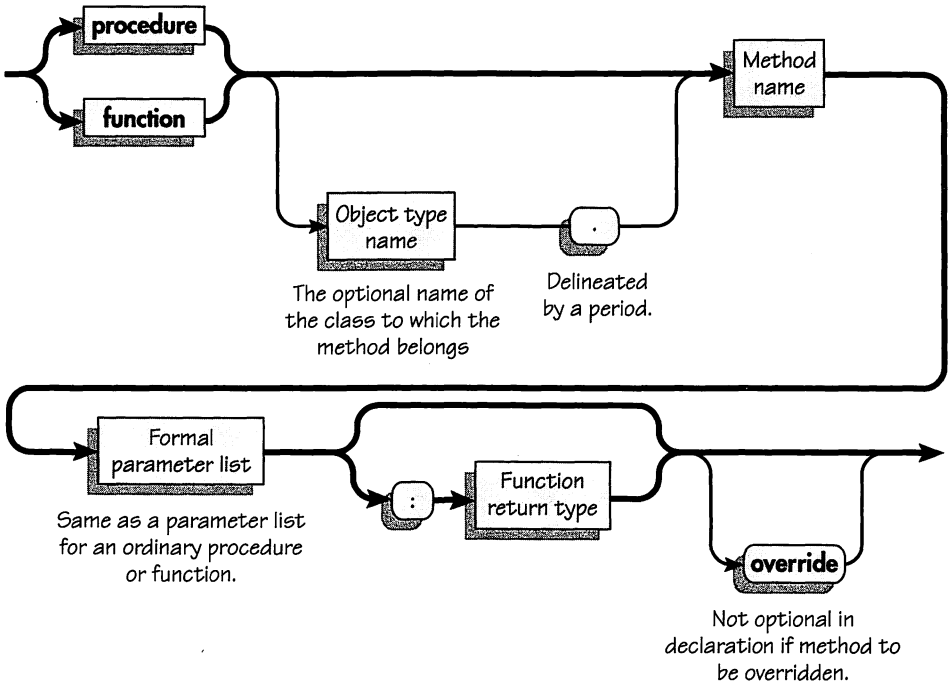


Adapted, by permission, from Symantec's *THINK Pascal Manual*

Figure 9-2.
Syntax of an object type (class) declaration including the heritage identifier, the field list, and the method list.

Figure 9-3 is a full diagram of the the syntax for a method heading in a declaration.

Method heading in a declaration



Adapted, by permission, from Symantec's *THINK Pascal Manual*

Figure 9-3.
Syntax of a method heading in a class declaration.

The corresponding method bodies appear below the object type declaration (after the declaration's *end* statement). In a main program, the method bodies are given in the declaration part after the object type declaration. In a unit, the type declaration, including the instance variable list and the method headings, is given in the unit's interface part, and the method bodies are given in the unit's implementation part.

If a method inherited from an ancestor is to be overridden, the method heading must include the keyword *override*. The keyword is required in the method heading in the class declaration but is optional in the method body.

Even though the method bodies are given separately from the object type declaration, the scope of an instance variable or a method in the object type declaration extends to the method bodies as well as the object type declaration itself. The scope of an instance variable or a method also extends to the object class's descendants. The spelling of an instance variable or method identifier must be unique throughout an

object hierarchy. The scope of instance variables and methods also extends to component designators (dot-notation accesses) and to statements in which the identifiers are used.

Variable (object reference) declaration

Variables declared of an object type are called "object references," "reference variables," or "instances." Given an object type called *TButton*, a reference variable of that type would be declared as

```
var
  aButton: TButton;
```

Such declarations are like any other Pascal variable declarations.

Declaration and Implementation of Methods

Methods are declared in the same style as forward declarations for procedures and functions. The method is declared, as in a forward declaration, inside the object type declaration (but without the *forward* directive). Then the method heading is repeated elsewhere with the implementation of the method's full body.

When the method heading is repeated in the implementation, it must be related to its heading in the object type declaration (must extend the object type's scope over the body) by a qualification of the method name with the object type's name. To qualify a method name with its object's type name, put a dot (.) between the object type name and the method name. For example,

```
procedure TSphere.Display;
```

In the method body heading, repeating the method's parameter list and function return type is optional. If the method was overridden, repeating the *override* keyword with the method body heading is optional.

Figure 9-4 on the next page diagrams the syntax of a method body, including a breakdown of the method heading's syntax.

Method Calling (Message Sending)

Methods greatly resemble ordinary procedures and functions, but calling them requires a different syntax.

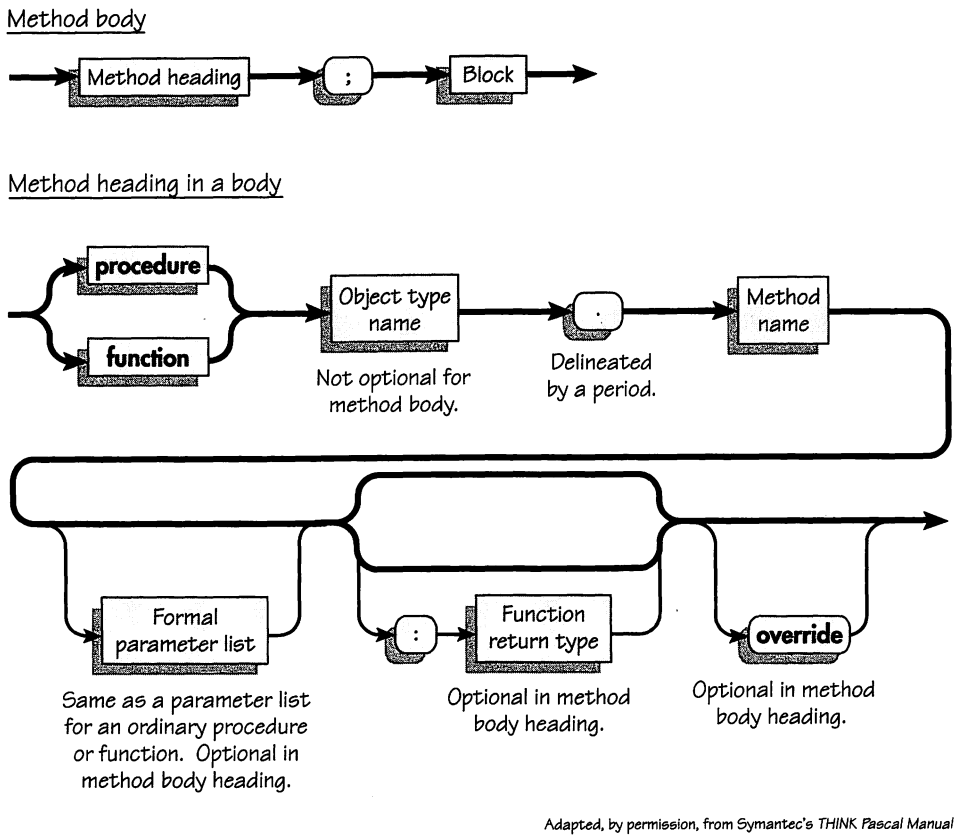
The reference variable prefix

We call a method (send a message) by prefixing the method name with the name of the object whose method it is, using a record-style dot notation. For example,

```
aSphere.Display;                { Display message to aSphere }
vol := aSphere.VolumeOf;        { VolumeOf message to aSphere }
```

or

```
aSphere.SetRadius(r);           { SetRadius message to aSphere }
```



Adapted, by permission, from Symantec's THINK Pascal Manual

Figure 9-4.
Syntax of a method body and of the method heading in a method body.

The reference variable *self*

Inside an object—that is, inside the code of an object's methods—we can call another method of that same object without the reference variable prefix. Optionally, we can use the special reference variable *self* as the prefix, as in *self.Display*. This means the same as a simple *Display* would. Using *self* in this way, though, does improve clarity in some complex methods.

The *inherited* keyword

By using the *inherited* keyword, we can call an ancestor's method that our object type has overridden. This keyword tells the compiler to use the ancestor's version of the method instead of the current object's. We can use *inherited* to call only our immediate ancestor's version of the method. Any version declared higher up in the hierarchy can't be called this way. If a method gets overridden two or three times in a hierarchy, some of the versions in higher classes will be inaccessible from lower classes. A call using *inherited* looks like this:

```
inherited ProcMethodName(parameters);
```

or this:

```
result := inherited FuncMethodName(parameters);
```

The *inherited* keyword can be used only inside an object class's method code.

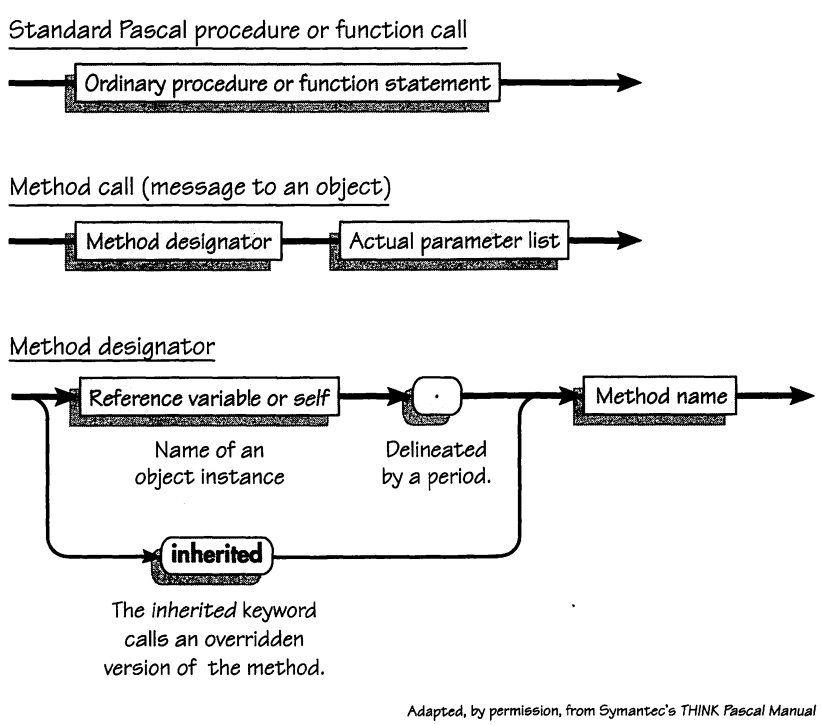
Outside an object's methods—from some other object or a group of ordinary Pascal statements not part of some object—we can call a method only by prefixing it with an object variable identifier. There's no other way for the compiler to know whose method we mean without the object name as a prefix.

Calling function methods

Calling methods that are functions is otherwise similar to calling ordinary functions:

```
result := aFunction(parameters);                    { Ordinary function call }
result := anObject.aFunctionMethod(parameters);     { Function method call }
```

Figure 9-5 diagrams the syntax for message sending.



Adapted, by permission, from Symantec's *THINK Pascal Manual*

Figure 9-5.
Syntax of an ordinary procedure or function call and of a method call, with a breakdown of the method designator syntax.

with statements

Outside an object, we can use an object reference variable in a *with* statement, just as we can a record-type variable:

```
with anObject do
  begin
    field1 := 3;                    { Access anObject's field }
    Display;                        { Call anObject's method }
  end;  { with }
```

Inside the code of an object, using a *with* statement this way is pretty pointless:

```
with self do
  begin
    field1 := 3;
    Display;
  end;  { with }
```

We can achieve exactly the same effect with either of these forms:

```
field1 := 3;
Display;
```

or

```
self.field1 := 3;
self.Display;
```

The first version acts as though the code were surrounded by an implicit *with* statement. Because using *self* is optional, there's no need for a *with* statement.

The *self* identifier

The special identifier *self* is an object reference that points to the object within which it is used. Every time we call a method, *self* is passed with the call as if it were part of the parameter list. It's only implicitly there, however; we don't actually list it among our parameters.

Outside the context of an object's method code, *self* would make no sense. Inside that code, *self* is entirely optional as a qualifier in instance variable accesses and calls to the object's own methods.

Although explicit use of *self* is almost never required, programmers can make good use of *self* in certain special situations. As an example, consider the discussion of two-way communication between objects in Chapter 8 (also demonstrated in real programs in Part 2 as well as in the example for radio button clusters in Chapter 4).

Assignment Compatibility

Descendant objects can be assigned to ancestor object reference variables, but not vice versa. This extension of Pascal's assignment compatibility rules makes polymorphism possible. Object reference variables can also be assigned the value *nil*.

Type Coercion of Object Reference Variables

When a polymorphic object variable "contains" an object of some descendant type, you can use type coercion (typecasting) to convert the variable to the type of its actual contents. The variable must contain the type you are casting to or any type descended from it.

```
var
  aFood: TFood;
  anotherFood: TFood;
  anApple: TApple;
  aFruit: TFruit;
⋮
  aFood := anApple;
  ⋮
  aFruit := TApple(aFood);          { Typecast to actual type }
  anotherFood := TFruit(aFood);     { Typecast to descendant type }
```

In this example, the object that was assigned to *aFood* earlier is now also referenced by *aFruit* and by *anotherFood*, cast in each case to a legal type. Figure 9-6 shows how several variables can reference the same object.

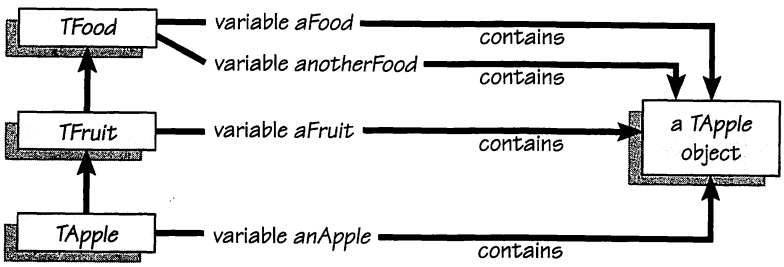


Figure 9-6.
Contained object. Each variable "contains" (references) the same actual TApple *object.*

The *Member* Function

Object Pascal's *Member* function returns *true* if its first parameter is a member of the class listed in its second parameter. The function header looks like this:

```
function Member(r, t): Boolean;
```

where *r* is an object reference (the name of an instance), such as *anApple*, and *t* is a type identifier of a reference type (the name of a class), such as *TFruit*. For example,

```
Member(anApple, TFruit)
```

returns *true* because *anApple is-a Tfruit*. So does

```
Member(anApple, TApple)
```

but not

```
Member(anApple, TPear)
```

Member can be handy as a test before you try to typecast an object. You can see whether the actual object contained in an object reference variable is the type you expect. For example,

```
if Member(aFruit, TApple) then
  anApple := TApple(aFruit);
```

This statement first tests to see whether the typecast is OK and then typecasts the contents of *aFruit* to an object explicitly of type *TApple*. You can use the same approach to extract the actual type of a variable of any class higher in the hierarchy, including *TObject*.

To determine a particular object's position in its hierarchy, use the *Member* function several times, starting with the most specific class:

```
if Member(appetizer, TApple) then
  ⋮
else if Member(appetizer, TFruit) then
  ⋮
else if Member(appetizer, TFood) then
  ⋮
```

This will use the first class that returns *true* (the most specific).

Don't overuse *Member*. It's easy to misuse it, as Kurt Schmucker points out in *Object-Oriented Programming for the Macintosh* (Schmucker 1986a). Consider this convoluted code, paraphrased from an example that speaks to the same point in Schmucker's book:

```
if Member(aFruit, TGrannySmith) then
  PeelGrannySmith(TGrannySmith(aFruit))
else if Member(aFruit, TMcIntosh) then
  PeelMcIntosh(TMcIntosh(aFruit))
else if Member(aFruit, TApple) then
  PeelApple(TApple(aFruit));
```

Each *if* statement tests to see whether the actual object in *aFruit* is a particular type—first a Granny Smith apple, then a McIntosh apple, and then any kind of apple at all. (Note that the specific types are tested for before the more general one.) If *Member* returns *true*, a procedure (not a method!) is called with *aFruit* as its parameter. Before being passed to the *Peelxxx* procedure, *aFruit* is typecast to the appropriate type.

But because *TFruit*, *TApple*, *TGrannySmith*, and *TMcIntosh* are all in the same object hierarchy, it makes sense to have a *Peel* method declared in *TApple*, which all of *TApple*'s subclasses inherit (or override). Then all the convoluted code above can be replaced by this:

```
aFruit.Peel;
```

On the Heap

Now we'll look at how objects function in memory, at object structure, and, later, at how runtime binding can be implemented.

Different implementations of Object Pascal might differ in the ways they implement and manage objects. At this writing, for example, Symantec says it has changed the runtime binding mechanism in THINK Pascal between versions 2 and 3 and might change it again for version 4. Keep the hypothetical nature of this discussion in mind. We're not looking at how a particular Object Pascal implements objects or does runtime binding.

The idea is to show how it might work. This scheme is simple and would be fairly fast, but at the cost of considerable space, especially for classes with lots of methods. Let's not pretend that it's the best thing since sliced bread but, rather, use it to get a feel for how polymorphism and runtime binding can be managed.

Object Reference Variables

An object reference is implemented on the Macintosh as a handle, a pointer to a pointer to a block of storage in the application heap. The handle itself points to a master pointer, which in turn points to a block. When memory gets shuffled around or compacted by the Memory Manager, handled blocks can safely be moved, in a way that's transparent to applications. The master pointer changes to reflect the block's new location, but the handle still points to the same master pointer location.

A handle has to be doubly dereferenced in order to access the block it points to. In Object Pascal, this dereferencing is done for us when it comes to object references. We shouldn't try to dereference an object reference.

Objects in the Heap

An object instance is allocated as a block of storage in the heap, pointed to by its object reference (a handle). The block contains space for the object's instance variables. It also contains a link of some kind to the instance's class information. Figure 9-7 on the next page shows our object in the heap with the handle, the object block, the block header, the class link, and data. The instance's block of memory does not contain the method code, nor even the addresses of methods. That approach would take up too much space for each instance of a class. Instead, the instance's link to the class information can be followed to get at a dispatch table for the class's methods.

Because objects of descendant classes inherit all the instance variables declared anywhere above them in the class hierarchy, each object instance must have its own space in the heap for each instance variable. Figure 9-8 on the next page shows the blocks for several objects in a class hierarchy with repeated instance variables in the same order. Descendant objects take up larger blocks than objects higher in the hierarchy. The order of fields in the blocks remains the same all through the hierarchy,

with new fields added at the bottom. Any object reference in a hierarchy thus points to a block of memory that starts with the same fields, even though the block can have more fields tacked on at the end.

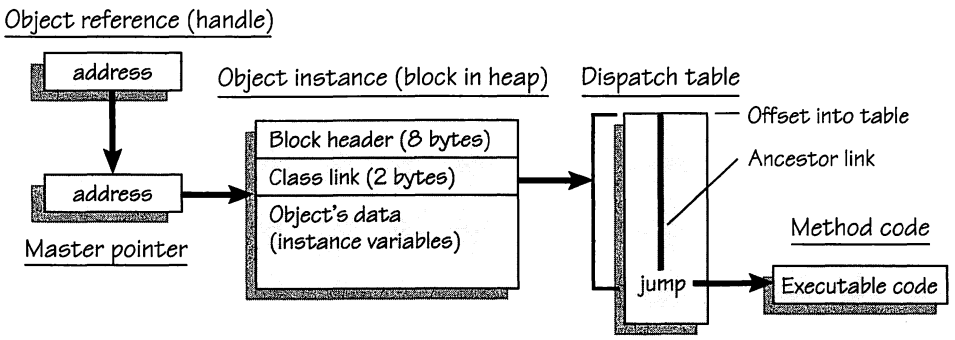


Figure 9-7.
A hypothetical object in the heap.

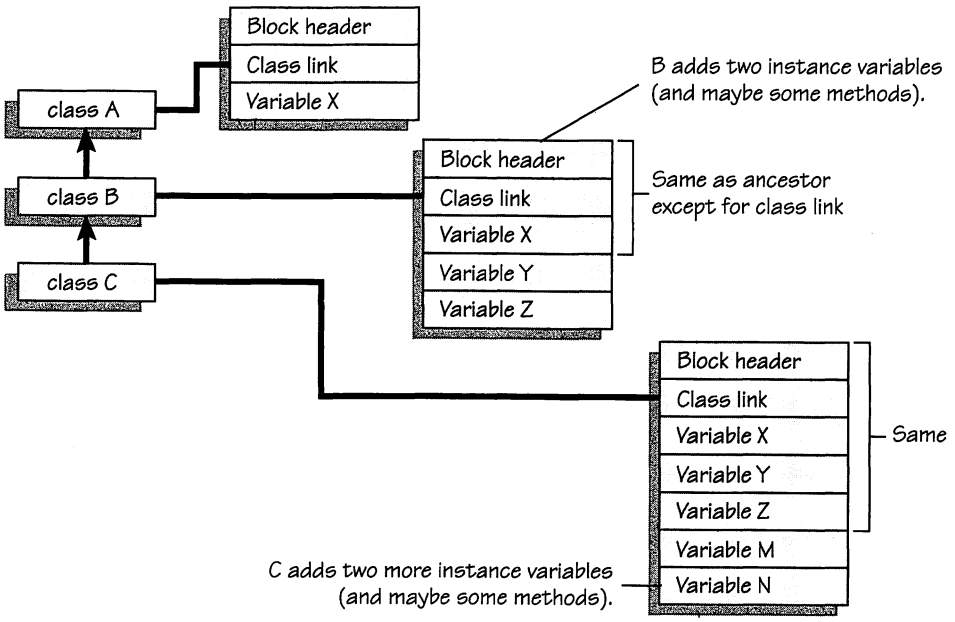


Figure 9-8.
Descendants in the heap.

Structure of an Object Block in the Heap

An object instance is allocated memory from the heap much as any other dynamic data structure is. We'll use the structure of an object as implemented in THINK Pascal as an example. Keep in mind that this is undocumented information, subject to change in future versions.

At the start of the block is a standard 8-byte block header that carries information needed by the Memory Manager and various Toolbox routines. Following the header is the content part of the block. In THINK Pascal 3.0, there is enough room for the object's instance variables (both inherited and new) plus an extra 2 bytes. That's it. Figure 9-9 diagrams this byte-by-byte structure of an object block in the heap.

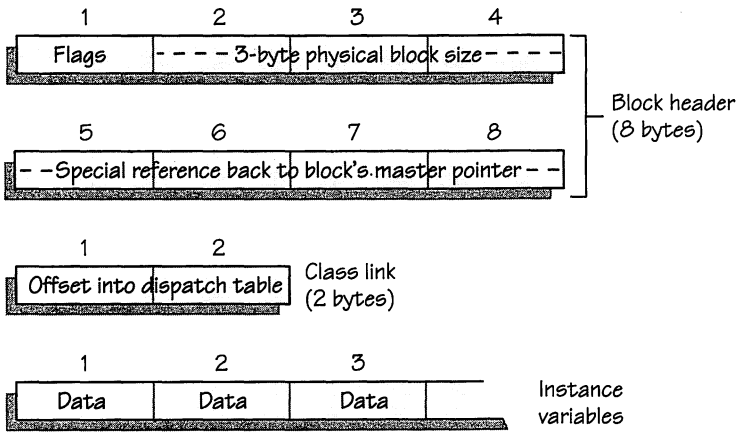


Figure 9-9.
Object block structure.

Let's look at what's in a block for an object declared the way this one is:

```
type
  A = object(TObject)
      i: Integer;                    { One instance variable }
      procedure MethodA1;
      { Also inherits TObject's methods }
      {  but overrides Free }
      procedure Free;
      override;
    end;  { Class A }
```

```
B = object(A)
    j: Integer;                    { Two more instance variables }
    c: Char;

    { Inherits A.MethodA1 }
    procedure MethodB1;
    { Also inherits TObject's methods }
    {  but overrides A.Free }
    procedure Free;
    override;
  end;  { Class B }
⋮
var
  anObj: B;
⋮
  New(anObj);                      { Allocates the block in the heap }
```

Here's the structure of the block for *anObj*, an instance of class *B*:

- Block header (8 bytes).
  Byte 1: flags byte of header.
  Bits 1–4 tell whether block is relocatable and so on.
  Bits 5–8 tell the number of unused bytes in block.
  Bytes 2–4: 3-byte physical size of block (including header). Here it contains 10H (16 decimal), for header: 8 bytes + class link: 2 bytes + variable i: 2 bytes + variable j: 2 bytes + variable c: 2 bytes.
  Bytes 5–8: reference to block's master pointer in a special form—not an actual address or pointer.
- Contents (8 bytes, in this case). The block's master pointer points to the beginning of the content—not to the beginning of the header.
  Bytes 1–2: class link.
  Bytes 3–4: variable *i*, a 16-bit integer inherited from class *A*.
  Bytes 5–6: variable *j*, a 16-bit integer.
  Bytes 7–8: variable *c*, a character in 2 bytes.

Note that field *i*, inherited from class *A*, comes first in the *B* object's field list. Consult *Inside Macintosh,* II-9, for more details about block structure.

The class link

The most interesting item in this block is the 2-byte class link. It's not an address or a pointer or a handle; instead, it's an integer value that gets used as an offset from some known value (such as the address stored in a register or the start of a CODE resource or the application's heap zone) to the beginning of class *B*'s method dispatch table. Figure 9-10 shows how the class link value can be used to compute dispatch table location.

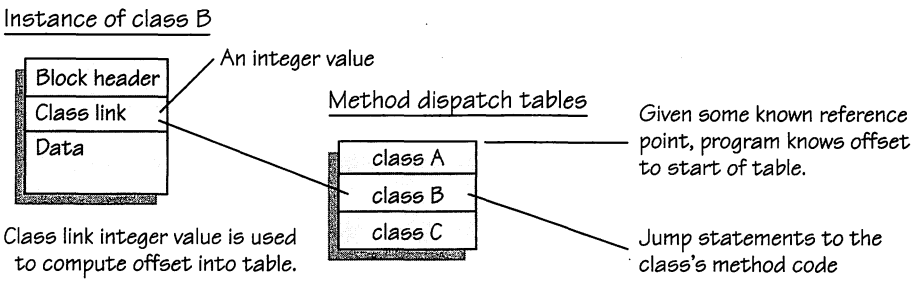


Figure 9-10.
An object's class link.

How Runtime Binding Works

When a method is called for an instance of a class, the instance's class link is used to reach the class's method dispatch table.

The method dispatch table hierarchy

If the class does not have the code for the method that has been called (probably because the method is inherited), the link to the class containing the inherited method is followed, and that class's method dispatch table is used to locate the body of the method that has been called. Figure 9-11 illustrates this search through the method dispatch tables for the method that has been called. The correct table entry for the method must be found, or an error occurs. As soon as the correct entry is found, the program "jumps" to the method's code and executes it.

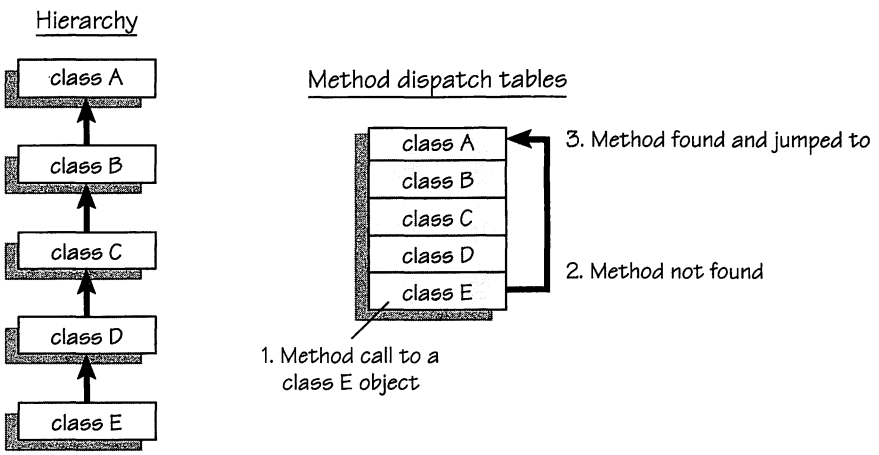


Figure 9-11.
The lookup order in the hierarchical dispatch tables. The method is defined by class A *and isn't overridden in the hierarchy.*

Compiled classes

When a class is compiled, the compiler builds a method dispatch table for the class. (If there are several classes—or many—in a hierarchy, their tables are laid out contiguously.) A class's table probably contains at least two items:

- A link or offset to the table for the class's immediate ancestor.
- A machine language jump instruction (JMP) for each method in the table. This instruction enables the program to jump to and execute the method's code.

The compiler also generates code for each method call in the source code. The code for each call must

- Push the method's parameters (if any) onto the stack
- Push an address to return to after the method call
- Calculate the location of the method dispatch table for the object whose method is being called
- Look up or calculate the particular method's offset within the dispatch table (first method, second method, third method, and so on)
- Jump to the address of the calculated dispatch table entry

Polymorphism

When polymorphism is not involved, the compiler can resolve which method has been called and generate the necessary code to set up the stack with the method's parameters and the return address, then jump to the method's address, and finally clean up the stack after the call returns. This would be a direct call, with no dispatch table lookup.

Polymorphic objects must be bound at runtime, not compile time. If we have

```
var
  aFood: TFood;
  anApple: TApple;
  aSteak: TSteak;
⋮
  aFood.Prepare;
```

the compiler has no way of telling what object will be referred to by *aFood* at runtime and thus no way of telling which object is the target of the call to *Prepare*. Is it *TFood*'s *Prepare* method? *TFruit*'s? *TApple*'s? *TSteak*'s?

To handle this, the compiler must generate code to examine the actual object referenced by *aFood* at runtime, follow that object's link to its class information, check the method dispatch table there, and, if the method is inherited by that class, then trace the method on up the hierarchy until it's found.

Let's look at one way we could implement runtime binding using the object structure we've outlined.

A Hypothetical Runtime Binding Mechanism

Again, remember that the mechanism we're looking at doesn't necessarily correspond to any actual implementation of Object Pascal. It should give us a general sense of how runtime binding might be done.

The Method Dispatch Table

As it compiles class declarations and method bodies, the compiler creates a jump table for dispatching calls to methods. The table contains groups of executable jump statements, one group of jump statements per class. For a given class hierarchy, say one featuring, in descending order, the classes *TObject*, *A*, and *B*, there's a group of jump statements for *TObject*, a group for class *A*, and a group for class *B*, in that order. Each group contains a jump statement for each method in the class.

At the start of each group, an integer offset value tells (indirectly) how far the group's beginning is from the start of the whole hierarchy in the table. We'll call this the group's "ancestor link." Class *A*'s ancestor link is an offset back to *TObject*'s group. Class *B*'s ancestor link is an offset back to class *A*'s group. Figure 9-12 shows what this part of our hypothetical dispatch table might look like.

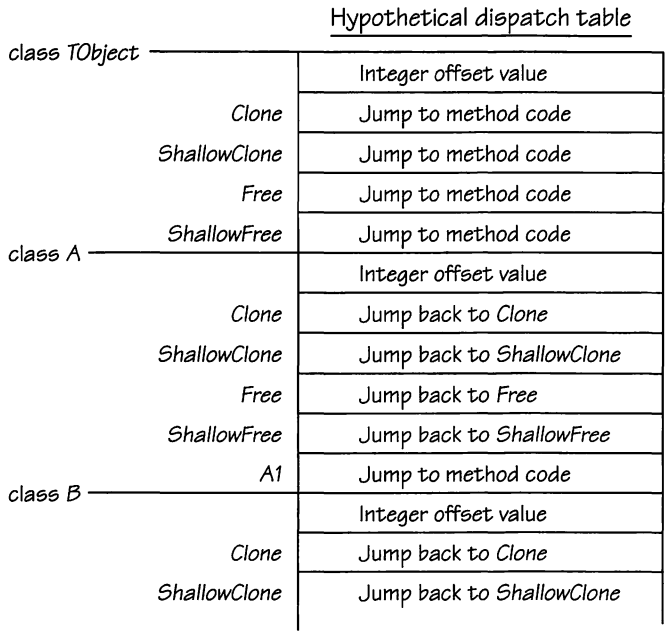


Figure 9-12.
Contents of the dispatch table.

Each jump instruction is 6 bytes long: 2 bytes for the instruction itself and 4 bytes for the address to jump to.

Suppose our hierarchy contains the following methods:

- class *TObject*
  - *Clone* (#1)
  - *ShallowClone* (#2)
  - *Free* (#3)
  - *ShallowFree* (#4)
- class *A*
  - *MethodA1* (#5)
  - Also overrides *TObject.Free*
- class *B*
  - *MethodB1* (#6)
  - Also overrides *A.Free*

Note that class *A* overrides *TObject.Free* and that class *B* overrides *A.Free*. The numbers next to the method names are the "method numbers" we'll use to compute offsets into the method dispatch table.

For simplicity, each group has a jump statement for each of the class's methods, starting with the methods it inherits and then adding any new methods. Our method dispatch table might look like the one shown below.

At some specific offset from a known point in memory:

	Table group for class *TObject*	
tablestart	0	*TObject*'s ancestor link—none
	Jump to address 1	method1 *TObject.Clone*
	Jump to address 2	method2 *TObject.ShallowClone*
	Jump to address 3	method3 *TObject.Free*
	Jump to address 4	method4 *TObject.ShallowFree*
	Table group for class *A*	
class *A*	26	class *A*'s ancestor link—back 26 bytes
	Jump back 26	to *Clone*'s jump statement
	Jump back 26	to *ShallowClone*'s jump statement
	Jump to address 3a	overridden *Free* method—new address to jump to
	Jump back 26	to *ShallowFree*'s jump statement
	Jump to address 5	*MethodA1*

	Table group for class *B*	
class *B*	32	class *B*'s ancestor link—back 32 bytes
	Jump back 32 + 26	method1 *Clone*
	Jump back 32 + 26	method2 *ShallowClone*
	Jump to address 3b	overridden *Free* method—new address to jump to
	Jump back 32 + 26	method4 *ShallowFree*
	Jump back 32	method5 *MethodA1*
	Jump to address 6	method6 *MethodB1*

Each class starts (after its ancestor link) with its inherited methods, one jump statement for each. Unless it overrides a method, it uses a relative jump, backward to the table entry for that method in the class from which it's inherited. Thus, for example, the entry for method *B.Clone* should jump back 58 bytes to the table entry for method *TObject.Clone*. That's because class *B* inherits method *Clone* from class *A*, which inherits it in turn from *TObject*. The 58-byte jump is the sum of class *B*'s and class *A*'s ancestor links (32 plus 26)—the distance back to the original method. The compiler would set up the table to take this kind of inheritance into account. Again, refer to Figure 9-12.

For an overridden method such as *Free* in class *A*, a jump to class *A*'s own *Free* method code is inserted in the table instead of a relative jump back to the inherited version. New methods in a class get their own addresses to jump to, of course, as with *MethodA1* in class A.

How the table is used for real objects

Now suppose we have the following code in an application:

```
var
  anAObj: A;
  aBObj: B;
⋮
  New(aBObj);          { Assume that aBObj gets initialized }
  ⋮
  anAObj := aBObj;     { B instance assigned to A-type variable }
  ⋮
  anAObj.Free;         { Polymorphic method call }
```

We have a polymorphic assignment of a *B*-type object instance to an *A*-type variable. Later, there's a method call to free whatever object is in the *A*-type variable.

Because of polymorphism, the compiler can't tell what kind of object is in the *A*-type variable, so it must defer binding until runtime. To do so, it generates code that

- Traces the *anAObj* reference variable to the instance it contains
- Gets the instance's class link value (first two bytes)
- Computes the location of the class's part of the method dispatch table

- Adds an offset to reach the *Free* method's jump statement:
  (the method # – 1) * 6 bytes per jump statement + 2 bytes for ancestor link
- Pushes any parameters for the method onto the stack
- Pushes a return address onto the stack
- Jumps to the address of the method

At runtime, the generated code follows this trail, through the *B*-type object in the *A*-type variable to class *B*'s method table section and the jump instruction for *B*'s *Free* method. *B* has overridden *A.Free*, so the jump is to the actual code of *B.Free*. When the *Free* method finishes executing, the return address on the stack is used to return to where we left off—right after the *anAObj.Free* statement.

But suppose that *B* had not overridden *A*'s *Free* method but simply inherited it. Then the jump instruction in the table for *B.Free* would be a relative jump back to *A.Free*, and from there to *A*'s *Free* code. And, if *A* had not overridden *TObject.Free*, the jump for *B.Free* would be a longer jump, going back two classes to *TObject.Free*. The jump at *B.Free* would jump back by the sum of class *B*'s and class *A*'s ancestor link values in the table.

Thus the program automatically traces back up the hierarchy to find a real jump to method code.

What about *inherited*?

Suppose that, inside the method code for *B.Free*, we have a call to

```
inherited Free;
```

This means that we should invoke class *A*'s *Free* method as part of the operations of *B*'s *Free* method.

How would the method dispatch table be used to support this call? The compiler would generate code to trace where *self* points to and use that address to retrieve class *B*'s class link value. (We couldn't simply access *B*'s class link value from inside the object—it's not an instance variable.)

From there, we'd jump to the *B.Free* entry in the method dispatch table minus the value of class *B*'s ancestor link. In other words, when the compiler computed the location in the table, it would subtract the ancestor link value it found at the class link offset and jump to that new offset instead. Then it would jump to class *A*'s *Free* method code.

This approach handles all the requirements:

- Reaching inherited method code
- Implementing overrides with new addresses in the method dispatch table
- Linking each class to its immediate ancestor, if any
- Handling the *inherited* mechanism

The approach also minimizes what must be stored in each object instance—no jump instructions or addresses there, only one integer offset value (class link) per object. This approach should be fairly fast because, although we do trace through the hierarchy in the method dispatch table to find inherited methods, we don't have to search—the jump statements get us there automatically.

Can this mechanism be improved upon? Undoubtedly. It's intentionally as simple as possible, but a programmer developing a real compiler would want to find ways to optimize the method lookups and perhaps the table size.

The scheme we've looked at here simply lets us see how runtime binding might work. Compiler programmers are right not to document the mechanism, of course. It might have to be changed in future versions, potentially wreaking havoc on any application that assumes too much about the mechanism and tries to play tricks with it. It's safest to consider the structure of objects and the runtime binding mechanism as merely part of the object's encapsulated innards—best left alone.

Actually, compilers and linkers do quite a bit to optimize method calls. In two cases, THINK Pascal's linker, for example, generates a direct call to the method code rather than tracing through a dispatch table. It does a direct call if the method is "monomorphic"—that is, nonpolymorphic (neither overrides other methods nor is overridden itself). And it generates a direct call to the correct method when the *inherited* keyword is used. (See page 20 in Symantec's *THINK Pascal Object-Oriented Programming Manual.*)

Efficiency

Now let's look at the efficiency issues we might expect to come up if we use objects.

Code Size

When compiled, the object version of a simple program will usually be somewhat larger than a standard Pascal version designed with reusability in mind. An Object Pascal program will use Pascal units in addition to the main program, so we can expect the longer source code to result in a slightly larger executable file.

We should balance this expectation, though, with our expectation that large programs, using lots of objects, will show reduced code bulk because of inheritance. To the extent that a program uses classes subclassed from other classes, it should be able to save considerable code because the subclasses inherit much of what they need from their ancestor(s). In the long run, and for most programs, this kind of reusability through inheritance should tip the scales in favor of OOP, as attested by a number of sources, including Grady Booch's *Object-Oriented Design: With Applications* (Booch 1991). Besides, most good compilers, such as THINK Pascal's, now also have "smart" linkers that automatically strip out any code that is never actually referenced. This further reduces code bulk.

Speed

Object-oriented programs are reputed to run more slowly because method calls require a higher overhead. There's more to do than in straight procedure calling—especially moving through the hierarchy to locate inherited methods. Of course, there are also many cases, those in which polymorphism is not a factor, in which a smart compiler can compile method calls as straight procedure calls.

The relative speeds of OOP programs and standard programs is a pretty arguable area. For a more detailed and technical discussion, see Kurt Schmucker's *Object-Oriented Programming for the Macintosh,* page 11 (Schmucker 1986a) and Brad J. Cox's *Object Oriented Programming: An Evolutionary Approach,* page 115 (Cox 1986a)—Cox is writing about the Objective-C language. Cox points out one advantage of a "hybrid" OOP language like Object Pascal or Objective-C: When message sending slows a program too much, we can rewrite parts of the program in standard code, giving up some object-orientedness to gain execution speed. In the following section on optimizing, we'll look at that and at other ways to optimize Object Pascal programs when optimizing is called for.

Optimizing

When programs use too much memory, or run too slowly, we have to optimize them somehow to improve their efficiency. (Usually we can't improve size and speed at the same time; they tend to be mutually exclusive.)

We'll look at four levels of optimization:

- Being careful how we call the methods of *self*
- Reducing message sending by violating encapsulation
- Reducing message sending by rewriting objects in standard Pascal
- Improving speed by recoding selected program parts in assembly language

Calling *self*'s Methods

One very simple optimization we should always do will make our methods as fast as possible. Inside a method, we can access an object's instance variables either

- Indirectly by calling *self*'s methods:

```
radius := self.RadiusOf;
```

or

- Directly with record-style dot notation for fields:

```
radius := fRadius;  { Or radius := self.fRadius }
```

The second approach is acceptable OOP etiquette, in spite of the strong advisability of preserving an object's encapsulation by always accessing its instance variables indirectly, through methods, not directly. It's OK here because we're inside the object—inside its encapsulation.

Consider a method such as *TBouncingBall.Display*. One version of it might have lots of direct field accesses (see the items in italics):

```
procedure TBouncingBall.Display;
  var
    top, left, bottom, right, r: Integer;   { Rectangle for drawing }
    pos: Point;                             { Current location of ball }
begin
  pos := self.fCenter;                      { Get ball's current location }
  r := Round(self.fRadius);                 {  and its radius }

  { Get rectangle that bounds ball }
  top := self.fWheresTheGround - pos.v - r;
  left := pos.h - r;
  bottom := self.fWheresTheGround - pos.v + r;
  right := pos.h + r;

  { Paint ball inside that rectangle }
  PaintOval(top, left, bottom, right);
end;
```

We might also develop a version of *TBouncingBall.Display* that calls *self*'s methods to get and set field values instead of making direct field accesses (see the items in italics):

```
procedure TBouncingBall.Display;
  var
    top, left, bottom, right, r: Integer;   { Rectangle for drawing }
    pos: Point;                             { Current location of ball }
begin
  pos := self.CenterOf;                     { Get ball's current location }
  r := Round(self.RadiusOf);                {  and its radius }

  { Get rectangle that bounds ball }
  top := self.GroundLocOf - pos.v - r;
  left := pos.h - r;
  bottom := self.GroundLocOf - pos.v + r;
  right := pos.h + r;

  { Paint ball inside that rectangle }
  PaintOval(top, left, bottom, right);
end;
```

This version would slow down considerably from the overhead of lots of unnecessary method calls—unnecessary because, inside the object, we are free to access its instance variables directly.

Inside an object, it's always a good idea to use direct field accesses rather than method calls to *self* (when that's possible, of course).

The principle of encapsulation still makes it preferable to use method calls for other objects. When we're outside, we call methods. When we're inside, we don't. Users of a class are always outside. Class designers are the only ones who can get inside.

Violating Encapsulation (on Purpose)

Having established that principle, we nevertheless can use a second level of speed optimization for OOP programs by selectively forgetting about our guidelines on the encapsulation of objects.

Suppose that we're writing a main program that calls methods of an object called *theBall*, telling it to display itself, drop itself, report its location, and so on.

```
theBall.Display;
theBall.Drop;
where := theBall.LocationOf;
```

The first two calls will have to remain as they are. We can't display the ball by changing one of its instance variables, nor can we drop it that way. We have to send messages.

But we can speed up the third message a bit by letting ourselves break the "rules" of encapsulation. We aren't supposed to know what's inside *theBall* unless we call one of its methods to ask. But Pascal lets us get around that by simply accessing the instance variables inside the object directly.

```
where := theBall.fCenter;        { Get }
theBall.fRadius := 3.0;          { Set }
```

In some cases, this kind of deliberate rule-breaking for the sake of optimizing might gain us enough extra speed to be worth it.

Replacing Objects with Standard Pascal Code

Most of the time, we want to use objects because they offer the advantages of realism, information hiding, extensibility, and code reusability. But sometimes we have to give up those advantages, at least to some degree, for the sake of optimization.

Given the claim that OOP code is slower than standard code, we can save time in some cases by replacing some or all of the OOP code in a program with standard Pascal code. The fact, for example, that a class is an abstract data type (ADT) might be one source of its attraction for us. But object classes are not the only way to write ADTs in Pascal.

Instead of creating an abstract data type as an object type, we can do it with a standard Pascal record type. We still define the operations allowed on the type with procedures and functions, but they aren't methods. The type declaration and the procedures and functions are syntactically separate. We saw an example of this in Chapter 3.

Because we can't use inheritance, we might have to combine some of our instance variables and procedures from an ancestor class (or standard Pascal versions of them) with the standard Pascal record type declarations.

The standard Pascal version should run faster than its object-oriented counterpart. It would run even faster if we selectively replaced procedure and function calls with direct record field accesses—violating encapsulation of the ADT just as we did when it was expressed as an object class.

Of course, sometimes giving up OOP is giving up a lot.

Using Assembly Language

If our Pascal supports external assembly language modules or in-line code, as most do, and if we know enough assembly language, we can always isolate the slower parts of the program and recode them in assembly language. This is the traditional way to optimize slow code, and of course it works. We won't go into it here because the various Object Pascal compilers function differently in this regard. But we mention it for completeness. (Besides, Apple's assembler is object oriented.)

Optimizing Objects for Memory Usage

The basic observation for controlling how much memory our OOP code will use is that adding instance variables adds bytes to every single object of the class. On the other hand, adding methods is relatively cheap. Method code is stored in only one place, no matter how many objects call it. More methods do require more space in the method dispatch table, but at least the storage requirements don't explode as we instantiate more objects.

Sometimes an object must carry information that really pertains to all objects of the same class. If the information is that general, we can use constants to let the object "know" the information without using an instance variable. We used this technique in Chapter 6 to allow auto parts objects to report their "types" via methods. Because all hood ornaments are of the same basic kind ("hood ornament"), there's no need for every hood ornament instance to devote an integer or a string to its "type" information.

Another thing we can do to conserve space in objects is to delay allocating any subordinate storage they require. For example, if our objects contain string-type instance variables, we can declare the instance variables as handles to strings. Then we can have the object's methods allocate the actual strings only when they're needed—and deallocate them when they're not needed. In general, we can use handles to various kinds of data to good advantage this way.

Objects in Libraries

A virtue of OOP, as we've seen, is that if a class doesn't quite do what we need, we can write a subclass of it that does. By adding instance variables and methods and overriding other methods, we can create our own version of the class, which still gains a lot of leverage by inheriting from the original class. Moreover, we can create our subclass without recourse to the source code for the original class.

Provided we understand the interface of the superclass, we can write our subclass even if we don't have the source code for the superclass. The superclass might already have been compiled into a library file—but we can still subclass it.

What's a Library?

A library, like a unit, contains procedures and functions, but it consists of already-compiled code. And a library can be made up from more than one unit.

To create a library, we make a project containing the units we want. To avoid linker problems with multiply defined symbols in THINK Pascal, we need to remove the Runtime.lib and Interface.lib files from the project. A library project must not contain a main program—that is, the keyword *program* can't appear in a library project. In THINK, we use the Build Library command in the Project menu to create the library.

All Macintosh Pascals provide for the creation of libraries, although the process is different in the THINK and MPW environments. See the documentation for your environment for details.

Using a Library in a Program

Once we've created and compiled a library, we can add its file to our program's project. Normally, we also add a special file called an interface file that provides declarations for the constants, types, variables, functions, and procedures in our library. Figure 9-13 diagrams the relationship between the interface file and a single-unit or multiple-unit library.

The interface file is itself a unit. Its interface part declares all the names we can access from the library in our program. Its implementation part doesn't provide code for procedures and functions. Instead, it lists the procedures and functions, with the keyword *external* after them:

```
function MyFunction(param: AType): AType;
external;
```

The *external* keyword tells the compiler that the function's code is already compiled and can be found elsewhere. The linker then locates the code in the library.

As an example, suppose that we've created a library called MyLibrary.lib. (The .lib extension is conventional in THINK. MPW uses .o files.) Our interface to the library

is in the unit *MyLibeIntf*, in file MyLibeIntf.p. Let's say that the library contains only one function—the one we looked at above. In the interface part of the interface file, we declare

```
function MyFunction(param: AType): AType;
```

and in the implementation part we repeat that declaration line followed by the keyword *external*, as above.

In the project, we add the library file, MyLibrary.lib, and then add the interface file, MyLibeIntf.p. (Order doesn't matter.) Then the program can access the function provided by the library.

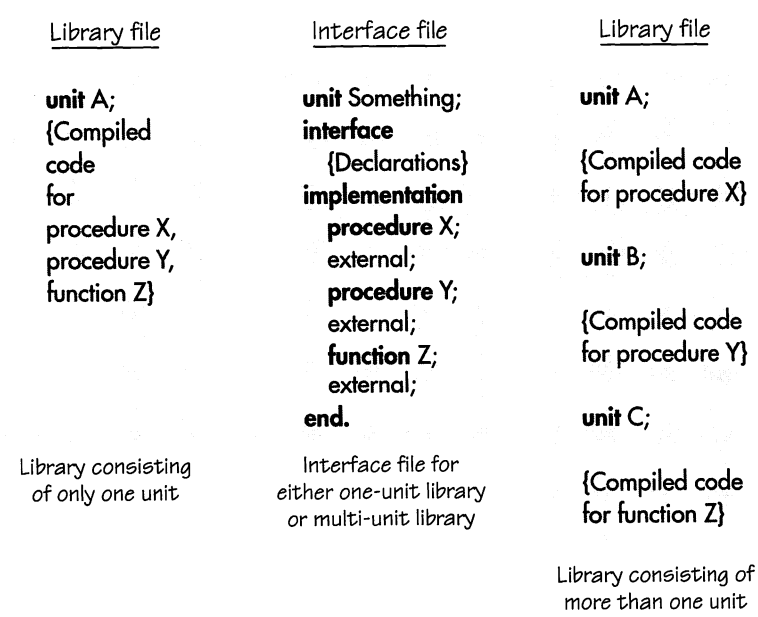


Figure 9-13.
Code libraries and interface file.

Problems with Objects in Libraries

In MPW Pascal and TML Pascal, object classes work in libraries just as other types do. You'll run into no special problems.

But, as of version 3.0, THINK Pascal still uses a method-dispatching mechanism that can cause some difficulties with libraries. (In THINK Pascal 2.0, you couldn't use objects in libraries very well at all.)

The best way to explain the THINK problem is to look at a hypothetical library that uses class *TObject*. Actually, we can compile such a library, but when we try to use it in a program that also needs *TObject*, the linker complains of multiply defined symbols.

Here's the situation:

In our library, we declare some classes whose immediate ancestor is class *TObject*, so, to compile the library, we have to add ObjIntf.p, the file defining *TObject*, to the project.

Then, in our program, we add the compiled library file and its interface file. Of course, we also have to add ObjIntf.p to the project if other classes in the program need *TObject*. And there's the rub. *TObject* has already been compiled into the library—but it's not declared in the library's interface because it was already declared in ObjIntf.p. So we can't simply access *TObject* from the library for use in other classes declared in the program. (Besides, what if we had two libraries, both of which contained code for *TObject*?) When we add both the library and ObjIntf.p to the project, the linker complains.

And not only *TObject*

Because so many classes declare *TObject* as their ancestor, it's the class with which we'll have the most trouble in relation to THINK's problem. But the problem applies equally to any class that must be used to support building a library of other classes and that must also be added to a program project that uses the library so that the program can declare subclasses of it. One example is class *TErrorHandler*, which we looked at in Chapter 8.

Working around

THINK might fix the problem in a future version, but in the current version of THINK Pascal we still have to work around the difficulty, using THINK's *$J* compiler switch.

Essentially, we can put {*$J+*} and {*$J–*} compiler directives around a set of declarations to prevent THINK from generating the *selprocs* and *methprocs* that THINK uses for method dispatching. This prevents multiple-definition problems with the linker later, when we try to incorporate our library into a program.

We can use *$J* either in the library or in the interface file for the library. Using *$J* in the library means that the library expects the interface file to supply the *selprocs* and *methprocs*. Using *$J* in the interface file means that the library is expected to supply the *selprocs* and *methprocs*. We can design our code to work either way. We'll usually require the *selprocs* and *methprocs* in the interface file, not the library. This makes sense with objects because we can't predict what programs the library will be used with and which programs will need the same classes for their own purposes. Nor can we guarantee that some other library won't have provided the same information. The scheme we consider next will ensure that our libraries provide only what they're supposed to provide and that they rely on our main program to supply the code for elements like *TObject*.

The workaround

Let's look schematically at a technique that keeps an object class used to support a library—such as *TObject*—from fouling up the works.

The idea is to use conditional compilation directives to tell the compiler which chunks of code to compile under which circumstances. We'll compile *ObjIntf*—or units containing other problem classes—one way when building our library and then compile it differently when it's recompiled as an interface file that's part of a program.

First, we'll need a compiler variable to control conditional compilation:

IsLibrary is set to *true* if we're compiling a library and to *false* otherwise. To set this switch for a given compilation, we can either put a compiler directive like

```
{$SETC isLibrary = true}
```

in our source code or, better, in THINK, choose the Compiler Options command from the Project menu and type *isLibrary = true* into the dialog box. We'll have to change the value of the switch from time to time as we compile various projects.

When compiling a library, we ensure that *TObject*'s declaration is bracketed by the directives {*$J+*} and {*$J–*}. If we're compiling *TObject* into a program instead of a library, we compile without the *$J* switches. Listing 9-1 shows what our specially modified ObjIntf file looks like for this technique.

```
unit ObjIntf;
interface

{$IFC UNDEFINED isLibrary}
{$SETC isLibrary := false}
{$ENDC}

{$IFC ISLIBRARY}          { ← Which condition is in force? }
{$J+}                     { ← If isLibrary is true, then use $J }
                          {   and compile the next section of code }
  type
    Str30 = string[30];   { Added by Chuck Sphar, June 1990 }
  type
    TObject = object
        function ShallowClone: TObject;
        function Clone: TObject;
        procedure ShallowFree;
        procedure Free;
```

Listing 9-1. *(continued)*
Specially modified ObjIntf file.

Listing 9-1. *continued*

```
        { Method added by Chuck Sphar, June 1990 }
        function TypeOf: Str30;
      end;
{$J-}                  { ← End $J around declarations }
{$ELSEC}               { ← Else compile the next section instead }
                       {  without $J directives }
  type
    Str30 = string[30]; { Added by Chuck Sphar, June 1990 }
  type
    TObject = object
        function ShallowClone: TObject;
        function Clone: TObject;
        procedure ShallowFree;
        procedure Free;

        { Method added by Chuck Sphar, June 1990 }
        function TypeOf: Str30;
      end;
{$ENDC}                { ← End of conditional compilation }

implementation         { ← Now for the implementation part }
{$IFC ISLIBRARY }      { ← Start new conditional compilation }
                       { ← If isLibrary is true, then set up for the linker }
                       {  to find the code externally }
  function TObject.ShallowClone: TObject;
  external;            { ← Note the external keyword }

  function TObject.Clone: TObject;
  external;

  procedure TObject.ShallowFree;
  external;

  procedure TObject.Free;
  external;

  function TObject.TypeOf: Str30;
  external;
{$ELSEC}               { ← But if isLibrary is false, then compile the following }
                       {  instead to create an interface file for the library }
  function TObject.ShallowClone;
  begin
                       { Actual method code omitted here to save space }
  end;
```

(continued)

Listing 9-1. *continued*

```
function TObject.Clone;
begin
  Clone := self.ShallowClone;
end;

procedure TObject.ShallowFree;
begin
  DisposHandle(Handle(self));
end;

procedure TObject.Free;
 begin
  self.ShallowFree;
end;

function TObject.TypeOf: Str30;
begin
  TypeOf := 'TObject';
end;

{$ENDC}                    { ← End of conditional compilation }

end.  { ← End of unit }
```

This code allows us to generate two different kinds of compiled code:

- A library
- An interface file for a library

Using this arrangement

A library contains two kinds of code: code for the procedures and functions that the library will provide to its users and supporting code such as *TObject*'s that the library needs in order to compile. For example, a library supplying shape objects for a drawing program contains the shape classes and their methods. *TObject* must also be present when we compile the library because the shape classes descend from it.

When we compile the library, we have to supply its needs by including ObjIntf.p in the project—but we want to do that without actually compiling *TObject*'s method code into the library. We'll supply it later, from outside, because the program using the library might need to declare *TObject* for its own needs.

Our scheme is to set up both *ObjIntf* and any other problem units composing the actual library using conditional compilation so that they are compiled for the library. Then we can change our compiler switch and compile *ObjIntf* and the other units in a different way for use in a main program.

When we compile *ObjIntf* with *isLibrary = true*, the library's compilation needs will be satisfied. Then, in the main program, when we compile *ObjIntf* with *isLibrary = false*, *TObject*'s method code will be compiled. Even though *ObjIntf* has been compiled twice (or perhaps more times—other libraries could use it, too), the linker finds the code only once. Everybody's happy.

Setting up the library's conditional compilation

We've seen how to set up *ObjIntf* for conditional compilation. Classes that present the same problem as *TObject* would be handled in the same way. For example, when we build a library for our linked list component in Part 3, the unit *UErrorHandler* must be set up in the same way as *ObjIntf*—because both the library and the project must include the unit.

But how do we set up the library itself? It has to be conditionally compiled, too. When we use it in a main program, we'll include both the compiled library (a .lib file) and its interface file in our project. We need to compile the library code one way to produce the library itself and another way to produce an interface file.

The setup is similar to, but not identical to, the setup for *ObjIntf*. We'll consider it schematically, for some hypothetical library:

```
unit X;
interface
  uses
    ObjIntf, ...;

{$IFC ISLIBRARY}

  { Class declarations, etc. }

{$ELSEC}
{$J+}

  { Same declarations }

{$J-}
{$ENDC}

implementation

{$IFC ISLIBRARY}

  { Method bodies--complete }
  { This reverses what we did for ObjIntf }

{$ELSEC}

  { Method headers with external directives }
  { This reverses what we did for ObjIntf }

{$ENDC}
end.  { X }
```

NOTE: *Write your compiler directives the way they're written in our code here—don't put an extra space in front of the dollar sign.*

The setup we've just looked at is for library elements that aren't like *TObject*. The element will be declared and implemented only in the library. In the interface file that goes with the library, we don't want to generate code for the library classes and their methods, so we use the *$J* switch in the interface, but not in the library.

Libraries with more than one unit

If the library consists of more than one unit, we'll need to set each unit up identically. However, because a multi-unit library has a one-unit interface file, we won't be able to use conditional compilation the way we have here. In the units that we compile for the library, we must give the full object class declarations with no *$J* switches, and the full method bodies with no external directives.

Then, when we prepare an interface file for the library, we have to collect all the public declarations (of the classes and other types that we want to export) from all the units used into our interface file's interface section. We put a {*$J+*} directive at the top of the declarations in the interface part and a {*$J–*} directive at the bottom of the declarations. Likewise, we gather all the (public) method, procedure, and function headers from all units into the implementation section of our interface file and label each with the *external* keyword.

The exception to this is any unit, such as *ObjIntf*, that needs to be included both in the library and in the using project. We must set such units up as we did *ObjIntf*.

Summary

If this chapter has accomplished its intentions, it has given you a deeper view "inside" Object Pascal and will serve as a reference.

We've taken a detailed look at the full syntactic extensions that Object Pascal brings to standard Pascal. The railroad diagrams in the first section of the chapter, along with the many examples of code in the book, will help you write correct Object Pascal code.

We've gone inside to see how objects are (hypothetically) implemented by the Pascal compiler. This discussion stayed general because different compilers might do things differently and the same compiler might change its strategy from one version to the next.

We've briefly discussed several efficiency issues that come up for Object Pascal code, along with some approaches to optimizing code.

And we've dealt with a problem THINK Pascal has with using objects in compiled libraries.

PART

2

OBJECT-ORIENTED APPLICATIONS

In Part 2, we'll look at techniques for using object classes in the design of thoroughly object-oriented programs. We'll focus on Crapgame, a simple example program written almost completely with objects, and on PicoApp, a tiny application framework modeled loosely on Apple's MacApp and Symantec's THINK Class Library. We're going to go inside a simple application framework to see how objects are used to build the framework and how OOP techniques are used to extend the framework to create real applications.

After you complete this part of the book, you'll be able to move on to one of the commercial application frameworks, either Apple's industrial-strength MacApp or THINK's somewhat smaller and somewhat less powerful TCL. Kurt Schmucker's *Object-Oriented Programming for the Macintosh* (Schmucker 1986a) and David A.

Wilson, Larry S. Rosenstein, and Dan Shafer's *Programming with MacApp* (Wilson 1990) focus on using the MacApp application framework. Our focus will be on building one in the first place. We'll investigate what goes on under the hood by developing a relatively simple model. PicoApp doesn't do everything in the same way that MacApp or TCL does, but its general architecture and mechanisms are close enough to MacApp's and TCL's that your transition to MacApp or TCL will be considerably easier if you choose to follow up.

In Chapters 10 through 17, we'll go through all the stages of developing a medium-sized, complete, and completely OOP program. In Chapter 10, we'll look at Crapgame and its high-level design—in particular, at how we decide which objects to use. In Chapter 11, we'll first derive PicoApp from Crapgame because PicoApp represents what's general about Macintosh applications and Crapgame represents what's specific. After we've developed the key concepts and structure of PicoApp, we'll go back and derive Crapgame from PicoApp in the same way that a MacApp programmer builds a working application on top of the MacApp foundation. In Chapter 12, we'll flesh out PicoApp by going inside its major object classes: *TPicoApp*, *TPicoDoc*, *TWindow*, and *TPicoView*. Then, in Chapter 13, we'll see how PicoApp interacts with the user, relaying user-generated events to our application's views and controls. In Chapter 14, we'll look at memory management and other application-level topics. In Chapter 15, we'll learn how to develop the views an application shows to the user and how to display drawing in those views. To demonstrate the drawing, we'll develop PicoSketch, a second small "PicoApplication." In Chapter 16, we'll go through the steps of using PicoApp.

Finally, in Chapter 17, we'll round out Crapgame by developing the rest of its object classes within the PicoApp framework.

10

DESIGNING OOP APPLICATIONS

In this chapter, we'll take up OOP application design techniques as we design a simple game program called Crapgame. We'll look at

- OOP design vs. traditional design
- OOP characteristics that affect application design
- An informal OOP application design strategy
- The reuse of object classes
- The design of new classes with reusability in mind
- Alternative design approaches
- PicoApp, a tiny application framework

Not only will we do the high-level design for our complete OOP application, but later we'll also build the application on top of PicoApp, a small application framework. Although it is far simpler and less capable than either, PicoApp is generally modeled after Apple's MacApp and Symantec's THINK Class Library.

What's the point? After we've abstracted the PicoApp classes from our Crapgame program, they'll be available for reuse in other simple applications. PicoApp will become the basis for new applications, and we'll look at how it might be extended and improved, too.

Just for fun, let's take an early look at the final main program we'll develop for our Crapgame program over the course of the next several chapters (Listing 10-1). The main program is extraordinarily short—it leaves all the work to objects.

```
{$S Main}
{$I-}
{ This directive requires that we initialize Toolbox managers ourselves, }
{  for greater portability to MPW; see call to InitTheMac, below }

program CrapGame;          { File is CrapgameMain.p }

 uses
   ObjIntf, UPAGlobals, UMyGlobals, UPARoutines, UPAIntf, UButton,
   UHyperButtons, URandomStream, UDice, UPlayer, UGame, UPlayerList,
   UCrapsViews, UCrapgame, UCrapsApp;

 const
   kMasterPtrBlocks = 8;      { Call MoreMasters 8 times}
                              { Accept default reserve sizes }
 var
   app: TCrapsApp;
begin
  { Install a memory-manager object in gMemory; also install an }
  {  error-handler object }
  { If installation is successful, create and run the application }

 InitTheMac;                  { Defined in UPARoutines }
 if NewMemoryObject(kTempReserveSize, kEmergencyReserveSize,
     kMasterPtrBlocks) then
   begin
     app := NewCrapsApp;      { Create the application }
     app.Run;                 { Run it }
   end;
end.  { CrapGame }
```

Listing 10-1.
Crapgame's tiny main program.

The display screen we'll create for Crapgame is shown in Figure 10-1.

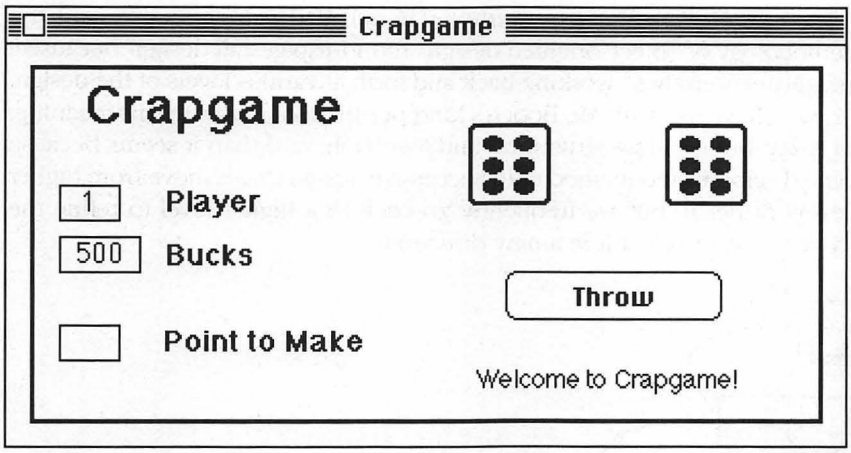


Figure 10-1.
The Crapgame playing screen.

OOP Design vs. Traditional Design

OOP application design differs from traditional procedural application design. A number of peculiarly OOP factors influence how we do it.

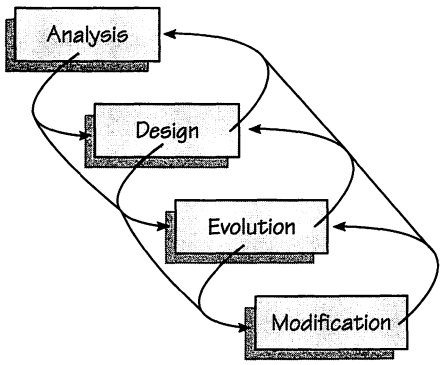
Inheritance and Reusability

OOP gives us lots of leverage through inheritance and class reusability. Given a good class library, we can often produce at least some of the classes we need simply by reusing or perhaps subclassing library classes. In Crapgame, we'll of course reuse the *TButton* class we created in Chapter 4. And we'll use the *TRandom* class from Chapter 6 as the basis for a dice class.

Object Modeling and Design

Objects are capsules containing both data and methods, so they're ideal vehicles for modeling the real-world objects in our problem domain. Object modeling affects the whole direction of the application design—we'll work from both the bottom up and the traditional top down at the same time. For Crapgame, we'll first decide what objects we need and then decide what instance variables and methods they require. And we'll plan the mechanisms by which the objects will interact to do their jobs, a task that goes more or less from the top down. Then, working from the bottom up, we'll develop one object at a time, often in isolation from the others, test the objects individually using a simple driver program, and integrate the objects into the final application. We'll use an informal methodology for this design process and consider other, more formal methodologies.

In *Object-Oriented Design: With Applications* (Booch 1991), Grady Booch calls the general methodology of object-oriented design "round-trip gestalt design" because it is "incremental and iterative," working back and forth at various levels of the design. Figure 10-2, which we use with Mr. Booch's kind permission, illustrates his meaning. OOD feels fuzzy, but it's more structured and less haphazard than it seems because there is some discipline and method to it. Successive design stages move from higher to lower levels of detail, but we frequently go back to a higher level to refine the design in a new way or orient it in a new direction.



Recreated, by permission, from Grady Booch, *Object-Oriented Design: With Applications*

Figure 10-2.
Object-oriented design (OOD)—"round-trip gestalt design" that is "incremental and iterative."

Generalization

Reusability cuts two ways. Even as we reuse objects developed earlier, we'll look for eligible new parts of the application that we can make general enough to be reusable in future applications. In particular, while writing Crapgame we'll develop PicoApp, our simple application framework. Just as we design specific objects to represent dice, players, and so forth in the game, we'll design reusable object classes to represent various features of the Macintosh user interface. Figure 10-3 shows generic and programmer-supplied aspects of application development.

PicoApp consists of a reusable "application object" that knows how to do the standard Macintosh interface things (menus, windows, mouse handling); a reusable "document object" that can be the basis for document subclasses in other applications; a reusable "window object" that performs all the normal window functions; and a reusable "view object" that represents the functionality of some part of the content region of a window in a general way.

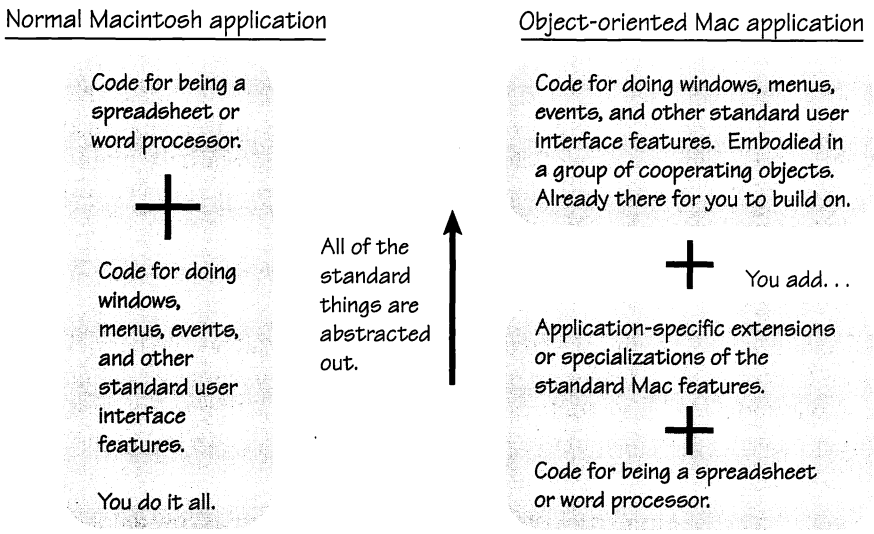


Figure 10-3.
Traditional application design and coding and object-oriented design and coding.

We'll use the same process of generalization to develop several other potentially reusable object classes in addition to the PicoApp classes, including a reusable "player list object" that can manage an open-ended list of player objects, helping to manage turn-taking in all sorts of games or other applications that require turn-taking. We'll see how these object classes work and where they come from later in Part 2.

Object-Oriented Requirements Analysis

The first phase of any application design, whether formal or informal, is an analysis of the users' requirements.

We can view developing an application as a process of translating from the real world of the problem domain into the more abstract world of software, the solution domain. Interestingly, because the analyst is looking directly at the problem domain in the real world, what he or she sees is a collection of interacting real-world objects. As it turns out, requirements analysis is basically object-oriented already and fits neatly into the rest of object-oriented design.

What we see in the problem domain for Crapgame looks something like this: Several players (maybe in an alley, maybe in Caesar's Palace) huddle around a surface on which, one at a time, they roll a pair of dice. They place bets on the outcome of the shooter's throws. When the current shooter loses, he or she passes the dice to the next shooter. A player who runs out of money is out of the game.

We require a translation of that scene into a software form that lets one or more players roll dice according to the rules of craps.

Looking at it informally, we can see some obvious candidates for objecthood—things we'll want to model directly as objects in Crapgame: players and dice. But what other objects might we need? Before we consider that question we might need a slightly more formal methodology for picking objects out of the problem domain and developing them as object classes.

About Craps

Craps, of course, is a dice game played in casinos and in back alleys and barracks the world over. Its structure and rules are fairly simple in general, although casino play uses a complicated table layout for bets because most of the action is in side betting by the other players rather than in the shooter's rolling the dice. We'll skip the fancy betting layout, and we'll use Hoyle for the rules.

Any number of players can participate. We'll limit our game to as few as one or as many as five. One player, the shooter, rolls the dice, and he or she and the others bet on the outcome. A shooter who loses ("craps out") passes the dice to the next shooter. We'll implement betting in a very simplified way, keeping track of players' "scores" by adding or subtracting increments of $50 or $100, depending on how the dice go. (The code in unit *UCrapsPlayer* on the code disk works out the details.)

The shooter's main aim is to establish a "point" with the dice and then match that point on a subsequent roll. The shooter rolls repeatedly until he or she either "makes" the point or rolls a 7 ("craps"). A shooter who makes his or her point rolls again to set a new one.

Before a point is set, a shooter can roll a "natural"—a 7 or an 11. A shooter who rolls a 7 or an 11 on his or her first roll, for instance, wins any current bets. If the shooter then rolls another 7 or another 11, he or she wins again, and so on. When the shooter finally sets a point, he or she must make the point before a natural is possible again.

The shooter can crap out before setting a point by rolling a 2, a 3, or a 12 before setting his or her point. The shooter loses any current bets and gives up the dice to the next shooter.

What We Require

Crapgame is a simple enough program, so our requirements analysis can be short. Given the nature of craps, let's try to lay out the overall form and nature of our application in terms of what we want to accomplish with it.

Our Crapgame must provide for players who take turns shooting the dice. Two dice are used. No score is kept except for the players' winnings and losses. To keep it simple, we won't implement any actual betting in this version.

We have to account for turn-taking, dice-rolling, and quitting, as well as for letting the players know the outcome of each roll. We intend the game to be played by as few as one and as many as five human users. This recommends a display that shows whose turn it is, what the dice show, what the current point-to-make is, and so on. And we need some way for the human players to throw the dice. This suggests an input method such as a mouse-clickable button, a menu, or a keystroke.

We'll also make it possible for players to play game after game without the application's terminating—a game ends when everybody but one player has no more money and that player craps out—and to set various options, such as the number of players for a game and the "stakes" with which each player starts the game.

Some Initial Decisions

A program's design can be influenced by the hardware and software environment it will run in and by other environmental factors involving the users and their requirements. In this section we'll make some tentative decisions based on where and how we plan to develop the Crapgame application.

We're implementing on the Macintosh, so our display will be a Macintosh window, and our input device a button that can be clicked with the mouse.

Our dice, of course, have to roll random numbers from 1 through 6. Computers can't generate truly random numbers, so we'll use pseudorandom numbers. A sequence of pseudorandom numbers appears random but will actually repeat at some point. We'll periodically "randomize" the random number generator behind our dice by "reseeding" it from the system clock. We'll go into this in more detail in Chapter 17.

We have lots of options for displaying the results of the shooter's dice rolls. We could simply print the number for each die on the screen. Or we could be more graphic, displaying actual pictures of dice. Our solution will be to use Macintosh icons.

We'll have menu items for starting a new game with the old, or default, options in force, starting a new game with new options, and quitting the application.

We'll use a "Throw" button for rolling the dice. We already have a button class, so we'll simply subclass to fill in its *DoClick* method.

An Informal OOD Methodology

Over the years, a number of object-oriented design approaches have been proposed, some of them informal, others highly structured, technical, and formal indeed. After requirements analysis, the key phases in any of these OOD methodologies are (a) identifying the objects to develop, (b) designing the objects, including their instance variables and methods and the mechanisms by which they interact and communicate, (c) using inheritance and other reuse strategies to gain leverage, (d) testing the objects, and (e) integrating the objects and testing the integrated application.

As Grady Booch suggests in *Object-Oriented Design: With Applications,* though, the designer often works by developing simplified, sketchy versions of some classes, returning to them later to refine, perhaps time and time again. As the designer's understanding of the problem, of the ramifications of his or her solution, and of the nature of his or her objects deepens, he or she keeps reworking the classes, tuning them. (Take another look at Figure 10-2's illustration of this process on page 232.) I can testify that my experience developing the many classes in this book fits Booch's model, particularly as I designed groups of cooperating object classes.

We'll take a simple, informal OOD approach in this chapter. We begin with the question: What objects do we need for Crapgame?

Data Objects

The most important objects in our (or any) program will be those derived from the data. A personnel program, for example, would begin from the personnel data. An airline reservation system would start with data about airplanes, airports, cities, departure times, and so forth. Many of these data items could be objects.

In Crapgame we might find it hard to identify such data items because the program won't manipulate information to any important degree. The values rolled with our dice probably come closest, and, as it turns out, that data is exactly what drives the program. Dice are rolled, the numbers on their faces are read, and decisions are made accordingly.

Does that mean we want to use a "number" object or a "face" object? Not likely—that's too myopic a view. A rolled die, provided it's on a flat surface, naturally lands with one side up and another down, and of course the upper face is the most important to us. But we see the entire die, not only one face, and we can roll the die only as a whole—we can't roll a face. This is a characteristic of a die, the first real-world object for Crapgame we've noted so far. We'll start the design with die objects.

A die object knows how to roll itself and report the number on its upper face. It should also know how to display itself—ideally in a graphic way.

Other Objects

What other objects are we going to need besides dice?

In this simple example, we could simply analyze by brute force and trust to intuition. But, instead, let's try out an informal design technique that's a combination of the methods suggested by R. J. Abbott's "Program Design by Informal English Descriptions" (Abbott 1983), William Lorenson's "Object-Oriented Design" (Lorenson 1986), and Grady Booch's OOD book (Booch 1991).

Design Method Overview

In an OOP system such as Object Pascal, the basic design steps, once we've done our preliminary requirements analysis, are as follows (mostly from Lorenson):

1. Identify the objects (classes).
2. Identify each object's attributes (data fields, or instance variables).
3. Identify each object's operations (methods).
4. Begin to develop the mechanisms by which the objects work.
5. Identify the communication between objects (messages).
6. Test the design with "scenarios."
7. Apply inheritance where it makes sense to.
8. Integrate the objects and test the integration.

These steps tend to loop back again and again as the design process continues.

We've already made some efforts, in our requirements analysis, to identify the Crapgame objects. So far, we have identified the dice. To identify the other objects we need, we'll borrow a technique from Abbott, with a bit of an assist from Booch. The idea is to carefully specify the problem we need to solve, then develop a one-paragraph informal strategy for the solution (in English), and then mark the nouns in the paragraph. The nouns (and possibly the noun phrases) should suggest objects. We'll work with the paragraph again to identify other elements of our program.

1. The Objects (Classes)

To identify the Crapgame objects, we first define the problem, following Booch. For Crapgame, this consists of the kind of requirements analysis we did above, which might end with a statement of what the program does:

> Players take turns rolling dice according to the rules of craps.

Let's express that in the form of a problem to be solved by our program:

> Develop a program that lets players take turns rolling a pair of dice according to the rules of craps.

The next step is to come up with an informal strategy for solving the problem. In our case, that means describing our crapgame in noncomputer terms:

> Crapgame lets one through five players repeatedly take turns rolling a pair of dice. [Rules: A turn ends when the "shooter" "craps out" by (a) rolling a 2, a 3, or a 12 initially (before having set a point) or (b) rolling a 7 before "making his or her point." A player wins—and continues to roll—if he or she (a) rolls a 7 or an 11 initially or (b) establishes a "point" that he or she must then "make" and then makes it by rolling the same number again.] Players use a mouse to click buttons to roll the dice and to select commands from menus to start new games, set new options, and quit the application. Results of player rolls, useful messages, and information such as the current player and the point he or she must make are displayed on the screen.

This informal statement of our strategy takes into account the constraints we're under—such as using a mouse for input. To avoid confusion, we use, as much as possible, the same terms to describe the same things rather than synonyms.

Now let's take the same passage—we can skip the rules—and mark its nouns and noun phrases:

> Crapgame lets one through five *players* repeatedly take *turns* rolling a pair of *dice*....*Players* use a *mouse* to click *buttons* to roll the *dice* and to select commands from *menus* to start new *games,* set new *options,* and quit the *application. Results* of player *rolls,* useful *messages,* and *information* such as the current *player* and the *point* he or she must make are displayed on the *screen.*

Let's do a little listing and grouping:

players, player (real-world object)
turns, turn
dice, die (real-world object)
mouse (real-world object)
buttons, button (real-world object)
menus, menu
games, game
options, option
application
results, result
rolls, roll
messages, message
information
point
screen (real-world object)

Some of these nouns really are objects. Some are actions or attributes in disguise—the English language is pretty slippery. Some of the nouns are abstract—compared to the real-world objects we've marked. And some of the nouns might well be objects, even real-world objects, but they are part of the problem, really, not part of the solution.

What's What

We know we need *die* objects.

We probably also want *player* objects. Players know what the current point is, know the rules of craps, know how to roll dice, and bear a useful but abstract resemblance to the human players who will use the game when it's finished.

We can rule out the mouse. It's merely a way for the human players to click or press buttons on the screen. It's not really part of the game. As one of the constraints we acquire by working on the Mac, the mouse is more a part of the "problem space" than a part of the "solution space." All we need are the objects that we have to put into software to make our game work. Extraneous objects need not apply.

We will use a *button,* though, and, as we've already seen in Chapter 4, it's a good idea to make it an object. Our button will be drawn on the screen by software, and, also through software, will respond to mouse clicks just as real-world buttons would respond to finger presses. We'll let the human players use the button to control the flow of the game.

What about menus? It's certainly possible to create menu objects—see David B. Curtis's "Menus as Objects in TML Pascal" (Curtis 1988) and the THINK Class Library—but we won't use menu objects here. As is usual in Mac applications, the application will handle menus.

The *game* itself and the *application* itself, oddly enough, will be objects. Apple's MacApp uses an application object that manages the event loop, and so will we. And to implement the player's ability to play game after game without restarting the program, the application object will destroy the game object of a completed game and allocate a new game object in its place. A game object knows how to keep track of players, how to set up its display (by creating more objects), and how to manage a game until its end.

The options are the number of players for a given game (1–5) and the amount of money each player starts a game with. The players can see these options by means of a dialog box; the options needn't be objects.

Although the screen is in somewhat the same category as the mouse—an external side effect of the fact that we're using a computer—we should remember that we want a highly modular program here. Creating one or more "display objects" to coordinate all the screen-displaying in the program is a good idea. We'll call the display objects *views*—another idea from MacApp. And the views, of course, are displayed in a *window.*

Many of the rest of our nouns are really disguised verbs or adjectives. "Turns," for example, are actions taken by the players. In fact, we'll probably need a "take-a-turn" method in the player object class. "Results" are simply the numbers produced by the dice, so they look like a poor candidate for objecthood. "Rolls," likewise, are really actions associated with the dice. "Information" refers to the current point and player number, and to the messages we want to display to let human players know what's going on. These sound like parts of the view objects rather than objects in their own right. "Point" is another number—the value established by rolling and displayed for reference. None of these nouns from our list sound like good candidates for objecthood.

As far as our initial analysis goes, then, these are the objects we'll develop for Crapgame:

dice
players
buttons
views
window
game
application

Where the names are plural, it's really the singular form we're interested in. We could, for instance, create a "dice" object class, but we'd probably find it a bit cumbersome and less general than it should be. Object classes such as "die" and "button" look like strong prospects for reusable objects, so we'll want to make them as general and reusable as we can.

2. The Objects' Attributes

Now that we've decided what objects to use, we need to figure out the data each needs to carry in its instance variables. It's common to refer to the instance variables in an object class as the object's attributes, or characteristics. Let's take up our objects one by one.

Die. A die object's attributes are the number of faces it has—some dice have more or fewer than six—and the number on the face that's currently up. A die should also be able to display itself, so it will need a rectangle defining where it does that.

Player. A player object will be identified by a number, so that's one of its attributes. A player object should also be responsible for displaying itself, probably by displaying its player number in a rectangle on the display. And a player will need fields for keeping its own score, storing the current point to match, and so on.

Button. We've already seen the attributes of *TButton* in Chapter 4.

View. A view object is responsible for displaying some part of the visible aspect of a program—what users will see on the screen (the program name, the die face values, the button, labels for information, any decorative embellishments, and so on). We'll take the approach that each displayable "object" on the screen—text string, player information, each die, the button—is represented by its own view object. The application will manage a collection of view objects to produce the whole display. The main attributes of a view object are the display rectangles allotted for whatever it has to draw: title, box, decoration, and so on. See Figure 10-1 earlier in this chapter for the results of the view objects' work.

Window. Each view object is associated with a particular window. The window object needs a list of its views.

Game. A game object is responsible for creating nearly all the other objects; therefore, the game object's instance variables are containers for the other objects—object references of the various types, such as *TButton* and *TDie.* The game object will have containers for from one through five player objects, two die objects, a button object, a window object, and one or more view objects.

Application. An application object is responsible for creating a new game object each time the user asks for one (by means of a menu), so a container for the current game object is one of the application object's instance variables. The application object probably also tracks the current option settings. And it manages such Macintosh user interface paraphernalia as menus, an event loop, window updates, an "about" dialog box, desk accessories, and so on, but most of those won't entail instance variables—the application object's methods will handle the user interface.

As we get further into the program's development, we might find that we need attributes that we haven't thought of yet. But these at least are a good, thoughtful start.

3. The Objects' Operations

Next, we decide what operations each object should be able to perform. This will give us the object's methods.

Where object classes are designed for maximum generality and reusability, as in our button and die object classes, we'll want methods to set and read the values of most of the objects' instance variables. We can't fully predict what future clients using the classes might need, so we give the classes as much flexibility as possible, including a complete interface for each as in an abstract data type (ADT). But for a custom-built, use-once object class such as our player objects, we can waive that requirement and create only the methods the object will actually use.

Finding the Operations

One aid to deciding what the objects' methods should be is to check the verbs in our informal solution strategy, just as we checked the nouns to find the objects themselves. Here's the strategy paragraph without rules again, this time with verbs (and verblike forms) marked:

> Crapgame *lets* one through five players repeatedly *take turns rolling* a pair of dice.... Players *use* a mouse to *click* buttons to *roll* the dice and to *select* commands from menus to *start* new games, *set* new options, and *quit* the application. Results of player rolls, useful messages, and information such as the current player and the point he or she must *make are displayed* on the screen.

Some of these verbs look pretty unhelpful ("lets," "use"), but "take turns" is something that players do. We'll want a "take-turn" method in the player class. A player also makes his or her point, although that is part of taking a turn. Clicking is something a player does to buttons—but maybe a more useful way to view that is as

buttons being clicked. In other words, a button object must be able to test whether it has been clicked. Sure enough, a "clicked" function-type method is one of *TButton*'s methods. Rolling is something that a player does to dice, but rolling is more usefully viewed as a property of dice rather than of players. We'll want a "roll" method in our dice class. A player quits the game and sets options by choosing from the menu. These actions will be taken care of as methods of the application object. Quitting is probably based on a flag, or signal, to the game object that its event loop should end. Options are set in a dialog box by the application object. Finally, displaying is primarily a function of the view objects, including the button, player, and dice objects; thus, all of those objects will need display methods.

At the least, then, we'll need the following methods for our objects:

Die

Initialize—number of faces, number initially showing.
Display.
Set and Get display rectangle, number of faces.
Roll.
Randomize—reseed the random number generator.

Player

Initialize—number, rectangle, score, and point to match.
Take turn ("Play")—this is where we put the rules of craps.
Display.

Window

DisplayViews.
Grow, Drag, Move, and so on.

Game

Initialize—create objects (leading to display and start of the game).
Various methods to manage the player list.
Clean up—dispose of objects and other allocated space.

Application

Initialize—create game object; let user choose options.
MainEventLoop—get events.
DispatchEvent—dispatch events to the right objects.
DoNew—set up a new game.
DoKeyCommand—handle command keys for menus.
Quit—handle quit choice from menu.
Display—call the game object in response to updates.
DoUpdate—handle updates, including calling Display.
DoActivate—handle window activations.
DoContent—handle a mouse-down in the window's content area.
Run—loop, creating new games, until user quits.

All of these objects must be able to call *Free* to dispose of themselves when they're no longer needed, so—directly or indirectly—all are descendants of *TObject*.

And several of the objects must be able to handle mouse clicks, so they must be descendants of *TEvtHandler*. The application object gets events in its *MainEventLoop* method and dispatches any it can't handle itself, such as menus. The dispatcher passes a non-menu mouse-down to the first object in the event-handler chain, which passes it to the button object so that it can see whether the click "belongs" to it. The objects in this event-handler chain must have *DoMouseCommand* methods to respond to clicks. We'll discuss this flow-of-control mechanism in the next chapter.

In order to specify what methods our view objects will need, we have to be more specific about what view objects we require. Looking at the display in Figure 10-1 on page 231, we can identify five distinct display items, some simple and some complex. We'll use one view object to draw the title, "Crapgame." The "Throw" button will be a view object because it has display capability. Likewise, each of the dice. Finally, each player object can display its "Player" number, its score ("Bucks"), and its current "Point to Make," as well as messages to the human players such as "Welcome to Crapgame," so the player objects will be view objects, too. We've already seen the methods for the player, die, and button objects. The view objects basically need initialization and display methods.

We realize, of course, that as we get into developing the individual object classes we might discover other methods we need, and we might yet need more objects as well.

4. Object Mechanisms

By this point in the design process, part of the mechanism of the game program is beginning to show through the details.

Mechanisms, according to Booch (1991), are the devices and arrangements we use to make groups of objects cooperate to get some task done.

Parts of our mechanism here include

- Communication from the button to the player
- Communication between the player and the dice
- Communication among various objects and the display
- Communication among various objects and the game

It appears, at this stage, that the game will function more or less this way:

The game object creates a set of player objects and die objects and a button object. These display themselves (one player at a time). When the shooter clicks the Throw button, the game object tells the current player object to play. The player object tells its dice to roll themselves and report their results, which the player object uses to determine its next action. Then the player object can do various things, depending on whether its throw crapped out, set a point, matched a point, and so on. If a player

object craps out or runs out of money, it tells the game to pass the dice to the next player object or to remove the current player object from the player list. When the user chooses a menu option, the application object tells the game object to free itself. As it frees itself, the game object tells all its subordinate objects to free themselves as well.

Figure 10-4 shows the flow of actions, from the Throw button to the player, to the dice, to the display. We'll see this mechanism get fleshed out as we proceed.

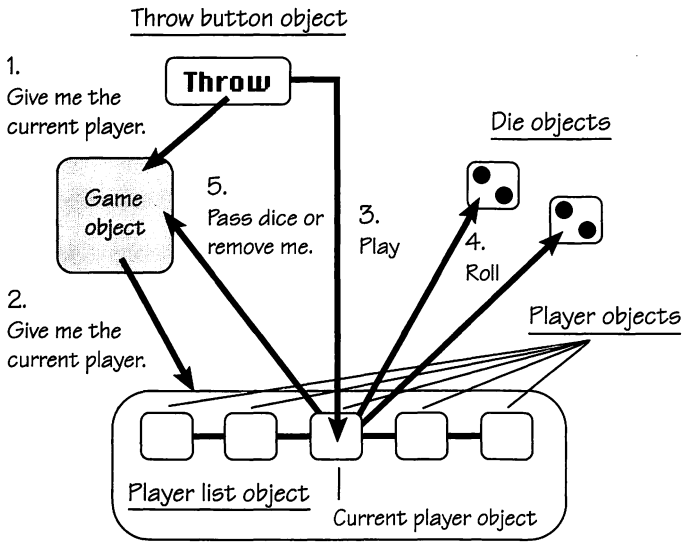


Figure 10-4.
The Crapgame mechanism.

5. Object Communication (Messages)

Our list of methods for each object already lets us know what the messages will be—messages are simply method names (with parameters).

But communication is still an issue in a multi-object program such as Crapgame, and we need to look at several interesting problems. We need to know which objects need to communicate with each other (send each other messages) and whether any of that communication is two way.

Let's briefly consider three kinds of control flow in Crapgame: the flow of events to event-handler objects, the flow of Display messages, and the flow of actions ensuing from a button click.

The application object will operate its event-handler loop to get events from the Mac. It dispatches these either to itself or to other objects. Our application object will handle many events itself, including mouse-downs in the menu bar, Command-key

menu selections, updates, activates, and MultiFinder suspend and resume events. It will pass other mouse-downs and key-downs along the event-handler chain. Sooner or later, they're passed to at least one view object—normally, to the Throw button. Thus, the application object communicates with a sequence of objects down the chain, many of them view objects.

When a game starts, the window must be created and its contents drawn. The window is created by the game object. When there's an update event, the contents of the window must be redrawn. The application passes updates to the current top window, which sends Display messages to all its view objects. Each view object draws itself in the designated rectangle in the window. Thus, again, the flow is from application to game to views.

When a user clicks the Throw button and the button processes the mouse-down event, determining that it has been clicked, a new flow of actions occurs, some of the actions two way. The button needs to know which player object is the current shooter, so it calls the game object to get the current player. Then the button calls that player to have the player object roll the dice and process the results. The player calls the dice, which produce pseudorandom numbers and display themselves. Then the player object sums the dice results and analyzes the outcome, responding accordingly. The player then displays results. If a player craps out, it must call the game object to signal passing the dice to the next player. And if a player runs out of money, it must call the game to have itself removed from the player list. Thus, the button must communicate with the game object and the current player object. The player object must communicate with the dice objects and the game object.

Because the game object creates both the button object and the player objects, which in turn must call methods of the game object, we get a two-way flow of communication and a resulting question of where to declare each object class so that it "knows" the right classes.

We reviewed the use of superclasses to solve problems of this sort in Chapter 8. We'll see in a later chapter how all of this gets implemented.

6. Object Class Testing

An object class is not really complete, of course, until it has been tested.

About Testing

One convenient aspect of object-oriented development as we are practicing it here is that classes are often independent enough to be tested all by themselves before we have to complicate matters by integrating them with other classes.

The main objective in testing a new class is to give every method in the class a good workout. And, if a method receives parameters or returns a function result, we need to try the method out with a range of parameters, both good ones and bad ones. The idea is to develop a number of scenarios to test all of the class's functioning.

The goal of a good tester is to break the software being tested. If we can make a class break by passing it bad parameters, by initializing it incorrectly, and so on, we can fix it. The result will be a better object class.

One consequence of the boom in OOP these days is going to be lots of libraries of reusable software components in the form of object classes. Such classes as our die and button classes, for instance, can be picked up and reused in completely different programs. Class libraries are already available for Object Pascal: TCL from Symantec, MacApp from Apple, and—at least for students and hobbyists—our own PicoApp. Many different programmers can borrow (or buy) an object class to use in their programs. Such classes must be certified for reliability. To be able to claim reliability, object class suppliers must do thorough testing. The alternative to testing, a difficult one, is to prove mathematically that a class is correct. We won't go into that subject in this book.

How to Test Objects

How do we go about testing a new object class? The technique is simple enough. We write a small "test driver" main program that

- Includes our class's unit in its *uses* clause
- Declares one or more variables of our class's type
- Creates and initializes the objects for the variables
- Sends all possible messages to the objects, including messages with a range of parameters
- Tries to break the object class by making its methods err

The main program might have to emulate the actions of any other objects with which the class we're testing might communicate. In some cases, we might need a dummy object to stand in for some of those not yet developed. The dummy would provide a minimal version of the as yet undeveloped objects' services.

A test scenario is a set of messages that fully "exercise" one of the object's methods or a group of its related methods. The scenario tries out not only good data but bad data as well, such as boundary cases (where the method should respond one way on one side of the boundary and a completely different way on the other side), data known to be unacceptable to the method, faulty initialization, and so on.

As we use our test driver, we will almost inevitably encounter bugs. As we do, we fix them and then run the driver again. Eventually, we run out of things to try.

A whole software-engineering literature has grown up around software testing, of course, and Roger S. Pressman's *Software Engineering: A Practitioner's Approach* (Pressman 1987) is a good place to start.

Using Stubs

In some cases, of course, we have to test more than one class at a time. If two classes are declared in the same unit, we might need to test them together.

In such cases, we might have to finish both classes at least partly before we can start testing. One technique that can help, though, is to defer dealing with some of the methods until later. We do this by temporarily leaving the method declarations for those methods out of the class declaration or by putting in their declarations but writing their bodies in the implementation as stubs. A stub has everything the method needs except the code in the method body. If the method is a function, it might return some dummy result for the time being. Here are some sample stubs:

```
implementation

  procedure A.M;              { No statements in body }
  begin
  end;

  function A.N: B;            { Returns type B }
  begin
    N := someValue;           { A default return of some sort }
  end;
```

These stubs satisfy the compiler but do nothing until we come back to implement them more fully. They help by simplifying the testing and not getting in the way as we test other methods.

7. Applying Inheritance

As we develop object classes such as Crapgame's button, player, die, view, and game classes, we need to be on the lookout for ways to get some mileage out of inheritance.

We've already seen several examples of the kind of serendipity that inheritance can engineer. By applying inheritance during class development, we can reduce code size and enjoy the advantages of reusability, including the benefit of all the testing that has been done on a class.

As we design new classes, we shouldn't forget that an existing class might already have almost what we want in the way of instance variables and methods. We can gain considerable leverage by using subclassing to design our new class. The new class can inherit what's useful and alter the rest by adding new fields and methods, by overriding methods it can't use as is, and by judiciously reusing code from the ancestor class by means of the *inherited* keyword. This kind of subclassing is a powerful approach and one of the best features of OOP.

Sometimes we'll make a very different use of inheritance. Instead of making a subclass of some existing class, we'll "abstract out" the essence of a class, creating a new superclass over the original. We can push the hierarchy upward, generalizing

and building classes that can serve as platforms for other kinds of subclassing. This is exactly the approach we'll take in pulling PicoApp out of Crapgame.

We use inheritance in several ways in designing Crapgame. In Chapter 13 we'll implement event handling. The basic idea is to make the application, game, view, and button objects "event handlers." Back in Chapter 4, we abstracted the event-handling property out into an abstract superclass called *TEvtHandler* (paralleling MacApp). Each event-handler object—in addition to whatever else it is—is an event handler by virtue of its descent from *TEvtHandler*. Figure 10-5 shows the inheritance hierarchy of our event handlers.

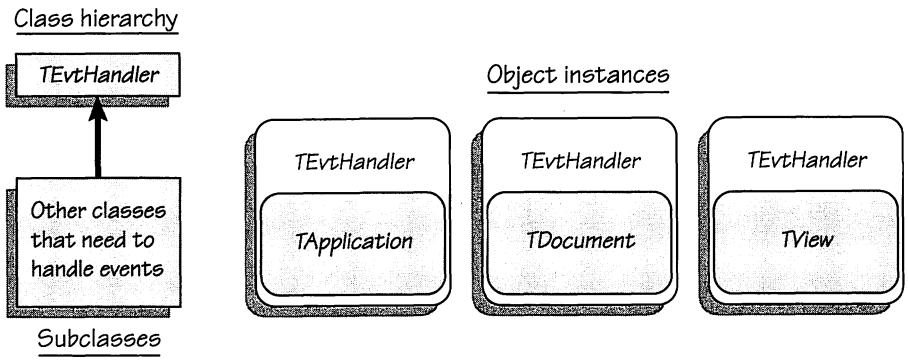


Figure 10-5.
The event-handler hierarchy. An instance of each class descended from TEvtHandler is-an *event handler with full event-handling powers through its inherited instance variables and methods.*

The Throw button object we use in Crapgame is a descendant of class *TButton*. The subclass fills in how the *DoClick* method works and adds an outlet instance variable so that the button can communicate with the game object.

Moreover, after we implement PicoApp, *TButton* becomes a subclass of *TPicoView*, from which it inherits methods like *Clicked*, *TrackMouse*, and *DoMouseCommand*. A button is a displayable object, so this is a natural kind of heritage. All view objects are event handlers, so all of them come with mouse click–handling abilities. Much of what was originally implemented in *TButton* has been pushed upward into its ancestor, *TPicoView*. *TButtons* do have some differences from other views—for instance, they can't handle double or triple clicks—but the button class is now much simpler than it was.

Inheritance also helps us solve a scope problem that would otherwise prevent two-way communication among objects. The button object must send a Play message to the current player object. But *TButton* has to be declared above *TCrapsPlayer* (a descendant of the abstract class *TPlayer*), introducing a visibility problem. By giving

the button object an outlet instance variable of type *TPlayer*, we allow it to hold a reference to a *TCrapsPlayer* object. On the code disk, the abstract class *TPlayer* is declared in the unit *UPlayer*, and the real player object class, *TCrapsPlayer*, overrides *TPlayer*'s *Play* method. At runtime, the actual player object is called by means of the button's *TPlayer* outlet instance variable.

Similarly, the button object must send the game object a CurrentShooterOf message. Again, *TButton* is declared above *TCrapGame*, so the button object can't "see" the game object. By giving the button object an outlet instance variable of type *TGame* (another abstract class), we allow it to hold a reference to a *TCrapGame* object at runtime and call the game object's method by means of the outlet.

The methods of both *TPlayer* and *TGame* are implemented as stubs. The real functionality is added by their descendants. See the discussions "Scope and Communication Among Objects" and "Outlets: A Corollary to Handling Scope Problems" in Chapter 8.

8. Integrating Our Objects

After all the objects have been developed, and after we've tested the ones that can be tested standalone, it's time to integrate them into a functioning whole—and then test the whole.

Actually, integration is usually, to use Booch's phrase again, an incremental and iterative part of object-oriented development. Typically, we'll start with a relatively simple system based on early versions of our most important classes. We'll prepare a simple working prototype of the application—that's our first, but not our last, integration. Later, we'll continually build more sophisticated prototypes that include new classes as well as revised ones, again integrating our classes. This continues incrementally and iteratively until we reach the final version of the application.

In the first version of Crapgame that I built, I used our application and game objects to integrate everything. During development of the other objects, I gradually developed the internals of these objects in a test driver. When everything worked, I transferred that code from the driver to a new object class—first *TCrapsApp* and then *TCrapGame*. The driver was then replaced by the tiny main program we've seen at the beginning of this chapter plus the new *TCrapGame* object type. At each stage, the program "worked," although with each new level it became more fully developed and integrated.

Later, as I developed PicoApp and then the version of Crapgame based on it, I worked in a similar fashion, from the simple to the more complex in a series of prototypes.

Even after Crapgame was integrated, I had to work with it to improve the display and the timing of informative messages to the human players. In THINK Pascal, this kind of compile-run-fix-compile... cycle was easy because the THINK system handles so

much of the overhead. But even when I "built" the program (compiled it as a standalone, double-clickable Mac application), I still had to work on it some more. It ran enough-faster as a standalone than as a program-in-development inside THINK that the message timing was off again.

Finally, I subjected the newly integrated standalone Crapgame program to some rigorous user testing. Thanks to Pam Sphar and Aaron Goodman for their attempts to break the Crapgame. Each did, at least once.

Other Design Approaches

The informal amalgam of the Abbott, Lorenson, and Booch approaches we've looked at in this chapter is not the only way to design OOP programs. Nor, as presented here, is it likely to be the best approach for commercial-grade software development, although hobbyist programmers should find it sufficient. For programmers who know something about data flow techniques, David M. Bulman proposes in "An Object-Based Development Model" (Bulman 1989a) a more formal design approach he calls "model-based development." Unlike most other OOP designers, Bulman believes that selecting the objects is not a trivial step that can be done by simple inspection or by writing an informal description of the problem. He also contends that, although objects consist of both data and operations on the data, at the design stage it's the operations that count. Bulman's article is worth reading.

Among the languages Grady Booch discusses in his OOD book is Object Pascal of the Macintosh variety. The principal design techniques of his "round-trip gestalt design" approach for finding objects are object-oriented analysis and domain analysis. Booch also presents a serious MacApp-based optics application using Object Pascal. Serious OOP developers should read Booch.

Developing PicoApp

Having laid out the design for a specific application, Crapgame, we now need to ask some questions about it. Our goal is to see what can be reused from Crapgame the next time we need to write a simple Macintosh application.

The end result will be a tiny class library, or application framework, somewhat like MacApp or TCL. A word of warning, though—what we'll develop here is really about at the level of the little skeleton programs, or application shells, we all kept around in pre-OOP days to use as templates for building applications. Such shells usually implement basic Mac initialization, a simple event-loop and event-dispatching mechanism, the basis for menu handling, and not much else. To use such a shell, we copy it and start inserting code wherever it's needed to create our application.

That same process is essentially what MacApp and TCL programmers do, although they do it with object classes, using subclassing and overriding rather than physically altering the application shell. The shell in this case is a set of object classes that the programmer subclasses to fill in details and override default behaviors. MacApp

and TCL, of course, are a lot more than that—each implements a large proportion of what's standard about the Macintosh user interface. MacApp does more than TCL (and is by now much more robust and field tested), but both—and PicoApp—work in essentially the same way.

What PicoApp Has

PicoApp implements a number of features of the Mac interface:

- Menus—in such a way that you can add your own
- An event loop—that is MultiFinder aware
- An event dispatcher—that you can extend or override using numerous "hooks" (overridable methods)
- A mechanism for passing events down an event chain to the objects that can process them, such as buttons
- Command-keys for menu items
- A simple system of "views" for handling your application's drawing and updating (could be extended for printing)
- A document class that can serve as the basis for your documents, mediating among files, windows, and views
- Default code for handling the New, Open, and Quit items in the File menu
- Basic memory management
- And more

What PicoApp Lacks

Of course, PicoApp isn't much. Unlike TCL and MacApp, it doesn't support:

- Color
- Multiple monitors
- Printing
- File handling (saving, reading, and reverting to previously saved versions of documents)
- Scrolling views and scrollbars
- TextEdit objects, let alone extended TextEdit
- Palettes, floating or otherwise
- Dialog box objects
- And much, much, much more

To use PicoApp, you still have to do a lot of the work yourself. PicoApp does give you a head start, and it is highly extendable. Add more classes.

Why PicoApp?

We have two reasons for developing a simple PicoApp.

PicoApp demonstrates the process of abstraction we'll use to create highly reusable object classes. This is the first step toward developing an extensive class library, which in the long run will greatly simplify our programming.

Second, we can look at PicoApp as a set of "training wheels" for using the more capable application frameworks MacApp and TCL. After looking at the structure of a simple framework and using it to develop an application, we'll be ready to move on to either MacApp or TCL. In the long run, this is probably the more important of our PicoApp goals.

Summary

In this chapter we took a first look at Crapgame, a complete, thoroughly OOP application.

We developed some tips on finding the objects needed by a program, deciding what instance variables and methods those objects would need, deciding how the objects would communicate with each other by messaging, testing the object class designs, making use of inheritance, and integrating the objects into a whole program.

The most important point of this chapter has been that OOP program design, unlike traditional application design, works incrementally and iteratively, from both the bottom up and the top down, rather than simply from the top down as traditional structured programming with its waterfall development cycle does.

Projects

- Try out the design methods described in this chapter on another program. A simple card game might be a good project. If you aren't well versed in graphics programming, simplify the display aspects. For instance, card objects might display themselves simply by printing their numbers and suits rather than by drawing elaborate pictures. Don't forget to try using inheritance if it seems suitable. And feel free to reuse anything you've seen so far, such as our *TButton* class or *TRandom*.
- If you choose instead to write some other dice game, borrow the *TDie* class. If it doesn't quite suit you, subclass it to change it.

You might want to read the rest of Part 2 before you do much beyond the high-level phase of your design.

11

PICOAPP: A TINY APPLICATION FRAMEWORK

Having laid out a general design for the Crapgame application, in this chapter we'll develop the overall architecture of the application and refine its design with some detail. And we'll begin the process of generalizing from Crapgame to develop PicoApp. We'll see

- The architecture of Crapgame
- What an application framework is
- How PicoApp is derived from Crapgame
- How PicoApp can be used in new applications

Structuring an Application with Objects

Let's consider three approaches to constructing an application program with objects:

- Writing a traditional Pascal main program that takes care of setup and has a main event loop, adding objects where needed
- Writing a collection of object classes in which various aspects of the user interface behavior are intermixed with the specifics of what the application does—such as managing the game of craps
- Writing classes such as application, document, and window that handle the user interface, and other classes such as player, die, and button that perform the actions of the application

We've used something like the first approach in our button example in Chapter 5. And THINK Pascal comes with an OOP demo called ObjectDraw that's built on the same principle.

I tried the second approach in the very first version of Crapgame, which you won't see here. The original version of Crapgame had a game object, player objects, dice, and buttons. But it had no application object.

The third approach is the one we'll take in this chapter and in the rest of Part 2. And we'll generalize our application architecture so much that it will be reusable for other applications such as the second small PicoApplication, called PicoSketch, that we'll develop in Chapter 15.

Crapgame Architecture

The work in Crapgame is divided among three kinds of objects:

- An application object
- A game object
- Several view objects

Figure 11-1 shows the Crapgame objects and their relationships to one another.

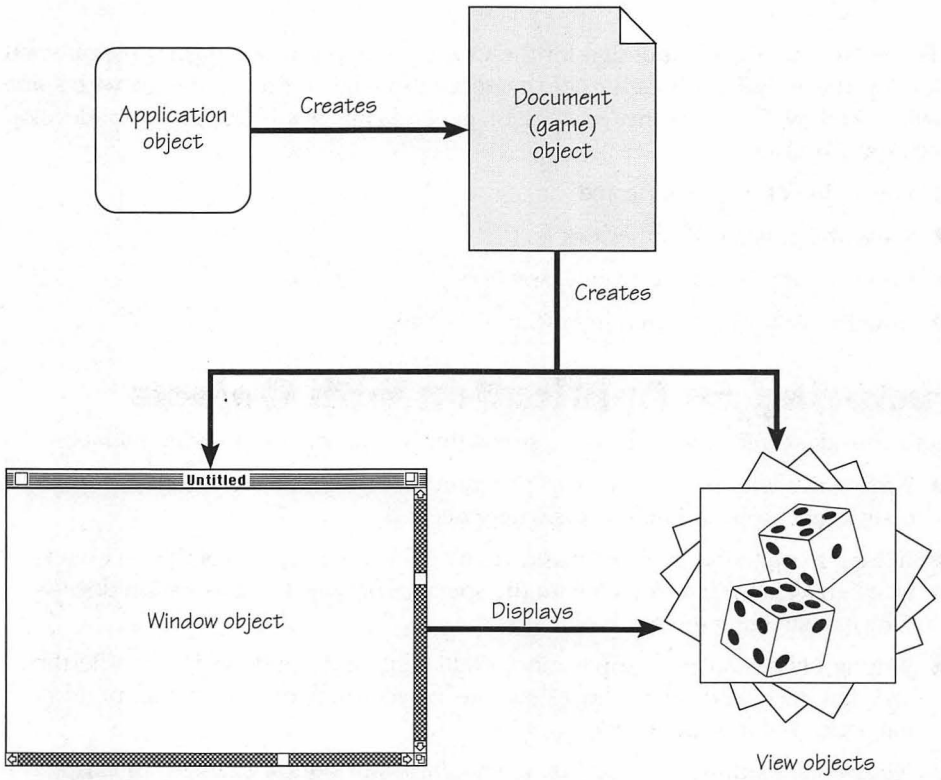


Figure 11-1.
The several kinds of objects in Crapgame and their interaction.

The application object is responsible for creating a shell within which the users can play game after game. The shell provides menus for starting a new game, for choosing different options for the next game, for quitting the application, and for closing the current game without quitting. The application object also takes care of updating and activating the game window and processing other events in general.

The game object is responsible for creating and initializing the objects that provide the display and carry out the game. The game object also serves as a communications center and conduit for event passing among the other objects. And it creates the game's window and views.

The other objects are almost all view objects, responsible for, among other things, displaying information for users. Most of the view objects also have other functions. The button object turns mouse-down events into player actions. The player objects embody the rules of the game, keep score, and manage dice-throwing. The die objects return pseudorandom numbers.

The only other major object in Crapgame is the player list object. Its responsibility is to keep a linked list of player objects and to keep track of whose turn is next. Underlying the player list object is a simple linked list like the one we use to link *TEvtHandler* objects. One object in the list points to the next, using an object reference in an instance variable to do the pointing. Later, in Part 3, we'll develop a much more powerful linked list software component that can be plugged into any application. But for now we'll get by with simpler lists.

Crapgame at Work

When Crapgame is run, it puts up menus, creates a game object that in turn creates the view objects, and runs the game. One or more users take turns according to the rules of craps. During a turn, a user clicks the Throw button to roll the dice and rolls repeatedly until he or she craps out. The player's score (in accumulated dollars) is displayed, along with the current point to match and the current player's player number. When the user quits, Crapgame destroys the objects it has created.

Underneath this action, there's a lot going on:

- Events, such as mouse clicks, are passed down an event chain from object to object until they reach the object that handles them.
- When the Throw button detects that it has been clicked, it asks the game object for the current player, and the game object, in turn, gets a player object reference from the player list object. Then the button object sends the player object a Play message. The player object responds by sending a Roll message to each die object. The die objects generate pseudorandom numbers and return them to the player object. The player object then analyzes the rolls and takes appropriate action, updating its display information, posting messages, and alerting the game if it craps out, or posting an alert and asking the game to remove it from the player list if it runs out of money.

- When a user clicks in the window's close box or chooses Close from the File menu, the game object frees its views, including the window, and then frees itself.
- If a user then chooses New Game or New Options from the File menu, the application object creates a new game object, the game object creates the views, and a new game commences. The event loop starts getting events and passing them down the chain.
- If a user chooses Quit from the File menu, the current game object frees itself (and its subordinate objects), and the application terminates.
- If a user chooses About Crapgame from the Apple menu, an About dialog box appears. It has buttons leading to three subordinate dialog boxes, and the user closes it by clicking the Done button.
- If a user running Crapgame under the Finder starts a desk accessory from the Apple menu, the Edit menu is enabled. The Close command in the File menu also works for the desk accessory.

Deriving PicoApp from Crapgame

Having looked at a picture of the Crapgame objects and how they interrelate, let's ask a fundamental question: What does Crapgame have in common with all other Macintosh applications?

From that question comes the genesis of PicoApp.

General Application Features of Crapgame

Consider these statements about Mac application characteristics and their relationship to Crapgame:

- All true Mac applications have an event loop, which gets events from the Mac and dispatches them to other parts of the application for handling.
- All Mac applications use windows for display, and the application takes care of updating and activating windows as needed. (Not quite true—a few applications don't use windows—but close enough.) Typically, you can drag the windows around on the desktop, and sometimes you can resize and zoom the windows. (Crapgame allows dragging but not resizing or zooming, although these capabilities are built into PicoApp.)
- All Mac applications (well, almost all) provide menus for giving commands. In particular, almost all applications provide Apple, File, and Edit menus. The Apple menu normally contains an About command. The File menu normally has New, Open, Close, and Quit commands plus, possibly, Save, Save As, and Print commands. The Edit menu normally has Undo, Cut, Copy, Paste, and, sometimes, Clear commands. Menu commands are dimmed when their actions are not available. Often menu commands have command-key equivalents.

- Besides menus, (virtually) all Mac applications take user input from various kinds of controls, such as buttons and scrollbars, using the mouse. Many applications also provide keyboard substitutes for some control manipulations. (Crapgame lets you "click" the Throw button by pressing either the Return or the Enter key.)
- All Mac applications (almost all, anyway) support desk accessories.
- All (recent) Mac applications support MultiFinder, so they can run under either MultiFinder or the Finder.
- Most Mac applications create and manipulate documents of some kind. Documents mediate among files, windows, and views of data. (Crapgame has no documents *per se* and does not use files, but it does have the game object, somewhat analogous to a document object, that manages game data and mediates between that data and the window and views.)
- Most Mac applications support cutting, copying, and pasting by means of the Clipboard. (Crapgame has an Edit menu, which it enables only for desk accessories if it's run under the Finder. PicoApp processes Edit menu items by calling SystemEdit to pass them to a DA if appropriate; if not, it calls the *Cut*, *Copy*, *Paste*, or *Clear* method, but each is a stub. Crapgame chooses not to override those methods.)
- Many Mac applications support the Undo (and Redo) command, at least to some extent. (So far PicoApp doesn't, but it does pass Undo menu commands to desk accessories, and it provides an *Undo* method in case we want to override it to implement Undo in some way.)
- All good Mac applications support memory management to keep the application from crashing while they try to allocate memory for a document or another object. The usual approach is to set up a "memory reserve" from which the application can draw if the Memory Manager is unable to supply enough space in the heap. If the memory reserve gets too small, the application can take various memory-saving steps: It can ask the user to close a document; it can disable menu items that can cause memory allocation requests; and in extreme cases it can ask the user to quit the application (and presumably buy more RAM).
- Today, with the advent of color monitors for Macintoshes and machines like the Mac II series and the Mac LC, many Mac applications support color.

Crapgame has most Mac program features, in one way or another, except for undoing actions and color. So far we haven't provided a mechanism for Undo, but we'll at least discuss some approaches to implementing it. And some of our PicoApp objects are somewhat more limited than they ought to be. Our document objects, for instance, can't print or save to a file yet, and our view objects are very simple indeed.

Crapgame-Specific Features

Of course, Crapgame does many things that most other Macintosh applications don't:

- Crapgame puts up a custom introductory dialog box. Many applications do this, but the content for each application's opening dialog box is highly specific. Crapgame's introductory dialog box, shown in Figure 11-2, uses animation for a little fun.

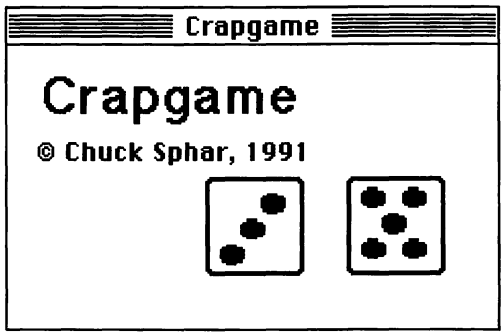


Figure 11-2.
Crapgame's opening dialog box.

- Crapgame includes an About Crapgame dialog box, similar to the About dialog boxes in other applications. But the structure of Crapgame's About dialog box is atypical in that it branches to three subdialog boxes. And, of course, its content is specific to Crapgame. (The subdialog boxes display information about the program, about the rules, and about scoring. Each dialog box is activated by a button in the primary About dialog box.)
- The content of Crapgame's File menu is partially specific to this particular application. Close and Quit are pretty standard. New Game and New Options are equivalent to most applications' New commands. They cause a new game object to be created. Of course, the exact nature of what's opened is Crapgame specific.
- Naturally, some of Crapgame's objects and their actions are not generic at all: player and die objects, the button's *DoClick* method, the player list object, many of the game object's methods. In addition to what is generic about it, Crapgame implements the rules and behaviors of a game of craps, after all.

Division of Labor

The Crapgame design began life with these fundamental object classes: *TCrapsApp* (the application object), *TCrapGame* (the game object), *TCrapsPlayer* (the player object), *TPlayerList*, *TDie*, and *TThrowButton*.

At this point, we can abstract out all the generic functionality that applies to all Mac applications. Figure 11-3 shows the derivation of PicoApp superclasses from Crapgame's classes. From the design of class *TCrapsApp*, we create a new superclass, *TPicoApp*, to which we move all the generic application methods—*MainEventLoop*, *DispatchEvent*, and most of the other event-handling methods, as well as methods relating to the menus and to initializing and quitting the application.

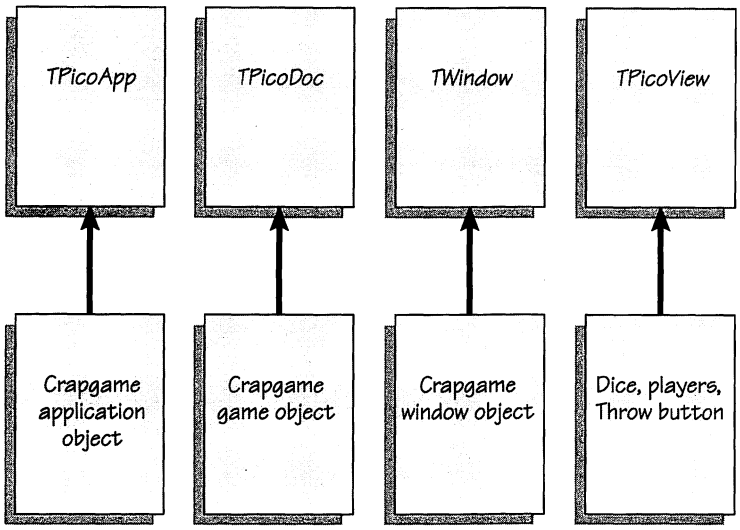


Figure 11-3.
Deriving PicoApp from Crapgame.

From the design of class *TCrapGame*, we create a new superclass, *TPicoDoc*, to which we move all the generic document methods, such as those that create and manage windows and views.

From the design of classes *TDie*, *TCrapsPlayer*, and *TThrowButton*, we create a new superclass, *TPicoView*, which embodies the display-related and mouse-handling activities of those classes. We also create PicoApp's *TWindow*, a new subclass of *TPicoView*, which operates a standard Macintosh window for PicoApp's document object, displaying the document's views.

Of course, many of the methods moved up to superclasses become stubs there and are then overridden in their Crapgame-specific subclasses. But a lot of functionality moves up, too.

Besides creating the new superclasses, we make those superclasses descendants of class *TEvtHandler*, the class that embodies event handling in PicoApp. All descendants of *TEvtHandler*—hence the application, the document, and all views—are event-handler objects, with methods for handling mouse, keyboard, and other events (although in Crapgame not all the objects actually handle events).

Figure 11-4 shows the relationships among the old and new classes. By deepening the hierarchy with superclasses, we build in a general level that presides over the specifics.

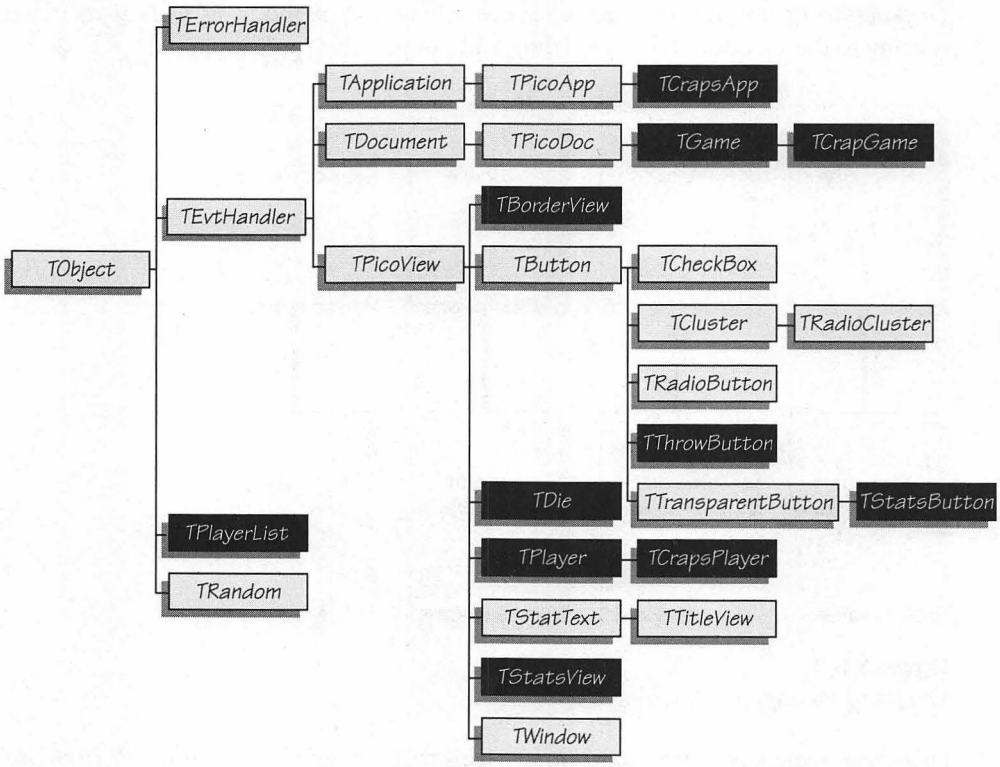


Figure 11-4.
The PicoApp-Crapgame object class hierarchy. The names of Crapgame-specific classes are inverted.

Where's PicoApp? It's composed of all of the nonspecific classes in the hierarchy, from *TObject* (*TEvtHandler*'s superclass) down to *TPicoView*, *TPicoApp*, and *TPicoDoc*. (Technically, *TButton*, *TStatText*, *TWindow*, and some of their subclasses are also a part of PicoApp, although they're subclasses of *TPicoView*.) Together, these classes make up a small object class library, which, if compiled, has the functionality of a basic Macintosh program (but nothing more). The PicoApp library is a reusable collection of objects, capable of being the foundation of many applications.

Application Frameworks

PicoApp is a small, demonstration-model application framework, the same sort of creature as Apple's MacApp and THINK's TCL. (In the MS-DOS world, Borland has released a framework under Turbo Pascal 6.0 called Turbo Vision.)

An application framework is a library of classes that encapsulate the generic workings of an application. The classes can handle application startup chores, put up basic menus, manage a list of documents, provide basic document functions such as window creation, get and dispatch events to various objects in the library, and handle application shutdown.

To use an application framework, we subclass several of its main classes (in PicoApp's case, its application class, its document class, and several view classes) to add our own methods and override methods we don't want to inherit as is. We usually write a few new classes and fit them into the application's architecture. Figure 11-5 illustrates the relationship between PicoApp's classes and the programmer-supplied classes for Crapgame. The result, PicoApp classes plus Crapgame classes, is a fully functional Macintosh application that does something more or less useful.

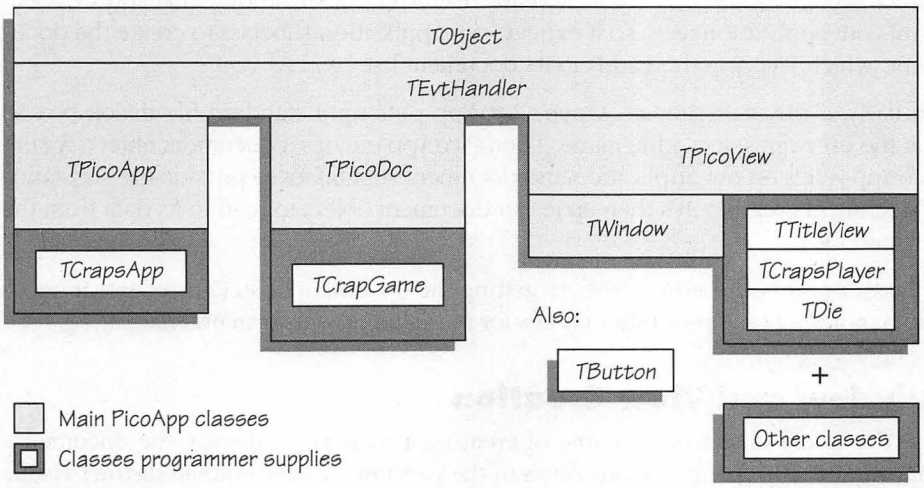


Figure 11-5.
PicoApp framework with added classes. The programmer supplies TCrapsApp, TCrapGame, TTitleView, TCrapsPlayer, TDie, *and other application-specific classes.*

One advantage of PicoApp over the more powerful MacApp and TCL application frameworks is that we can use PicoApp with only 1 megabyte of RAM. TCL requires 2 megabytes; MacApp requires 2 to 4 megabytes.

PicoApp, MacApp, and TCL differ in many of their details, but they all operate on the same basic principles, and using them involves essentially the same process, which we'll go into in more detail now.

What PicoApp Does

Let's look at the aspects of a Macintosh application that PicoApp can hand us on a platter.

Startup

On startup, PicoApp creates an application object. That object, in turn, puts up an Apple menu and the File and Edit menus, using MENU resources and an MBAR resource that we supply. Our application subclass can add more menus if they're needed. PicoApp also initializes a number of global variables.

PicoApp then provides several "hooks," which we can use to insert any additional startup behavior our application needs.

Finally, PicoApp begins its event loop and starts taking input from the user.

Document Creation

When the user chooses New from the File menu, PicoApp creates a new document object and has it initialize itself. Actually, PicoApp doesn't know what sort of documents our application uses, so it expects our application subclass to create the document, which PicoApp then adds to its document list.

Similarly, if the user chooses Open, PicoApp puts up a standard file dialog box so that the user can select a file name. Then PicoApp creates a document object. Again, PicoApp relies on our application and document subclasses to provide the substance of document creation. It's then up to our document object to read in its data from the selected file.

From this point on, PicoApp sees to getting the document object any events it needs and handles a number of other chores for it, including cursor maintenance.

Window and View Creation

The document object is in charge of creating a window to display the document's data and installing one or more views in the window. A view is some distinct visible entity in a window, such as a graphic, a button, or a scrollbar. Of course, it's up to us to decide what our views do, and our document subclass has to handle the view creation. But PicoApp supplies the window and sees to it that views get the events they need and are notified when they need to draw themselves.

Event Handling

PicoApp provides a complete event loop and dispatching mechanism to get events from the user and send them to the appropriate objects. Basically, PicoApp uses three "submechanisms" to do this.

First, PicoApp's event-handler chain is a linked list of objects that handle mouse clicks, keypresses, and certain other kinds of events. Each event is passed along the chain until an object can claim the event and respond to it. The chain usually includes one or more view objects and possibly a document object.

Second, PicoApp allows view objects in the current window object's view list to check whether the cursor needs to change shape. Once each time through the event loop, PicoApp lets each object in the list check whether it needs to alter the cursor. Figure 11-6 shows the two views in Crapgame that change the cursor from an arrow to a pointing finger.

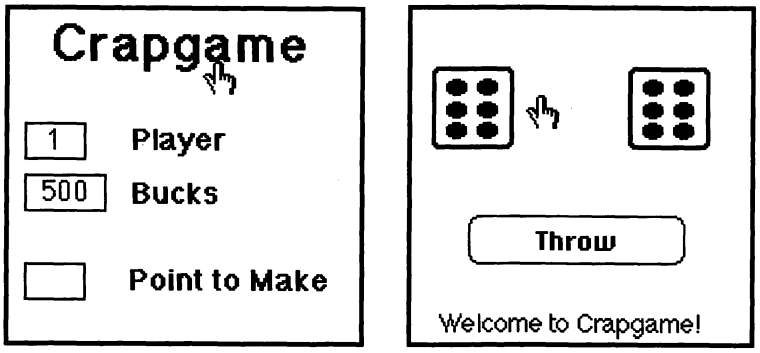


Figure 11-6.
Two views in which the cursor takes the shape of a pointing finger.

Third, some other events, such as updates and activates, are also channeled through window objects. For example, upon an update event, the affected window is notified. It then tells its list of view objects to redraw themselves. Other window actions, such as resizing and zooming, are also passed through the window to its views in case they need to take some related action, such as resizing themselves to fit.

Window Closing

When the user closes a window, by either clicking its close box or choosing Close from the File menu, PicoApp notifies the document associated with the window. The document takes care of disposing of any data storage it has in the heap and tells its window and its views to dispose of themselves. The document also asks the application to remove it from the application's document list. And the appropriate menu commands are adjusted.

Shutdown

When the user chooses Quit from the File menu, PicoApp sends close messages to all open documents. The document objects ask their windows to close themselves, free their view objects, and so on. PicoApp allows us to add any other shutdown behavior we need. And, finally, it terminates the application.

Using PicoApp in Other Applications

How can you reuse PicoApp to write completely new applications? By reversing the process through which we created PicoApp in the first place. For a complete new application, we need to add several classes to those in the PicoApp library:

- The application's specific application subclass—like Crapgame's *TCrapsApp*
- The application's specific document subclass—like Crapgame's *TCrapGame*
- One or more specific view subclasses—like Crapgame's *TCrapsPlayer*
- Any other classes needed to build a complete model of the software solution—like Crapgame's *TPlayerList*

To add these classes, we subclass the main PicoApp library classes. Each subclass declares its PicoApp ancestor. Thus, *TCrapsApp* declares *TPicoApp* as its immediate ancestor class in the heritage spot, and by means of inheritance and overriding, the labor is divided between the library class and the application-specific class. Figure 11-7 illustrates the relationship between the library class ancestor and the application class.

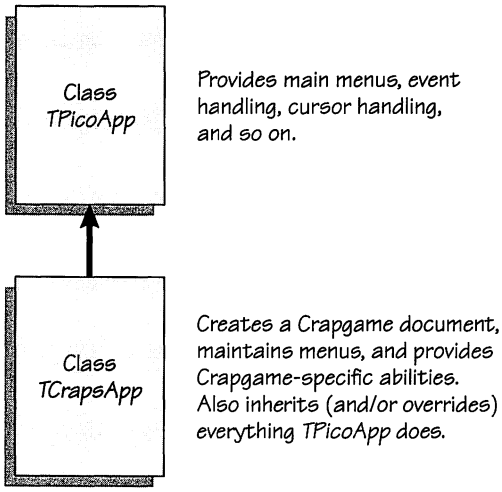


Figure 11-7.
Building an application object by combining the PicoApp application class and the Crapgame application class.

Overview: Steps in Using PicoApp

To build an application such as Crapgame on top of PicoApp, we need to follow the steps listed on the next page.

1. Subclass a number of PicoApp's classes, overriding certain methods.
2. Create additional classes needed for the application.
3. Create a resource file and associate it with the application.
4. Compile and debug.

Now let's take a look at how we'd carry out these steps as we built Crapgame on top of PicoApp.

Subclass *TPicoApp*. Our application subclass is called *TCrapsApp*. It can inherit quite a few of *TPicoApp*'s fields and methods, but it also overrides some methods and adds some of its own. The result is an extended version of *TPicoApp*, specialized for running a game of craps.

Subclass *TPicoDoc*. *TPicoDoc* is an abstract class from which we subclass our document class called *TCrapGame*. We'll refer to an object of class *TCrapGame* as the "game object." The game object is a document in the sense that it mediates among the game's data, its window, and the views that display the game information to users. *TCrapGame* is a specialized document designed to manage game players, dice, and other aspects of a game of craps.

Subclass *TPicoView*. *TWindow* works fine without subclassing, but the view objects we want to display inside the window need to be developed from *TPicoView*. This is a matter of analyzing the planned display, singling out independent elements of it. We select seven view objects: the Throw button, the dice, the current player, the title "Crapgame" at the top of the window, a border around the other views, a transparent statistics button overlying the dice images, and a completely alternate view depicting dice statistics that replaces the other views when the statistics button is clicked. Figure 11-8 shows some of the views used in Crapgame, each a separate subclass of *TPicoView*. *TThrowButton* is a subclass of *TButton*, which is in turn a subclass of *TPicoView*. *TStatsButton* is a subclass of *TTransparentButton*, a specialized subclass of *TButton* that provides a button without a display.

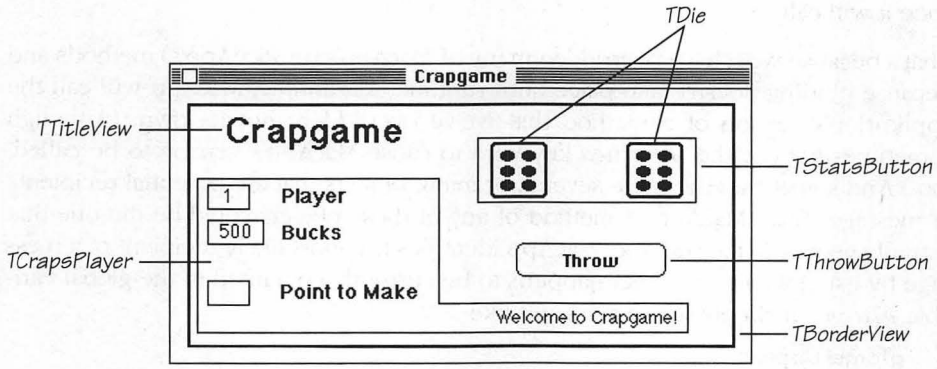


Figure 11-8.
Some of Crapgame's view objects.

For each of these views, we need to flesh out a display method, and we might need to fill in a few other details. Some views, for instance, need to change the cursor's shape. The statistics button, for example, needs to change the cursor to a pointing finger when it comes over the button, so the button must notify the application of this need. Many of our views need to respond to mouse clicks. The title view, for example, is set up to demonstrate the effects of clicking, option clicking, double clicking, and triple clicking, so we need to fill out its *DoClick*, *DoDoubleClick*, and *DoTripleClick* methods.

Although we haven't implemented scrolling in PicoApp yet, some views would also need to enable scrolling of their contents. And some would need to respond to activate, deactivate, grow, and zoom messages in various ways. The mechanism for these latter features is available in PicoApp, but Crapgame doesn't make use of them.

Create a resource file. PicoApp has a basic resource file, but we need to copy it to a file for the particular application, Crapgame. Then we need to add other resources, such as a cursor (CURS) and several icons (ICON), for Crapgame. We also need to edit a number of the resources copied from PicoApp's resource file: the resources associated with providing a desktop icon (BNDL, FREF, STR#, and a "signature" resource that enables the Finder to create an icon); several of the MENU resources; the MBAR resource; and so on. We'll look at these in more detail in Chapter 16.

Compile and debug the program. After we've assembled or created all of these pieces, we need to compile, debug, and test the application.

When we begin to compile and debug an application, the method calls can be confusing at first. MacApp programming has been called "programming in the closet" (Wilson 1990), and the same can be said of PicoApp programming. Most of the time MacApp (or PicoApp) is in control, handling menus and windows, getting and dispatching events, and so on. But there are many things that an application framework can't do. When it encounters one of them, it calls the application's method code to do the job. The trouble is, we can't easily tell when MacApp will call our code, or which code it will call.

That's because we'll have overridden many of MacApp's (or PicoApp's) methods and because binding doesn't take place until runtime. At runtime, MacApp will call the application's version of a method that we've overridden, not its own. (Although sometimes we use the *inherited* keyword to cause MacApp's version to be called, too.) And sometimes there are several, or many, objects that are potential recipients of messages from MacApp. A method of any of these objects could be the one that actually gets called at runtime. MacApp identifies the most likely recipient of a message by using whatever object happens to be currently contained in the global variable *gTarget*. It simply sends messages like

```
gTarget.Display;
```

PicoApp doesn't use a *gTarget* variable, but the results of its message passing are similar.

Part of the confusion stems from our tendency to think of *TPicoApp* and *TCrapsApp* as defining two separate objects. In fact, they define one application object. Figure 11-9 shows the relationship between the two classes and their one object result.

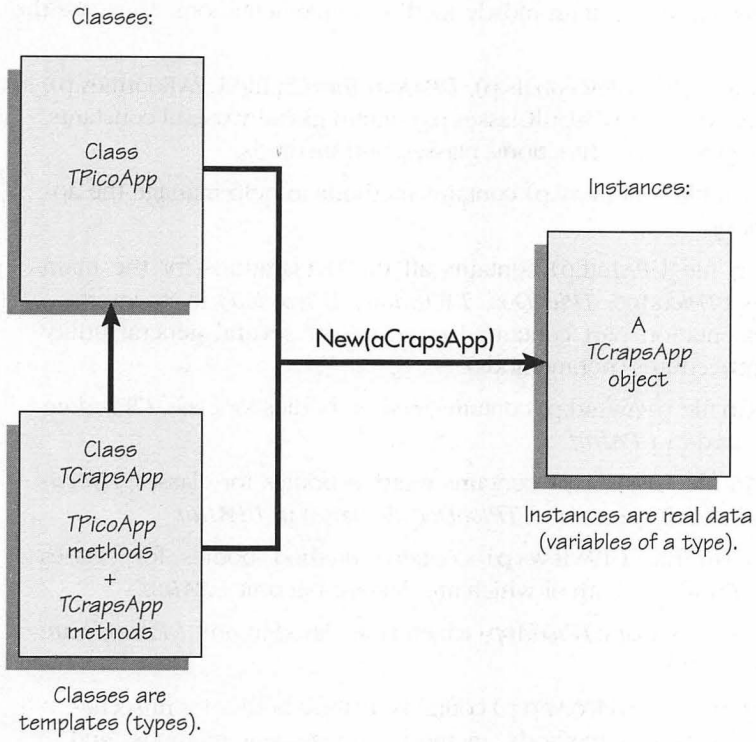


Figure 11-9.
Two classes, one object. A TCrapsApp *object is a member of two classes.*

When we create a new *TCrapsApp* object in our tiny main program for Crapgame, the new object has all the fields and methods we've designed for *TCrapsApp*, and by inheritance, it also has all the fields and methods of *TPicoApp* (except those methods we've overridden). The one and only application object *is-a TCrapsApp*, but it also *is-a TPicoApp*.

An analogy drawn from biological inheritance might help us keep this two-classes, one-object arrangement straight. Having inherited your father's nose and your mother's eyes, you still are just you. You have features like your parents'—you don't literally have the nose off your father's face.

And an application object inherits features from its application class ancestors. But there's only one application object in a given OOP application. It's primarily and above all a *TCrapsApp* object (or something similar) with TPicoApp overtones.

The same principle holds for our subclasses of *TPicoDoc*, *TPicoView*, and other PicoApp classes.

PicoApp's Units

The bulk of PicoApp is contained in a group of eleven Pascal units plus several "freestanding" units we can add to our projects as we need them. Some units contain mostly declarations. Others contain mostly method implementations. Here are the eleven Pascal units:

- Units *UPAGlobals* (in file UPAGlobals.p), *UPARoutines* (in file UPARoutines.p), and *UPAUtilClasses* (in file UPAUtilClasses.p) contain globally useful constants, variables, types, procedures, functions, classes, and methods.
- Unit *UMemory* (in file UMemory.p) contains methods to help manage the application's memory.
- Unit *UPAIntf* (in file UPAIntf.p) contains all the declarations for the main PicoApp classes (*TPicoApp*, *TPicoDoc*, *TWindow*, *TPicoView*) in its interface part. Its implementation part contains the bodies of several general utility functions and procedures (not methods).
- Unit *UPAWind* (in file UPAWind.p) contains method bodies for class *TWindow*, which was declared in *UPAIntf*.
- Unit *UPADoc* (in file UPADoc.p) contains method bodies for classes *TDocument*, declared in *UPAGlobals*, and *TPicoDoc*, declared in *UPAIntf*.
- Unit *UPAViews* (in file UPAViews.p) contains method bodies for classes *TPicoView* and *TStatText*, both of which are declared in unit *UPAIntf*.
- The method bodies for class *TPicoApp* (which is declared in unit *UPAIntf*) are spread across three units:
  - Unit *UPicoApp* (in file UPicoApp.p) contains method bodies for miscellaneous application-class methods, memory-management methods, and event-handler chain methods.
  - Unit *UPAMenus* (in file UPAMenus.p) contains method bodies for menu-creation and menu-update methods of class *TPicoApp*.
  - Unit *UPAEvents* (in file UPAEvents.p) contains method bodies for event-handling methods of class *TPicoApp*.

The several freestanding units can be incorporated in PicoApp as we need them:

- Unit *UButton* (in file UButton.p) contains classes *TButton*, *TCluster*, *TRadioButton*, *TRadioCluster*, and *TCheckBox*.
- Unit *UHyperButtons* (in file UHyperButtons.p) contains class *TTransparentButton*. Eventually, it should contain several other classes.
- Unit *URandomStream* (in file URandomStream.p) contains class *TRandom*, a random number–generator object useful in many applications.

In addition to these units, PicoApp expects a unit *UMyGlobals* to be declared above all PicoApp units except *UPAGlobals*. This is where we can put any global variables,

constants, and so forth that our application might need. Some menu-related global constants must be declared in *UMyGlobals*. Even if our application didn't need any globals of its own, we would still be required to provide a basic unit with the right name. (PicoApp provides a skeleton version of *UMyGlobals* we can begin with.)

Other Units Used by PicoApp

Because all of PicoApp's objects ultimately descend from *TObject*, its unit, called *ObjIntf* (in file ObjIntf.p) must be declared above PicoApp. PicoApp uses a modified version of the *ObjIntf* unit that comes with THINK Pascal. It's in file ObjIntf.px (x for "extended") on the code disk.

What Else?

In addition to the units that make up PicoApp, our project will also contain units for our own objects and other code. These units are declared below all of PicoApp. Their *uses* clauses must list *ObjIntf*, *UPAGlobals*, *UPAUtilClasses*, *UPAIntf*, and possibly *UMyGlobals* and *UPARoutines*. See the project list for Crapgame in Figure 11-10.

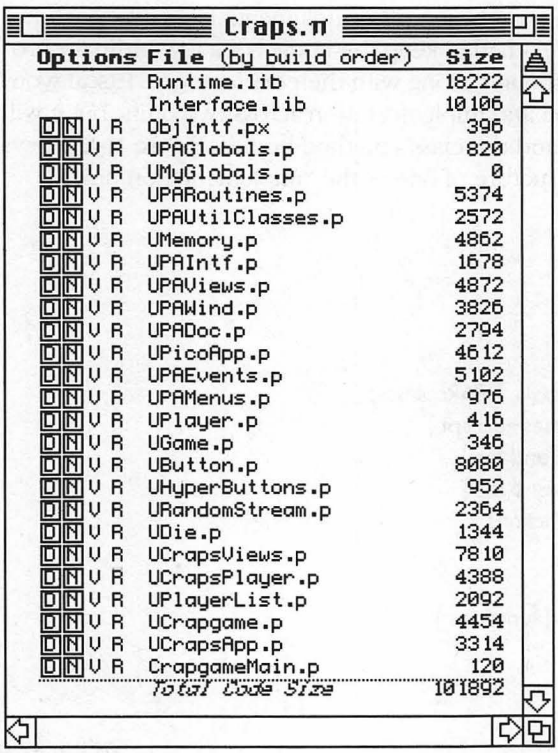


Figure 11-10.
The THINK Pascal project window for Crapgame, showing compilation order.

Because *UPAIntf* contains all the main PicoApp class declarations, the PicoApp units below it in the project all list it in their *uses* clauses. All of them list *UPAGlobals* for reference to PicoApp's global types, variables, and routines. Many of them also list *UMyGlobals* and *UPARoutines*. All of them also list *ObjIntf*, for *TObject*.

About the Way Units Are Broken Down

I designed PicoApp using THINK Pascal on a Macintosh SE with 1 megabyte of RAM and a 20-megabyte hard disk, so it was impractical to put all of PicoApp into one unit—there wouldn't have been enough RAM for THINK's already-large compiler to load and compile the unit. Besides, one of the design goals was to keep PicoApp small enough for use by programmers who don't have enough RAM or hard disk space for MacApp or TCL.

Originally, my solution was to put each class in its own unit, as I've advocated in other parts of this book. But that was rather cumbersome. The list of units in the project was very long, and the *uses* lists in the units were also quite long.

I finally decided to put all the interface elements (declarations) into the interface part of a unit *UPAIntf* and postpone the method bodies. I broke the method bodies up into several units, putting them into those units' implementation parts. Each implementation unit uses unit *UPAIntf*. I had to keep the bodies of a few ordinary procedures and functions in *UPAIntf*, though, along with their declarations. Pascal won't let us split a procedure's declaration and implementation across two units, but it will let us separate a class declaration and the class's method bodies. Figure 11-11 shows the structure of *UPAIntf* and the structure of one of the "implementation units."

```
unit UPAIntf;

interface
  uses
    ObjIntf, UPAGlobals, UMyGlobals, UPARoutines;
  { Declaration for all PicoApp classes--except }
  { freestanding ones like TButton--and for }
  { global constants and variables and for }
  { useful utility procedures and functions }

implementation
  { Bodies for utility procedures and functions }

end.  { UPAIntf }
```

Figure 11-11. *(continued)*
The unit breakdown for PicoApp. Large PicoApp classes are spread over multiple units.

Figure 11-11. *continued*

```
unit UPAViews;

interface
  uses
    ObjIntf, UPAGlobals, UPARoutines, UPAIntf;
  { Empty interface section--see UPAIntf }

implementation
  { Method bodies for some of the classes declared in UPAIntf }
  { Others are in other units that resemble this one }

end.  { UPAViews }
```

This solution keeps the number of files down, keeps the *uses* clauses shorter, and still keeps the files small enough for compiling and debugging under THINK Pascal. When we compile under THINK and encounter an error, we need enough RAM to open the file so that we can look at where the error occurred and use THINK's debugging tools.

The solution is similar to the approach taken in MacApp, where a technique like this is used:

```
unit USomething;                    { This is a MacApp example }

interface
  uses
    unitNames;
  { Class and other declarations }

⋮

implementation
 {$I USomething.incl.p}

⋮
end.  { USomething }
```

Here MPW Pascal's *$I* (meaning "include") directive is used to physically include the named file (USomething.incl.p) at compile time. Yet the mass of material is neatly split between two files.

This mechanism is not available in THINK, which lacks an "include" directive, so I resorted to the mechanism I've just described.

THINK's own TCL uses a scheme similar to PicoApp's. Each class's method code is in a separate unit—in the unit's implementation part—but all the declarations are in a

single large unit called *TCL*—in a file called TCL.p. The difference is that PicoApp confines the implementation code to fewer files. As PicoApp grows (after the publication of this book), I'll undoubtedly need to spread its code over more files and units to keep them small enough to use in 1 megabyte.

Summary

In this chapter, we've looked at the general shape of PicoApp and learned how new applications can use it as a framework on which to build their own functionality.

Projects

- Having started your own application design in the previous chapter, start thinking now about how you'd build on PicoApp to create it. You'll want to study the PicoApp code in the folder PicoApp on the code disk: in files UPAGlobals.p, UPARoutines.p, UPAUtilClasses.p, UMemory.p, UPAIntf.p, UPAWind.p, UPADoc.p, UPAViews.p, UPicoApp, UPAMenus.p, UPAEvents.p, and, of course, UButton.p. You'll probably also want to reserve most of your real decisions until you've read through the rest of Part 2.
- Consider how you would design an application such as Crapgame using lots of object classes but without using PicoApp. Contrast that with how you might design the application without objects.

12

PICOOBJECTS

In this chapter we'll survey the main object classes that make up PicoApp. We'll look at

- PicoApp's class declarations
- PicoApp's important features and abilities
- The "hook" methods on which we can hang variations of PicoApp's abilities
- Aspects of the Macintosh user interface that we haven't yet implemented in PicoApp

The main PicoObjects are the application class *TPicoApp*, the document class *TPicoDoc*, the window class *TWindow*, and the view class *TPicoView*.

PicoApp includes a number of other object classes, most of which we'll look at in later chapters as they come up: *TButton*, which we've already seen (although it's now a subclass of *TPicoView*); *TStatText*; and *TEvtHandler*, which we looked at in Chapter 4.

In Chapter 17, we'll go into Crapgame's subclasses of the PicoApp classes, with an eye toward what Crapgame inherits from each PicoApp class, what Crapgame overrides, and what Crapgame must add. In Chapter 11 we saw, in a general way, how to subclass Crapgame from PicoApp. In Chapter 16, we'll identify what to subclass and go into the details of how to do it.

This chapter is an overview. In subsequent chapters we'll consider the particulars of how PicoApp handles events and maintains the cursor shape, how views are used for user interaction and display, and how the various PicoObjects function together as a system.

Class *TPicoApp* and the Idea of an Application Object

TPicoApp is PicoApp's application object. Some part of a Macintosh application has to handle application-level activities: managing menus, creating and managing new documents and their windows, and handling startup and quitting chores—among other things.

There's no reason, of course, why these activities can't be handled by a conventional main program. In fact, that's exactly how they were handled for the button demonstration in Chapter 5. So what's the value of having a special object class to play the role of application? Inheritance and overriding.

As an object, an application can bequeath many functions to more specialized subclasses, like *TCrapsApp*. *TCrapsApp* can focus on being a Crapgame, leaving the generic functions to the methods it inherits from *TPicoApp*. Even better, *TCrapsApp* can override any of *TPicoApp*'s methods that aren't quite right. It can add services that *TPicoApp* can't possibly anticipate. *TPicoApp* can also provide services, implemented as methods, for *TCrapsApp* and other classes to call upon, such as removing documents from the document list when they close.

For a standard main program, most of those services would be impossible. The only way to extend and modify a standard main program is by direct alteration of its source code. In principle, there's no reason why *PicoApp*, on the other hand, couldn't be already compiled as a library and still be highly extensible without our touching (or even seeing) its source code.

The *TPicoApp* Class Declaration

Listing 12-1 is the declaration of class *TPicoApp*. Note that it's quite a big class, with more than 50 methods. You'll find a fully commented version on the code disk.

```
type
  TPicoApp = object(TApplication)          { See units UPicoApp, UPAEvents, }
                                           {  and UPAMenus for methods }

    { kSizeOfPicoApp = kSizeOfTApp + 42 = 44 bytes }

    fQuitting: Boolean;                    { True if quitting the application }
    fWindow: WindowPtr;                    { Most recently selected by mouse }

    { Document list instance variables }
    fNewestDoc: TPicoDoc;                  { New document to be put in list }
    fDocList: TPicoDoc;                    { Document list }
    fNextDocID: Integer;                   { Next ID number to assign to a }
                                           {  new document }
    fDocCount: Integer;                    { Number of documents in the }
                                           {  document list }
    fDocsAllowed: Integer;                 { Maximum number of documents }
                                           {  application can open }
    fClosingWindowClosesDoc: Boolean;      { True if closing window closes
                                           {  document }
```

Listing 12-1. *(continued)*
The declaration of class TPicoApp.

Listing 12-1. *continued*

```
    fFileTypes: SFTypeList;          { File types application can open }
    fNumTypes: Integer;              { Number of types }
    fAlertPosted: Boolean;           { True if we've told user about low memory }

procedure TPicoApp.IPicoApp (fileTypes: SFTypeList; numTypes: Integer);
function TPicoApp.DocCountOf: Integer;
procedure TPicoApp.MakeMenus;
procedure TPicoApp.AdjustMenus;
procedure TPicoApp.FixMenus;
override;
procedure TPicoApp.FixFileMenu;
override;
procedure TPicoApp.FixEditMenu;
override;
procedure TPicoApp.DoAbout;
procedure TPicoApp.ShowIntro;
function TPicoApp.MakeDocument: TPicoDoc;
procedure TPicoApp.DoNew;
procedure TPicoApp.DoOpen;
function TPicoApp.UniqueDocID: Integer;
procedure TPicoApp.MainEventLoop;
procedure TPicoApp.AdjustCursor;
procedure TPicoApp.DoEachLoop;
procedure TPicoApp.LowMemorySituation (takeSteps: Boolean);
override;
procedure TPicoApp.DoWithLowMemorySituation (takeSteps: Boolean);
procedure TPicoApp.UnloadSegments;
procedure TPicoApp.DispatchEvent (event: EventRecord);
function TPicoApp.DoMenuCommand (theMenu, theItem: Integer): Boolean;
override;
function TPicoApp.DoKeyCommand (event: EventRecord): Boolean;
override;
function TPicoApp.DoMouseCommand (event: EventRecord): Boolean;
override;
procedure TPicoApp.Close (window: TWindow;  doc: TPicoDoc);
procedure TPicoApp.DoBeforeClosing;
procedure TPicoApp.DoAfterClosing;
function TPicoApp.RemoveDoc (whichDoc: TObject): TObject;
override;
procedure TPicoApp.DoBeforeQuitting;
procedure TPicoApp.CloseAllDocs;
```

(continued)

Listing 12-1. *continued*

```
    procedure TPicoApp.Quit;
    function TPicoApp.DoUpdate (event: EventRecord): Boolean;
    procedure TPicoApp.DoWithUpdate (event: EventRecord);
    function TPicoApp.DoActivate (event: EventRecord): Boolean;
    procedure TPicoApp.DoWithActivate (event: EventRecord);
    procedure TPicoApp.DoWithDeactivate (event: EventRecord);
    procedure TPicoApp.Drag (winObj: TWindow; where: Point);
    procedure TPicoApp.DoWithDrag;
    procedure TPicoApp.Grow (event: EventRecord; winObj: TWindow);
    procedure TPicoApp.DoWithGrow;
    procedure TPicoApp.DoMultiFinderEvent (event: EventRecord;
        whichWindow: WindowPtr);
    procedure TPicoApp.DoWithSuspend (event: EventRecord);
    procedure TPicoApp.DoWithResume (event: EventRecord);
    procedure TPicoApp.DeskToPrivateScrap;
    procedure TPicoApp.PrivateToDeskScrap;
    procedure TPicoApp.DoIdle (event: EventRecord);
    override;
    procedure TPicoApp.DAUndo;
    procedure TPicoApp.DACut;
    procedure TPicoApp.DACopy;
    procedure TPicoApp.DAPaste;
    procedure TPicoApp.DAClear;
    procedure TPicoApp.Run;
    procedure TPicoApp.DoApplicationSetup;
    procedure TPicoApp.DoApplicationCleanup;
    procedure TPicoApp.RegisterEvtHandler (eh: TEvtHandler);
    override;
    procedure TPicoApp.RemoveEvtHandler (eh: TEvtHandler);
    override;
    procedure TPicoApp.RegisterIdleHandler (eh: TEvtHandler);
    override;
    procedure TPicoApp.RemoveIdleHandler (eh: TEvtHandler);
    override;
    procedure TPicoApp.RegisterMenuFixer (eh: TEvtHandler);
    override;
    procedure TPicoApp.RemoveMenuFixer (eh: TEvtHandler);
    override;
    function TPicoApp.TypeOf: Str30;
    override;
    { Inherited methods: Free, ShallowFree, Clone, ShallowClone }
    { Also other TEvtHandler methods }
  end;  { Class TPicoApp }
```

Ancestry and Inheritance

TPicoApp is a subclass of *TApplication*, which is a subclass of *TEvtHandler*, which in turn is a subclass of *TObject*, as shown in Figure 12-1.

Thus, *TPicoApp* inherits event-handling methods and methods for cloning and freeing itself. It also overrides some of the methods of its ancestors and adds many methods of its own.

TApplication is a highly abstract class that provides little functionality. It's used because of Pascal scoping rules. Some of the classes that must be declared ahead of *TPicoApp* (such as *TPicoDoc* and *TPicoView*) must themselves refer to the application object for communication. We supply a global variable *gApplication* (of type *TApplication*), which gets initialized in *TPicoApp.IPicoApp*:

```
gApplication := self;                    { Refers to our application object }
```

This gives all PicoApp objects a simple way to refer to the application object as they need to. They can simply send a *TPicoApp* message:

```
gApplication.MethodName(parameters);
```

Depending on what message you're trying to send it, you might sometimes have to typecast *gApplication* (usually to *TPicoApp* or *TYourApp*). *gApplication* is declared as type *TApplication*, so you need to typecast to convince the compiler that *gApplication* has, say, a *TPicoApp* method that *TApplication* lacks, such as *AdjustCursor*.

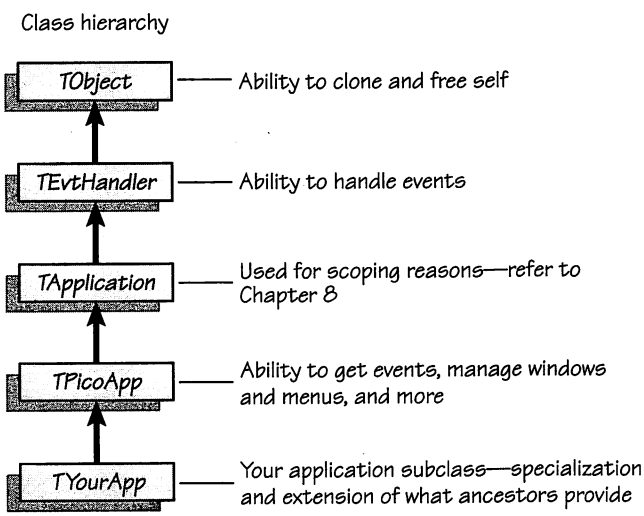


Figure 12-1.
The application object class hierarchy.

What *TPicoApp* Does

As we've seen, the functions of *TPicoApp* include

- Startup activities such as setting up the menu bar
- Event getting and event dispatching
- Document list maintenance
- MultiFinder compatibility
- Desk accessory support
- Numerous default behaviors, overridable if needed

When the short main program we saw in Chapter 10 runs, it simply

- Creates an application object (whose type is a subclass of *TPicoApp*)
- Initializes the application object
- Calls the application object's *Run* method

When *Run* terminates, so does the application. For a given Macintosh application, there will be one and only one application object.

Initialization

We don't directly call method *IPicoApp*. Instead, we call it as part of the initialization method of its subclass—*TCrapsApp.ICrapsApp*, for example. This is standard procedure for initialization methods. Each subclass does its own initialization and also calls its immediate ancestor's initialization method. In this way, initialization works its way up the entire object hierarchy and all the instance variables of the lowest subclass in the hierarchy get initialized properly, including those the subclass inherits. When we write the initialization method of a *TPicoApp* subclass, we must remember to call *IPicoApp* from it, as we've done in *TCrapsApp*. (Note that *ICrapsApp* doesn't call its ancestor's initialization method by using the keyword *inherited*. Because *IPicoApp* has a distinctive name, in contrast to, say, *ICrapsApp*, *TCrapsApp* inherits *IPicoApp* without an override. The handiness of these distinctive names is one reason we don't call our initialization methods "Init.")

IPicoApp first sets the instance variable *fQuitting* to *false*. It then sets up an empty document list. It also sets a number of important global variables such as the *gApplication* variable and the drag rectangle and size rectangle used for windows. It initializes its *TEvtHandler* fields. And it calls *TPicoApp.MakeMenus* to set up the menu bar.

Additionally, *IPicoApp* calls *DeskToPrivateScrap*. If our application uses a private scrap instead of directly using the desk scrap (Clipboard), we must override *DeskToPrivateScrap* so that it knows how to copy the desk scrap into our private scrap initially. (See *Inside Macintosh*, I-451.) That lets our application transfer data in by means of the Clipboard. *DeskToPrivateScrap* is also called by *DoMultiFinderEvent* on a MultiFinder resume event.

The code for *TPicoApp.IPicoApp* is in UPicoApp.f in the folder PicoApp, Part 2 on the code disk.

Menu setup

PicoApp assumes that its resource file contains three MENU resources, one each for the Apple, File, and Edit menus, and an MBAR resource. When we copy these resources into our own application's resource file, we will almost certainly need to edit them using ResEdit. (See Chapter 16 for details.)

TPicoApp.MakeMenus follows the standard process of setting up the menu bar by getting the MBAR resource, enabling or disabling each menu, and finally calling *DrawMenuBar*.

Of course, our application might have other menus besides the ones specified in the MBAR resource. Later, when we look at *TCrapsApp*, we'll see how we would go about adding other menus and getting the application to use them.

The code for *TPicoApp.MakeMenus* is on the code disk.

We'll look more closely at both menu setup and menu maintenance in Chapter 13.

Event Handling

TPicoApp contains quite a few methods related to event handling, including those it inherits or overrides from *TEvtHandler*.

The two most important methods are *MainEventLoop* and *DispatchEvent*, which we'll look at in the next chapter, as we explore how *TPicoApp* is involved in the event-handling process.

Document creation

TPicoApp can't create any documents itself, of course, because it has no way of knowing about our document types. But our subclass of *TPicoApp* (*TCrapsApp*, for example) can override *TPicoApp.MakeDocument* to create a document. Then, when *TPicoApp.DoNew* or *TPicoApp.DoOpen* needs to have a document made, it calls our method, after which it installs the document in its document list. *TPicoApp* doesn't have to know about our document type to be able to manage the document list. We'll look more closely at document creation in the context of our focus on the relationship between *TCrapsApp* and *TPicoDoc*.

Cursor maintenance

TPicoApp provides a simple mechanism that lets a view object adjust the cursor if the cursor comes over the view object's space. On each pass through the event loop, *AdjustCursor* is called. If the cursor is over the front window, *AdjustCursor* sends a DoCursor message to the window, which then sends the same message to its views. We'll go into the details of this mechanism in Chapter 14.

What *TPicoApp* Doesn't Implement

PicoApp is pretty rudimentary compared to MacApp or TCL. We won't have some features of the Macintosh user interface, such as

- Save—for saving documents on disk (See the discussion of *TPicoDoc* later in this chapter.)
- Print—for printing documents (See the discussion of *TPicoDoc* later in this chapter.)
- Undo (But we'll discuss a possible implementation later.)

TPicoApp Hooks

Our implementation code for *TPicoApp* contains a rather large number of stub methods. Why bother to include them?

These methods are "hooks" we can hang something on if we want to. The idea is that we can override any or all of these hook methods in our application subclass in order to add special functionality to *TPicoApp*.

For example, we've provided a *DoEachLoop* method. It gets called once each time through the event loop. *TPicoApp*'s version is a stub because there's nothing we know of to do during the loop that isn't already done by *MainEventLoop*. But, by overriding *DoEachLoop* in our application subclass, we can ensure that special tasks requiring periodic action get executed. We simply put them in our overriding *DoEachLoop*. For example, we could use *DoEachLoop* to call *TEIdle* to blink the insertion pointer—or to let our TextEdit object do that.

In general, *TPicoApp* provides hooks anywhere we think they could potentially be useful someday. There's a *DoWithUpdate* method in case we need to do something extra in conjunction with what the *DoUpdate* method already does. Similarly, there are *DoWithActivate* and *DoWithDeactivate* methods, which actually have predictable uses although we haven't needed to use them in Crapgame. When a window is activated, we might need to highlight some current selection in it or show the insertion pointer for text. Perhaps we'll call the TextEdit routine *TEActivate* from here. Or, when a window is deactivated, we might need to remove the highlight from the selection.

Here's a list of the 20 hook methods in the *TPicoApp* class:

- *DoEachLoop*
- *DoBeforeClosing* to prepare for closing a document
- *DoAfterClosing* to clean up after closing a document
- *DoBeforeQuitting* to prepare for quitting the application
- *DoWithUpdate*
- *DoWithActivate*
- *DoWithDeactivate*

- *DoWithDrag*
- *DoWithGrow*
- *DoWithSuspend*
- *DoWithResume*
- *DAUndo* to fix the Edit menu after a DA handles Undo
- *DACut* to fix the Edit menu after a DA handles Cut
- *DACopy* to fix the Edit menu after a DA handles Copy
- *DAPaste* to fix the Edit menu after a DA handles Paste
- *DAClear* to fix the Edit menu after a DA handles Clear
- *DeskToPrivateScrap* to copy from the Clipboard to a private scrap, if any
- *PrivateToDeskScrap* to copy from a private scrap, if any, to the Clipboard
- *DoApplicationSetup*, called by *Run*
- *DoApplicationCleanup*, called by *Run*

Other PicoApp classes also provide hook methods. Class *TWindow*, for example, has a *DoWithPostGrow* method in case we want to do something useful after a window's size has changed.

Class *TPicoDoc* and the Idea of a Document Object

TPicoDoc is PicoApp's document object. Most Macintosh applications work with documents of some kind. We can always point out applications that don't, but most of them do.

Let's consider, for instance, what happens when we open a word processing document.

When we choose New from the File menu, a window appears with a blinking insertion pointer, waiting for us to enter text for a new document. Then, when we choose Save As, the window's contents are written out to a file.

Later, if we choose Open from the File menu, we can select the file we saved earlier from the standard file dialog box. When we do, the file is read in to become a document in memory. A window is opened to display the document's contents, and we can scroll through it, editing anywhere we like. At any point, we can save the current version of the text, revert to the previous version on disk, or print the current version. When we finish, we close the window, which closes the document and optionally causes its altered contents to be written out to the disk again, replacing the old version.

Clearly, there is a distinct relationship among the disk file, the window, and the data being displayed in the window. Figure 12-2 on the next page shows the relationship of the document to these entities.

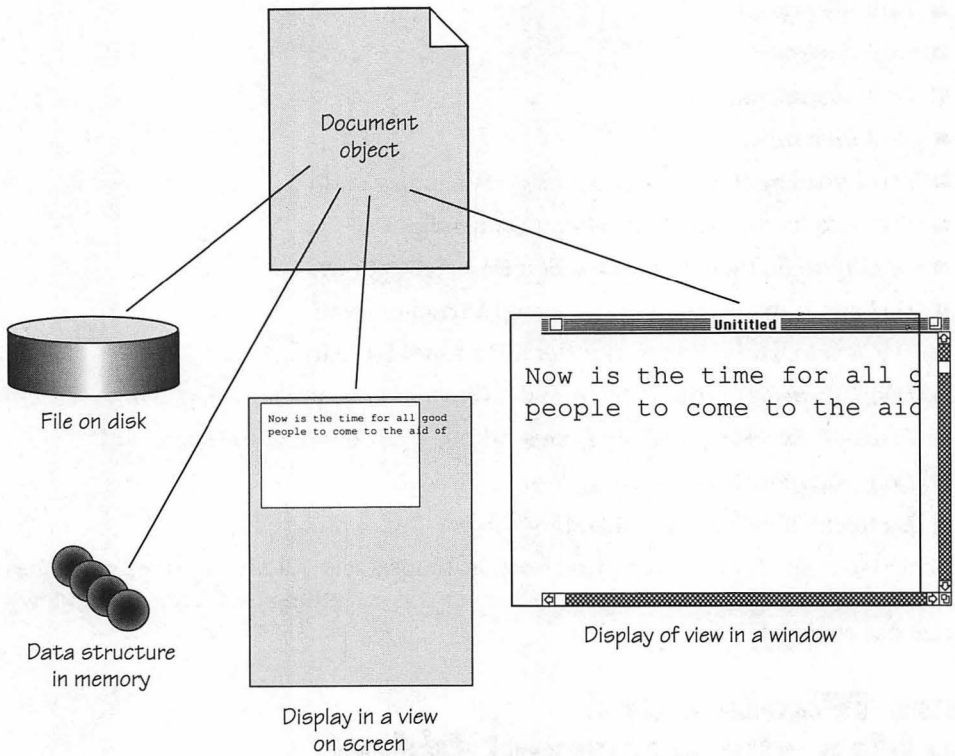


Figure 12-2.
The document object creates and manages the file, the data structures, the view(s), and the window.

In PicoApp terms, we say that the document object "mediates" among the entities. Its job is to handle saving the contents on disk, reading them in, printing them, and causing them to be displayed in a window. Thus, a fully functional document object should have methods for

- Creating a window
- Manipulating the data
- Writing the data to disk
- Reading the data from disk
- Printing the data
- Creating a view or views in which the data is displayed in the window
- Reverting to an older version

The document object is responsible for creating and initializing the window and view objects it needs for displaying its data. And it's responsible for creating and

maintaining the data structures that store the data (such as a *TextEdit* record or a drawing bitmap). In PicoApp, we're so far implementing only part of that functionality.

The *TPicoDoc* Class Declaration

Listing 12-2 is the declaration of class *TPicoDoc*.

```
type
  TPicoDoc = object(TDocument)    { See unit UPADoc for methods }

      { kSizeOfPicoDoc = kSizeOfTDoc + 20 = 38 bytes }

      fWindowClosed: Boolean;     { True if document is open but window }
                                  {  is closed }
      fItsDocID: Integer;         { For searching the document list }
      fCurrentViews: TPicoView;   { Reference to the current view objects }
      fItsWinObj: TWindow;        { The document's window object }
      fWinHasClose: Boolean;
      fWinHasZoom: Boolean;
      fWinHasGrow: Boolean;
      fWindowDirty: Boolean;      { True if user has modified document }
                                  { Views probably set this }

      function TPicoDoc.IPicoDoc: Boolean;
      procedure TPicoDoc.SetDocID (id: Integer);
      override;
      function DocIDOf: Integer;
      procedure TPicoDoc.OpenDoc (fileName: Str255; vRefNum: Integer);
      override;
      procedure TPicoDoc.ReadDoc (fileName: Str255; fileRefNum: Integer);
      function TPicoDoc.MakeViews: Boolean;
      procedure TPicoDoc.SwitchViews (var theViewRect: Rect);
      function TPicoDoc.MakeWindows (hasClose, hasZoom, hasGrow: Boolean):
      Boolean;
      function TPicoDoc.CurrentWindowOf: WindowPtr;
      function TPicoDoc.CurrentWindowObjectOf: TWindow;
      procedure TPicoDoc.FixMenus;
      override;
      procedure TPicoDoc.FixFileMenu;
      override;
      procedure TPicoDoc.FixEditMenu;
      override;
```

Listing 12-2. *(continued)*
The declaration of class TPicoDoc.

Listing 12-2. *continued*

```
    procedure TPicoDoc.RegisterMenuFixer (eh: TEvtHandler);
    override;
    procedure TPicoDoc.RemoveMenuFixer (eh: TEvtHandler);
    override;
    procedure TPicoDoc.Undo;
    procedure TPicoDoc.Cut;
    procedure TPicoDoc.Copy;
    procedure TPicoDoc.Paste;
    procedure TPicoDoc.Clear;
    procedure TPicoDoc.Close (winObj: TWindow; closeDoc: Boolean);
    procedure TPicoDoc.DoBeforeClosing;
    procedure TPicoDoc.FreeViews;
    procedure TPicoDoc.FreeAlternateViews;
    procedure TPicoDoc.Free;
    override;
    function TPicoDoc.TypeOf: Str30;
    override;
  end; { Class TPicoDoc }
```

Ancestry and Inheritance

TPicoDoc descends from *TDocument*, which descends from *TEvtHandler* and, in turn, *TObject*, as shown in Figure 12-3.

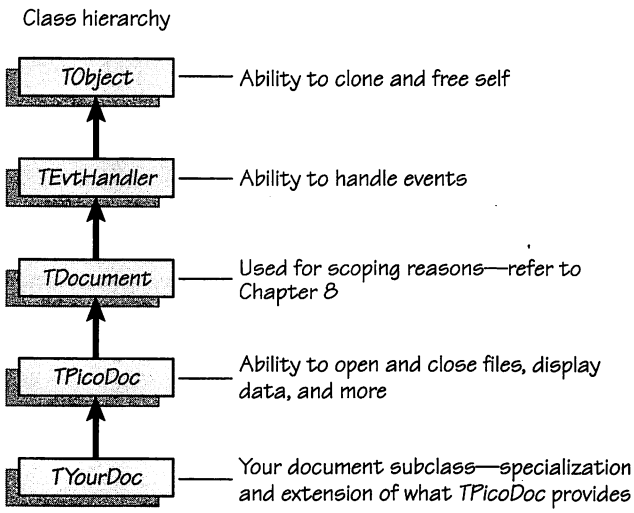


Figure 12-3.
The document object class hierarchy.

Thus, *TPicoDoc* inherits event-handling methods and *Clone* and *Free* methods. *TDocument*, like *TApplication*, is an abstract class used for Pascal scoping reasons. Several classes that must be declared above *TPicoDoc* must nevertheless refer to document objects for communication (for example, *TWindow* and *TPicoView*). By declaring *TDocument* above these classes, we make it possible for them to refer to an instance variable such as

```
fItsDocument: TDocument;
```

Then, given polymorphism and runtime binding, we can assign a *TPicoDoc* object to that variable at runtime.

What *TPicoDoc* Does

As we've seen, the functions of *TPicoDoc* include

- Creating a window
- Installing a list of views in the window
- Telling the window and views to free themselves on closing
- Other activities that we aren't implementing now (reading from and writing to disk, printing)

Some of the document functions must be handled by a subclass of *TPicoDoc*, such as *TCrapGame*. In particular, *TPicoDoc* can't know what views are needed to display the document's data. So our document subclass must override *MakeViews* to make the views and link them in a simple linked list. Then *TPicoDoc*'s *MakeWindows* can install the views in the window object it has created. We'll look at view creation in more detail in Chapter 15.

Initialization

When a document object is initialized, it must call *IPicoDoc*, register itself as an event handler (if desired), and create its window(s) and views.

As we've seen, any class's initialization method should always call the initialization method of its immediate ancestor. This ensures that all instance variables get properly initialized.

If we want our document object to receive events from the application, we can call

```
gApplication.RegisterEvtHandler(self);
```

from our document subclass's initialization method, which adds the document to the chain of event-handling objects. Of course, some document objects might not need to be event handlers (even though they descend from *TEvtHandler*). If we do have a document register itself as an event handler, we must be sure before freeing the document to have it remove itself from the chain by calling

```
gApplication.RemoveEvtHandler(self);
```

Calling *MakeWindows* results in the creation of whatever window(s) the document needs and a call to *MakeViews*.

Making windows

PicoApp provides facilities for creating windows via the *TWindow* object we'll look at in the next section of this chapter. Making windows is a complicated process, although PicoApp does most of it behind the scenes. Basically, though, we instantiate a *TWindow* object and tell it what kind of Macintosh window to put on the screen.

Our first step as application programmers is to override *TPicoDoc.MakeWindows* to make our window(s). We make our document's *TWindow* object(s) by calling either of the utility functions *NewSimpleWindow* and *NewSpecialWindow* (defined in the unit *UPAIntf*).

NewSimpleWindow creates a *TWindow* object and calls the first of *TWindow*'s two initialization methods, *ISimpleWindow*, which in turn calls the utility routine *MakeSimpleWindow* to make an actual standard Mac window. (The window can have a size box, a zoom box, and a close box.) The *WindowPtr* value for the Mac window is placed in the *TWindow* object's *fWindow* instance variable.

Note that class *TWindow* has two different initialization methods, so you can initialize a *TWindow* object in either of two ways: with a simple Mac document window (a default with only a few variations that you can specify) or by calling the utility function *NewSpecialWindow*, whose parameters let you specify everything about the window according to your needs. Giving a class multiple initialization methods is a good way to make it flexible and gives you an alternative to passing lots of parameters.

The Mac window made by using *NewSimpleWindow* is titled "Untitled-*n*" and appears in a designated rectangle in the desktop. A window's title shows the ID number "*n*" of its document. We can rename the window by calling the Toolbox procedure *SetWTitle* in our *MakeWindows* override, as we've done in Crapgame.

Our *MakeWindows* override must also call *inherited MakeWindows* so that *TPicoDoc*'s method can create our document's views.

By the way, we haven't considered windows with scrollbars. That's because scrollbars are going to be treated as special objects in PicoApp—they're views, although so far we haven't implemented them, and they're attached not to a window but to another view within a window. To set up a normal scrolling window, we'd create a scrolling view with one or two scrollbar objects attached to it and install the whole thing in the window. This approach also makes it easy to set up scrolling panes within a window. See the next section for more on this topic.

It's possible that our application might need to open two or more windows for a single document. For instance, one window might display the document's data, and a second window might display statistical information about the document. The possibilities are endless, although most documents do nicely with one window. If a document needs multiple windows, the required override of *TPicoDoc.MakeWindows* can use PicoApp's *NewSimpleWindow* or *NewSpecialWindow* routine to make as many windows as we need, of whatever kinds. We'd also need to make

some changes in our *TPicoDoc* subclass to accommodate storing and dealing with two (or more) *TWindow* objects for the same document.

This illustrates a principle common to MacApp and TCL as well as to PicoApp. The application framework accommodates standard behaviors as a matter of course. At the same time, it doesn't prohibit nonstandard behaviors—but we have to work considerably harder to accomplish them.

Making views

Some documents need only one view. Others might need several. In particular, if a document is too large to display completely in the window, as most are, we'll also need a "scroller" object and one or more scrollbars. We're treating these objects as views in PicoApp, as they are in both MacApp and TCL. A scroller object would translate among the coordinate systems of the window, the document, and another view that contains the data. We can visualize a scroller as a transparent sheet overlying most of the content region of a window. It might actually be the view that displays what the user types or draws. But it's a special subclass of the usual view, one that happens to know how to scroll the data and to interact with scrollbar views. For now, we're leaving scrolling as a project.

After creating views, the document passes a reference to its view list to the window object, which handles view display and other chores.

What *TPicoDoc* Doesn't Implement

At this point, we've done nothing to implement printing, saving on disk, reading from disk, or reverting. We should be able to generalize printing so that PicoApp can do it for us, making use of a view's ability to draw itself and having it do so into a printer port instead of into an ordinary window. Writing to and reading from disk will require that the document subclass override those methods because different documents use different file formats, but it might be possible to generalize some of those tasks. For an instructive discussion of this point, see the chapter "Input/Output with Streams" in *Programming with MacApp* (Wilson 1990). Crapgame doesn't need these features, so we're saving time and space by not implementing them.

TPicoDoc Hooks

Like *TPicoApp*, *TPicoDoc* provides several hook methods for us to override if we need to implement them.

Not all stub methods are really hooks. We generally define a hook as a stub method we can choose to override or not. *MakeViews* is a stub method, but we must override it if our application uses any views—which almost every application does. Some of the hooks we list on the next page are actually methods we haven't gotten around to implementing yet. We left them as hooks to make it easier to implement them later.

Here's a list of the *TPicoDoc* hooks:

- *DoBeforeClosing* for anything that needs to be done before the document closes, such as putting up a dialog box or saving the file.
- *SwitchViews* for some documents, such as *TCrapGame* objects, that might have two or more alternative lists of views. In *TCrapGame* we'll override this method to switch the views on command.
- *FreeAlternateViews* for alternate view lists. PicoApp takes care of freeing the standard view list, but we'll need to override this method in order to free alternate view lists.
- *ReadDoc* for reading in our document's data from disk.
- *Undo* for implementing some kind of undo.
- *Cut*, *Copy*, *Paste*, *Clear* for implementing Clipboard support.

Class *TWindow* and the Idea of a Window Object

Class *TWindow* is PicoApp's window object. *TWindow* is actually a subclass of *TPicoView* because it's an object that draws itself on the screen. *TWindow* has little else in common with other views because of the way it's used and its extra powers.

Basically, a window is a place to display other views, so, besides creating a Mac window frame, its main task is to maintain a view list (given to it by its document) and to pass view-related messages to the views. Among other tasks, the window gets update and activate events from the application and passes them to its views so that they can respond by drawing themselves. Similarly, the application uses the window to pass cursor-changing messages to its views. The window also manages normal window actions such as sizing, zooming, and dragging; and it lets its views know about those events in case they need to resize themselves or do something else useful at the time of the event.

Because the Macintosh implements windows with a Pascal record of type *WindowRecord*, which is pointed to by a variable of type *WindowPtr*, a *TWindow* object naturally contains a *WindowPtr*-type instance variable. However, in an interesting twist inspired by TCL, we also install a reference to the *TWindow* object inside the Window Manager's *WindowRecord* variable. *WindowRecord* variables come with a handy 4-byte field called a *refCon*, which is provided for the use of programmers. We can install anything we like in the *refCon* field, so, because an object reference is 4 bytes long, we install our *TWindow* there. Thus, the *WindowRecord* variable points to the *TWindow* object, and vice versa. This has lots of handy consequences, as we'll see in Chapter 15.

The *TWindow* Class Declaration

Listing 12-3 is the declaration of class *TWindow*. It has nearly 30 methods.

```
type
  TWindow = object(TPicoView)    { See unit UPAWind for methods }

      { kSizeOfWinObj = kSizeOfViewObject + kSizeOfMacWindow }
      {  + 10 = 2266 bytes }
      { Inherits view fields and methods }
      { fItsWindow (type TPicoView) not used here }
      { fViewRect is the Mac window's port Rect }

      fWindow: WindowPtr;          { Actual window managed by this object }

      { This window's characteristics }
      fHasGrow: Boolean;           { True if window has size box }
      fHasZoom: Boolean;           { True if window has zoom box }
      fHasClose: Boolean;          { True if window has close box }

      procedure TWindow.ISimpleWindow (itsDoc: TDocument; hasGrow, hasZoom,
          hasClose: Boolean; itsRect: Rect);
      procedure TWindow.ISpecialWindow (itsDoc: TDocument; winType: Integer;
          custom: Boolean; ownStorage: Ptr; title: Str255; behind: WindowPtr;
          bounds: Rect; refCon: Longint; visible, hasGrow, hasZoom,
          hasClose: Boolean; itsRect: Rect);
      procedure TWindow.SetViews (v: TPicoView);
      function TWindow.ViewsOf: TPicoView;
      function TWindow.HasSizeBox: Boolean;
      procedure TWindow.Display;
      override;
      procedure TWindow.Hide;
      procedure TWindow.DisplayViews;
      function TWindow.DoCursor (where: Point): Boolean;
      override;
      function TWindow.WindowPtrOf: WindowPtr;
      function TWindow.PortRectOf: Rect;
      function TWindow.DocumentOf: TDocument;
      function TWindow.DoUpdate (event: EventRecord): Boolean;
      function TWindow.DoActivate (event: EventRecord): Boolean;
      override;
      procedure TWindow.DoWithPreGrow (newSize: Longint);
      procedure TWindow.DoWithPostGrow (newSize: Longint);
      procedure TWindow.DoPreGrow (newSize: Longint);
      procedure TWindow.DoPostGrow (newSize: Longint);
      procedure TWindow.Grow (event: EventRecord; partCode: Integer;
          zooming: Boolean);
      override;
```

Listing 12-3. *(continued)*
The declaration of class TWindow.

Listing 12-3. *continued*

```
    procedure TWindow.GrowSubviews (event: EventRecord; partCode: Integer);
    procedure TWindow.Zoom (event: EventRecord; whereHit: Integer);
    procedure TWindow.Close;
    procedure TWindow.Drag (where: Point);
    procedure TWindow.Move (p: Point; front: Boolean);
    procedure TWindow.DoWithResume (event: EventRecord);
    procedure TWindow.DoWithSuspend (event: EventRecord);
    function TWindow.TypeOf: Str30;
    override;
  end;  { Class TWindow }

  { Two nonmethod functions to return different kinds of window objects }
  function NewSimpleWindow (itsDoc: TDocument; hasGrow, hasZoom,
        hasClose: Boolean; itsRect: Rect): TWindow;
  function NewSpecialWindow (itsDoc: TDocument; winType: Integer;
        custom: Boolean; ownStorage: Ptr; title: Str255; behind: WindowPtr;
        bounds: Rect; refCon: Longint; visible, hasGrow,
        hasZoom, hasClose: Boolean): TWindow;
```

Ancestry and Inheritance

TWindow is a subclass of *TPicoView* and thus also of *TEvtHandler* and *TObject*, as shown in Figure 12-4.

TWindow inherits all its ancestors' instance variables and methods, so we should seldom, if ever, have to subclass *TWindow*. (Someday we might subclass it to get a class of color windows.)

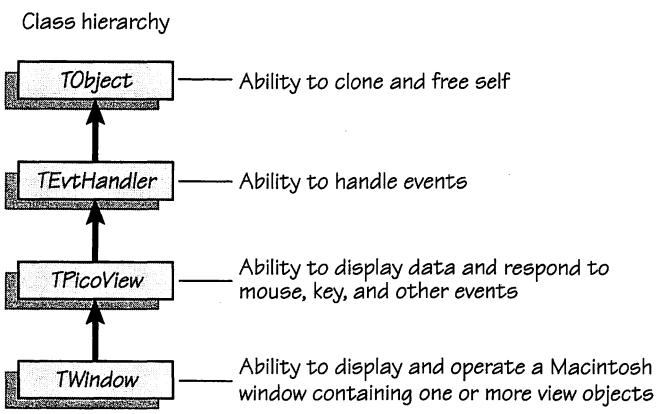


Figure 12-4.
The window object class hierarchy. You'll seldom need to subclass TWindow.

What *TWindow* Does

As we've seen, the functions of *TWindow* include

- Creating a Macintosh window
- Managing the document's view list
- Sizing, zooming, dragging, moving, activating, updating
- Letting the views know of the events that affect the window

Managing the view list

When the document creates its window object, it also creates a list of view objects. This is a simple linked list, with each view object pointing to the next through its *fSubView* instance variable. The document calls *TWindow.SetViews* to install a reference to the view list in the window object.

From there on, the window manages that list. In particular, as events affecting the window and its views come in, they are directed to the window, which passes them on to the views as needed.

Much of the window's response to events, such as redrawing the frame as needed, is done automatically by the Window Manager. But if the user drags the window, changes its size, or otherwise affects it, the *TWindow* object is called to carry out the chores (often by calling Window Manager routines).

The two most important window/view events are updates and activates. A change in a window's contents generates an update event. The application gets the update event, checks its event record to see which window the update is for, gets a reference to the Macintosh window record's *TWindow* object, and sends the window a DoUpdate message. *DoUpdate*, in turn, sends each view object in the view list a Display message. Drawing is actually clipped to what really needs redrawing, of course—this is the default way for views to display themselves. Instead of explicitly sending them Display messages someplace, we simply wait for an update event. Here and there in Crapgame, we force an update by calling *InvalRect* on some part of a view when its contents change. And, in a few places in Crapgame, we draw directly rather than wait for an update—but only when speed is critical.)

When a window becomes the new front window, or when the old front window is displaced by a new one, one window gets an activate event and the other gets a deactivate event. The window frames typically reflect activation or deactivation by changing their appearance somewhat. In some cases, the views inside a window also need to make changes as their window activates or deactivates. For example, a view displaying text with some text selected would need to highlight the selection on an activate and unhighlight it on a deactivate. An activate/deactivate event causes the window object's *DoActivate* method to be called. *DoActivate* notifies the views of the event by sending them a DoActivate message. Views that need to respond to those events can do so; other views can ignore the event.

The views might also be affected when the window's size changes because of a grow or zoom event. Some views don't respond to these events, but others might need to grow or shrink proportionally as the window does. In these cases, the window object sends its views Grow messages so that they can adjust themselves as needed.

What *TWindow* Doesn't Implement

TWindow is actually a pretty complete representation of a Macintosh window. It has instance variables for referring to its document, to the Macintosh window record, and to its view list. It also has Boolean flag fields to indicate whether the window has a size box, a zoom box, and a close box. When we create the window object, we set these flags by passing a Boolean parameter for each to the initialization method, so there isn't much left to be implemented.

TWindow Hooks

TWindow's hook methods let us customize certain window actions. Here's a list of the *TWindow* hooks:

- *DoWithResume* to do anything we want when a MultiFinder resume event occurs
- *DoWithSuspend* to do anything we want when a MultiFinder suspend event occurs
- *DoWithPreGrow* and *DoWithPostGrow* to take our own actions while expanding or shrinking a window

Class *TPicoView* and the Idea of a View Object

Class *TPicoView* is PicoApp's view object. We can define a view most broadly as "any distinct visible entity." Views usually appear inside windows' content regions, but we can foresee the use of views on the desktop—we might sometime want to "minimize" an application window, for instance, turning it temporarily into an icon on the desktop as in the NeXT computer's NextStep operating system. (Of course, we'll have to keep MultiFinder in mind.)

Distinct visible entities include all sorts of things: graphics, text strings, buttons, scrollbars, pop-up windows, sliders, dials, icons—even windows themselves. In the extreme, we could even count the menu bar as a view, although in PicoApp we won't treat the menu bar as a view, or even as an object. (But see THINK's Class Library.) Because buttons are views, our *TButton* classes are declared as subclasses of *TPicoView*. And PicoApp comes with one built-in *TPicoView* subclass: *TStatText*, representing a displayable text string. Decorations such as drop shadows and borders can also be views. (See TCL's *CBorder* class.)

In TCL, each view has its own grafPort to draw into, and the coordinates for what it draws are measured in terms of that grafPort rather than in terms of the window. This is a good approach for implementing separate window panes such as Microsoft Word's split-paned windows or THINK Pascal's LightsBug window. TCL even has a view subclass called *CPane* and a subclass of that with scrolling capabilities. A future version of PicoApp will probably have something similar.

MacApp, on the other hand, uses the window's grafPort for drawing. Before doing the drawing for a particular view, you call a *Focus* method, which changes the origin of the grafPort to coincide with the upper left corner of the view. When drawing is over, the origin is reset to where it was.

In PicoApp, we've adopted a simpler view model. A view in PicoApp is simply a graphical entity of some kind, and we choose to display it in a rectangle relative to the window's coordinate system.

Views have a variety of properties, but the two most important are the way in which they display and the way in which they handle events.

Naturally, the most crucial thing a view object does is to display itself in its designated view rectangle in a window. *TPicoView* can't foresee what our views need to draw, of course, so its *Display* method is a stub and we must subclass *TPicoView* and override *Display* to implement the display aspect of our views. Views should probably also be able to hide themselves. And they should be able to resize themselves as needed to fit the window.

A great deal of the user interaction with Macintosh applications is done through views—clicking buttons, editing text, dragging scrollbar thumbs, and so on. Therefore, views come equipped to handle mouse and other events, although not every view object will necessarily do so. Views inherit a *DoKeyCommand* method from *TEvtHandler* in case they need to take keyboard input. An example of a view that takes keyboard input would be a TextEdit view into which the user types. (Recall our "key-eater" object from Chapter 5.) Views also inherit a *DoMouseCommand* method. They add to it a *Clicked* method to determine whether the user has clicked inside a particular view (and how many times if so) and a *TrackMouse* method to track the mouse in and out of the view, highlighting or adjusting it as required. If a view is clicked, its *DoMouseCommand* method then sends a DoClick, a DoDoubleClick, or a DoTripleClick message. In *TPicoView*, the *DoClick*, *DoDoubleClick*, and *DoTripleClick* methods are stubs that we must override in our view subclass to specify what the view does when clicked once, twice, or three times.

Views also know how to change the shape of the cursor as it passes over their territories.

Views are an extremely important element of what a Macintosh application does, and much of our development time and effort in a PicoApp (or MacApp or TCL) application will go toward developing views. But *TPicoView* implements most of the more important—and standard—aspects of views for us.

The *TPicoView* Class Declaration

Listing 12-4 is the declaration of class *TPicoView*. It has more than 20 methods.

```
type
  TPicoView = object(TEvtHandler)     { See unit UPAViews for methods }

      { kSizeOfViewObject = kSizeOfEvtHandler + 50 bytes = 64 bytes }

      fItsWindow: TPicoView;          { Window the view is attached to }
      fItsDoc: TDocument;             { Document it belongs to }
      fSubview: TPicoView;            { View--or chain of views--owned by }
                                      {  this view }
      fViewRect: Rect;                { This view's display rectangle }
      fClickRect: Rect;               { This view's clickable area }
      fGrowthRight: Integer;          { Greatest size if window grows }
      fGrowthBottom: Integer;
      fCursor: CursHandle;            { Cursor to show over the view if change }
                                      {  desired }
      fIsCursorAdjustor: Boolean;     { True if view needs cursor adjustment }

      { Mouse-click variables--default to false in IPicoApp, so reset }
      {  variables in your subclass's initialization method after calling IPicoApp }
      fWantsClicks: Boolean;          { True if a single mouse click has meaning }
                                      {  to the view }
      fWantsDoubleClicks: Boolean;    { True if a double click has meaning to }
                                      {  the view }
      fWantsTripleClicks: Boolean;    { True if a triple click has meaning to }
                                      {  the view }
      fCloseEnough: Integer;          { How many pixels apart clicks of a }
                                      {  double click can be }
      fOldMouse: Point;               { Last position of mouse }

      procedure TPicoView.IPicoView (r: Rect; itsDoc: TDocument;
          wantsEvents: Boolean);
      procedure TPicoView.SetViewRect (r: Rect);
      procedure SetGrowthProperties (right, bottom: Integer);
      procedure TPicoView.SetSub (sub: TPicoView);
      procedure TPicoView.SetWindow (w: TWindow);
      function TPicoView.RectOf: Rect;
      function TPicoView.SubviewOf: TPicoView;
      procedure TPicoView.Display;
      procedure TPicoView.DoBeforeGrow (event: EventRecord;
          partCode: Integer; zooming: Boolean);
```

Listing 12-4. *(continued)*
The declaration of class TPicoView.

Listing 12-4. *continued*

```
    procedure TPicoView.Grow (event: EventRecord; partCode: Integer;
        zooming: Boolean);
    procedure TPicoView.DoAfterGrow (event: EventRecord; partCode: Integer;
        zooming: Boolean);
    function TPicoView.DoActivate (event: EventRecord): Boolean;
    function TPicoView.DoCursor (where: Point): Boolean;
    procedure TPicoView.Hilite (hiliteYes: Boolean);
    procedure TPicoView.DoBeforeTrackingMouse (whereNow: Point);
    procedure TPicoView.DoWhileTrackingMouse (whereNow: Point);
    procedure TPicoView.DoAfterTrackingMouse (whereNow: Point);
    function TPicoView.TrackMouse (var time: Longint; var place: Point): Boolean;
    function TPicoView.Clicked (event: EventRecord; var clicks: Integer): Boolean;
    function TPicoView.DoMouseCommand (event: EventRecord): Boolean;
    override;
    procedure TPicoView.DoClick (event: EventRecord);
    procedure TPicoView.DoDoubleClick (event: EventRecord);
    procedure TPicoView.DoTripleClick (event: EventRecord);
    function TPicoView.TypeOf: Str30;
    override;
    function TPicoView.GenericTypeOf: Str30;
    override;
  end;  { Class TPicoView }
```

Ancestry and Inheritance

TPicoView is descended from *TEvtHandler* and thus also from *TObject*, as shown in Figure 12-5, and inherits their fields and methods.

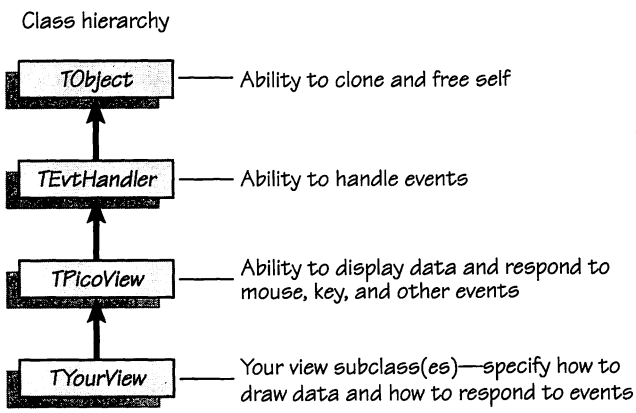


Figure 12-5.
The view object class hierarchy.

What *TPicoView* Does

As we've seen, the functions of TPicoView include

- Displaying some graphic entity on the screen
- Adjusting the cursor, if desired
- Handling events such as mouse clicks and keypresses

Display

In PicoApp, views display themselves inside a view rectangle in local window coordinates. Unlike TCL, PicoApp doesn't (yet) give each view object its own grafPort to draw into and doesn't change the origin of the grafPort.

Because display depends on the nature of our application and our data, we must subclass *TPicoView* for each of our views and override its *Display* method.

Cursor adjustment

If any of our views needs to change the shape of the cursor when it's over that area, it can do so by installing a cursor resource at initialization. Each time through the event loop, the current front window's views are polled to see whether they want to change the cursor. This lets the views test where the mouse is at frequent intervals. If a view finds that the mouse is in its airspace, PicoApp will change the cursor to the one stored in the view.

Event handling

By letting the application know that it wants events, a view can handle mouse clicks, keypresses, and other events. To get the view to handle the events, we must override one or more of the handler methods *DoClick*, *DoDoubleClick*, and *DoTripleClick*.

What *TPicoView* Doesn't Implement

The most important feature not implemented is a subclass of *TPicoView* that knows how to scroll data. We'll probably need at least two subclasses, to handle graphics and text. And we haven't implemented scrollbars for our windows yet, either.

TPicoView Hooks

TPicoView has six real hook methods:

- *DoBeforeTrackingMouse* to set clipping and so on for drawing or selecting
- *DoWhileTrackingMouse*, useful for drawing, selecting, and so on
- *DoAfterTrackingMouse* for cleanup after drawing or selecting
- *DoClick* to specify how to handle a single click
- *DoDoubleClick* to specify how to handle a double click
- *DoTripleClick* to specify how to handle a triple click

TPicoView also has a number of unimplemented stub methods that we might or might not want to override:

- *Grow*, *DoBeforeGrow*, and *DoAfterGrow* to resize the view to match the window
- *DoActivate* to activate or deactivate the current selection, and so on
- *Hilite* to specify what our view does as the mouse is tracked in and out of it

Summary

In this chapter, we briefly surveyed the four main object classes of PicoApp: *TPicoApp*, the application object; *TPicoDoc*, the document object; *TWindow*, the window object; and *TPicoView*, the view object.

We'll go deeper into the heart of these classes in the next few chapters. Meanwhile, this chapter should serve as a quick reference and overview of the classes.

Projects

- Try your hand at implementing one or more of the missing elements of class *TPicoDoc—Save*, *Print*, and *Undo*. (We'll explore some of these methods in more detail later.)
- Try your hand at implementing generalized file read/write operations for *TPicoDoc*. Obviously, you can't implement the whole thing, since different applications read/write different data types and use different file structures. But you can perhaps provide some support for what your document subclasses need to do. Hint: Look at what MacApp or TCL does. Or read Chapter 22 of *Programming with MacApp* (Wilson 1990).
- Try your hand at implementing a *TPicoView* subclass that can scroll graphics. And one that can scroll TextEdit text. Also implement a scrollbar object that can manage a scrollbar (horizontal or vertical) and report its setting as requested by your scroller object.

13

INSIDE PICOAPP: THE APPLICATION

What's in This Chapter

Now that we've taken an overview of PicoApp and its four main classes, let's dive inside to see how PicoApp will handle a good deal of the Macintosh user interface for us. We'll cover

- Application startup
- Menus
- Event handling
- Application shutdown

In the next chapter, we'll cover more application-level topics.

Application Startup

In PicoApp, the functions of the application object are divided into two phases: After creating a new application object in the standard Pascal main program, we call its initialization method; after the object is initialized, we call its Run method.

In Chapter 11, we noted that the actual runtime application object is an instance of some subclass of *TPicoApp*—such as *TCrapsApp*. The application object inherits all of *TPicoApp*'s methods except those that its subclass overrides. Any application subclass naturally has its own *ISomethingApp* method for initialization, and part of what that method does is to call *IPicoApp* for general initializations. *IPicoApp* initializes the application object's *fQuitting* field to *false*; creates a new, empty document list; initializes a number of global variables; calls the initialization method of *IPicoApp*'s own superclass, *TApplication*; and sets up the application's menus.

The call to *TPicoApp*'s *MakeMenus* really calls the subclass's version of the method (if any), which overrides *TPicoApp*'s. If your application has menus other than the Apple, File, and Edit menus, you do need to override *MakeMenus*. Your *MakeMenus*

should begin its work by calling *inherited MakeMenus*, which calls *TPicoApp*'s version to do its basic Apple, File, and Edit menus setup. Then your version of the method does some housekeeping chores for the other menus your application needs.

OOP code frequently uses these kinds of method layering:

- A subclass's initialization method almost always calls the initialization method of its immediate ancestor, which calls the initialization method of its immediate ancestor, and so on up the hierarchy. This ensures that all instance variables get properly initialized.
- You subclass and override a method, but your overridden version at some point calls the version it overrides with the *inherited* keyword. This is a way to build upon, or modify, what a method does. Our immediate example is *MakeMenus*.

Menu Setup

PicoApp knows about its three standard menus—Apple, File, and Edit. The PicoApp way, after all, is to know about standard, general things. Your subclasses add the particular, nonstandard things. PicoApp expects your application's resource file to contain MENU resources for the three standard menus. Of course, the versions of these resources supplied in the file PicoApp.rsrc are "average" menus, containing the items that most Mac applications use. You'll have to use ResEdit to edit your copies of these menu resources in order to add, remove, or rename any of the menu items. In Crapgame, for example, we don't need a Print item in the File menu, so we've removed it from Crapgame's copy of the File MENU resource.

With a little trickery, though, PicoApp can set up the menu bar not only with the menus it knows about—Apple, File, and Edit—but also with any menus your application adds besides those. PicoApp accomplishes this trick (as TCL does) by requiring you to provide not only MENU resources for all menus other than the Apple, File, and Edit menus but also an MBAR resource that contains the IDs of the MENU resources in your application's resource fork. With that resource information, PicoApp can build the menu bar and display it.

Of course, PicoApp doesn't thereby know anything about the details of your menus. It has information for processing commands from the Apple, File, and Edit menus, but you have to add menu-item constants to unit UMyGlobals (see the next section). You also have to obtain handles to your own menus. You get handles to your own menus by overriding *TPicoApp.MakeMenus*. You can extract your menu handles from the menu list that PicoApp has already set up. Then your menu-handling code can use the handles to process mouse clicks in your menus.

TPicoApp.MakeMenus looks like this:

```
procedure TPicoApp.MakeMenus;
  var
    theMenuBar: Handle;
```

```
begin
  { Get the application's MBAR resource and make a menu list }
  theMenuBar := GetNewMBar(kMenuBarID);
  if theMenuBar <> nil then
    SetMenuBar(theMenuBar);
  { Else error }

  { Get handles to individual menus and add desk accessories to Apple menu }
  gAppleMenuHandle := GetMHandle(kAppleMenuID);
    { Get reference to Apple menu }
  AddResMenu(gAppleMenuHandle, 'DRVR');
  gFileMenuHandle := GetMHandle(kFileMenuID);  { References to other menus }
  gEditMenuHandle := GetMHandle(kEditMenuID);
  EnableItem(gFileMenuHandle, 0);
  DisableItem(gEditMenuHandle, 0);
  RedrawMenus;

  { Note:  Subclasses should call "inherited MakeMenus" first }
  { Then they should get handles to the menus it knows about }
end;  { TPicoApp.MakeMenus }
```

This code relies on there being an MBAR resource with ID = *kMenuBarID* (which equals 1) in your application's resource fork. It gets the MBAR, uses it to get all the menus the MBAR refers to, installs the menus in a menu list, and makes that list the current menu bar. The rest is housekeeping. *MakeMenus* calls *GetMHandle* to extract handles to the individual menus from the menu list so that you can refer to the menus in various ways later. The code also calls *AddResMenu* to add all desk accessory resources (of type DRVR) to the Apple menu. Then it enables the File menu commands, disables the Edit menu commands, and draws the menu bar. Notice that a special routine called *RedrawMenus* is called to draw the menu bar. To reduce flicker, it waits for the Mac's vertical retrace period before calling *DrawMenuBar*.

Crapgame doesn't need more than the standard menus, so it doesn't override *TPicoApp*'s *MakeMenus*. (Neither does PicoSketch, the PicoApplication we'll develop later.) But here's what such an override might look like:

```
procedure TYourApp.MakeMenus;
  override;
begin
  inherited MakeMenus;          { Do this first }
  gToolMenuHandle := GetMHandle(kToolMenuID);
  gFooMenuHandle := GetMHandle(kFooMenuID);
  gStyleMenuHandle := GetMHandle(kStyleMenuID);
  { Do anything else, such as adding FONT resources to a Font menu }
end;  { TYourApp.MakeMenus }
```

The override first calls *TPicoApp.MakeMenus* (using *inherited*) as required. Then it gets reference handles to your other menus.

Item number constants

PicoApp defines a group of integer constants for its Apple, File, and Edit menus:

```
{ Menu item numbers:  Some of these might need changing for particular }
{  programs as the number and positioning of items in the File menu }
{  change }
{ Legend:  PA = PicoApp; CG = Crapgame }

const

  kAboutItem = 1;          { PA: Apple menu--only item number needed }
  kNewItem = 1;            { PA: File menu--standard item }
  kOptionsItem = 2;        { CG: New Options item in Crapgame File menu }
  kCloseItem = 3;          { PA: standard item, but number can change }
  kQuitItem = 4;           { PA: standard item, but number can change }
  kOpenItem = 0;           { Needs a number if menu has an Open item }
  kSaveItem = 0;           { Needs a number if menu has a Save item }
  kSaveAsItem = 0;         { Needs a number if menu has a Save As item }
  kPageSetupItem = 0;      { Needs a number if menu has a Page Setup item }
  kPrintItem = 0;          { Needs a number if menu has a Print item }
  kUndoItem = 1;           { PA: Edit menu--standard items }
  kCutItem = 3;
  kCopyItem = 4;
  kPasteItem = 5;
  kClearItem = 6;
```

Some of these numbers are standard, especially the Apple menu's About item and the Edit menu items. But the File menu items might have to be renumbered to reflect the structure of your File menu. For example, File-Quit is originally numbered 3, but if you add your own items, as we did above, this number might change. We've added a *kOptionsItem* specifically for Crapgame, numbering it 2, which requires renumbering some of the other items. The number scheme corresponds to the order and number of items in the actual MENU resource PicoApp reads in.

Note that *TPicoApp*'s code refers to all of these identifiers, so they must be defined even if your application's menus don't contain all of these items. To deal with items that aren't used, we set their values to *0* here.

These definitions are therefore placed in the unit *UMyGlobals*, which your application must supply. You can copy PicoApp's *UMyGlobals*, edit it, and replace PicoApp's version with your own in your application project. PicoApp's *UMyGlobals* is in a file called UMyGlobals.p/PA.

Menus from Other Objects

Normally, the application object does all the menu making. But in PicoApp, other objects, such as documents or views, can put up their own menus in addition. An object simply gets and inserts its menu into the menu bar and redraws the menu bar.

Menu Updating ("Fixing")

Once menus are up, an application needs a mechanism for keeping the right menus and menu items enabled or disabled as program conditions change. PicoApp's menu-updating (or "menu-fixing") mechanism resembles MacApp's and TCL's.

Setting Up the Menu Bar

When the user clicks the mouse in the menu bar, PicoApp immediately disables the File and Edit menus—all items. At the same time, it calls the *AdjustMenus* method of your application subclass to disable the other menus it knows about. At the end of this process, all menus are disabled except for the Apple menu. The idea is to disable everything and then let certain objects reenable what's pertinent to their needs at the moment. By the time the menu is actually pulled down to show its items, the appropriate items are enabled. Although the application object must communicate with several other objects to get the menu-fixing job done, it takes surprisingly little time. Normally, the user won't notice any delay.

Having disabled everything, the *AdjustMenus* method sends the application object a FixMenus message. This invokes the *FixMenus* method, which examines the environment and reenables selected menu items accordingly. "Environment" in this context means the state of the application at the time: What windows are open? Is the front window an application window or a desk accessory? Is the application suffering from a low-memory condition? Other states are possible.

Who Fixes What

The application object can enable a few key menu items: the File menu's New, Open, and Quit commands. If the front window is a DA, the application can enable the Edit menu commands Undo, Cut, Copy, Paste, and Clear for the DA's use.

Beyond that, the application has to call on other objects for help. It keeps a list of other "menu-fixer" objects to which it sends FixMenus messages after it does its own menu fixing. The menu-fixer object list usually consists only of the document object connected to the front window (if that's an application window). Figure 13-1 on the next page shows the relationships among application document windows and view objects. When a new window comes to the front, its document object replaces the old one in the menu-fixer list. The application object always sends its FixMenus message to the front document.

The document's job

The document object fixes the menu items it can and then passes the torch to any of its views that need to fix menus. Menu items important to documents include the File menu's Close, Save, Save As, Revert, Page Setup, and Print commands (and possibly others) as well as the Edit menu's commands. Some of those items might not exist in a particular application, but the document object can fix the ones that do.

If a document object receives a FixMenus message, the implication is that the document (with its window) is open; so the document object enables the Close command in the File menu. It can also check the document's status to see whether it needs saving. If the document is "dirty" (has been changed since the last save), the document object enables the Save command. If the document has never been saved, the document object enables the Save As command. If the document supports printing, the document object enables the Page Setup and Print commands.

Other menu fixers

Each document object has a mechanism for letting its views and other subordinate objects "register" themselves as menu fixers. Some views might need to enable menu items, and they might need to make other adjustments to the menus as well—checking or unchecking a menu item, changing an item's text, and so on.

To register itself, a view object can send its document object a RegisterMenuFixer message. In response, the document object will add the view object to its menu-fixer list. The application object and each document object keep their own menu-fixer lists, which work in conjunction, as shown in Figure 13-1.

When the document object gets a FixMenus message, it first fixes the menus it can and then sends the first fixer in its list a FixMenus message. Each object in the list

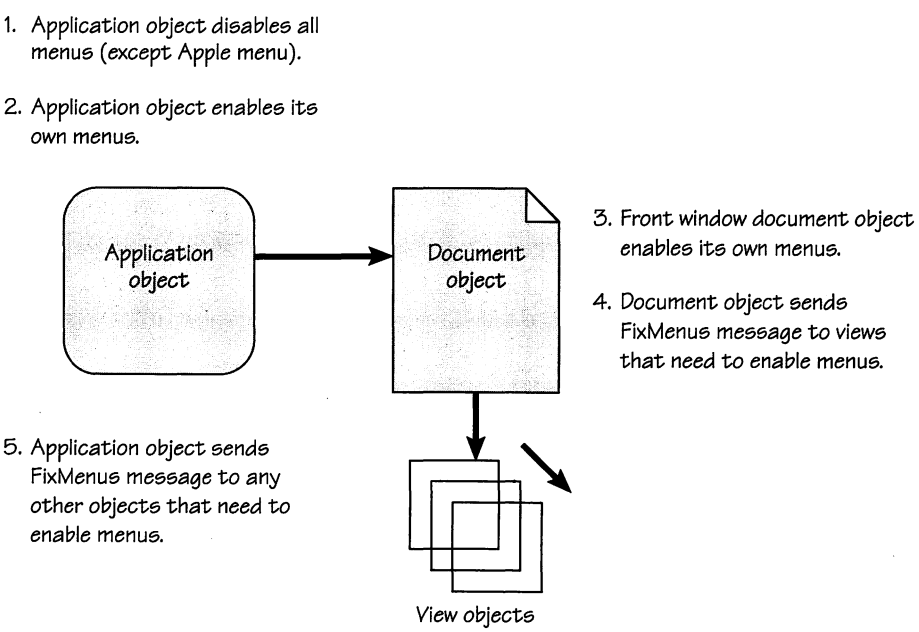


Figure 13-1.
Menu-fixer lists.

passes the message to the next object in the list after first fixing the menus it needs to. The mechanism for sending messages along the list is similar to the event-passing scheme we'll look at in "The Event-Handler Chain" later in this chapter.

At the level of the application object's menu-fixer list, it's possible to add other objects besides the current document to the menu-fixer list. Such objects can register themselves in the same way that views register themselves with their documents, by sending the application a RegisterMenuFixer message. Application-level menu fixers that aren't documents are added to the end of the list. Documents are added at the head. This ensures that any other objects your application chooses to put into its menu-fixer list will be called after the currently active document object.

Other Methods

Menu-fixing objects inherit their menu-fixing methods from *TEvtHandler*. Each has a *FixMenus* method and supporting *FixFileMenu* and *FixEditMenu* methods. The menu-fixing object first gets a FixMenus message. Then its *FixMenus* method calls the methods to fix the File and Edit menus. *FixMenus* looks like this:

```
procedure TPicoApp.FixMenus;
  override;
begin
  FixFileMenu;
  FixEditMenu;
  inherited FixMenus;     { Pass to front document and its views }
  { Overrides can add other menu-fixing code for other menus }
  {  or other menu items and then call "inherited FixMenus" }
end;  { TPicoApp.FixMenus }
```

When you declare a new class that will fix menus, you can also add methods to fix any other menus that objects of that class might care about. For example, if your object works with a Tools menu, it should have a *FixToolsMenu* method, which the class's override of *FixMenus* calls in addition to the other menu-fixing methods.

An example menu-fixing method

Here's an example of a menu-fixing method that takes care of some items on the File menu—*TPicoApp*'s *FixFileMenu*:

```
procedure TPicoApp.FixFileMenu;
  override;
begin
  EnableItem(gFileMenuHandle, kQuitItem);
  { See whether more open documents are allowed }
  if (fDocsAllowed <= kUnlimitedDocs) or (fDocCount < fDocsAllowed) then
    begin
      if not gLowMemory then
```

```
            begin
              if kNewItem <> 0 then
                EnableItem(gFileMenuHandle, kNewItem);
              if kOpenItem <> 0 then
                EnableItem(gFileMenuHandle, kOpenItem);
            end;
        end;
        { Default enables Close only for desk accessories at application level }
        if (FrontWindow <> nil) & IsDAWindow(FrontWindow) & (kCloseItem <> 0) then
          EnableItem(gFileMenuHandle, kCloseItem);
    end;  { TPicoApp.FixFileMenu }
```

This method always enables File-Quit. Then it checks to see whether the application allows unlimited documents or sets some limit. The instance variable *fDocsAllowed* will be *-1* if there's no limit; if there is a limit, *fDocsAllowed* tells what the limit is (a value greater than *0*). The method checks to see whether the current number of open documents is less than *fDocsAllowed* and indicates whether more documents can be opened. If more documents can be opened, the method then checks to see whether a low-memory condition is in effect. If not, the method can safely enable the File-New and File-Open commands, provided that the application defines *kNewItem* and *kOpenItem* as nonzero constants in *UMyGlobals*.

The method enables File-Close only if the front window belongs to a DA and the Close command exists.

FixFileMenu methods of other menu-fixing objects take care of the File menu's other commands in their turn.

What the User Sees

Again, this whole mechanism is transparent to the user. When the user clicks in the menu bar, he or she doesn't watch the disabling and then the selective reenabling of the menus and their items by a whole chain of objects. Instead, the user sees the menu he or she has chosen open so that one of its enabled items can be chosen.

The mechanism affirms one of our most important OOP principles: Whenever possible, delegate the work to the objects. Each can do the job in its own way.

Event-Handling Action Chains in PicoApp

In PicoApp (and Crapgame), events are processed by means of several chains of actions:

- Initialization chain. The application object creates and initializes the document object(s). Each document object creates and initializes its own window, view, and other supporting objects. In Crapgame, the application object creates a game object (document), which creates a window and a number of views, including player objects, dice objects, and button objects. The game object

creates a player list object in which to store the players. (The tiny main program creates and initializes the application object.)

- Event chain. The application object's event loop gets events and dispatches them. The application object handles some events itself, particularly menu events. The rest are passed on to other objects. Mouse-downs, key-downs, and most other events are passed along a linked list of "event-handler" objects we'll call the "event-handler chain." The chain includes application, possibly document, and selected view objects. Cursor-adjustment messages travel down the front window's list of views. Idle messages and menu-fixing messages (to enable/disable menu items) travel down special chains. In Crapgame, the event-handler chain includes the application object, two button objects, and the title view object. The list of views that adjust the cursor includes the "stats button" overlying the dice images and the title view object that demonstrates how to implement multi-clicking.
- Display chain. The first update event, after the window is created, causes the window's *DoUpdate* method to be called. The method in turn sends a Display message to each view object in the window's view list. Subsequent updates follow the same path. Crapgame doesn't have to alter or augment this process in any way.
- Play chain. In Crapgame, when the Throw button object detects that it's been clicked, it sends a message to the game object asking it to report the current player. Then the Throw button object sends that player a Play message. The player object, in its turn, sends Roll messages to the die objects and several kinds of messages to the game object.
- View switch chain. Recall that our game object actually keeps two different chains of view objects. One chain contains the views of the game-playing window we saw in Chapter 10. The other contains views of statistics about how the dice are performing for each player. The same window can display two or more chains of views alternately. The user switches from the game-playing view chain by clicking a transparent button overlying the dice. This replaces the play view with the statistics view and expands the transparent button so that it covers the whole window. The user can use the transparent button to go back to the play view and continue as before. The game object has a *SwitchViews* method that toggles the two view chains in response to the transparent button's *DoClick* method.
- Free chain. When the user closes a document, the document object sends a RemoveDoc message to the application object. The application object then disconnects the document object from its document list—but doesn't free it. The document object frees its window and its list(s) of views and then finally frees itself. If the user is quitting, the application object sends a Close message to each document object in the document list, and then the application object frees itself and the application terminates. In Crapgame, closing the game window or choosing File-Quit ends the current game and frees its document.

As the player objects free themselves, they also free their statistics objects, which constitute the second view chain.) The completion of a game also closes and frees the document and all of its subordinate objects.

The Event-Handler Chain

In a Macintosh application, there has to be a way to get user-generated events and dispatch them to the code that can handle them. When the user clicks the mouse in the menu bar or in a button or in some text, or when the user presses a key, something needs to handle the event.

In Chapter 4, when we developed *TButton*, we also looked briefly at a class called *TEvtHandler*, loosely modeled on a similar class in MacApp. *TEvtHandler* provides a field and several methods for linking a number of event-handler objects in a simple, singly linked list—the event-handler chain—and passing various event types along that list.

The objects in the event-handler chain are all of type *TEvtHandler*, of course. But each is also an object of some subclass of *TEvtHandler*. Any object class that must handle events is declared as a subclass—directly or indirectly—of *TEvtHandler*. In PicoApp, the application, document, and view classes are all descendants of *TEvtHandler*, as shown in Figure 13-2.

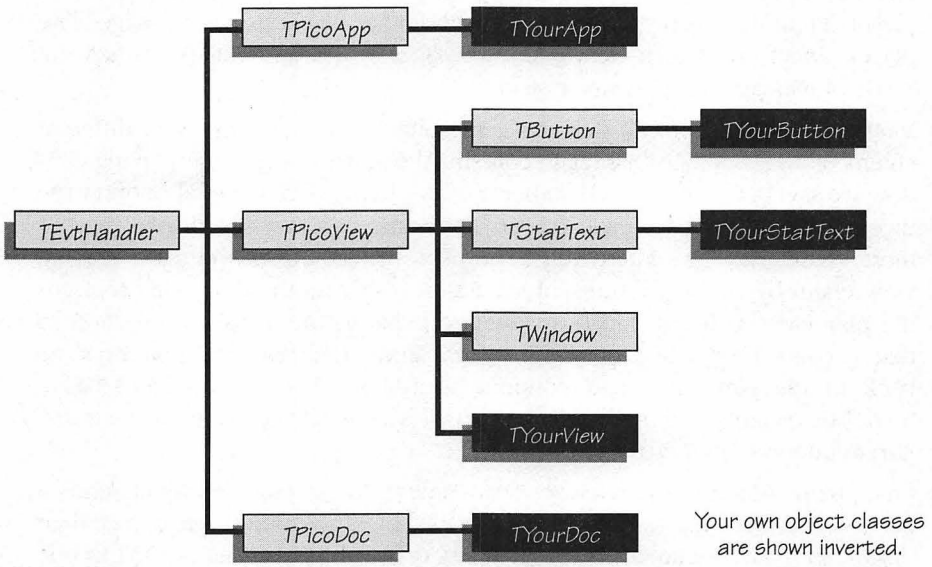


Figure 13-2.
The event-handler hierarchy, including application-specific classes.

The application object and all document objects are event handlers. So are all view objects—including buttons and pieces of so-called static text. Windows are views, and thus technically also event handlers, but we'll never put a window object into the event-handler chain. Windows play a different role.

What we've seen so far is the object class hierarchy of event handlers—how the various event-handling classes inherit their abilities from *TEvtHandler*. But we need to look at event handlers in a completely different way as well—as links in the event-handler chain. Figure 13-3 shows one possible configuration of the PicoApp event-handler chain and the actual Crapgame event-handler chain.

The event-handler chain is the flow of control in a Macintosh OOP application. The chain shows which objects pass events to which other objects. Of course, the event chain shown in Figure 13-4 on page 312 is not the only way to implement event dispatching, and, frankly, PicoApp's event-handler chain mechanism is best suited to small and medium-sized applications. If the event-handler chain gets too long, it will take too long to pass events down it. A bit later we'll look briefly at event handling in MacApp and TCL. We'll also look at some possible ways to improve the situation in PicoApp.

The event-handler chain is really several chains, although all contain the same objects. Menu events take one route. Mouse-downs other than those in the menu bar take another route. Ditto for key-downs and other events. We'll look next at *TEvtHandler* and the most important of the several chains.

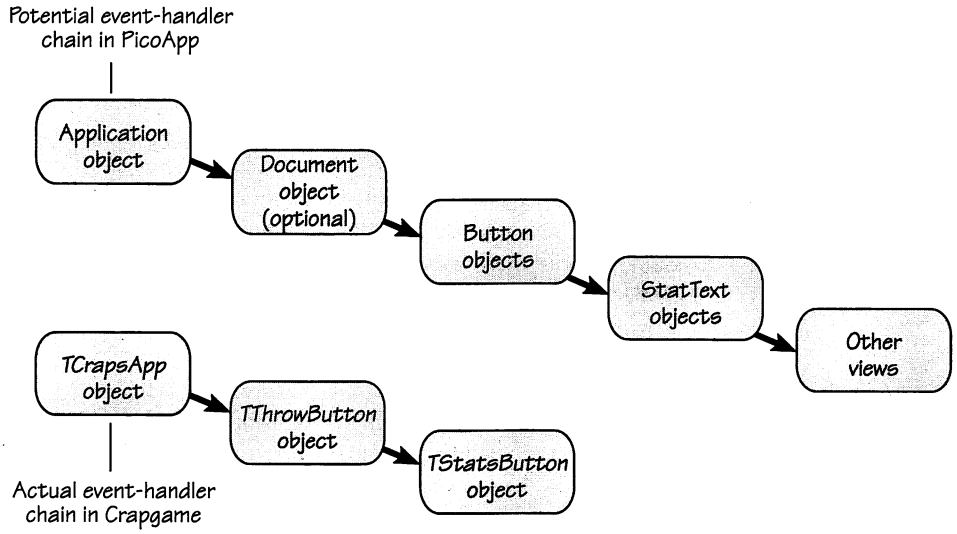


Figure 13-3.
The PicoApp and Crapgame event-handler chains.

The *TEvtHandler* Class Declaration

Here's what PicoApp's class *TEvtHandler* looks like:

```
type
  TEvtHandler = object(TObject)        { Abstract class }

    { kSizeOfEvtHandler = 14 bytes }

    fNextHandler: TEvtHandler;      { Next in a chain of event-handler objects }
    fNextIdleHandler: TEvtHandler;  { Next in a chain of idle handlers }
    fNextMenuFixer: TEvtHandler;    { Next in a chain of menu fixers }

    procedure TEvtHandler.IEvtHandler (handler: TEvtHandler);
      { Initializes the fNextHandler field }
      { Actually called only by the application object in PicoApp }
    procedure TEvtHandler.SetNextHandler (handler: TEvtHandler);
      { Another way to set an object's fNextHandler--that is, to make this object }
      {  point to another event-handler object in a chain of such objects }
    function TEvtHandler.NextHandlerOf: TEvtHandler;
      { Returns a reference to this object's next handler--the value of }
      {  fNextHandler }
    procedure TEvtHandler.FixMenus;
      { Objects that need to adjust menus, enabling or disabling }
      {   them or altering their text, should override this method }
    procedure TEvtHandler.FixFileMenu;
      { Abstract method for enabling menu fixer's File menu items }
    procedure TEvtHandler.FixEditMenu;
      { Abstract method for enabling menu fixer's Edit menu items }
    function TEvtHandler.DoMenuCommand (theMenu, theItem: Integer): Boolean;
      { Objects that respond to menu events should override this }
      {  to handle those events }
    function TEvtHandler.DoKeyCommand (event: EventRecord): Boolean;
      { Objects that respond to key-down or autokey events should }
      {  override this to handle those events }
    function TEvtHandler.DoMouseCommand (event: EventRecord): Boolean;
      { Objects that respond to mouse-down events }
      {  should override this to handle those events }
    function TEvtHandler.DoHandleEvent (event: EventRecord): Boolean;
      { Objects that handle "other" events, such as network }
      {  and "alien" (programmer-defined) events, can override }
      {  this method to handle the events }
    procedure TEvtHandler.DoIdle (event: EventRecord);
      { Objects that want to handle idle events should override }
    function TEvtHandler.TypeOf: Str30;
    override;
      { Returns the string 'TEVTHANDLER' }
```

```
function TEvtHandler.GenericTypeOf: Str30;
  { Returns the string 'OBJECT' }
end;  { Class TEvtHandler }
```

TEvtHandler and MacApp

The idea of *TEvtHandler* and its name comes from Apple's MacApp object-oriented application framework, in which it's a larger class and where many things are done rather differently than in PicoApp's version. In MacApp, a method like *DoMouseCommand* returns a *TCommand* object, which is passed back to the application object. The application object executes a method of the "command object" in order to carry out the actual response to an event. In MacApp, an object doesn't usually respond directly to an event. It creates a command object that knows what to do and passes it to the application object to be carried out.

If that sounds like heavy going, you'll be glad that we'll keep it much simpler. We will look briefly at a similar use of "command objects" that might help us someday implement in PicoApp an undo mechanism something like MacApp's.

Event Passing

The application object must get events from the event queue and then pass them to event-handling objects in the application, where one of the objects can handle the event.

How does the application object pass events to other objects? First, it sorts them into event types by means of a *case* statement in the *TPicoApp.DispatchEvent* method. Mouse events are handled by the application object's calling its own *DoMouseCommand* method. Key events are handled by *self.DoKeyCommand*. And so on.

How does a method process the event it's handed? By following a chain of event-handling objects through the *fNextHandler* link. In addition to creating the objects, initializing the application includes setting up the chain of links from object to object. The application object's *fNextHandler* might be set to point to a document object whose *fNextHandler* in turn is set to point to, say, a button object. Once the chain is set up, events can be passed along it from object to object so that each object has a crack at handling the event. The last object in the chain has the value *nil* in *fNextHandler*.

DoMouseCommand

For example, when a mouse-down event occurs, the application object calls its own *DispatchEvent* method, which first checks to see whether the click is in the menu bar, close box, size box, or zoom box—an event the application should handle. If it isn't, the application calls its own *DoMouseCommand* method, which passes the mouse-down event to its ancestor's *DoMouseCommand* method:

```
DoMouseCommand := inherited DoMouseCommand(event);
```

Having decided that it can't handle the event, the application object tries the *DoMouseCommand* method it would have inherited from *TEvtHandler* if it hadn't overridden that method. The keyword *inherited* means "Call my immediate ancestor's version of the method." Is that the button object's *DoMouseCommand*? No, not yet. We're still at the application object level, the first link in the chain.

Here's what *TEvtHandler.DoMouseCommand* looks like:

```
function TEvtHandler.DoMouseCommand(event: EventRecord): Boolean;
begin
  if self.fNextHandler <> nil then
    DoMouseCommand := self.fNextHandler.DoMouseCommand(event)
  else
    DoMouseCommand := false;
end;  { TEvtHandler.DoMouseCommand }
```

Remember that, although this is a *TEvtHandler* method, our application object, button objects, and so forth inherit it. It's their method, too. So *self* means the application object or the button object. And the *fNextHandler* field the method uses is the one the object inherited from *TEvtHandler*. This is what we mean when we say that one of these objects *is-an* event handler. (Notice that the method checks to see whether there really is a next handler object. If not, it simply reports that it can't handle the event, either.)

Calling *inherited DoMouseCommand* is thus falling back on a *TEvtHandler* version of the method, one that knows only how to pass the event on to the next object in the event-handler chain. The application object thus ends up calling the *DoMouseCommand* method of whatever object *fNextHandler* refers to—in this case a button object—passing the event as a parameter. Figure 13-4 shows how the sequence of calls goes.

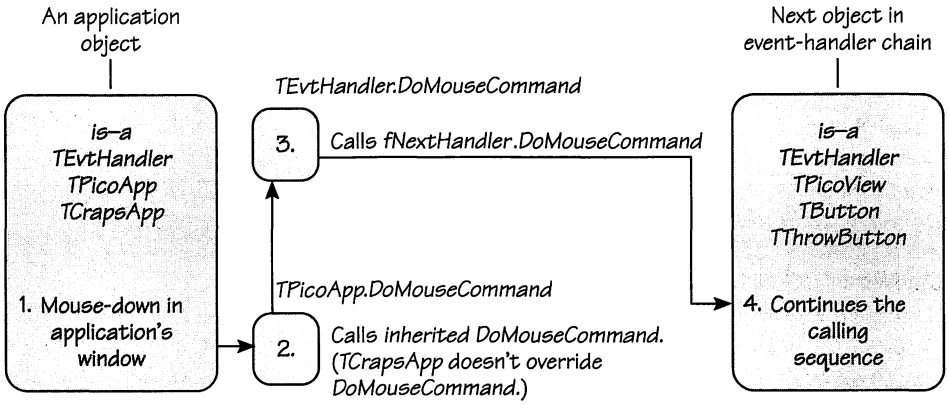


Figure 13-4.
Passing an event along the event-handler chain until a handler is found.

If the next object can't handle the event, it passes the event along the chain with the same kind of call, using its own *inherited DoMouseCommand*:

```
DoMouseCommand := inherited DoMouseCommand(event);
```

If this call returns *true*, the event has been handled by some object down the line. If *false*, the event has not been handled. The *DoMouseCommand* method of each object in the chain can post an error message if no object has been able to handle the event, or take some corrective action, or simply report to its own previous handler object that the event was not handled. The rule-of-thumb sequence for an object in an event-handler chain is first try to handle the event yourself; if you can't handle the event, pass it along the line via *inherited*. As a consequence of this sequence, a handler can assume that no object before it in the chain is able to handle the event. Success short-circuits further movement along the chain.

So an event might be passed from the application object to a document object, which passes the event to one or more control objects, and so on. As the *DoMouseCommand* calls return, each object can tell whether one of the objects along the chain was able to handle the event. Figure 13-5 on the next page diagrams the chain of calls in Crapgame as a mouse-down event is passed to the Throw button.

TPicoApp.DoMouseCommand--(*TCrapsApp* doesn't override the method)
TEvtHandler.DoMouseCommand--to pass along the chain
TButton.DoMouseCommand--(*TThrowButton* doesn't override the method)

If *TCrapsApp* were to override *TPicoApp.DoMouseCommand*, the first call would be to *TCrapsApp.DoMouseCommand*, then on to *TPicoApp.DoMouseCommand*, then on to *TEvtHandler.DoMouseCommand*, and so on. The process is similar with *TCrapGame* and *TThrowButton*, neither of which needed to override the default *DoMouseCommand* supplied by its superclass.

In this way events reach our button objects. If that seems roundabout, that's because it is. But if that seems slow, it isn't. With the relatively short event-handler chain in Crapgame, mouse-downs in the Throw button are acted on quickly, even on a slower machine such as a Macintosh SE.

TEvtHandler's Methods

All that most of *TEvtHandler*'s methods do is pass a call along a chain of handler objects, as we saw. The code for all of them—except *FixMenus* and *MakeMenus*—is parallel to the code for the *DoMouseCommand* method.

Let's look at some of the other methods.

DoMenuCommand

The *DoMenuCommand* method allows objects to respond to menu selections. The application object should call its *DoMenuCommand* method (an override of *TEvtHandler*'s) to process its About dialog box and its desk accessories. The application object's *DoMenuCommand* method will probably process other menu commands, such as File-New, File-Open, and certainly File-Quit.

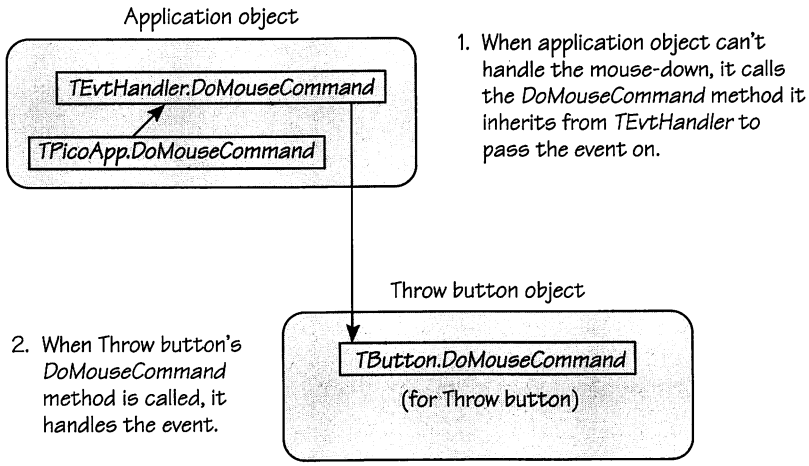


Figure 13-5.
Traversing the event chain in Crapgame—how a mouse click is passed to the Throw button.

Other objects can respond to menu commands; a document object, for instance, would probably respond to File-Save, File–Save As, File-Revert, File-Print, and perhaps other menu commands. If the application object can't handle a menu event, it dispatches the menu event (a mouse click in a menu) to other objects in the event-handler chain. Presumably, one of them will be able to handle the event.

In general, a button doesn't carry out an action as a result of the user's making a menu choice, so *TButton* simply inherits the stub *DoMenuCommand* method from *TEvtHandler* and doesn't override it.

Any object in the chain can override *DoMenuCommand*. Figure 13-6 illustrates how the document object can take on some menu-handling chores and even add its own menus and handle their items.

DoKeyCommand

The *DoKeyCommand* method processes keypresses. *TPicoApp.DoKeyCommand* checks the event to see whether the Command key was down while the key was pressed. If the Command key was down, the method checks to see whether the Menu Manager can use the key-down to select a menu command. If the key-down is not a menu-command key, *DoKeyCommand* passes the key-down along the event-handler chain to see whether the document object or a view object can handle it.

The mechanism for passing events along parallels the way *DoMouseCommand* works, with one exception. We implemented *TButton* so that a button could be made the "default" button. Pressing Enter or Return is equivalent to clicking the button if the button is the default button. In Chapter 4 we looked at how buttons implement their *DoKeyCommand* methods.

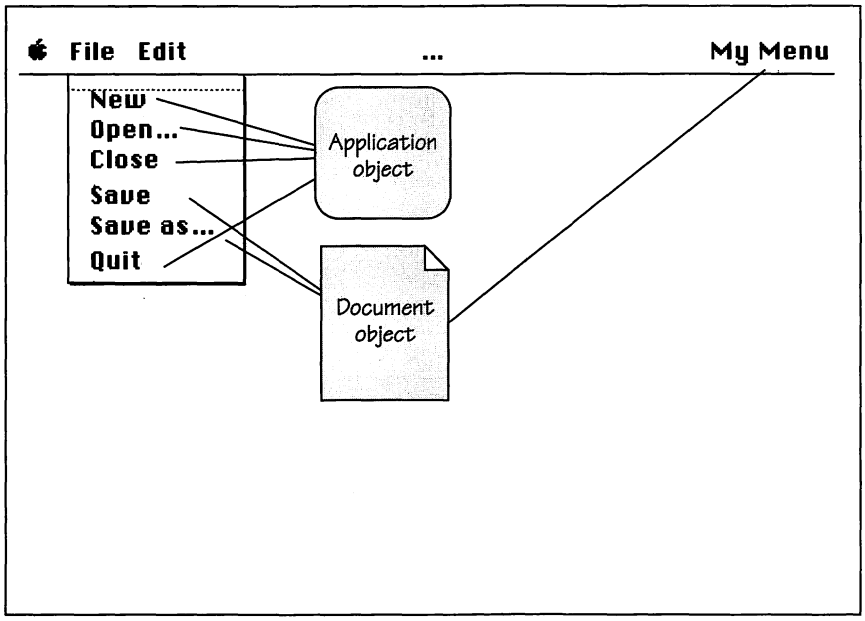


Figure 13-6.
Menu setup by multiple objects. The document object helps the application object handle New and Open by means of its MakeDocument *method, and the document object and other objects can also put up their own menus and handle their items.*

DoHandleEvent

The *DoHandleEvent* method is for processing non-mouse, non-key, non-menu events, such as disk insertions, network events, and "alien" events. Alien events are those that an application defines for its own use. Any such events that the application can't handle are passed on to other objects.

We won't define any action for buttons or other Crapgame objects to take in response to such events.

FixMenus

The *FixMenus* method, which we looked at earlier in this chapter, under "Menu Updating (Fixing)," provides a centralized way to enable or disable menu items.

DoIdle

The *DoIdle* method provides a way for objects to take advantage of idle time to do useful work. Idle messages are passed to objects that have registered themselves, just as event handlers do, with the application. This puts them into an idle-handler chain separate from the event-handler chain. Idle events (null events) have to traverse only a short chain of objects—a good thing because idles occur frequently.

Getting and Dispatching Events

Now that we've seen how events reach their destination objects, let's see how the application object gets and dispatches them in the first place. Figure 13-7 is good preparation for understanding the process.

Two *TPicoApp* methods are involved (neither of which your application ever needs to override, in all likelihood): *MainEventLoop*, which gets one event, and *DispatchEvent*, which dispatches an event to some appropriate event handler—usually somewhere along the event-handler chain.

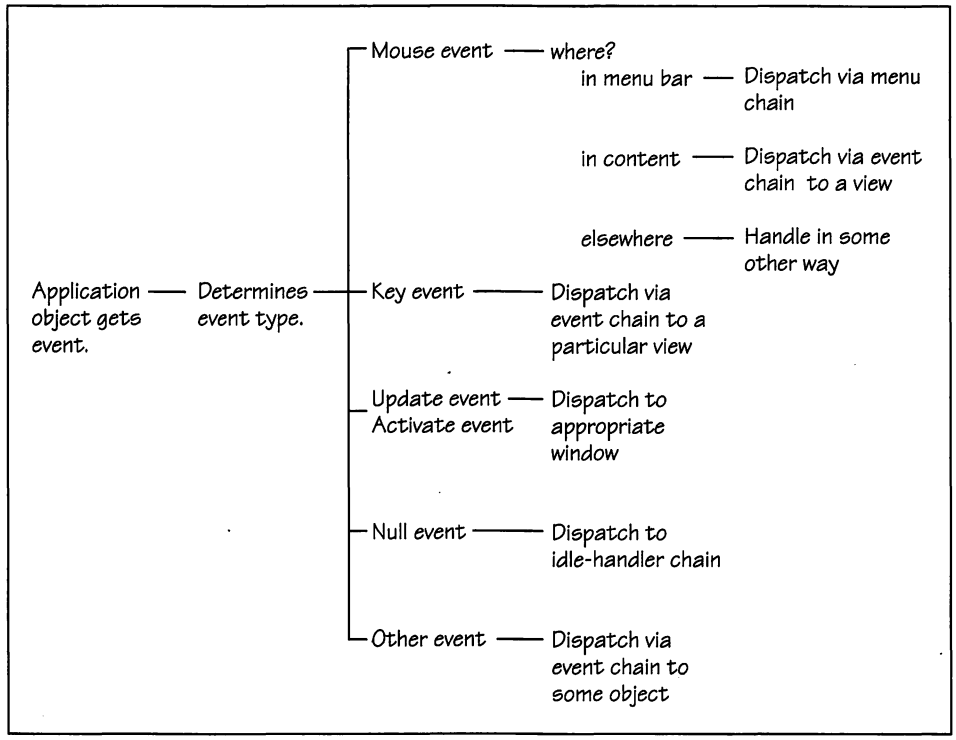


Figure 13-7.
Event dispatching—how an event is dispatched and passed along until it reaches an appropriate handler.

MainEventLoop

The *MainEventLoop* method does some hocus-pocus to find out whether certain Toolbox traps related to MultiFinder are available—mainly *WaitNextEvent*. If *WaitNextEvent* is available, the event loop will call *WaitNextEvent*, which, if MultiFinder is actually running, waits until there's an event for the application object, allowing processor time for background processes. If MultiFinder isn't running, *WaitNextEvent* behaves the way good old *GetNextEvent* does, immediately getting the next

event from the Mac's event queue. And if the Toolbox traps aren't available (because we're running on a machine with old ROM), the event loop uses the traditional *GetNextEvent*.

Then the event loop gets one event (a null event if nothing else) and sends the application object a DispatchEvent message.

If *WaitNextEvent* is unavailable, the event loop also calls *SystemTask* to allot some time to desk accessories and other drivers needing time, such as a clock DA.

Four additional calls are made each time through the event loop. The first is to *CheckMemory*, which alerts the user if a low-memory situation exists.

The second call is to *AdjustCursor*, which allows the application's view objects to adjust the cursor when the mouse pointer is moved into their territories on the screen. We'll go into cursor maintenance in detail in Chapter 14.

The third call is to a *DoEachLoop* method, which in *TPicoApp* is a stub. Application subclasses can override it if they need to perform some additional action periodically. The overriding version is the one that's called.

The fourth call is to an *UnloadSegments* method, which an application subclass should override to provide its own calls to the Toolbox routine *UnloadSeg*. (See *Inside Macintosh*, II-59.) Actually, we plan to replace this method eventually with a fancier scheme in our *TMemory* class, which we'll get to in the next chapter.

Listing 13-1 is the code for *TPicoApp.MainEventLoop*.

```
procedure TPicoApp.MainEventLoop;
  const
    kWNETrapNum = $60;      { Trap number for testing MultiFinder }
    kUnImplTrapNum = $9F;   { Also used in testing for MultiFinder }
  var
    hasMultiFinder: Boolean;  { True if MultiFinder available on this machine }
    sleep: Integer;           { Time to yield to other applications under }
                              {  MultiFinder }
    doIt: Boolean;            { True if there is an event to process }
    event: EventRecord;       { The event }
    theErr: OSErr;
    myWorld: SysEnvRec;       { For testing whether MultiFinder is available }
begin
  FlushEvents(everyEvent, 0);
  { Check the environment--is MultiFinder available? }
  theErr := SysEnvirons(1, myWorld);
```

Listing 13-1. *(continued)*
The procedure TPicoApp.MainEventLoop.

Listing 13-1. *continued*

```
    if (myWorld.machineType >= 0) and (NGetTrapAddress(kWNETrapNum,
        ToolTrap) = NGetTrapAddress(kUnImplTrapNum, ToolTrap)) then
        hasMultiFinder := false     { MultiFinder is not available }
    else
      hasMultiFinder := true;       { MultiFinder is available, so use WaitNextEvent }
    sleep := 10;                    { You might want to alter this }

    { Loop repeatedly for incoming events }
    repeat
      gMemory.CheckMemory;          { Handle low-memory situations }
      AdjustCursor;                 { Let views in front window change cursor }
      DoEachLoop;                   { Anything you need to do each time through }
                                    {  loop? }
      if hasMultiFinder then        { Check for MultiFinder }
        doIt := WaitNextEvent(everyEvent, event, sleep, nil)
      else                          { Just the Finder, so use GetNextEvent }
        begin
          SystemTask;
          doIt := GetNextEvent(everyEvent, event);
        end;
      if doIt then
        DispatchEvent(event);       { Decipher and deal with the event }

      UnloadSegments;               { Override to unload your segments }
    until fQuitting;
  end;  { TPicoApp.MainEventLoop }
```

DispatchEvent

After *MainEventLoop* has gotten a new event, it passes the event to *DispatchEvent*, which farms events out to various application methods and other objects for handling.

Before deciding what to do with the event, *DispatchEvent* takes steps to determine which window and document objects will be affected. If the event was a mouse-down, *DispatchEvent* finds out precisely where it occurred and in which window. That information makes it easy to find out whether the window is a PicoApp window and, if it is, to get a reference to the window object and, from that, a reference to the associated document object. These references are then used to pass various event-oriented messages to the window or the document.

After these preliminaries are out of the way, *DispatchEvent* uses a large *case* statement to determine what type of event it's dealing with and to dispatch the event to an appropriate handler.

Listing 13-2 is the code for *TPicoApp.DispatchEvent*:

```
procedure TPicoApp.DispatchEvent (event: EventRecord);
  const
    kMultiEvt = 15;              { MultiFinder suspend or resume }
  var
    whereHit: Integer;           { Value specifying where event occurred }
    hitInAppWindow: Boolean;     { True if in content or frame area of application }
                                 {  window }
    windowKind: Integer;         { Value of window's windowKind field }
    myWindowKind: Boolean;       { True if window is a PicoApp window }
    mResult: Longint;            { Menu and item chosen }
    theMenu, theItem: Integer;   { Menu and item extracted from mResult }
    ok: Boolean;                 { Auxiliary variables }
    r: Rect;
    globalPt: Point;
    oldPort: GrafPtr;
    i: Integer;
    theWinObj: TWindow;          { Target objects for dispatch }
    theDoc: TPicoDoc;
begin
  globalPt := event.where;
  whereHit := FindWindow(event.where, self.fWindow);
  hitInAppWindow := InFrontAppWindow(whereHit, self.fWindow);
  if (whereHit = inMenuBar) or hitInAppWindow then
    ExtractWinObj(self.fWindow, theWinObj, theDoc);
  myWindowKind := hitInAppWindow;

  { Process the event }

  case event.what of
    mouseDown:
      begin
        { Mouse-down in the menu bar--user selecting a menu item }
        if whereHit = inMenuBar then
```

Listing 13-2. *(continued)*
The procedure TPicoApp.DispatchEvent.

Listing 13-2. *continued*

```
    begin
      self.RegisterMenuFixer(theDoc);          { Possibly nil }
      self.AdjustMenus;
      mResult := MenuSelect(globalPt);
      theMenu := HiWord(mResult);
      theItem := LoWord(mResult);
      ok := self.DoMenuCommand(theMenu, theItem);
    end

  { Mouse-down in any window's drag bar--drag the window }
  else if whereHit = inDrag then
    begin
      if not DoRearWindow(self.fWindow) then
        begin
          if myWindowKind then
            self.Drag(theWinObj, globalPt)  { Have TWindow do it }
          else                              { Ordinary window drag }
            DragWindow(self.fWindow, globalPt, gDragRect);
        end;
    end

  { Mouse-down in a system window (DA)--let system handle it }
  else if whereHit = inSysWindow then
    begin
      SystemClick(event, self.fWindow)
      CleanUpAfterDA  { Fix Menus if DA has closed }
    end
      { If not a DA window, user just closed last DA }

  { Mouse-down in a window's close box--user closing window/document }
  else if whereHit = inGoAway then
    begin
      if not DoRearWindow(self.fWindow) then
        if TrackGoAway(self.fWindow, event.where) then
          self.Close(theWinObj, theDoc);
    end

  { Mouse-down in a window's size box--expand or shrink the window }
  else if whereHit = inGrow then
    begin
      ok := DoRearWindow(self.fWindow);
      windowKind := WindowPeek(self.fWindow)^.windowKind;
      { If window is a documentProc type without a size box, }
      {  treat it as a mouse-down in content area }
```

(continued)

Listing 13-2. *continued*

```
        if not ok & myWindowKind and (windowKind = kPicoKindLow) and
            (not theWinObj.HasSizeBox) then
          ok := self.DoMouseCommand(event)
        else                                { It has a size box, so expand it }
          self.Grow(event, theWinObj);
      end

    { Mouse-down in a window's zoom box--zoom the window }
    else if (whereHit = inZoomIn) or (whereHit = inZoomOut) then
      begin
        if not DoRearWindow(self.fWindow) then
          if TrackBox(self.fWindow, globalPt, whereHit) then
            theWinObj.Zoom(event, whereHit);
      end

    { Mouse-down in content area or scrollbar of a window--let }
    {  the document and its views handle the click }
    else if whereHit = inContent then
      begin
      if not DoRearWindow(self.fWindow) then
        ok := self.DoMouseCommand(event)
      end

    { Mouse-down in the desktop--just beep }
    else if self.fWindow = nil then
      Sysbeep(1);

  end;  { mouseDown }

{ Mouse-up event--DoMouseCommand should handle if needed }
mouseUp:
  ok := self.DoMouseCommand(event);    { Application defined }

{ Keypress--see whether any object wants it }
keyDown, keyUp, autoKey:
  ok := self.DoKeyCommand(event);

{ Disk insertion, ejection, and so on--see whether any object wants it }

diskEvt:
  ok := self.DoHandleEvent(event);          { Application defined }
```

(continued)

Listing 13-2. *continued*

```
    { Update event for fWindow--let window and views handle it }
    updateEvt:
      begin
        self.fWindow := WindowPtr(event.message);
        ok := self.DoUpdate(event);
      end;
    { Window activating or deactivating--let window handle it }
    activateEvt:
      begin
        self.fWindow := WindowPtr(event.message);
        ok := self.DoActivate(event);
      end;
    { Communications network activity--some object should override DoHandleEvent }
    {  if it needs to process such events }
    networkEvt:
      ok := self.DoHandleEvent(event);        { Application defined }
    { Device driver event--some object should override DoHandleEvent }
    {   if it needs to process such events: }
    driverEvt:
      ok := self.DoHandleEvent(event);        { Application defined }
    { User suspending or resuming using this application under MultiFinder-- }
    {  process the switch }
    kMultiEvt:
      self.DoMultiFinderEvent(event, self.fWindow);
    { Programmer-defined event--some object should override DoHandleEvent }
    {  if it needs to process such events }
    app1Evt, app2Evt, app3Evt:                { app4Evt = MultiFinder event }
      ok := self.DoHandleEvent(event);        { Application defined }
    { The event the operating system posts when nothing is happening }
    { DoIdle does nothing, but objects can override it to do idle-time }
    {  processing if needed }
    nullEvent:
      self.DoIdle(event);                     { Application defined }
    otherwise
      ;
  end;  { case }
end;  { TPicoApp.DispatchEvent }
```

The cases in the first group involve mouse-down events and use a sequence of *if* statements to determine where a mouse click took place, using the variable *whereHit*, returned by *FindWindow*. The cases are for a click

- In the menu bar
- In the drag bar of a window
- In a system window (such as a desk accessory)
- In a window's close box
- In a window's size box
- In a window's zoom box
- In a window's content region (including scrollbars, if any)
- Simply somewhere in the desktop

In most of these cases, *DispatchEvent* calls another method to handle the event from there. For instance, a mouse click in a window's close box uses the Toolbox function *TrackGoAway* to track the mouse to be sure that it has really been clicked within the close box. If it has, *DispatchEvent* calls the *TPicoApp.Close* method.

The rest of the *case* statement is for events other than mouse-downs and operates in much the same way. Notice that many kinds of events call the same method, the *DoHandleEvent* method, which *TPicoApp* inherits from *TEvtHandler*. At this level, *DoHandleEvent* is a stub, which an application can override to handle one or more of these kinds of events. For example, if an application needs to do something in response to both network and driver events, it can override *DoHandleEvent* and use an *if* statement or a *case* statement to determine which kind of event is being passed and then handle each appropriately. Within *DoHandleEvent*, code needs to test which kind of event the method is dealing with by examining the *what* field of the *event* parameter it is passed.

In addition to providing handlers for all the standard kinds of events listed in *Inside Macintosh*, I-244, *TPicoApp* provides for three "application-defined" events that you can define for yourself. These are also handled by *DoHandleEvent*. *TPicoApp* can also test for a MultiFinder event (using *app4Evt*).

There's also a handler for null events (*DoIdle*), in case your application needs to get some work done while the event loop is just idling. By overriding *DoIdle* in your application subclass and "registering" your idle handler with the application, you can provide for such idle-time processing. Objects that need to do the actual idle processing will override *DoIdle*, which they inherit from *TEvtHandler*. HyperCard uses idle-time processing to maintain the blinking insertion point in text fields and to update the clock if you have the time displayed. *DoIdle* will be called every time there's nothing else going on, so it gets called a lot. *TPicoApp*'s version is a stub.

As soon as an event handler returns, *DispatchEvent* ends and the main event loop cycles again.

Mouse Commands and Menu Commands

TPicoApp overrides the *TEvtHandler DoMouseCommand* and *DoMenuCommand* methods. *DoMenuCommand* does some processing, if it can, before events are dispatched along the chain.

DoMouseCommand simply passes an event along to the next handler in the chain.

```
function TPicoApp.DoMouseCommand (event: EventRecord): Boolean;
  override;
begin
  DoMouseCommand := inherited DoMouseCommand(event);
    { Simply pass to next handler }
end;  { TPicoApp.DoMouseCommand }
```

Let's take a brief look at *TPicoApp*'s override of *TEvtHandler.DoMenuCommand*. It works much as *DoMouseCommand* does except that it's designed to handle three different menus and consequently has a *case* statement. Listing 13-3 shows the code.

```
function TPicoApp.DoMenuCommand (theMenu, theItem: Integer): Boolean;
  var
    DAName: Str255;                    { Name of the DA }
    DA: Integer;                       { Dummy for OpenDeskAcc result }
    front: WindowPeek;                 { Window variables }
    frontKind: Integer;
    theWinObj: TWindow;
    theDoc: TPicoDoc;
    oldPort: GrafPtr;
    pass: Boolean;                     { Used to pass event along chain }
begin
  case theMenu of
    kAppleMenuID:
      if theItem = kAboutItem then     { Handle About box }
        DoAbout
      { Or handle a desk accessory }
      else if (theMenu = kAppleMenuID) & (theItem > kAboutItem) then
        begin
          if not gLowMemory then       { Open if not a low-memory situation }
            begin
              GetPort(oldPort);
              GetItem(gAppleMenuHandle, theItem, DAName);
              DA := OpenDeskAcc(DAName);
              EnableItem(gEditMenuHandle, 0);
```

Listing 13-3. *(continued)*
The function TPicoApp.DoMenuCommand.

Listing 13-3. *continued*

```
            RedrawMenus;
            SetPort(oldPort);
          end
        else
          NotEnoughMemoryAlert;
      end;  { Apple }

    kFileMenuID:
      case theItem of
        kNewItem:              { Create a new document }
          DoNew;

        kOpenItem:             { Let user open a document of appropriate type }
          DoOpen;

        kCloseItem:            { Allow closing of windows }
          { Application closes its own windows--DAs handled here }
          begin
            if FrontWindow <> nil then
              begin
                frontKind := WindowPeek(FrontWindow)^.windowKind;
                if frontKind < 0 then
                  begin
                    CloseDeskAcc(frontKind)
                    DisableItem(gEditMenuHandle, 0);
                    RedrawMenus;
                  end
                else
                  begin
                    if InAppWindowRange(frontKind) then
                      begin
                        self.fWindow := FrontWindow;
                        { Get the window's winObj and its document }
                        theWinObj := TWindow(GetWRefCon(self.fWindow));
                        theDoc := TPicoDoc(theWinObj.DocumentOf);
                        self.Close(theWinObj, theDoc);
                      end;  { If InAppWindowRange }
                  end;  { Else no front window }
              end;  { If FrontWindow <> nil }
          end;  { CloseItem }

        kQuitItem:
          Quit;
```

(continued)

Listing 13-3. *continued*

```
        otherwise                { Not item in this menu--pass to next handler }
          DoMenuCommand := inherited DoMenuCommand(theMenu, theItem);
      end;  { File }

    kEditMenuID:
      begin
        pass := false;
        case theItem of
          kUndoItem:
            if not SystemEdit(kUndoItem - 1) then
              pass := true;      { It's an application Undo call }

          kCutItem:
            if not SystemEdit(kCutItem - 1) then
              pass := true;      { It's an application Cut call }

          kCopyItem:
            if not SystemEdit(kCopyItem - 1) then
              pass := true;      { It's an application Copy call }

          kPasteItem:
            if not SystemEdit(kPasteItem - 1) then
              pass := true;      { It's an application Paste call }

          kClearItem:
            if not SystemEdit(kClearItem - 1) then
              pass := true;      { It's an application Clear call }
          otherwise
            ;
        end;  { case }
        if pass then             { Undo, Cut, Copy, Paste, or Clear not for a DA }
          DoMenuCommand := inherited DoMenuCommand(theMenu, theItem);
      end;  { Edit }

    otherwise                    { Not this menu--pass to next handler }
      DoMenuCommand := inherited DoMenuCommand(theMenu, theItem);
  end;  { case }
  HiliteMenu(0);                 { Turn menu selection off }
end;  { TPicoApp.DoMenuCommand }
```

Code for the Apple menu item handles either the application's About dialog box or the opening of a desk accessory.

Code for the File menu handles the New, Open, Close, and Quit commands. The next event-handler object is called to handle other File menu items. This might be a document object. But keep in mind that the *TPicoApp* version of *DoMenuCommand*

will be the *second* version called. The application subclass's *DoMenuCommand* will be called first, and it will handle most menu items unknown to *TPicoApp*. If the application subclass's version of the method can't handle the menu item, it calls its *inherited DoMenuCommand*, which invokes the version shown in Listing 13-3.

Code for the Edit menu is really all hooks. Undo, Cut, Copy, Paste, and Clear commands are passed to any open desk accessory using the *SystemEdit* call. If the DA can handle the call, it does, returning *true*. If *SystemEdit* returns *false*, the DA doesn't want the call, so we pass the menu event to the front document object, which calls its own *Cut*, *Paste*, and so on. These are document hook methods. In PicoApp they're all stubs, but your document subclass can override them to tailor Edit menu handling to your liking.

At the end, we call *HiliteMenu(0)* to turn off the menu item's highlighting after we've handled it.

An Integrated Command System

Frankly, PicoApp's event-handling mechanisms are still rather piecemeal. It's also worth considering MacApp- or TCL-style command mechanisms as an alternative.

Both MacApp and TCL translate raw events—mouse clicks in a menu, for example—into integer codes, which they then pass along the event chain. So, for instance, where a menu click in PicoApp is conveyed with two integers (one to identify the menu and another to identify the item in the menu), in MacApp or TCL it would be simplified to one—represented by a *kSaveCmd* constant, say.

Instead of having separate methods for mouse-downs, menu commands, and key-downs, each document and view object would have a *DoCommand* method that processes the commands the object "knows" about. For example, consider this abbreviated *DoCommand* method:

```
procedure TSomeDoc.DoCommand(theCommand: Integer);
begin
  case theCommand of
    kCloseCmd:  Close;
    kSaveCmd:  Save;
    kPageSetupCmd: PageSetup;
    :
    otherwise
      inherited DoCommand(theCommand);  { Pass it on }
  end;  { case }
end;  { TSomeDoc.DoCommand }
```

Each object class overrides *TEvtHandler.DoCommand* to set up a case statement detailing those commands it can process. To process them, *DoCommand* calls other document or view methods. If one object's *DoCommand* can't handle a command, it passes the command on to the next object in the chain.

Caution: Objects in Multiple Handler Lists

An important word of caution is in order—even for our current, piecemeal version of PicoApp. There's great danger of error if we put an object into two or more separate object lists (such as our special chains for DoCursor messages, idle events, and menu fixing).

As Figure 13-8 demonstrates, our chains are built by linking objects themselves together. One object has an instance variable that "points" to the next object. Usually it's *fNextHandler*.

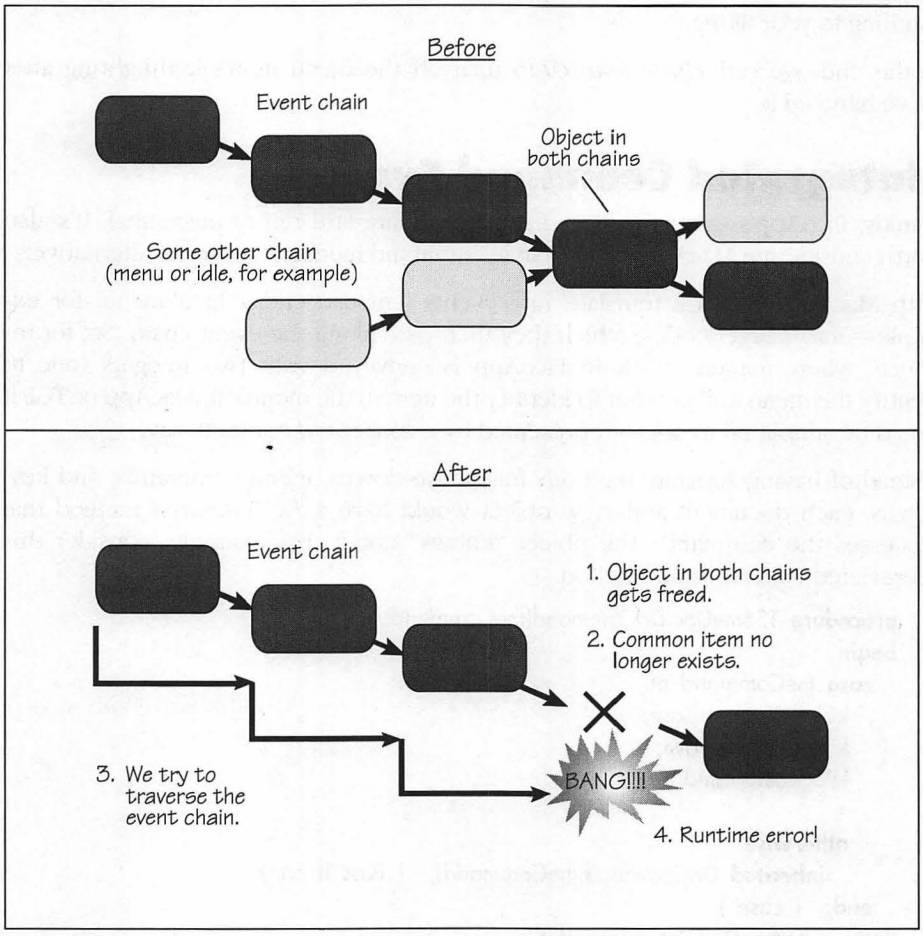


Figure 13-8.
Interlocking chains. Two chains have a common element, which, when freed, causes an event traversing one of the chains to generate a runtime error.

Suppose that one of our view objects is in the event-handler chain so that it can get mouse events and that we also install it in the idle-handler list. If we free the object while it's in a list, without removing it first, the list's integrity is shot. And if we remove an object that's in several lists from only one or two lists, but not from all, and then free it, the remaining lists will be damaged.

As soon as the application tries to traverse a damaged list, it will stumble over the missing object and generate a runtime error. Figure 13-8 illustrates the breaking of a handler chain.

We must take care to call the appropriate removal methods—all of them—for objects in lists. We might even want to put in an automatic mechanism of some kind—say a method called by every object's *Free* method that first calls all the lists to see whether the object is in them and removes the object from every list it's in. This would keep us from inadvertently shooting ourselves in the foot.

Application Shutdown

When the user chooses Quit from the File menu (or presses Command-Q), the application calls *Quit*, which sets *fQuitting* to *true*, terminating the event loop. Here's the *Quit* method:

```
procedure TPicoApp.Quit;
begin
  self.DoBeforeQuitting;
  self.CloseAllDocs;
  self.fQuitting := true;                    { Quit the event loop }
end;  { TPicoApp.DoQuit }
```

Before quitting, the procedure calls *DoBeforeQuitting*, a hook method that lets us insert any special quitting behavior we need. The application subclass can override *DoBeforeQuitting*.

Then the procedure calls *CloseAllDocs*, which initiates a complex process. *CloseAllDocs* goes along the document object list, sending a Close message to each document. Each document, in turn, asks its window, views, and other subordinate objects to close and free themselves. Then the document calls *inherited Free* to free itself.

After the documents are closed, the application sets *fQuitting* to *true*, which terminates the event loop. Control subsequently returns to the *Run* method. Then, inside *Run*, the application executes the following lines:

```
DoApplicationCleanup;    { Subclasses can define this }
PrivateToDeskScrap;      { Copy private scrap, if we have one, to the desk scrap }
ClearMenuBar;
Free;                    { Free the application object }
```

DoApplicationCleanup is another hook we can override to define some special cleanup behavior. The application's final acts are to clear its menu bar and free itself. This returns control to the main program (which originally created the application object), which then terminates.

PrivateToDeskScrap is for applications that keep a private scrap. For example, in the original version of TextEdit, you had to copy data back and forth between TextEdit's private scrap and the desk scrap (the Clipboard). Recent versions of TextEdit don't use a private scrap, but many applications do. Your application subclass will have to override *PrivateToDeskScrap* (and its companion method, *DeskToPrivateScrap*) to fill in the details of what it does. (See *Inside Macintosh,* I-453.) PicoApp automatically calls *PrivateToDeskScrap* for you, but the default version does nothing. (You could even implement your *PrivateToDeskScrap* method so that it would ask the user whether he or she wanted to "Save large Clipboard?"—as in Microsoft Word—if the private scrap happened to be large.)

Notice how we have hooks both before and after the event loop terminates: *DoBeforeQuitting* and *DoApplicationCleanup.* Crapgame doesn't need either of these methods, but some applications might.

Summary

In this chapter, we explored how PicoApp works internally. The idea was to give you a clearer sense of how much and what kinds of work PicoApp does for you so that you'll have a better idea of what you have to alter or add for your own application.

We covered application startup, menu setup, event handling, and application shutdown. If you're thinking about moving on from here to MacApp or TCL, keep in mind that these application frameworks will probably do things a bit differently—but what you've seen here should give you a head start toward understanding their mechanisms for the same activities.

In the next chapter, we'll explore cursor adjustment, memory management, and "undoing."

Projects

- Currently, PicoApp has one event-handler chain for events such as mouse-downs and key-downs. If three windows, A, B, and C, are open and each has three views that need events, the application's event-handler chain has a total of nine objects linked into it—at the same time.

  Only the views of the front window can really use most kinds of events. To improve efficiency, we could

  - Put the event-handler chain in *TWindow* instead of in the application and have views register themselves *with their windows* as event handlers. (For a similar setup, see the menu-fixing mechanism discussed earlier in this chapter.)

- Have the application send events to the window, which would pass them down its event-handler chain.

  The application might still need to keep its own event-handler list, too, because some documents and other objects besides views might want certain kinds of events. The application could give those objects a chance at an event if the front window's event-handler chain hasn't claimed it.

  This would shorten the event-handler chain and speed event processing. Implement this scheme in PicoApp.

- Study the way events are processed in MacApp or TCL and implement something similar in PicoApp.

- In the *TPicoApp.MainEventLoop* method, we called *SysEnvirons* to get information about the Mac we're running on. We installed that information in a record variable called *myWorld*, which we accessed to get environmental information.

  To replace that approach, design and implement a *TMacintosh* class. Its function is to provide all sorts of environmental information about the currently running machine and system software. Your application object would create and initialize a *TMacintosh* object as its first act and install a reference to the object in a global variable, *gMacintosh*. Any object could then call the object's methods to learn the current screen dimensions; the height of the menu bar; what peripherals or display devices are attached; whether color, a hierarchical file system, and certain coprocessor chips are available; what processor, system level, and Finder version are running; and so forth. The *TMacintosh.IMacintosh* method could initialize the Toolbox managers, facilities needed for memory management, a global error handler, and so forth.

14

INSIDE PICOAPP: MORE ON THE APPLICATION

In Chapter 13, we explored event handling and other PicoApp features at the application object level. In this chapter, we'll look at other features at the same level:

- The document list
- Cursors
- Memory management
- A possible implementation of Undo
- *TPicoApp* design alternatives
- Global variables

Maintaining the Document List

Another important function of the application object is to maintain a list of all open documents plus methods for adding and removing documents from the list.

TPicoApp's document list is a simple linked list, like the event-handler chain, in which one object points to the next object through an object-reference field. In this case, the start of the document list is *fDocList*, an instance variable of type *TPicoDoc*. Each document object has a field, *fNextDoc*, that points to the next document of type *TPicoDoc* in the list. A document object can also contain a window object and a list of view objects. Figure 14-1 on the next page is a graphic representation of *TPicoApp*'s document list.

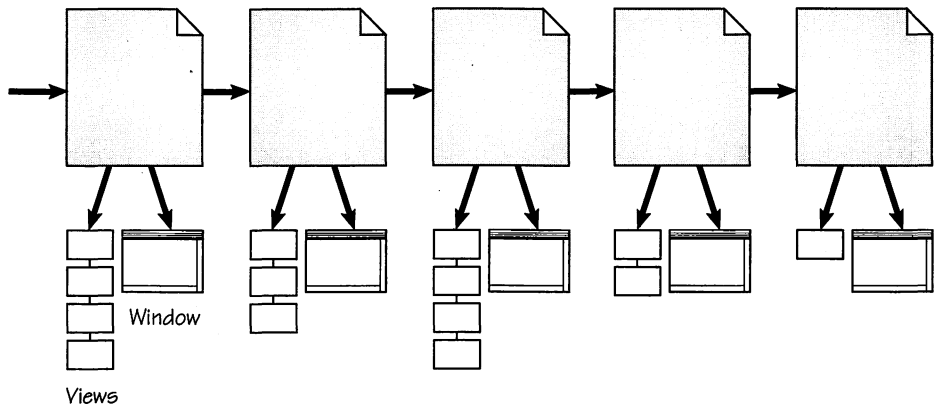


Figure 14-1.
PicoApp's document list—a linked list of document objects, each with its own window object and its own list of view objects.

Creating, Opening, and Adding Documents to the List

The *DoNew* and *DoOpen* methods take care of creating new documents and opening existing ones, respectively. Before either can be implemented, your application subclass must override *TPicoApp.MakeDocument* (which actually creates the new document object) because *TPicoApp* has no way of knowing about your document types.

DoNew creates a new, empty document object. *DoOpen* gets a file name from the user and creates a new, empty document object, which then, as part of its initialization, opens that file and reads it into RAM. After the file has been created, the document object calls *TPicoDoc.OpenDoc* and *TPicoDoc.ReadDoc* methods to open and read the file.

Both *DoNew* and *DoOpen* take care of adding a newly created document to the document list. It's added at the head of the list.

```
procedure TPicoApp.DoNew;
begin
  self.fNewestDoc := MakeDocument;
  if self.fNewestDoc <> nil then
    begin
      self.fNewestDoc.SetDocId(UniqueDocID);
      self.fNewestDoc.fNextDoc := self.fDocList;
      self.fDocList := TPicoDoc(self.fNewestDoc);
      self.fDocCount := self.fDocCount + 1;
    end;  { Else do nothing }
end;  { TPicoApp.DoNew }
```

In *TPicoApp*, *MakeDocument* only sets the global variable *gAllocatingDocument* to *true* and then returns *nil*. The application subclass must override it to create a document of the right type. Here's *TCrapsApp.MakeDocument* as an example of an overridden version:

```
function TCrapsApp.MakeDocument: TPicoDoc;
  override;
  var
    aDoc: TCrapGame;
begin
  { Call inherited version to set flag--returns nil }
  aDoc := TCrapGame(inherited MakeDocument);

  { Now make a real document }
  { Create a new game object }
  { If there was an old one, user must close it before New Game }
  {  and New Options menu items are available }

  aDoc := NewCrapgame(self.fNumberOfPlayers, self.fInitialStakes);
  MakeDocument := aDoc;  { So that game document gets added to document list }
end;  { TCrapsApp.MakeDocument }
```

DoOpen is similar to *DoNew*. It first uses the utility routine *GetFileName* to put up the standard file dialog box to get a file name, which it stores in a local variable, and then it sends a MakeDocument message to create and initialize a document object. As part of its initialization, the document object opens and reads in the file the user has named. Then *DoOpen* installs the document in the list. You can see the *DoOpen* code in the folder PicoApp, Part 2 on the code disk.

RemoveDoc and CloseAllDocs

TPicoApp also has a *RemoveDoc* method that document objects call when they're closing. The method uses a document number that was assigned to the document on creation (by means of the *SetDocID* method) to search the document list. If the document is found, it's removed from the list—but not freed. The document must free itself.

A related method is called when the user is quitting the application. *CloseAllDocs* tells each document in the document list to close itself. As part of that process, each document will call *TPicoApp*'s *RemoveDoc* method.

Maintaining the Cursor Shape

In many applications, it's customary to change the shape of the cursor as the mouse pointer moves over different parts of the screen. When the mouse pointer is moved over a text window, for example, the cursor changes to an I-beam shape. When the mouse pointer moves back over the scrollbars or other parts of the screen, the cursor changes back to the standard arrow shape.

Once each time through the event loop, the application object sends itself an AdjustCursor message, which works with the current mouse pointer location in the local coordinates of the current grafPort (usually the front window).

TPicoApp.AdjustCursor first calls *FrontWindow* to get a reference to the current front window so that it can send a DoCursor message to that window object and then gets the current mouse pointer location in the local coordinates of the front window. The window object, in turn, traverses its list of views, sending each view object a DoCursor message. Figure 14-2 diagrams this transmission of messages.

When a view object receives a DoCursor message, it uses the QuickDraw routine *PtInRect* to see whether the mouse pointer is over the view's click rectangle. If it is, the view calls *SetCursor* to set the cursor to whatever cursor is stored in the view's *fCursor* field. The view's initialization routine can install a special cursor there—or not—as needed. To set itself up as a cursor adjustor, a view object needs only to

- Have the resource ID of a CURS resource in the application's resource fork
- Load that cursor resource into *fCursor*
- Set its *fIsCursorAdjustor* field to *true*

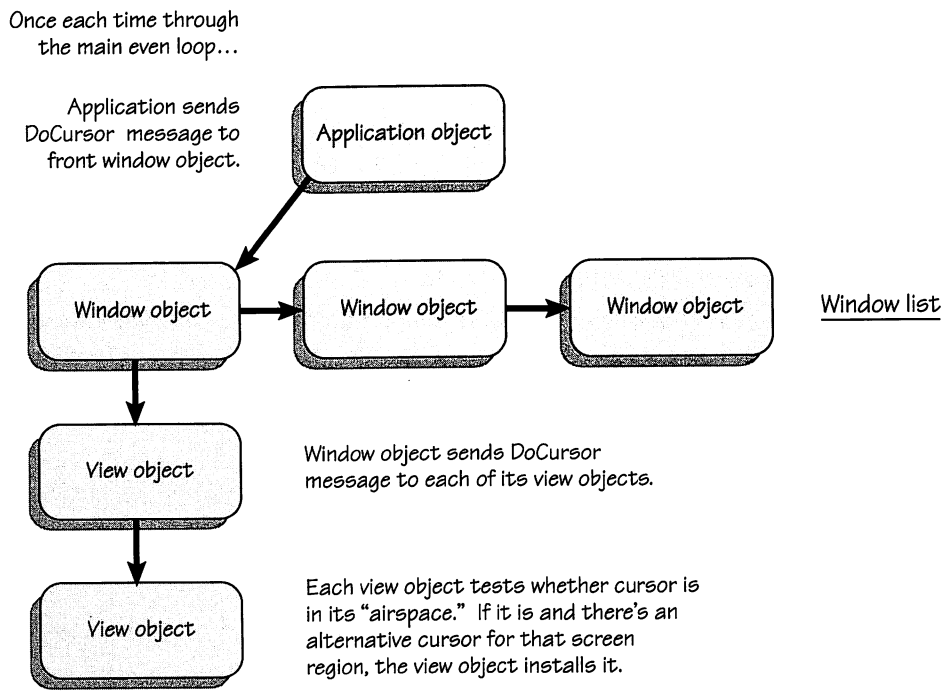


Figure 14-2.
How cursor messages are transmitted through PicoApp's structure.

The rest is automatic. Here are the view object calls for a typical cursor adjustor:

```
{ Replace default cursor with a finger cursor }
fCursor := GetCursor(kFingerID);
fIsCursorAdjustor := true;
```

where *kFingerID* is the ID of some specific CURS resource.

If the view object determines that the mouse pointer isn't over its "airspace," the view object does nothing.

Here's *TPicoApp.AdjustCursor*, which starts the process:

```
procedure TPicoApp.AdjustCursor;
  var
    cursorHandled: Boolean;
    whichWindow: WindowPtr;
    theWinObj: TWindow;
    localMouse: Point;
    oldPort: GrafPtr;
begin
  { The cursor adjustment chain = view list in front window }
  { If cursor is handled by a view, cursorHandled is set to true }
  cursorHandled := false;
  { Determine whether there is a front window, and whether it's a PicoApp window }
  whichWindow := FrontWindow;
  GetPort(oldPort);
  if whichWindow <> nil then
    SetPort(whichWindow);
  { Pretend a mouse click and see what window the cursor is over, if any }
  GetMouse(localMouse);
  if IsAppWindow(whichWindow) then
    begin
      theWinObj := TWindow(GetWRefCon(whichWindow));
      if theWinObj <> nil then
        { Allow views to adjust cursor }
        cursorHandled := theWinObj.DoCursor(localMouse);
    end;
 if not cursorHandled then
    InitCursor;          { Use arrow cursor }
  SetPort(oldPort);
end;  { TPicoApp.AdjustCursor }
```

TPicoView's *DoCursor* method, which all cursor-adjusting views inherit, picks up on the process initiated by *AdjustCursor* and goes on to do most of the work:

```
function TPicoView.DoCursor (where: Point): Boolean;
  var
    wasHandled: Boolean;
    inRect: Boolean;
begin
  wasHandled := false;
  { Checks whether there is a new cursor to install, and whether mouse pointer }
  {  is over this view's airspace }
  inRect := PtInRect(where, self.fClickRect);
  if self.fIsCursorAdjustor & (self.fCursor <> nil) & inRect then
    begin
      SetCursor(self.fCursor^^);  { Yes, so change cursor }
      wasHandled := true;
    end;
  DoCursor := wasHandled;
end;  { TPicoView.DoCursor }
```

NOTE: *In the long run, we might want to take advantage of a program's ability, when it runs under MultiFinder, to update the cursor more efficiently via* WaitNextEvent. *The fourth parameter to that Toolbox call is a screen region in which we want the cursor to maintain its current appearance. If the cursor is ever outside that region, the* DoMultiFinderEvent *method can check the message field of the event record it gets for a value of* $FA *in the first byte. That indicates a "mouse-moved event"—which merely signifies that the mouse pointer is currently not in the designated region. A redesigned* DoMultiFinderEvent *method could call a* MouseMoved *handler to recalculate the mouse pointer region and change the cursor shape while the mouse pointer is in that new region. For now, though, our more object-oriented method is fine, even at the cost of checking the cursor every time through the event loop. See* Macintosh Revealed *(Chernicoff 1985–90).*

Multiple Documents and Memory Management

Scenario: You open document after document--eight, nine, ten of them. But when you try to open the eleventh, the application crashes. Maybe with a polite alert box, maybe with the rudest of all Macintosh symbols, the bomb. This is an unacceptable scenario, of course. Applications should be able to bump gently against the memory ceiling and just as gently recover. The user might have to close an existing document or two, but then he or she should be able to go on working. That's precisely what PicoApp attempts to manage. But it needs help from the programmer.

What Can You Do?

PicoApp has an elaborate memory management subsystem modeled fairly closely on MacApp's. It does a great deal of the work for you. But you must help by

- Structuring your main program as in Crapgame and PicoSketch
- Computing the memory needed by each of the objects you'll allocate
- Always testing that there is enough memory for an allocation by calling PicoApp's special "preflight" methods before calling *New*, *NewHandle*, and so on
- Working with your application to determine how large its memory "reserves" should be and sizing them accordingly

We'll look briefly at each of those tasks. For full details, consult the files in the folder PicoApp Notes in the folder PicoApp, Part 2 on the code disk.

Let Your Main Program Take Advantage of Memory Management

After calling *InitTheMac*, your main program calls a PicoApp routine called *NewMemoryObject*. This creates and installs an initialized instance of the *TMemory* class in the global variable *gMemory*, which provides methods and facilities that help you manage your application's memory effectively. Class *TMemory* is declared in unit *UPAGlobals* and implemented in unit *UMemory*.

If *NewMemoryObject* is successful, you then call your application object's own *New[MyApp]* function. You can model your application's function after similar functions in both Crapgame and PicoSketch. The function should use the preflight methods of class *TMemory* before allocating and initializing your application object. If your application object is successfully allocated, you send it a Run message. When that call terminates, so does the application. See files CrapgameMain.p and PicoSketchMain.p for examples.

Plan How Much Memory Your Objects Will Need

For every object your program will create, compute the amount of space it—and its dependent objects and data—will need. For example, a document object needs space for itself, for its window object, for the window object's *WindowRecord* data structure, for its view objects, and possibly for other objects and some amount of data. Computing the size of an object is easy enough. Add up the sizes of all the object's instance variables. Then add the sizes of its ancestor classes. Finally, add two bytes for the class link. PicoApp declares many common sizes for you as named constants in unit *UPAGlobals*. You can declare additional constants in your unit *UMyGlobals*.

A document object also needs to estimate the amount of data it might have to store. For instance, a text-editing object would need to store a TextEdit *TERec* data structure containing a handle to the text that's read in from a file or typed in by the user. The data estimation will be less exact than that for objects, but do your best.

Then, as we'll see in the next section, you must ensure that your application never allocates an object without first testing to see whether enough space will be available. In particular, when allocating a document, find out whether there's room not only for the document but also for its subordinate objects and its estimated data.

Preflight All Memory Allocations

If you allocate *any* memory with *New*, *NewHandle*, *NewPtr*—even with *SetHandleSize* and *SetPtrSize*—preflight the allocation first.

Preflighting is a technique good programmers use to make their programs more robust. The idea is to try out the allocation in a way that can't crash your program. If that works, then make the real allocation.

Preflighting is so necessary with objects because if a THINK Pascal memory-allocating procedure fails to find enough memory to satisfy the allocation request, it generates System error - 108 (*MemFullErr*) and your application crashes.

So, before you call *New*, you need to call class *TMemory*'s *Preflight* method. It tries out the allocation, returning *false* if it fails. That gives you a chance to recover, or at least to crash more gracefully. If *Preflight* succeeds, it returns *true*, and you can go ahead with a call to *New*, knowing that it will succeed. (*TMemory* also has the preflight methods for *PreflightHandle*, *PreflightPtr*, *PreflightHandleSize*, and *PreflightPtrSize* for ordinary handles and pointers and for resizing handles and pointers.)

Preflight calls a safe memory allocation function to allocate a handle the same size as your object. The Toolbox function *NewHandle* calls upon the Macintosh Memory Manager to do everything it can to find enough space. If *NewHandle* fails, it returns *nil*—not a bomb dialog box. If *Preflight*'s *NewHandle* call fails, *Preflight* returns *false*. But if *NewHandle* succeeds, *Preflight* then deallocates the new test handle so that its space will be available for your subsequent call to *New*.

In the process of locating memory, *NewHandle* repeatedly calls the application's "grow zone function." This is a programmer-supplied function that tries various strategies for finding or creating more space—by compacting the heap, shuffling blocks of memory around, purging unneeded resources and other data, and so on. You install the the grow zone function by calling the Toolbox routine *SetGrowZone*. (See *Inside Macintosh*, II-14, 42.) PicoApp supplies a grow zone function for you, called *PAGrowZone*, which is declared and implemented in unit *UPARoutines*.

The best aspect of PicoApp's grow zone function is that it calls several of class *TMemory*'s methods in its search for memory. Some of *TMemory*'s methods already do useful things in search of memory. You can tailor others to your own needs—by subclassing *TMemory* and overriding the methods to make them do what you want. This gives you a lot of control over how the memory management system tries to find more memory. But PicoApp still does most of the difficult work for you. If you faithfully preflight your memory allocations—all of them—PicoApp will do the rest.

The main way you cooperate with *Preflight* is to write a *New[ObjectType]* function (not a method) for each class you'll be making instances of. These functions should (a) declare a local variable of your object type, (b) call one or more preflight methods, specifying the amount of memory needed, (c) if successful so far, call *New*, *NewHandle*, or whatever to really allocate the object, (d) send the object an initialization message, and (e) return the new object as your function result. For examples, see the many *New[ObjectType]* functions in PicoApp and PicoSketch, such as *NewPicoSketchApp*, in unit *UPicoSketchApp*. Then use these functions to do your allocations.

Help PicoApp Set Aside Memory Reserves for a Rainy Day

Like MacApp and TCL, PicoApp sets aside two chunks of memory as reserves, which it can use to meet the needs of both the system and the application when memory gets tight. TCL calls its main reserve a "rainy day fund." MacApp and PicoApp call their reserve funds the "temporary reserve" and the "emergency reserve."

The temporary reserve ensures that the Macintosh system never fails to find enough memory to load such critical resources as your application's CODE resources. The system is bad-mannered enough to allocate space in your application heap without asking. Moreover, if it fails to find enough space, it crashes your program. So we set aside extra space to keep the monster satisfied and stave off such a disaster.

The emergency reserve is for absolute last-ditch situations. If all else fails, PicoApp will use up the emergency reserve (by purging it with the *EmptyHandle* Toolbox call). When PicoApp goes to the emergency reserve, it signals its distress by putting up an alert box that asks the user to close documents and delete unnecessary data (such as an overlarge Clipboard) until the shortage can be alleviated. To enforce discipline, PicoApp also disables all menu items that might result in more memory allocation requests before the crunch is over.

After the crisis passes, PicoApp reallocates both reserves and lets the user continue. It's far better to make the user close one or two out of many documents, say, than to give him or her a bomb dialog box.

Your job, as PicoApp programmer, is to set reasonable sizes for the temporary and emergency reserves and to override the appropriate menu-fixing methods for your objects to ensure that they disable the right menus when the global *gLowMemory* flag is *true*. Look at Crapgame's *TCrapsApp.FixFileMenu* method for an example of a menu-fixing method override.

You can size your reserves accurately only by setting reserve sizes (in your *NewMemoryObject* call in the main program) and experimenting with various loads on your application until it recovers smoothly from memory shortages. See the files in the folder PicoApp Notes on the code disk for further details of all of these issues and techniques.

Implementing Undo

According to the Macintosh user interface guidelines, any user action that affects a document's data should be undoable (and then redoable). Some actions, of course, are impossible to undo—such as saving the document on disk—and some actions that can't be undone warrant putting up an alert box:

> Warning: This action cannot be undone.

In such cases, the Undo item in the Edit menu should be disabled, but whenever possible, Undo should be implemented.

Currently in PicoApp, there is no real facility for undoing user actions. *TPicoDoc* has an *Undo* method, but it's a stub. With some work, we could implement an undo mechanism similar to MacApp's. We'll merely look at an outline of the scheme here.

Undo in MacApp

In MacApp, when a view's methods take some action that affects the document's data, the methods don't act directly. They create a "command object" and return it to the application, which then executes the command object's *DoIt* method. Subsequently, if the user chooses the Undo command (before another command object comes along), the application calls the current command object's *UndoIt* method. Then, if the user next chooses *Redo*, the application calls the command object's *RedoIt* method. Of course, the command object must preserve the data's previous state for undoing and redoing.

Command objects

A command object is an odd sort of creature that knows how to do a command. When the user chooses a menu item, for example, the *DoMenuCommand* method that eventually handles the item creates a command object that knows how to carry out the action. Suppose we choose a menu item that inserts the current date into a spreadsheet cell. The menu-handler method creates a command object that knows how to get the current date from the system, how to locate the current cell, how to save a copy of what the cell currently contains, and how to insert the date. Command objects can be used for just about any kind of action we can envision, including mouse drawing, typing, and so on.

Here's a sketchy version of what a MacApp-style command object class looks like:

```
type
  TCommand = object(TObject)
      procedure DoIt;
      procedure UndoIt;
      procedure RedoIt;
      procedure Commit;
    end;  { Class TCommand }
```

DoIt does the action—your subclass of *TCommand* overrides *DoIt* to specify what and how. You'd also supply any needed instance variables, references to objects, and so on. *UndoIt* reverses what *DoIt* does, and *RedoIt* does it again. *Commit* can be used to help handle actions that are particularly hard to undo. In a case in which *DoIt* would change the data structure in a way that would make undoing too difficult, you can "filter" the action—show the intended output without changing the data structure. Then, if the user moves on to another action, the previous action is "committed"—and the data structure is changed. Wilson, Rosenstein, and Shafer talk about this (Wilson 1990), and we'll go into it a little in the next section.

An undo scheme for PicoApp

We could use command objects in PicoApp, too, although slightly differently. Instead of directly taking action, a click-, key-, or other event-handling method would create a command object and then call an application method to handle the event. The application method would store the command object in a field *fLastCommand* and call the command object's *DoIt* method. *DoIt* would handle the event and then set the Edit-Undo item to read "Undo Typing" or something appropriate. You would specify what the item's name should be when you wrote the command object subclass. Figure 14-3 shows the view object creating a command object to pass to the application object and the application object using the command object to undo a command.

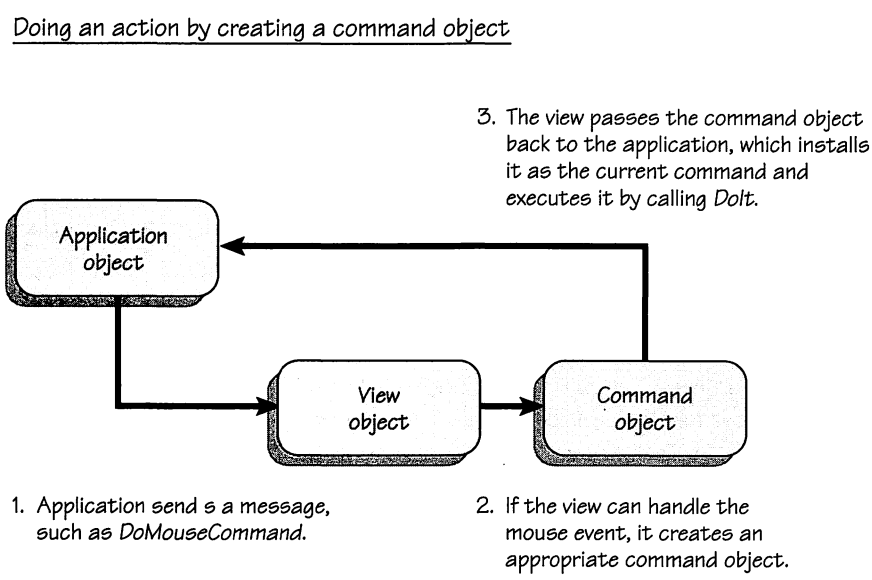
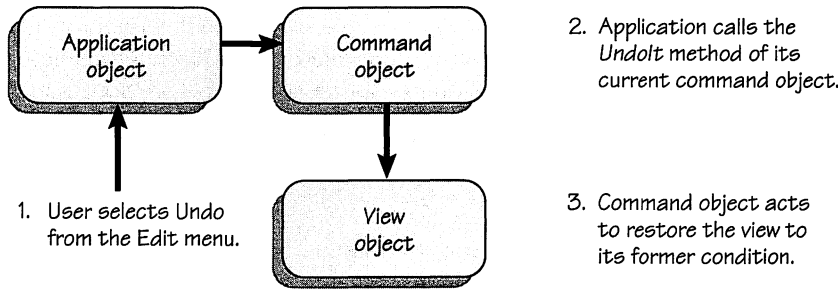


Figure 14-3. *(continued)*
Doing and undoing an action by using command objects.

Figure 14-3. *continued*
Undoing an action



If, before a new command object is installed in the application object's *fLastCommand* field, the user selects Edit-Undo, the application's *DoUndo* method is called by means of normal *DoMenuCommand* activity. *DoUndo* calls

```
fLastCommand.UndoIt;
```

where *fLastCommand* contains the most recent command object. This undoes the effect of the original call to *DoIt* and sets Edit-Undo to read "Redo Typing" or something appropriate.

Next, if the user selects Edit-Redo (before a new command is installed in *fLastCommand*), the application object's *DoUndo* method calls

```
fLastCommand.RedoIt;
```

This restores the undone data to its state before the call to *UndoIt* and sets Edit-Redo to read "Undo Typing" or whatever.

When a new command object is installed in the *fLastCommand* field, we start over.

Undoing Cut, Copy, Paste, and Clear

Undoing Cut involves pasting the data back. Undoing Copy involves clearing the Clipboard (although we might want to merely pretend that we have, to make Redo easier—see "Committing an Action"). Undoing Paste involves cutting the data again. Undoing Clear involves our having saved the cleared data in some alternative clipboard so that it can be retrieved. In order to execute all the variations of these commands, we really need a second clipboard in which to preserve the data's most recent state.

Committing an action

To implement a *Commit* capability in PicoApp, we could install a filter object in the command object. If the filter were not *nil*, the command object's methods would use the filter to show the changes but not make the changes in the data structure.

But if the filter were *nil*, the commands would go into effect immediately. If filtered, a command would be committed when the next command was installed in the application object's *fLastCommand*. (The application object's *DoCommand* method would need to save the previous command object long enough to commit it before installing the new command object.) To commit, we'd remove the filter and re-execute the command. For now, we won't go into the design of a filter object, but see Wilson, Rosenstein, and Shafer (Wilson 1990).

We've left some issues hanging: how to design filter objects, how to implement showing the changes without altering the data structure (as in a TextEdit record), the complete details of PicoApp *Undo*. We'll leave the actual implementation as a project.

PicoApp Design Alternatives

You'll notice that class *TPicoApp*, PicoApp's application class, is pretty large, with a lot of methods. Sometimes, in a programming environment with relatively little RAM, size can be a problem. As it stands now, PicoApp will compile in 1 megabyte, but an alternative design might make it a little smaller.

Let's very briefly consider other ways in which we might have handled class *TPicoApp*. An obvious approach is to break up that one monolithic class into several smaller classes that work together—to form the template of a composite application.

Some of the event-handling apparatus could be moved to a separate *TEventStream* object. Or the menu creation and updating methods could be moved into a separate *TMenuBar* object. Take a look at the THINK Class Library's *CBarTender* class for an example.

On initialization, the application object could create the other, cooperating objects, such as *TMenuBar*, and install them in instance variables or globals for easy reference. Then, to get things done, the application object would send messages to its partner objects. Figure 14-4 on the next page shows one possible design for a composite application object.

Our straightforward design is hardly the last word in what can be done with objects in building an OOP application framework. Be creative, and study not only the PicoApp code but the design of TCL, MacApp, and other frameworks.

Global Variables

In PicoApp, as in MacApp and TCL, we designate a global variable by prefixing a *g* to its name. For most of PicoApp's globals, see unit *UPAGlobals*. As it stands, PicoApp probably has too many globals. If we've been a bit sloppy, some of them might be demotable to instance variables of some object.

Ideally, we would have only a few globals. An important principle of software engineering is to hide details, to keep everything as local as possible, which protects code from corruption by outsiders.

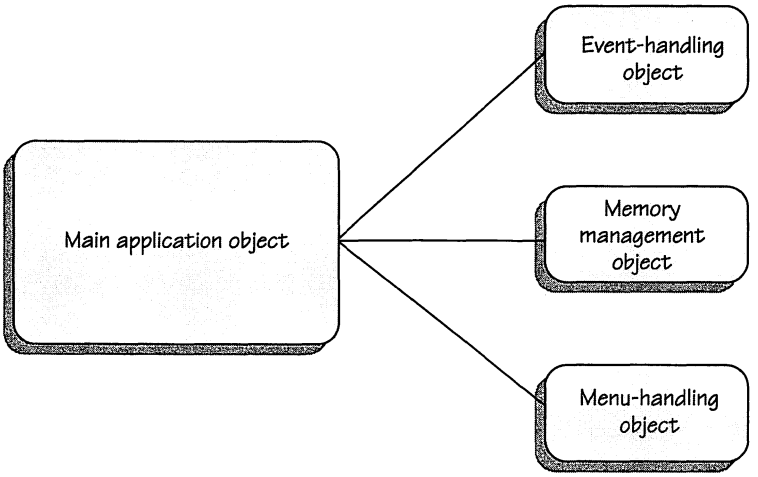


Figure 14-4.
A composite application. The application object coordinates the activities of its satellites, farming out portions of its responsibilities to them.

That's another way of saying "encapsulate." Objects are a vehicle for encapsulation, so objects are the likely source of a good solution to the problem of too many globals.

One way to reduce the number of globals is to encapsulate groups of them in objects. For example, in an earlier chapter we suggested that, as a project, you design a *TMacintosh* class. The application object would create one instance of the class and send it messages to get information like the current screen size, whether color is available, and so on. Anything having to do with the state of the Mac and its hardware and software environments would be initialized by, and obtained from, a *TMacintosh* object, by means of its methods.

Certain of PicoApp's globals would naturally fit into such an object. Other such objects should suggest themselves after you examine the globals. Two global objects currently used in PicoApp are *gApplication* and *gMemory*. Each encapsulates several instance variables that might otherwise have been global variables.

If you implemented a strategy of this sort, instead of lots of global variables you'd have two or three global object references, and you'd call their methods—just as we currently store our application object in *gApplication* and our memory object in *gMemory* and then call methods of those global objects.

Summary

In this chapter, we completed the work we began in Chapter 13. We looked at how PicoApp creates document objects and maintains them in a document list; at how it allows you to change the cursor shape as needed; at how it manages memory so that

your users don't run out in the middle of some crucial operation; at how you might implement an undo capability for PicoApp; and at how alternative designs for some of our PicoApp classes might work.

Projects

- Write a small PicoApplication that can both create and open documents of some kind (such as text-only documents). What parts of PicoApp did you have to override? How much work was required? How much work do you think would have been required without PicoApp's facilities? How would you extend PicoApp to make the job easier?
- Improve cursor-management so that you can control the cursor shape over not only a whole view but also over a specified region within a view.
- Implement at least part of the scheme we outlined for an undo mechanism.
- Redesign PicoApp as a composite application framework. Break up *TPicoApp* (and perhaps other classes) into several smaller classes that form a composite.
- Implement some global objects to reduce the number of global variables in PicoApp.

15

INSIDE PICOAPP: DOCUMENTS, WINDOWS, AND VIEWS

Having seen how PicoApp handles application object functions, let's turn to more details about document, window, and view objects:

- How the document object manages data, windows, and views
- How the window object mediates display
- How the view objects interact with the user
- How the three classes work together
- How PicoApp handles drawing with the mouse

And we'll look at a second sample PicoApplication called PicoSketch.

Overview: Document, Window, and View

Applications create documents. In your word processor, a document contains text data and displays it in a text editing window. In your spreadsheet, a document contains numbers, formulas, and labels as data and displays them in a grid inside a window. Thus, documents manage your data and arrange for its display and update in windows. What's actually shown in a window depends on the nature of the views.

Documents create one or more windows and one or more views. Windows are pretty standard "objects," familiar to all Macintosh users. But views are something special. If you can point to it, name it, or click it with the mouse, it's a view. Even windows are views, although of a very special kind.

Windows display views. In particular, when the application notifies a window of an update event, the window sees to it that all of its views draw themselves.

Views display data. Sometimes the data is in the form of controls, icons, and similar items. But often it's in the form of text, numbers, or graphics. A view knows what data it's supposed to display and how and where to display it. Its window tells it when to display it.

How Documents Work

A document, in OOP terms, is a pretty abstract entity. It's really a kind of control center that manages the storage, maintenance, and display of your program's data. It manages storage by reading your data from a disk file and saving it again after you edit. It maintains your data as you update it, keeping track of changes so that they'll be reflected accurately in the display and written out accurately to the disk. It manages data display through its window(s) and views, creating them and giving them access to the data—and letting them handle the dirty work. Figure 15-1 diagrams the document's mediation of file reading and writing, data structure maintenance, and data display.

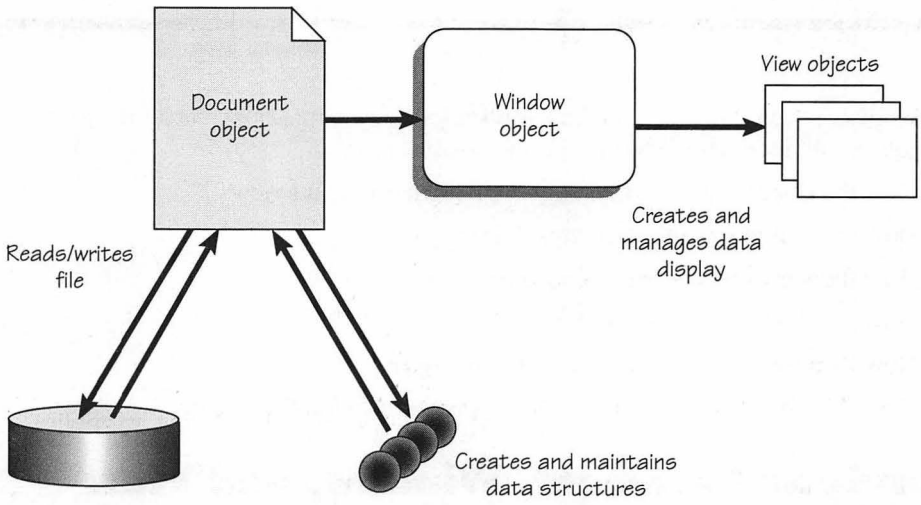


Figure 15-1.
Document interactions for storage, maintenance, and display.

Data Storage

Storage of data usually means putting it in a disk file. In PicoApp, MacApp, and TCL, the application object doesn't handle data storage. The document object does. The document object knows how to read a file and how to write one, using methods. Of course, the document object must have special knowledge of the format your data

takes on disk. Is it a text-only file; a file containing text plus formatting information (as in Microsoft Word, for instance); a file containing the bits or pixels of a painting; a file containing data about rectangles, circles, and other elements (as in MacDraw); or some other format? Sometimes an application makes more than one file format available. It's the document object that has methods for reading and writing your particular kind of data.

Although data storage is always a custom task, MacApp and TCL provide some help. TCL, for instance, provides classes *CFile* and *CDataFile*. By providing some basic file-handling abilities, the abstract class *CFile* gives you a head start in building your own disk file management. *CDataFile*, a subclass of *CFile*, goes further by providing a complete implementation for reading and writing raw bytes. But if you need to read or write structured data—records—you have to subclass *CDataFile* to provide translation between raw bytes and your structure. Wilson, Rosenstein, and Shafer (Wilson 1990) provide the class *TStream* and subclasses for MacApp that can translate raw bytes into a data structure. They also present the design of a class that knows how to read and write objects. Writing arbitrary objects to disk and reading them is not a trivial undertaking, and the design they present is worth study. So are the stream classes that come with Turbo Pascal versions 5.5 and 6.0 for the IBM PC, although of course they aren't suitable for use on the Mac.

PicoApp doesn't yet implement disk file management, but Wilson et al. (Wilson 1990) point the way in their discussion of Streamobject.

Data Maintenance

After a document has read in your data from disk or gotten the data by means of user input, it must manage data structures to contain it.

For text, the data structure might be a TextEdit record or something similar. For a spreadsheet, the data structure might be a sparse array implemented with pointers (or with cell objects). For a drawing program, the data structure might be an array or a linked list of shape objects.

The document object creates and manages these data structures during an application run and disposes of the structures afterward.

The document object also sees to it that view objects have access to the data they are to display.

When you design your document object classes, you need to provide the appropriate data structures and the methods to put data in and take it out. As always, this is one of the more important tasks in program design.

In Crapgame, for example, we provided a player list class, an instance of which can hold any number of player objects and maintain information about which player's turn is next. Class *TCrapGame* (our document) provides methods to add players, remove them, report who's up next, and so on.

Data Display

Display of data is managed indirectly in PicoApp. The document object is responsible only for creating the window and the view objects—and for giving them access to the data they display.

Windows are created by the document method *MakeWindows*, which in turn calls a method *MakeViews* to make the views.

In PicoApp, you have to override *MakeWindows*—to call utility routines that make the kind of window you need and to retitle the window. (PicoApp calls it "Untitled.") You also have to override *MakeViews* because PicoApp can't possibly know what your views are, how many there are, what they do, or anything else about your views. Making views will be your biggest task in writing your document object class.

In Crapgame, *MakeViews* makes two different sets of views. The main set, which we call the "main view list," contains the current player object, the throw and statistics buttons, the title text, a border, and the dice. An alternate view list contains the current player's statistics view object, for displaying that player's record of how the dice are rolling. *MakeViews* creates and initializes each view by calling a utility routine that calls *New* and asks the view to initialize itself. *MakeViews* then tells the view what window it belongs to and what view is next in the list after it.

Listing 15-1 shows Crapgame's overridden version of *MakeViews*, which sets up view rectangles and then creates the views one by one, linking the views into a view list as it goes.

```
function TCrapGame.MakeViews: Boolean;
  override;
  var
    player: Integer;
    nextPlayer, newPlayer: TCrapsPlayer;
begin
  { Set up view rectangles }
  SetUpRects(self);

  { Create game's play views:  border, title, dice, button, players }
  self.fBorderView := NewBorderView(2, self, self.fBorderRect);
  if self.fBorderView = nil then
    begin
      MakeViews := false;
      Exit(MakeViews);
    end;
```

Listing 15-1. *(continued)*
Crapgame's override of MakeViews.

Listing 15-1. *continued*

```
    self.fBorderView.SetWindow(self.fItsWinObj);
    { Make the border view the end object in the subview chain }
    self.fBorderView.SetSub(nil);

    self.fTitleView := NewTitleView('Crapgame', self, self.fTitleRect);
    if self.fTitleView = nil then
      begin
        MakeViews := false;
        Exit(MakeViews);
      end;
    self.fTitleView.SetWindow(self.fItsWinObj);
    { Put the title view in the subview chain }
    self.fTitleView.SetSub(self.fBorderView);

    { Create and initialize the dice }
    self.fDie1 := NewDie(6, self, self.fDie1Rect);
    if self.fDie1 = nil then
      begin
        MakeViews := false;
        Exit(MakeViews);
      end;
    self.fDie1.SetWindow(self.fItsWinObj);
    { Put die view in the subview chain }
    self.fDie1.SetSub(self.fTitleView);

    self.fDie2 := NewDie(6, self, self.fDie2Rect);
    if self.fDie2 = nil then
      begin
        MakeViews := false;
        Exit(MakeViews);
      end;
    self.fDie2.SetWindow(self.fItsWinObj);
    { Put die view in the subview chain }
    self.fDie2.SetSub(self.fDie1);

    { Create a Throw button object, passing a reference to the game object }
    {  to the button; the button calls the game }
    self.fThrowButton := NewThrowButton(self, self.fButtonRect);
    if self.fThrowButton = nil then
  begin
        MakeViews := false;
        Exit(MakeViews);
          end;
```

(continued)

Listing 15-1. *continued*

```
self.fThrowButton.SetWindow(self.fItsWinObj);
{ Make the button respond to press of Return and Enter keys }
self.fThrowButton.MakeDefault(true, false);
{ Put button view in the subview chain }
self.fThrowButton.SetSub(self.fDie2);

{ Create the game's statistics-view button }
self.fStatsButton := NewStatsButton(self, '', true, self.fStatsRect, false);
if self.fStatsButton = nil then
  begin
    MakeViews := false;
    Exit(MakeViews);
  end;
self.fStatsButton.SetWindow(self.fItsWinObj);
{ Put the statistics-view button in the subview chain }
self.fStatsButton.SetSub(self.fThrowButton);

{ Install the subview chain in fPlayViews }
self.fPlayViews := self.fStatsButton;

{ Create all the player-view objects }
self.fPlayerRects[1] := self.fPlayerNumRect;
self.fPlayerRects[2] := self.fPlayerTitleRect;
self.fPlayerRects[3] := self.fScoreRect;
self.fPlayerRects[4] := self.fScoreTitleRect;
self.fPlayerRects[5] := self.fPointRect;
self.fPlayerRects[6] := self.fPointTitleRect;
self.fPlayerRects[7] := self.fMessageRect;
self.fPlayerRects[8] := self.fStatsViewRect;

for player := 1 to self.fNumberOfPlayers do
  begin
    NewPlayer := NewCrapsPlayer(player, self, self.fPlayerRects, self.fDie1,
        self.fDie2, self.fStakes);
    if newPlayer = nil then
      begin
        MakeViews := false;
        Exit(MakeViews);
      end;
    self.fPlayers.AddPlayer(newPlayer);
      { Pass its player number and references to game (self) }
      {  and the dice }
      { Players must call game and must call dice in order to roll them }
  end;  { for }
```

(continued)

Listing 15-1. *continued*

```
    { Add the current player to the play views list }
    nextPlayer := TCrapsPlayer(CurrentShooterOf);
    { Point player at head of subview chain }
    nextPlayer.SetSub(self.fPlayViews);
    self.fPlayViews := nextPlayer;      { Point list at player }

    { Make fPlayViews the currently active view chain }
    self.fCurrentViews := self.fPlayViews;

    { Initialize fStatViews view chain }
    self.fStatViews := nextPlayer.StatsObjectOf;

    MakeViews := (self.fCurrentViews <> nil);
end;  { TCrapGame.MakeViews }
```

Notice how the view lists are built. Each view object's *fSubview* field points to the next view in the list. The last view has an *fSubview* of *nil*. Figure 15-2 shows a view list's structure.

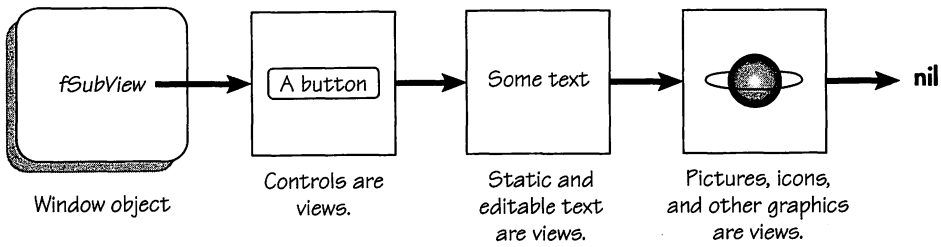


Figure 15-2.
The view list—a simple linked list of view objects. Each object has an fSubView *field that points to the next view object.*

Crapgame's View Switching

When the user clicks the transparent statistics button overlying the dice, Crapgame switches view lists by means of the method *SwitchViews*; that is, it replaces the current value of *TCrapGame.fPlayViews*—normally *fCurrentViews*, which points to the main view list—with *fStatViews*, which points to the current statistics object. Then the window redisplays its views, calling the statistics view to display itself. Clicking again (now anywhere in the window because the statistics button grows to cover the whole window in statistics view mode) reswitches the two view lists.

Documents and Menus

The application object puts up most of the menus and manages some of them, as we've seen. In particular, class *TPicoApp* is responsible for putting up an Apple menu, a File menu, an Edit menu, and any other menus referred to by the MBAR resource that you edited in your resource file. Your subclass of *TPicoApp* must obtain handles to menus other than the standard three.

TPicoApp provides methods for handling some of the menus. It can handle all of the Apple menu, putting up your About dialog box and running desk accessories. It can handle the Close and Quit commands in the File menu. It provides part of the apparatus needed for the New and Open commands in the File menu. Your application subclass is responsible for filling in the details of New and Open, via *MakeDocument*, because *TPicoApp* can't know about your document type. Who handles other menu items? Typically, your document objects do.

Document menu items

Once a document object has been created to manage a document, it is responsible for reading and writing the data, saving it, reverting to a previous version, and printing it, among other things. We've already reviewed most of those functions—they pertain to disk storage and appear as items in the File menu. We'll have a look at printing shortly. First, let's look at how documents handle some menu items.

Documents as menu handlers

Recall that any object class descended from *TEvtHandler* has a *DoMenuCommand* method. It can either inherit the method directly, if it merely wants to pass menu events along without handling them, or it can override *DoMenuCommand* to handle menu events itself.

TPicoDoc also provides hook methods that you can override to support the Edit menu's Cut, Paste, and Undo commands. To handle the menu items it's responsible for, your document object provides an overridden version of *DoMenuCommand* containing a *case* statement something like this:

```
case theMenu of
  kFileMenuID:
    case theItem of
      kSaveItem: DoSave;  { Menu items your document object knows }
      kSaveAsItem: DoSaveAs;
      kPageSetupItem: DoPageSetup;
      kPrintItem: Print;
      :
      otherwise
        ;
    end;  { case--File menu }
  kEditMenuID:
```

To handle each menu item, the *case* statement calls another document object method.

Putting up additional menus

Yes, most menus are put up and managed by the application object, but in PicoApp, it's entirely possible for other objects, such as documents and views, to put up menus of their own.

To put up its own menu, an object can use the Toolbox routine *GetMenu* to get a MENU resource from the resource file, then call *AddMenu* to put the menu in the menu bar, and finally call the PicoApp routine *RedrawMenus* (declared in unit *UPARoutines*). This would probably occur in the object's initialization method.

Of course, the object putting up the menu is also responsible for enabling items in that menu at the appropriate time. To be able to do that, the object class must override *FixMenus*. *FixMenus* should call a new method to fix items in the object's menu. For example, if an object puts up an Options menu, its *FixMenus* should call a *FixOptionsMenu* method, which you also add to the class.

Objects that put up their own menus are also responsible for processing menu commands received from the application by means of *DoMenuCommand*.

Neither Crapgame nor PicoSketch has such extra menus, but the potential is there if you should need it.

Printing

Your document object should have a *Print* method if it knows how to print its data. *Print* most likely functions by setting up a printing port and then getting the document's views to draw themselves into it. You'll probably have other methods to manage the Page Setup command and other aspects of printing. Ultimately, *TPicoDoc* should provide some of this capability.

Printing in MacApp and TCL

In MacApp, printing is handled by objects of class *TPrintHandler*. Special "handler" objects are associated with a document. They know how to print the document's data—perhaps various views of it. TCL does something similar, associating an object of class *CPrinter* with each document. The object knows how to manage communications between the document and the Macintosh Print Manager. The object puts up the necessary dialog boxes and manages a print record. The document provides a *PrintPageOfDoc* method to print each page. *CPrinter* calls the *PrintPageOfDoc* method at the appropriate time.

Printing in PicoApp

So far we haven't implemented printing in PicoApp. Look for it in a future instantiation. Meanwhile, be thinking about how you would implement it.

How Windows Work

In PicoApp, a view is associated with a particular window object. The window manages the flow of display-related events to its list of views.

Window Records and Window Objects

We've already gone into the trick we borrowed from TCL in order to integrate our *TWindow* objects neatly into the Macintosh's existing Window Manager: We make the standard Macintosh window record contain a reference to a *TWindow* object, which, recursively, contains a *WindowPtr* reference to the same window record. Figure 15-3 shows the relationship between the window record and the *TWindow* object.

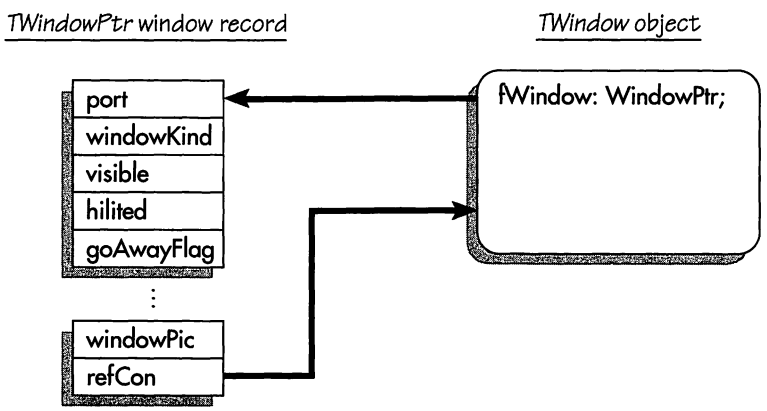


Figure 15-3.
WindowPtr *and* TWindow. *The standard Macintosh window record uses its* refCon *field to point to a* TWindow *object, which in turn has an instance variable of type* WindowPtr *that points to the window record.*

Associating a *TWindow* and a *WindowPtr*

When the document creates a window record, by calling the Window Manager routine *NewWindow*, it also creates a *TWindow* object. The window record's window pointer (of type *WindowPtr*) is stored in the *TWindow* object's *fWindow* field.

To make the other connection, we use a Window Manager routine called *SetWRefCon*, which installs a long integer value (or a pointer or a handle) in the window record's *refCon* field. The following code appears in *TWindow*'s *ISimpleWindow* method:

```
self.fWindow := MakeSimpleWindow(self.fViewRect, docNum, hasClose, hasZoom,
    hasGrow);
OffsetWindow(self.fWindow);
SetWRefCon(self.fWindow, Longint(self));  { Install winObj in refCon }
```

Notice how we have to typecast our object reference to a long integer in the *SetWRefCon* call to match what that routine expects.

Getting the Message

Window objects handle a number of kinds of events:

- Updates
- Activates
- Resizings and zooms
- Drags

When the application's *DispatchEvent* method gets one of these window events, it determines which window is involved. For mouse events, it's the window that was clicked. For updates and activates, it's the window obtained from the event record's message field.

We use *fWindow* to obtain references to a window's window object and its document object. Getting these references is an interesting process. Let's take a quick look at how you might do this:

```
if self.fWindow <> nil then  { Can't access a nil pointer }
  begin
    theWinObj := TWindow(GetWRefCon(self.fWindow));
    theDocObj := theWinObj.DocumentOf;
  end;
```

Be sure to check that *fWindow* isn't *nil* (as it would be for a mouse click in the menu bar or the desktop, say) because we can't get the *refCon* value of a *nil* window pointer. The first line inside the *if* block uses another Window Manager routine, *GetWRefcon*, which returns the contents of the window pointer's *refCon* as a long integer. So of course we have to typecast the long integer to a *TWindow*. Then, in the second line, we can call a method of the window object to get a reference to its document object.

Once we've established the connection between the window pointer and its *TWindow* object, we can use the *TWindow* reference to call *TWindow* methods, just as we used the reference to obtain the document reference in our example. In particular, we can call the window object's *DoUpdate*, *DoActivate*, *Grow*, *Zoom*, *Drag*, and other methods. Some of these methods in turn call Window Manager routines from the Toolbox such as *DragWindow* by passing the value of a *TWindow* object's *WindowPtr* field, *fWindow*, as a parameter to the Toolbox routine. Figure 15-4 on the next page shows the flow of event messages.

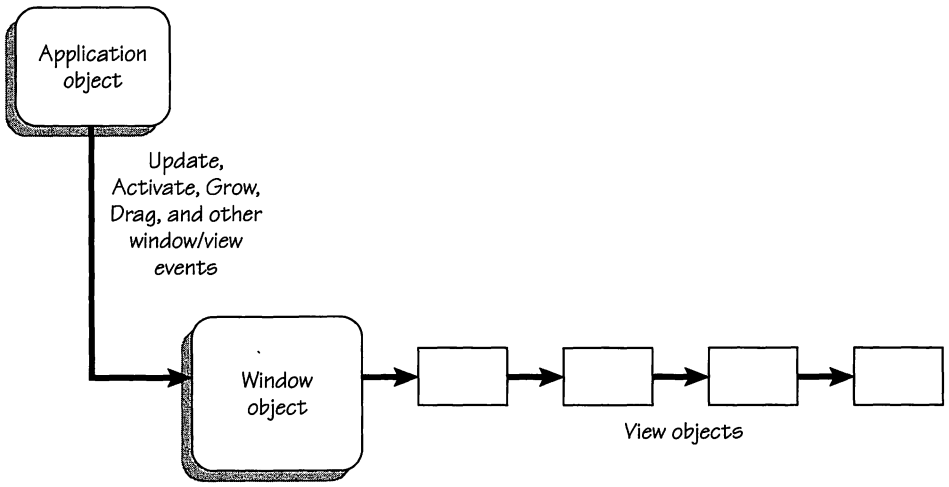


Figure 15-4.
Window messages. The application gets and dispatches window/view events to the appropriate window and then, if warranted, to the window's view list.

Display by Update

Sometimes, for speed and timing, you'll draw directly in a Macintosh program, calling QuickDraw routines or calling a view object's *Display* method, which in turn calls QuickDraw. The more usual way of drawing in a Macintosh program is to change the data in your document's data structure and then trigger an update event, often by calling the Toolbox's *InvalRect* or *InvalRgn* routine. When the event loop gets an update event, it dispatches the event to the affected window object.

In PicoApp, the application object dispatches an event by calling the *DoUpdate* method of the appropriate window object. *DoUpdate* then calls *DisplayViews*, which traverses the window's view list, sending each view a Display message. Even though we ask all views to redraw themselves, the actual drawing that gets done is clipped to the area(s) requiring redrawing.

Other *TWindow* Methods

TWindow has a lot of methods. Some are for accessing the data in the underlying window record. Some are for carrying out normal Window Manager actions, such as updating, activating, resizing, zooming, dragging, moving, or deactivating a window. Some are for manipulating the window list, moving the window object around in the list, and the like. And some are hooks. Some *TWindow* methods are used only by PicoApp, but you might find others useful.

Window Closing Is Document Closing

When the user clicks in a window's close box or chooses Close from the File menu, it's actually the window's document that ends up getting called.

TPicoApp.Close is called in two different situations: a mouse-down in a window's close box or the selection of the Close menu command, which is handled by *DoMenuCommand*.

The *Close* method uses the selected window's document reference (obtained through *fWindow*, as we saw earlier in this chapter) to call the document object's *Close* method.

The document object then

- Calls its window object's *Close* method to free itself and its window record
- Calls the application object's *RemoveDoc* method to get itself removed from the document list
- Calls its own *Free* method, which first calls *FreeViews* to free each view and then frees the document object

Thus, the window object gets called to close itself as part of the document-closing process. Figure 15-5 shows the flow of events in closing a window/document.

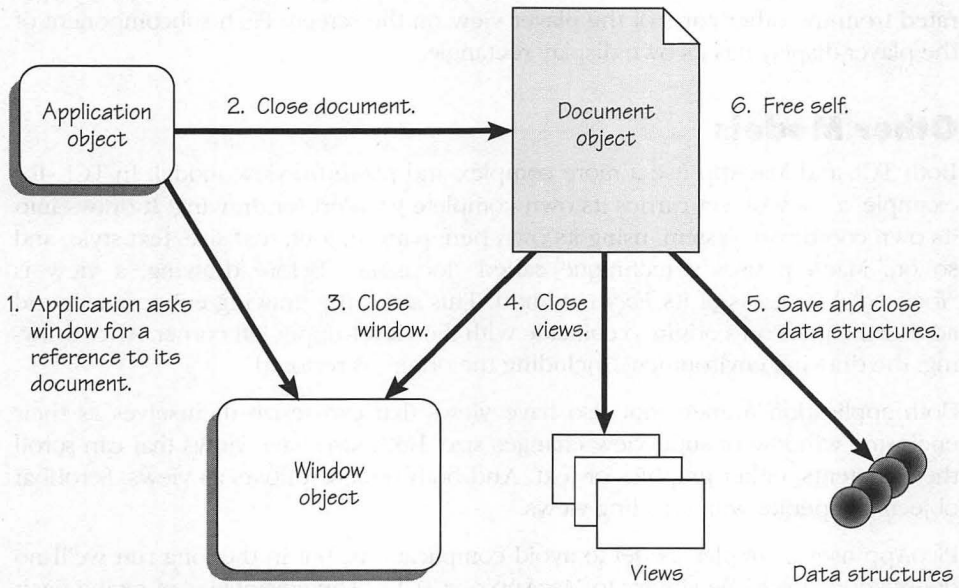


Figure 15-5.
Document and window closing.

How Views Work

Documents and windows do most of their work behind the scenes. The view objects are out front. They're responsible for two crucial and complementary tasks: displaying information, data, and controls for the user and taking user input by means of the mouse and the keyboard.

Views and Display

As we noted in Chapter 11, a view is simply some distinct visual entity on the screen. Windows, buttons, scrollbars, fields of text, and graphics qualify as views and are represented by view objects that know how to draw them.

The key to display is the view object's *Display* method, which the view object overrides from *TPicoView*. Of course, each view object's *Display* method works differently. A button object is a view object that draws a button. A static text object draws a string of text. And so on.

But the *Display* methods have some features in common. In all cases, the drawing occurs in one or more view rectangles. Every Crapgame view object, for instance, displays itself in the rectangle stored in its *fViewRect* field. Sometimes a view needs a more diverse region for display. Crapgame's player display consists of three stacked lines of text with boxes to the left of them plus a message display area that is separated from the other parts of the player view on the screen. Each subcomponent of the player display has its own display rectangle.

Other Models

Both TCL and MacApp use a more complex and powerful view model. In TCL, for example, a view object carries its own complete grafPort for drawing: It draws into its own coordinate system, using its own pen, pattern, font, text size, text style, and so on. MacApp uses a technique called "focusing": Before drawing, a view is "focused" by means of its *Focus* method. This saves the drawing environment and adjusts the grafPort's origin to coincide with the view's upper left corner. After drawing, the drawing environment, including the origin, is restored.

Both application frameworks also have views that can resize themselves as their enclosing window or superview changes size. Both also have views that can scroll their contents, either graphics or text. And both treat scrollbars as views. Scrollbar objects cooperate with scrolling views.

PicoApp uses a simpler model to avoid complications, but in the long run we'll no doubt adopt a scheme similar to MacApp's or TCL's. The advantages of giving each view its own grafPort, for instance, are certain to be worth the extra complexity.

As PicoApp is now, we have to manage a window's views in relation to the window's grafPort. Both MacApp and TCL use a more sophisticated "hierarchy" of views (not to be confused with an object class hierarchy) in which each view object has a

superview and possibly one or more subviews. One large view installed in the content region of a window might manage a number of subviews, each in a rectangular area of its superview. TCL and MacApp also provide great flexibility in the ways in which a subview and its superview relate to each other. Some subviews might scroll when their superview scrolls. Others might not. Some subviews might resize as their superview does. Others might not.

TCL's views

TCL provides class *CView*, which implements the event-handling aspects of a view, plus several subclasses:

- *CPane* implements resizable views and adds the display elements to *CView*.
- *CPanorama* provides for views larger than the available display area.
- *CScrollPane* mates a *CPanorama* with scrollbars to provide scrolling.

TCL also includes windows and the desktop as views. *CDeskTop* is the top of the view hierarchy, acting as an enclosure to all windows. A subclass of *CDeskTop*, called *CFWDeskTop*, implements floating windows. Other view classes include all controls (buttons, scrollbars, pop-ups), pictures, static text, edit text, size boxes, and borders.

MacApp's views

MacApp's view subclasses include

- *TTEView*, for text editing
- *TDialogTEView*, for text editing in dialog boxes
- *TDeskScrapView*, for displaying the contents of the Clipboard
- *TDialogView*, for putting up dialog boxes
- *TControl* (and all the buttons, radio clusters, icons, pop-ups, pictures, and other items under it)
- *TGridView*, *TTextGridView*, and *TTextListView*, for displaying things in grids and lists
- *TWindow* and *TInspectWindow*, for displaying and managing windows
- *TScroller*, a view that can scroll its data (or its subviews' data)
- *TListView*, for displaying lists of items, as in the list of files in the standard file dialog box

MacApp has at least 28 view subclasses.

PicoApp's views

PicoApp's views can resize themselves in relation to the window but, for now, can't scroll their contents. PicoApp currently provides a simple display mechanism, a relationship to the window and document, and the necessary event handling. For the time being, you'll have to implement other features yourself.

PicoApp currently has only a handful of view classes, including *TPicoView* itself. Clearly, there's lots of room for development here.

Resizable Views

Some Macintosh windows can grow and shrink and zoom as the user wants them to. When they do, some of their views should also grow or shrink or zoom while others should remain fixed.

PicoApp lets you specify whether a window has a size box and a zoom box. It also lets you specify for each view how the view behaves when its window grows or shrinks.

In PicoApp, views have two "growth properties"—one for the right edge of the view (*fGrowthRight*) and one for the bottom edge (*fGrowthBottom*). Mac windows grow right and down, so PicoApp's views do, too. You can set a view's growth properties either early, in the view's initialization method (after calling *IPicoView*), or later, by sending the view a SetGrowthProperties message.

A view's growth property instance variables can each take on several values to signify how the view should grow. If the view should remain fixed, the growth fields take the value *kViewNoGrow* (*–1*). If the view should grow without limit, the growth fields take the value *kViewGrowUnlimited* (*0*). If the view should grow only up to specified coordinates (a limit), the growth fields take values greater than *0*. The values are the window coordinates beyond which the view can't grow. Those values are upper limits, implying that resizable views can shrink as small as their window can. (The smallest possible window height and width are specified in the global constants *kMinVerticalWinSize* and *kMinHorizontalWinSize*, both currently set to 80 pixels.)

The PicoSketch application we'll look at shortly has three views. One view size is fixed: It has the right and bottom growth values *kViewNoGrow*. Another view can grow downward but not to the right: It has the right growth property *kViewNoGrow* but the bottom growth value *kViewGrowUnlimited*. The third view can grow an unlimited amount in either direction: It has the right and bottom growth values *kViewGrowUnlimited*.

When the PicoSketch window grows or zooms larger, the first view remains fixed, the second view remains fixed on the right but grows to fit the window on the bottom, and the third view grows to fit the window on both the right and bottom edges. You can observe this behavior in PicoSketch.

Display rectangle and click rectangle

Suppose the window shrinks smaller than one of its views, so that only part of the view shows. If the view's growth properties are unlimited, it will shrink too. If not, the view's display rectangle can be larger than the window frame.

Keep in mind that views are also event handlers, as we'll see in the next section. The user might click the mouse in the view, in the window frame, or outside the window frame. If the active part of the view, the part that responds to clicks, were larger than the window frame, we could have a program responding to a click that is outside the window but still inside the view. To keep that from happening, views have an *fClickRect* instance variable as well as an *fViewRect*. As Figure 15-6 shows, when the window shrinks smaller than the view, the view's click rectangle shrinks to fit the window. It's the click rectangle that's used to respond to clicks, not the display rectangle.

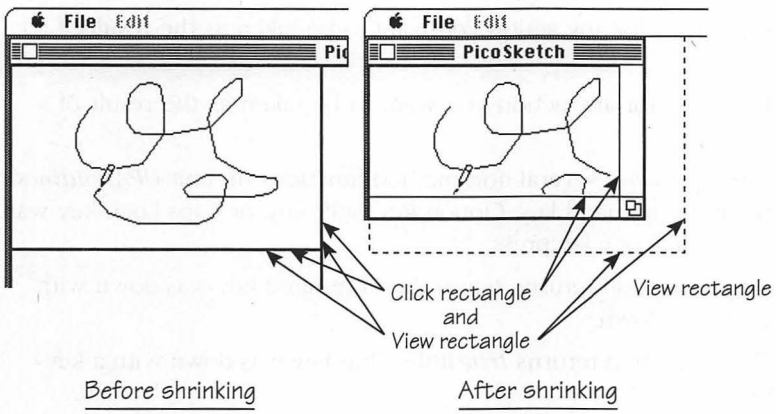


Figure 15-6.
The view rectangle and the click rectangle.

You can see how this is implemented in the *Grow* and *Zoom* methods of the application, window, document, and view objects in PicoApp. Hooks are provided at several points to let you alter the window's or the view's behavior at grow or zoom time.

Views and Event Handling

Views are really complete I/O facilities. Not only do they display information and data for the user, but they also take user input by means of controls. As implemented, *TPicoView* provides lots of event-handling capability for its subclasses:

- *DoMouseCommand*, for handling user input by mouse
- *DoCursor*, for adjusting the cursor as the user moves the mouse
- *Clicked*, for testing whether a particular view has been clicked (once, twice, or as many as three times)
- *TrackMouse*, for tracking the mouse while the mouse button is down
- *Hilite*, for highlighting the view, if appropriate, while the mouse button is down in the view's area

- *DoBeforeTrackingMouse*, a hook for any action you want to be taken just as mouse tracking begins
- *DoWhileTrackingMouse*, a hook for any action you want to be taken repeatedly as long as the mouse button is down in the view's area (This method can be used for mouse-driven drawing and painting or for selecting.)
- *DoAfterTrackingMouse*, a hook for any action you want to be taken immediately after the mouse button comes up in the view's area
- *DoClick*, a hook for any action you want to be taken as the result of a single click in the view's area—makes any view effectively a button if you like
- *DoDoubleClick*, a hook for any action you want to be taken as the result of a double click
- *DoTripleClick*, a hook for any action you want to be taken as the result of a triple click

Additionally, PicoApp provides several non-method functions (in unit *UPARoutines*), for testing whether the Command key, Option key, Shift key, or Caps Lock key was down at the time of a click or a keypress:

- The *CmdKeyDown* function returns *true* if the Command key was down with a key-down or mouse-down.
- The *ShiftKeyDown* function returns *true* if the Shift key was down with a key-down or mouse-down.
- The *OptionKeyDown* function returns *true* if the Option key was down with a key-down or mouse-down.
- The *CapsLockKeyDown* function returns *true* if the Caps Lock key was down with a key-down or mouse-down.

With all of that to work with, you can do some sophisticated event handling in your view subclasses.

The figures in Chapter 4 show much of the event-handling behavior of buttons, which in PicoApp are subclasses of *TPicoView*.

Examples

TButton has become a *TPicoView* subclass in PicoApp. It inherits most of the mouse-event–handling methods intact, overriding only *Hilite*. As a result, a button is a view that, when clicked, tracks the mouse, highlighting and unhighlighting as the mouse moves in and out of the button view's area, as long as the mouse button remains down. Some button subclasses—check boxes and radio buttons—override *Hilite* to implement their highlighting a little differently. Most *TButton* subclasses have to override *DoClick* to specify what happens when the user clicks the button. Buttons don't override *DoDoubleClick* and *DoTripleClick*. Double clicks and triple clicks are not reasonable events for a button to respond to.

Crapgame's title view, which displays the title string "Crapgame," handles mouse clicks in a very different way. It doesn't override *Hilite* at all—so no highlighting appears when the user clicks the title view. But it does override *DoClick*, *DoDoubleClick*, and *DoTripleClick*, just to demonstrate these actions. Listing 15-2 gives *TTitleView*'s three click methods.

Clicking once on the Crapgame title brings up an alert box suggesting that the user try various clicks. Clicking once with the Option key down brings up the game's animated introductory dialog box. Double clicking brings up Crapgame's About dialog box. And triple clicking brings up another alert box. Notice the use of the *OptionKeyDown* function in the *DoClick* method to modify the effect of a click.

```
procedure TTitleView.DoClick (event: EventRecord);
  override;
  var
    ok: Integer;
begin
  if OptionKeyDown(event) then
    { Show the application's introductory dialog box }
    TPicoApp(gApplication).ShowIntro
  else                    { Put up alert box suggesting what user can do }
    begin
      ParamText('Try Option-Click, Double Click, Triple Click', '', '', '');
      ok := NoteAlert(kAlertID, nil);
    end;
end;  { TTitleView.DoClick }

procedure TTitleView.DoDoubleClick (event: EventRecord);
  override;
begin
  { Put up the application's About dialog box }
  TPicoApp(gApplication).DoAbout;
end;  { TTitleView.DoDoubleClick }

procedure TTitleView.DoTripleClick (event: EventRecord);
  override;
  var
    ok: Integer;
begin
  { Put up an alert box }
  ParamText('This demonstrates triple clicking.', '', '', '');
  ok := NoteAlert(kAlertID, nil);
end;  { TTitleView.DoTripleClick }
```

Listing 15-2.
TTitleView*'s three click methods.*

View Methods

Let's look at some of the more important view methods.

DoMouseCommand

DoMouseCommand screens out events that don't pertain to its view object and then, if an event does belong to its view, determines how many clicks were involved and handles them. *DoMouseCommand* first checks to see whether the view even wants clicks of any kind. If the view does, *DoMouseCommand* sends a Clicked message to see whether there were any clicks and, if there were, how many. For one, two, or three clicks, *DoMouseCommand* calls a click-handler method. The view must both want clicks and get a *true* return from *Clicked* before *DoMouseCommand* calls a handler.

```
function TPicoView.DoMouseCommand (event: EventRecord): Boolean;
  override;
  var
    clicks: Integer;
    thePoint: Point;
    wantsClicks: Boolean;
begin
  wantsClicks := (self.fWantsClicks or self.fWantsDoubleClicks or
      self.fWantsTripleClicks);
  if wantsClicks & self.Clicked(event, clicks) then
    begin
      case clicks of
        1:
          DoClick(event);
        2:
          DoDoubleClick(event);
        3:
          DoTripleClick(event);
        otherwise
          ;
      end;  { case }
      DoMouseCommand := true;
    end
  else
    DoMouseCommand := inherited DoMouseCommand(event);
      { See whether other event handlers can take event }
end;  { TPicoView.DoMouseCommand }
```

This is now the *DoMouseCommand* method that *TButton* inherits unchanged from its immediate ancestor, *TPicoView*. A button "wants" only single clicks, of course, so only *DoClick* will be overridden to perform a task for a button. You'll see shortly how the *Clicked* method handles getting the right number of clicks.

Clicked

This version of *Clicked* is a bit more sophisticated than the one we saw for *TButton* in Chapter 4. *Clicked* must determine whether the mouse-down occurred in its view's *fClickRect* and then how many times the mouse was clicked. *Clicked* returns *true* if one or more clicks occurred in its view's area, and *false* if the clicks were outside its click rectangle. Listing 15-3 shows the code for *Clicked*.

```
function TPicoView.Clicked (event: EventRecord; var clicks: Integer): Boolean;
  var
    times, i: Integer;
    mouseTime, timeDown: Longint;
    mousePlace, placeDown: Point;

  { Nested functions SoonEnough, CloseEnough, and GetMouseDown are }
  {  declared here on the code disk }
  { We'll see them later }

begin { Clicked }
  clicks := 0;
  Clicked := false ;
  if event.what = mouseDown then
    begin
      { See whether click was intended for this view }
      mousePlace := event.where;
      GlobalToLocal(mousePlace);  { Need window coordinates }
      if PtInRect(mousePlace, self.fClickRect) then
        begin
          Clicked := true;          { We have at least one click }
          mouseTime := event.when;
          { See how many times to check for more clicks }
          if self.fWantsTripleClicks then
            times := 2              { Look for two more clicks }
          else
            times := 1;             { Look for one more click }
          { Check for indicated number of clicks }
          clicks := 1;
          for i := 1 to times do
            begin
              { Track mouse until button released for current click }
              if TrackMouse(mouseTime, mousePlace) then
                begin
                  { Wait for another mouse-down unless time runs out }
```

Listing 15-3. *(continued)*
TPicoView's Clicked *method.*

Listing 15-3. *continued*

```
                    if GetMouseDown(timeDown, placeDown) then
                      begin
                        { See whether mouse-down came soon enough and }
                        {  close enough }
                        if (SoonEnough (mouseTime, timeDown) & CloseEnough
                            (mousePlace, placeDown)) then
                          clicks := clicks + 1
                        else
                          { Was a mouse-down, but not soon enough or }
                          {  close enough }
                          Leave;  { Exit the for loop }
                      end  { If GetMouseDown }
                    else
                      { No mouse-down within time limit }
                      Leave;          { Exit the for loop }
                    end  { If TrackMouse }
                  else
                    { Mouse came up outside the }
                    {  tracking area }
                    Leave;            { Exit the for loop }
                end;  { for }
            end;  { If PtInRect }
        end;  { If event.what }
    end;  { TPicoView.Clicked }
```

To get its job done, *Clicked* calls a number of other methods and functions. The Toolbox procedure *GlobalToLocal* converts the mouse click location into local window coordinates. The QuickDraw call *PtInRect* determines whether the mouse click location is within the click rectangle. The *TrackMouse* method tracks the mouse as long as its button stays down, calling several hook methods as it tracks; if the mouse is still (or again) within the view rectangle when its button comes back up, *TrackMouse* returns *true*. The *GetMouseDown* function returns *true* if another click occurred within a reasonable interval, as determined by the Toolbox *GetDblTime* function, which gets a value that the user can change via the Control Panel to shorten or lengthen the time allowed for multiple clicks. The utility functions *SoonEnough* and *CloseEnough* report *true* if a new click occurred within the time limit and within a reasonable number of pixels from the previous click. (A view can set its *fCloseEnough* field to determine how *CloseEnough* responds. The field defaults to 5 pixels.)

Let's look at some of those functions—first at *SoonEnough* and *CloseEnough*. Both are declared as nested functions within *Clicked*. They simply compare the two times or locations they receive from *Clicked*, returning *true* if the second time or location is within range of the first:

```
function SoonEnough (startTime, stopTime: Longint): Boolean;
  { True if DblTime hasn't elapsed since startTime }
begin
  SoonEnough := ((stopTime - startTime) <= GetDblTime);
end;  { SoonEnough--not a method }

function CloseEnough (startPlace, endPlace: Point): Boolean;
  { True if mouse didn't move more than fCloseEnough }
  {  pixels between two successive clicks }
begin
  { Subtract the h and v fields of the two points }
  SubPt(startPlace, endPlace);
  CloseEnough := ((endPlace.h <= fCloseEnough) and (endPlace.v <= fCloseEnough));
end;  { CloseEnough--not a method }
```

As we've noted, the *GetMouseDown* utility function is responsible for detecting a second and possibly a third click. It determines how long to wait and then repeatedly calls *GetNextEvent* with a mask set so that it returns only mouse-downs. If *GetMouseDown* detects a mouse-down before the time runs out, it captures the time and place and returns *true*. *GetMouseDown* is also nested inside *Clicked*:

```
function GetMouseDown (var mouseDownTime: Longint; var mouseDownPlace:
    Point): Boolean;
  { True if second or third click detected }
  var
    timeNow, timesUp: Longint;
    ok: Boolean;
    mDownEvt: EventRecord;
begin
  { First, wait a reasonable time for another mouse-down }
  GetMouseDown := false;

  { Set a reasonable time to stop waiting for mouse }
  {  = time now (TickCount) + DblTime }
  timesUp := TickCount + GetDblTime;
  repeat
    { Check whether a mouse-down has occurred }
    ok := GetNextEvent(mDownMask, mDownEvt);
    if ok then          { If mouse button goes down, exit loop }
```

```
        begin
          GetMouseDown := true;
          { Note when and where mouse button went down }
          mouseDownTime := mDownEvt.when;
          mouseDownPlace := mDownEvt.where;
          { Get window coordinates }
          GlobalToLocal(mouseDownPlace);
          Leave;                    { Exit the repeat loop }
        end;

      { Current time: approximate mouse time if mouse button goes down }
      timeNow := TickCount;
    until timeNow > timesUp;        { When time up, stop waiting }
  end;  { GetMouseDown--not a method }
```

NOTE: *Because* GetMouseDown *automatically calls* GetNextEvent, *it's not really compatible with MultiFinder. Ideally, the function would determine whether MultiFinder is available on the current machine and then, if it is, call* WaitNextEvent *or else call* GetNextEvent. *This is the technique used in PicoApp's* MainEventLoop *method. The same actions need to occur both in the main event loop and in places like buttons, so a good approach would be to call, from anywhere you need to get events, a global non-method function that sees whether* WaitNextEvent *is enabled and then returns one event if possible.*

Conforming to MultiFinder is sometimes subtle and devious, as this case shows. We leave implementing the fix to interested readers.

TrackMouse vs. the Control Manager

Finally, let's see *TrackMouse*. It's similar to the mouse-tracking code in our original buttons, but now it has a number of hooks that allow an application to do work while the mouse is being tracked.

Why haven't we used the Toolbox's Control Manager? It does a nice job of tracking the mouse already, and undoubtedly does so faster. But we have two reasons for not using it. First, not all views are controls in the Toolbox sense, and we want to track the mouse in any view. Second, by writing the tracking code ourselves, we can insert a number of useful overridable hook calls: *Hilite*, *DoBeforeTrackingMouse*, *DoWhileTrackingMouse*, and *DoAfterTrackingMouse*. The extra flexibility is well worthwhile, and the loss of speed is pretty negligible. Moreover, all view subclasses can inherit or override methods like the hooks we mentioned—or even methods like *TrackMouse*, *Clicked*, *GetMouseDown*, and so on. Listing 15-4 shows our augmented *TrackMouse*.

```
function TPicoView.TrackMouse (var time: Longint; var place: Point): Boolean;
  var
    outOfRect: Boolean;
begin
  { See whether mouse was inside view at mouse-down }
  outOfRect := not PtInRect(place, self.fClickRect);
  fOldMouse := place;
  DoBeforeTrackingMouse(place);    { Does nothing unless overridden }
  if not OutOfRect then
    begin
      self.Hilite(true);           { Highlight view initially }
      { While mouse still down, track it in and out of view }
      while stillDown do
        begin
          GetMouse(place);         { See where mouse is now }
          { Perform some useful action while tracking-- }
          {  draw, for example }
          DoWhileTrackingMouse(place);  { Does nothing unless overridden }

          { It's in the click rectangle but goes out }
          if not outOfRect and (not PtInRect(place, self.fClickRect)) then
            begin
              self.Hilite(false);      { Turn off highlighting }
              outOfRect := true;
            end
          { Or it's out of the click rectangle but comes back in }
          else if outOfRect and PtInRect(place, self.fViewRect) then
            begin
              self.Hilite(true);       { Turn on highlighting }
              outOfRect := false;
            end;
          self.fOldMouse := place;
        end;  { while }

      { Once mouse released, see where and when it came up }
      if not outOfRect then
        begin
          time := TickCount;
          TrackMouse := true;      { Released inside; real click }
          self.Hilite(false);      { Turn final highlighting off }
        end
```

Listing 15-4. *(continued)*
TrackMouse.

Listing 15-4. *continued*

```
      else
        TrackMouse := false;        { Released outside; no click }
      end { If not out }
    else
      TrackMouse := false;
    DoAfterTrackingMouse(place);    { Clean up as needed--does }
                                    {  nothing unless overridden }
end;  { TPicoView.TrackMouse }
```

After determining whether the mouse is initially inside the click rectangle of the view, we send the hook message DoBeforeTrackingMouse in case the view wants to set something up. Actually, if the mouse wasn't in the view initially, *TrackMouse* would never get called. *Clicked* would screen this condition out. So, with the mouse button down in the view, we loop until the mouse button comes up. In the loop, we check where the mouse is now, send the hook message DoWhileTrackingMouse, and adjust highlighting depending on whether the mouse has moved outside the view or is still inside. Once the mouse button comes up, we see where it did. If it was still inside the view, we note the time and place so that we can return them in the *var* parameters, turn off any highlighting that might still be on, and end.

Unless your view subclass overrides them, the four hook methods in *TPicoView* are calls to do-nothing methods. The method calls are made, but nothing happens, so it's quick. *TButton* illustrates the use of *Hilite*. And we'll look at examples of ways to use the other hooks next.

PicoSketch: A Small Example of Views

We've already seen some of the capabilities of views (although we haven't gone into detail yet about how Crapgame's views do what they do). We've looked at views that modify the cursor shape: Crapgame's *TTitleView* and *TStatsButton* objects. We've looked at views that respond to clicks: Crapgame's *TThrowButton*, *TStatsButton*, and *TTitleView* objects. We've looked at views that respond to keypresses: Crapgame's *TThrowButton* object. And we've noted a variety of ways that views display themselves.

Our next example program, called PicoSketch, is another small application built on PicoApp. It shows how to use the *TPicoView* hook methods to do something useful—in this case, drawing with the mouse.

Let's briefly look at *PicoSketch*'s application, document, and view classes.

About PicoSketch

In addition to demonstrating views, PicoSketch illustrates mouse drawing, multiple documents and windows, memory management, and windows and views that grow and zoom.

We'll use the *TrackMouse* hooks, *DoBeforeTrackingMouse*, *DoWhileTrackingMouse*, and *DoAfterTrackingMouse*, to implement drawing with the mouse. Note, however, that this is a very rudimentary implementation. You'll see right away, for instance, that when something obscures a drawing in a view, the drawing is not preserved when the view is again updated. Ideally, we'd draw into an off-screen bitmap and then "blast" the bits of our drawing onto the screen. The bitmap would save the drawing for later updates. We've left out the bitmap to save room and because it's not especially object oriented. (But see class *CBitMap* in the THINK Class Library.)

Unlike Crapgame, PicoSketch lets the user open document after document, up to the limit of available RAM. This demonstrates PicoApp's built-in (although simple) memory management facilities. When the user tries to open one document too many for the amount of available RAM, an alert box warns the user that memory is low and PicoApp disables most menu items along the lines described in Chapter 14. Each PicoSketch document allocates a dummy data handle of 10,000 bytes to simulate documents with real data. Try opening documents in PicoSketch by choosing New from the File menu until you run out of memory.

Each PicoSketch document has a window with three views. In each such view the user can do a different kind of drawing, as described below. The user can resize a window with the size box in its lower right corner. The user can zoom windows in or out with the zoom box at the right end of the window's title bar. When a window grows, shrinks, or zooms, its views behave according to the growth properties we went into earlier in this chapter.

PicoSketch's *TPicoApp* Subclass

Class *TPicoSketchApp* is extremely simple because it doesn't need to do anything fancy beyond what PicoApp already provides. This subclass of *TPicoApp* adds one new method, *IPicoSketchApp*, and overrides two methods, *MakeDocument* and *DoApplicationSetup*.

The *MakeDocument* override is mandatory, of course, because we have to create a document of type *TPicoSketchDoc*, which PicoApp doesn't know anything about.

But PicoSketch puts up Apple, File, and Edit menus. It puts up an About dialog box. It handles desk accessories correctly. It gets and handles events. It knows how to handle windows. And it knows how to close documents and quit. All of this is still available because, as a subclass of *TPicoApp*, *TPicoSketchApp* inherits so much.

```
type
  TPicoSketchApp = object(TPicoApp)

      procedure TPicoSketchApp.IPicoSketchApp;
      procedure TPicoSketchApp.MakeDocument;
      override;
      procedure TPicoSketchApp.DoApplicationSetup;
      override;
    end;  { Class TPicoSketchApp }
```

PicoSketch's *TPicoDoc* Subclass

Like its application object class, PicoSketch's document object class is simple. It adds an initialization method and overrides five methods. *MakeWindows* and *MakeViews* are required overrides. *Free* is overridden to support freeing the document's views and window. *TypeOf* is overridden as usual in case it's necessary to know a *TPicoSketchDoc*'s actual type. (*TypeOf* returns the string 'TPICOSKETCHDOC'.) Finally, *ReadDoc* is overridden merely to show what a *ReadDoc* override looks like so that you can see how the File-Open menu item is implemented (although it isn't fleshed out in PicoSketch—we don't read any data).

Each PicoSketch document creates a single window, titled "PicoSketch," and displays three views in it.

We've overridden *MakeWindows* to rename the window, after calling *inherited MakeWindows* to do the dirty work. This is the same process we used in Crapgame. It's the process you'd always use with your PicoApp documents unless you wanted your windows to come up as "Untitled."

We've overridden *MakeViews* to create our three views and string them together in a simple linked list, which is used by the window object.

```
type
  TPicoSketchDoc = object(TPicoDoc)

      fTestHandle: Handle;        { Included to demonstrate memory management }
      fWatch: CursHandle;         { Handle to watch cursor }

      { This document's views }
      fSketchView: TSketchView;
      fRubberBandView: TRubberBandView;
      fFanView: TFanView;

      { The views' rectangles }
      fSketchRect: Rect;
      fRubberBandRect: Rect;
      fFanRect: Rect;
```

```
    function TPicoSketchDoc.IPicoSketchDoc: Boolean;
    function TPicoSketchDoc.MakeWindows (hasClose, hasZoom, hasGrow:
        Boolean): Boolean;
    override;
    function TPicoSketchDoc.MakeViews: Boolean;
    override;
    procedure TPicoSketchDoc.ReadDoc (fileName: Str255; fileRefNum: Integer);
    override;
    procedure TPicoSketchDoc.Free;
    override;
    function TPicoSketchDoc.TypeOf: Str30;
    override;
      { Returns the string 'TPICOSKETCHDOC' }
  end;  { Class TPicoSketchDoc }
```

PicoSketch's *TPicoView* Subclasses

Our three views are displayed in different parts of the same window.

TSketchView

The first, *TSketchView*, allows the user to do freehand drawing with a pencil tool. The cursor changes to a pencil while the mouse pointer is over this view. And we use *DoBeforeTrackingMouse*, *DoWhileTrackingMouse*, and *DoAfterTrackingMouse* to do the drawing.

```
type
  TSketchView = object(TPicoView)

    procedure TSketchView.ISketchView (r: Rect; d: TPicoDoc);
      { Initialize and get a pencil cursor resource }
    procedure TSketchView.Display;
    override;
      { Display the view's bounding rectangle }
    procedure TSketchView.DoBeforeTrackingMouse (whereNow: Point);
    override;
      { Set up for drawing while tracking the mouse pointer }
    procedure TSketchView.DoWhileTrackingMouse (whereNow: Point);
    override;
      { Draw while tracking the mouse pointer }
    procedure TSketchView.DoAfterTrackingMouse (whereNow: Point);
    override;
      { Finish the drawing after the mouse button comes up }
```

```
    procedure TSketchView.Free;
    override;
      { Free the view object }
    function TSketchView.TypeOf: Str30;
    override;
      { Return the string 'TSKETCHVIEW' }
  end;  { Class TSketchView }
```

TRubberbandView

The second view, *TRubberbandView*, allows the user to draw "rubberband" lines, as with a drawing program's line tool. While the mouse pointer is moved and the button is down, a line is drawn, erased, redrawn, erased, and so on, until the mouse button comes up. Then the line is drawn in its final position, from the initial mouse-down point to the final mouse-up point. Again, the cursor changes shape, to a set of crosshairs this time, and we use the hook methods to implement the drawing.

```
type
  TRubberBandView = object(TPicoView)

    fAnchor: Point;

    procedure TRubberBandView.IRubberBandView (r: Rect; d: TPicoDoc);
      { Initialize and set up the crosshair cursor }
    procedure TRubberBandView.Display;
    override;
      { Display the view's bounding rectangle }
    procedure TRubberBandView.DoBeforeTrackingMouse (whereNow: Point);
    override;
      { Set up for drawing while tracking the mouse pointer }
    procedure TRubberBandView.DoWhileTrackingMouse (whereNow: Point);
    override;
      { Draw while tracking the mouse pointer }
    procedure TRubberBandView.DoAfterTrackingMouse (whereNow: Point);
    override;
      { Finish the drawing after the mouse button comes up }
    procedure TRubberBandView.Free;
    override;
      { Free the view object }
    function TRubberBandView.TypeOf: Str30;
    override;
      { Return the string 'TRUBBERBANDVIEW' }
  end;  { Class TRubberBandView }
```

TFanView

The third view, *TFanView*, allows the user to do an interesting variant of rubber-band line drawing. As the mouse pointer moves, many lines are drawn from a common pivot point to every mouse pointer position detected by *TrackMouse*. The result is a pleasing free-form fan shape. The cursor changes to crosshairs, and we use the hook methods to implement the drawing.

```
type
  TFanView = object(TPicoView)

      fAnchor: Point;

      procedure TFanView.IFanView (r: Rect; d: TPicoDoc);
        { Initialize and set up the crosshair cursor }
      procedure TFanView.Display;
      override;
        { Display the view's bounding rectangle }
      procedure TFanView.DoBeforeTrackingMouse (whereNow: Point);
      override;
        { Set up for drawing while tracking the mouse pointer }
      procedure TFanView.DoWhileTrackingMouse (whereNow: Point);
      override;
        { Draw while tracking the mouse pointer }
      procedure TFanView.DoAfterTrackingMouse (whereNow: Point);
      override;
        { Finish the drawing after the mouse button comes up }
      procedure TFanView.Free;
      override;
        { Free the view object }
      function TFanView.TypeOf: Str30;
      override;
        { Return the string 'TFANVIEW' }
    end;  { Class TFanView }
```

You can see the full code for PicoSketch in the folder PicoSketch, Part 2 on the code disk. Figure 15-7 on the next page shows some output of the program, and Figure 15-8 shows the THINK Pascal project window for PicoSketch.

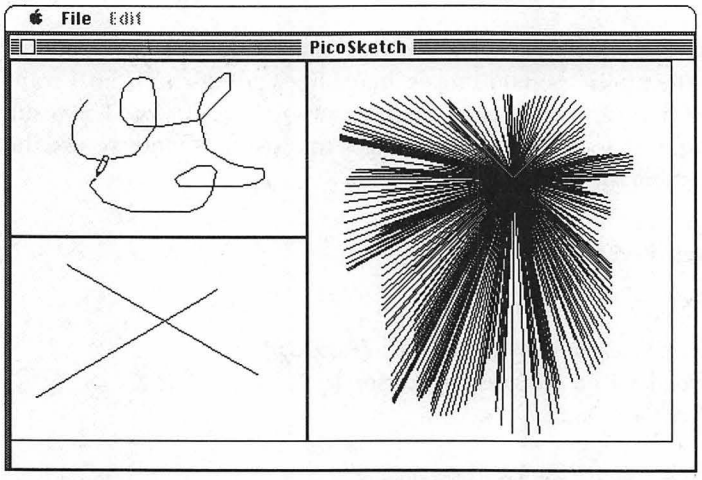


Figure 15-7.
PicoSketch window with three views subclassed from TPicoView: TSketchView, TRubberbandView, *and* TFanView.

PicoSketch.π

Options	File (by build order)	Size
	Runtime.lib	18222
	Interface.lib	10106
D N V R	ObjIntf.px	396
D N V R	UPAGlobals.p	820
D N V R	UMyGlobals.p	0
D N V R	UPARoutines.p	5374
D N V R	UPAUtilClasses.p	2572
D N V R	UMemory.p	4862
D N V R	UPAIntf.p	1678
D N V R	UPAViews.p	4872
D N V R	UPAWind.p	3826
D N V R	UPADoc.p	2794
D N V R	UPicoApp.p	4612
D N V R	UPAEvents.p	5076
D N V R	UPAMenus.p	976
D N V R	UPicoSketchViews.p	2536
D N V R	UPicoSketchDoc.p	1600
D N V R	UPicoSketchApp.p	438
D N V R	PicoSketchMain.p	120
	Total Code Size	70880

Figure 15-8.
Compilation order for PicoSketch.

Drawing in PicoApp

The Macintosh is a highly graphical computer. We can't leave a discussion of windows and views without saying something about drawing.

Views handle all drawing, and we've seen lots of drawing already. Crapgame draws its title, dice, button, and player information.

Views also handle drawing that the user does with the mouse.

Mouse Drawing

To accomplish drawing with the mouse, we have to track the mouse pointer and do the actual drawing in the process. That's what the mouse-tracking hooks we see in Figure 15-9 are for.

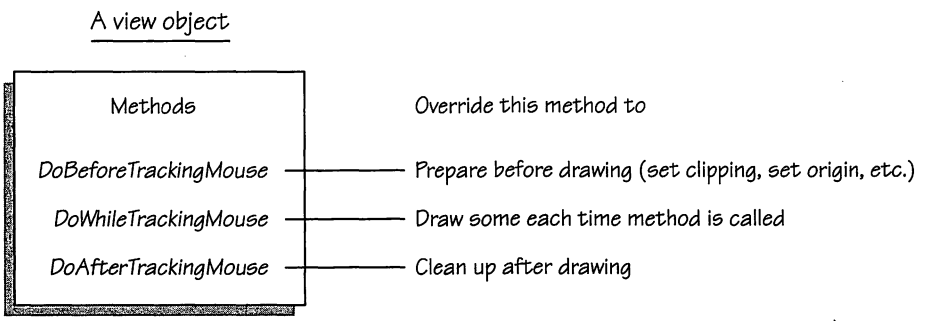


Figure 15-9.
The mouse-tracking hooks. These methods are called during the execution of a view's TrackMouse *method.*

In PicoSketch, to ensure that the drawing in a particular view doesn't get "outside the lines" of the view, we first clip all drawing to the click rectangle of the current view. We do that in *DoBeforeTrackingMouse*, using it somewhat as we would MacApp's *Focus* method. Then, after drawing, we reset the clipping region to what it was before. We do that in *DoAfterTrackingMouse*.

When a user starts to draw in a grafPort on the Mac, the pen is initially located at the grafPort's origin—in this case, the upper left corner of the window. If the user simply started to draw, the first line would run from the upper corner of the window to the mouse pointer's location. We have to manage moving the pen to where the mouse button went down before the user started drawing. We do that in *DoBeforeTrackingMouse*.

Here's a sample *DoBeforeTrackingMouse*:

```
procedure TSketchView.DoBeforeTrackingMouse (whereNow: Point);
  override;
  var
    r: Rect;
begin
  r := self.fClickRect;
  InsetRect(r, 1, 1);
  { Clip so that drawing is limited to the click rectangle of the view }
  ClipRect(r);
  { Move pen to where mouse touched down }
  MoveTo(whereNow.h, whereNow.v);
end;  { TSketchView.DoBeforeTrackingMouse }
```

And here's a sample *DoAfterTrackingMouse*:

```
procedure TRubberBandView.DoAfterTrackingMouse (whereNow: Point);
  override;
begin
  WaitForRetrace;
  LineTo(whereNow.h, whereNow.v);  { Draw final line }
  WideOpenClipRegion;              { Reset clip region }
end;  { TRubberBandView.DoAfterTrackingMouse }
```

This *DoAfterTrackingMouse* method calls two PicoApp support routines. *WaitForRetrace* waits for the Mac's vertical retrace period before drawing, which reduces flicker. *WideOpenClipRegion* resets the window's clipping region to its maximum value (–32767..32767 in both dimensions).

Sketching with the Mouse

The drawing itself takes place in the repeated messages to *DoWhileTrackingMouse*. Called repeatedly by *TrackMouse*, our override of the method takes the current location of the mouse pointer and connects it with a line to the previous point. It might seem that this would produce jerky, angular lines—a series of straight lines between angle points—instead of smooth curves. But keep the Mac's speed in mind. The consecutive tracking points are very close together, and *DoWhileTrackingMouse* gets called many, many times per second. The result is a fairly smooth looking curve in our sketch view. Here's our override of *DoWhileTrackingMouse*:

```
procedure TSketchView.DoWhileTrackingMouse (whereNow: Point);
  override;
begin
  { Connect new location with previous location, leaving pen there }
  WaitForRetrace;                  { Wait for vertical retrace for drawing }
  LineTo(fOldMouse.h, fOldMouse.v);
end;  { TSketchView.DoWhileTrackingMouse }
```

Drawing with a Rubberband Line Tool

Rubberbanding works by drawing a line and then erasing it. This process is repeated as the mouse pointer moves, with each previous line replacing the previous line and then disappearing, to be replaced by a new line reflecting the current mouse pointer location.

More precisely, during each call to *DoWhileTrackingMouse*, we first draw from the anchor point—the point at which the user pressed the mouse button—to the current mouse pointer location and then change the drawing mode to *patXor*, which blackens white bits and erases black bits. In that mode, we draw from the current mouse pointer location back to the anchor, effectively erasing the line. Of course, we then need to reset the drawing mode to normal (*patCopy*). New lines crossing old lines will erase the old lines' pixels where they cross.

On the next call to *DoWhileTrackingMouse*, we do it all again. Here's our override of *DoWhileTrackingMouse* for rubberband drawing:

```
procedure TRubberBandView.DoWhileTrackingMouse (whereNow: Point);
  override;
begin
  { Draw from the anchor to mouse pointer's current location }
  WaitForRetrace;                    { Wait to draw without flicker }
  LineTo(whereNow.h, whereNow.v);
  { Change drawing mode }
  PenMode(patXor);
  { Draw back to the anchor, erasing the old line }
  WaitForRetrace;
  LineTo(fAnchor.h, fAnchor.v);
  { Reset the pen to normal, which includes normal penPat }
  PenNormal;
end;  { TRubberBandView.DoWhileTrackingMouse }
```

Alternatives

Of course, this isn't the only way to implement mouse drawing. For instance, MacApp has the view object create a "mouse tracker" or "sketcher" object, which tracks the mouse pointer and does the drawing. We've gone for a simpler, although probably less flexible, solution in PicoSketch.

We could have used a single view in PicoSketch, with either a palette of tools or a tools menu. Then the user would select a tool and do all three kinds of drawing in the same view area—more like a real drawing program. With a single view, our overrides of *DoBeforeTrackingMouse*, *DoWhileTrackingMouse*, and *DoAfterTrackingMouse* would have to contain a selection mechanism. It could be a *case* statement, of course, with a case for each possible tool. But we could also get much more object-oriented than that, making use of polymorphism: Selecting a tool might

create a tool object, with its own tracking methods. The view would install the tool object as its current tool (by assigning the tool object to an *fCurrentTool* variable of an ancestor type):

```
fCurrentTool := theTool;
```

and call the tool's mouse-handling methods. To call the tool's mouse-handling methods, we'd access *fCurrentTool* from inside our drawing view's *DoWhileTrackingMouse* method. We'd send *fCurrentTool* a Draw message, say, which would draw one line segment. Figure 15-10 shows how this works in a MacDraw-like application.

The possibilities for implementing drawing are plentiful.

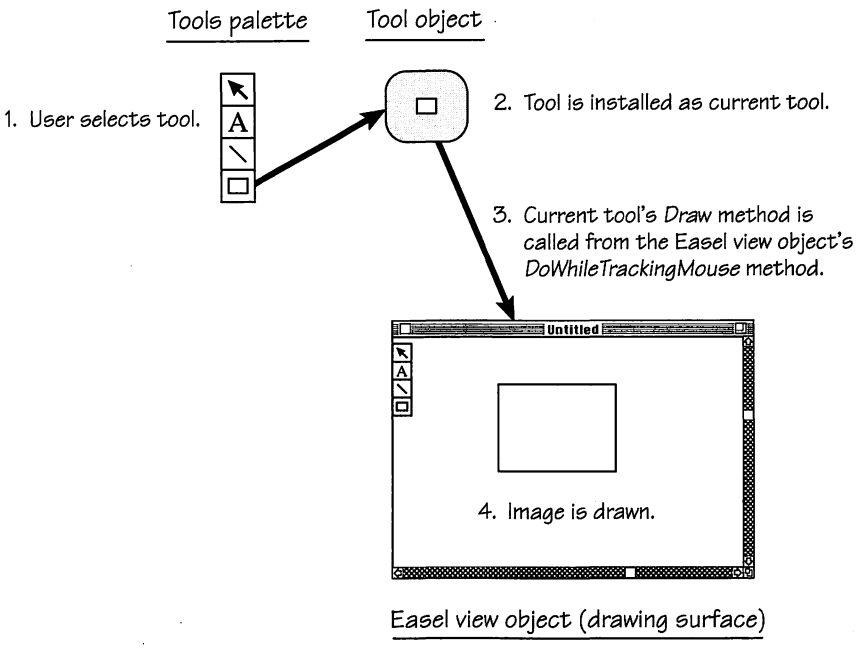


Figure 15-10.
A drawing tool selected, installed, and then called by the object's DoWhileTrackingMouse *method.*

Summary

In this chapter, we finished our inside tour of how PicoApp works. We saw what document objects, window objects, and view objects do. We saw how the three kinds of objects interact. And we looked at some of their most important methods.

We also examined a second small PicoApplication, PicoSketch—which gave us a second look at building applications with PicoApp.

And we discussed how you can implement mouse drawing in PicoApp, using the hook methods provided by *TPicoView* and called by its *TrackMouse* method.

In the next chapter, we'll go through the steps of creating new PicoApplications of our own.

Projects

- Add a printing capability to *TPicoDoc*.
- PicoApp so far omits an ability to open or print documents from the Finder. The user should be able to select document icons and choose either Open or Print from the Finder's File menu. Implement such a capability for PicoApp.

  Suggestions: In *TPicoApp.IPicoApp*, or in the utility routine *InitTheMac* (in unit *UPARoutines*), set global Boolean variables *gFinderOpen* and *gFinderPrint*, based on information obtained using the routines in *Inside Macintosh*, II-67. If Finder printing is called for, set up a minimal application object, have it create document objects that read in the chosen files, and send each document object in turn a Print message. Then quit. If Finder opening is called for, set up the normal application object and have it create as many of the chosen documents as possible. Modify the memory management code to ensure that no more documents are opened than can actually be worked on and that, if the user has tried to open more documents than memory will allow, he or she is notified with an alert box that fits the context (as determined by the global variable *gFinderOpen*).
- Modify PicoSketch to use tool objects. Design a window that displays a palette of tools (a view, of course). Select a tool from the palette, and install the tool object as the current tool in a drawing view. Will you have to modify or subclass *TPicoView* to have your tool's drawing method called, or can you fit that call into the existing scheme?
- Look at MacApp's and TCL's view classes. Design similar classes for PicoApp.

16

USING PICOAPP

Now we'll switch our focus from what PicoApp does and how it does it to how you can use it to write your own applications. We previewed this topic in Chapter 11. Now we'll fill in the details and look at examples. We'll go through, step by step,

- Setting up your project
- Preparing your application's resources
- Subclassing *TPicoApp*
- Subclassing *TPicoDoc*
- Subclassing *TPicoView*
- Writing your own additional classes
- Preparing your version of *UMyGlobals*
- Creating your main program
- Compiling your application and testing it
- Optimizing your application

Setting Up Your Project

In this step, you prepare the basic files and folders you'll need to create and compile your PicoApplication.

We'll define a "project" as the set of source code files, object code library files, and resource files needed to build your application. The setup particulars depend on which version of the language you use. For full details, see the documentation for your Pascal and its programming environment.

In either MPW Pascal or THINK Pascal, you'll need a folder to contain the files for your application. I've set up a folder on the code disk called Crapgame, Part 2. The folder Crapgame contains

- A TeachText (text-only) document called Compiling Crapgame
- The source code units for Crapgame—normally one file per object class
- The file Crapgame.rsrc

You will have to create a file THINK Pascal calls the "project file," which manages compilation order and other housekeeping chores. (We keep a few general-purpose units in the folder Common Units.) Under MPW or THINK, the project file ("dependency file" in MPW) specifies the order in which the files are to be compiled. In THINK, for example, you add the files to the project in an order determined by the dependence of some declarations on prior declarations. Figure 16-1 shows the THINK project file's window for Crapgame.

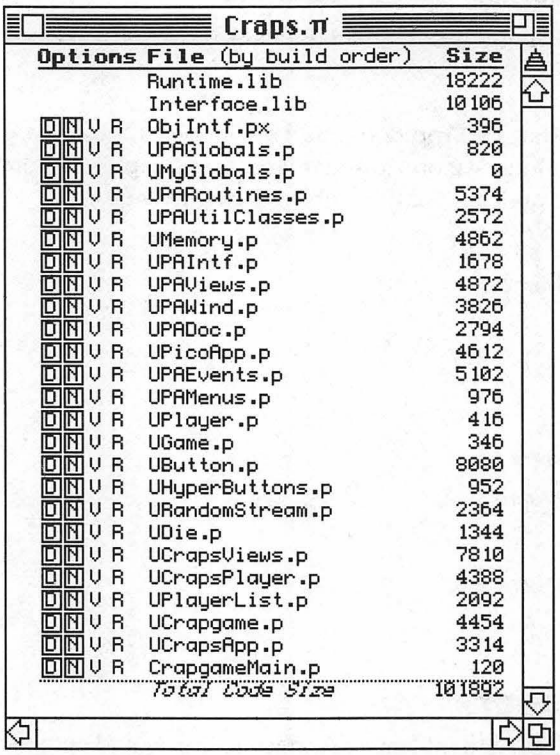


Figure 16-1.
The Crapgame project file window. Compilation order is a reflection of declarations depending on earlier declarations.

Adding PicoApp to Your Project

The PicoApp framework's files remain in their own folder. Some frequently used files are in the folder Common Units. In your project, above the files for your application, you'll add the PicoApp files ObjIntf.px, UPAGlobals.p, UPARoutines.p, UPAUtilClasses.p, UMemory.p, UPAIntf.p, UPAViews.p, UPAWind.p, UPADoc.p, UPicoApp.p, UPAEvents.p, UPAMenus.p, and, if you need them, UButton.p, UHyperButtons.p, and URandomStream.p.

Later, you'll need to copy the file UMyGlobals.p/PA, renaming the copy UMyGlobals.p. (See "Preparing Your Version of *UMyGlobals*" later in this chapter.) Put your UMyGlobals.p below UPAGlobals.p.

Preparing Your Application's Resources

You've probably gone far enough in your application design to know what resources it will need.

Copying File PicoApp.rsrc to File YourApp.rsrc

In the Finder, select file PicoApp.rsrc (in the folder PicoApp, Part 2). Choose Duplicate from the File menu. A new file named Copy of PicoApp.rsrc is created. Click its name and type its new name,

[YourApp].rsrc

where *YourApp* is the name of your application. The resource file for Crapgame, for example, is called Crapgame.rsrc.

It's a good idea to keep all the files for each project in one folder, so put this resource file in the folder that will contain your project's source code files and the project file. You can leave PicoApp's files in their own folder.

Editing the Resources in File YourApp.rsrc

ResEdit should be available with your Pascal system. So should its basic documentation. Start up ResEdit so that you can edit some resources and add others.

What to edit:

- The existing MENU and MBAR resources (for Apple, File, and Edit)
- The About dialog box DLOG resource, ID=1
- The BNDL, FREF, ICN#, and signature resources for your application's icons and its documents' icons (if any)
- The vers resource for version information used by the Finder

What to add:

- Any additional MENU resources you need (with IDs starting at 131)
- Any additional DLOG resources you need (plus their dialog item list resources)
- Any ALRT resources you need (plus DITLs)
- Any STR or STR# resources and any other resources you need

What to note:

- Item numbers of your items in the File, Edit, and other menus
- ID numbers of your MENU, DLOG, ALRT, STR, STR#, and other resources

Now let's look at some specific requirements and tips.

Editing the existing MENU resources

In the Apple menu resource, ID=128, change the name of the About item (item 1) to the name of your application. For example, we've changed it to About Crapgame.

In the File menu resource, ID=129, add, change, and rearrange the items to fit your needs. You might need to remove some File items, such as Print or Save. You might need to add some items specific to your application. And you might want to edit the names of some items. Write down the final sequence of items and their item numbers for later use in preparing the file UMyGlobals.p. (And see the discussion of preparing *UMyGlobals* later in this chapter.)

In the Edit menu resource, ID=130, add any items you need, such as Show Clipboard. You probably won't need to remove or rename any of the other items. Note the item numbers you've used, for later use in preparing the file UMyGlobals.p.

Adding other menu resources

If your application needs other menus besides Apple, File, and Edit, create new MENU resources for them. Start their resource ID numbers above 130. Note the items and item numbers in each for later use in preparing the file UMyGlobals.p.

To add a MENU resource, open your resource file in ResEdit, choose New from the File menu, and double-click MENU in the resource list that comes up.

The flags field in the menu resource (in the EnableFlgs text box) is a hexadecimal string whose digits, translated to binary, tell whether a particular menu or item is initially enabled. (See your documentation for ResEdit and *Inside Macintosh,* I-344.) The flags field has 32 bits, which translates to 8 hex digits. Bit 0 of the flags field is 1 if the whole menu is enabled; bits 1 through 31 will be 1's if all the individual items are enabled, 0's if not. Note that bit 0 is the rightmost bit in the number. Examples: Crapgame's File menu has the flags field set to $FFFFFFFF, which translates to 11111111111111111111111111111111—all items are enabled. PicoApp's Apple menu flags field $FFFFFFFB translates to 11111111111111111111111111111011—all but item 2 are enabled. Item 2 is the dividing line between the About item and the list of desk accessories.

Editing the existing MBAR resource

Already in your copy of PicoApp's one MBAR resource (ID=1) are entries for menus with IDs of 128 (Apple), 129 (File), and 130 (Edit). If you have additional menus, first create new MENU resources. (See "Adding other menu resources," above.) Then add entries in the MBAR resource for those menus. Start the resource ID numbers for additional menus at 131.

This lets PicoApp know about your menus so that it can load their resources and make them appear in the menu bar.

Editing the About dialog box resource

When you open the DLOG resource, ID=1, you'll see a miniature screen with a picture of the dialog box. Double-click the dialog box picture to open it for editing.

Edit the existing text string, add other strings, and rearrange them with the mouse. Newer versions of ResEdit also let you change the font and size of the text—one font and size for the whole dialog box. If you need text of different sizes, fonts, and styles, use a "user item." See *Inside Macintosh,* I-404.

You can change the dialog box's window title by choosing the Display as Text command from the Edit menu (or from the DLOG menu in later versions of ResEdit).

If you want a fancier dialog box, see the Dialog Manager information in *Inside Macintosh,* I-399. And see Crapgame's dialog boxes for a variety of examples of what you can do (including using a user item—see the code for the introductory dialog box, in method *TCrapsApp.ShowIntro*). If you do anything fancy with your dialog box, such as opening subdialog boxes from it, as in Crapgame, or adding user items, you'll have to override *TPicoApp.DoAbout* to manage those extras. See Crapgame's *DoAbout* method.

Adding other dialog box resources

If your application needs other DLOG resources besides the About dialog box, create new DLOG resources for them. Be sure to make their ID numbers greater than 1. Note which dialog box ID number corresponds to which dialog box. To add a DLOG resource, open your resource file and double-click DLOG in the resource list that comes up. Choose New from the File menu. When a picture of a dialog box appears, edit its contents by double-clicking the picture. You can add static or editable text, controls, icons, pictures, and user items by choosing New from the File menu. See *Inside Macintosh,* I-399.

As you edit the contents of your new DLOG resource, change the ID number of its accompanying DITL resource to match the DLOG's ID number. To do this, click the DLOG's contents, which creates a new window to display them. This window has the acronym "DITL" (dialog item list) in its title. Choose Get Info from the File menu, and change the DITL's ID number there.

You can adjust some characteristics of your dialog box, including its window name, by choosing Display as Text from the Edit menu (or from the DLOG menu in later versions). You can adjust other things, including your dialog box's ID number, by choosing Get Info from the File menu.

Setting up your application's desktop icon

You can make your application display its own double-clickable icon on the desktop by editing four of the resources found in your copy of PicoApp.rsrc: the BNDL, signature, FREF, and ICN# resources.

After editing these resources and compiling your program, you need to set your application's "bundle bit." We'll go into the details under "Setting the Bundle Bit of Your Application" later in this chapter.

Editing the vers resource

PicoApp's resource file includes a resource of type vers (for version). If your application has a vers resource, the Finder will use it to display information about the application when the user chooses Get Info. The vers resource usually contains the current version number of the application and a copyright notice. You can edit the copy of the vers resource in your application's resource file.

Adding ALRT resources

If your application needs to put up any alert boxes, you can either use the basic alert box supplied by PicoApp.rsrc or add more ALRT resources yourself. You might want several custom alert boxes for special purposes. Note the ID numbers of your ALRT resources.

Like a DLOG resource, an ALRT has an accompanying DITL resource. Be sure to change the ID number of each ALRT's DITL resource to match the ID number of the ALRT. See the earlier discussion of DLOG resources.

Adding STR or STR# resources

According to Apple, it's best to put the text displayed in your program (titles, error messages, and the like) into string resources. A resource of type STR holds one Pascal string. A resource of type STR# holds any number of strings. You'll need to note the ID numbers of your STR and STR# resources and the item numbers of particular strings in a STR# resource. You can use these resources in your application by calling *GetString*, passing the ID number of the STR resource. The file UPAGlobals.p includes a comment section explaining the scheme that PicoApp uses for numbering the STR# resources that contain error messages.

Adding any other resources you need

PicoApp can't anticipate all the resources your application will need—you might want to add more. Both Crapgame and PicoSketch use cursor resources, of type CURS. See the chapter "The Resource Manager" in *Inside Macintosh* (I-103) for other resource types you might use. You can use ResEdit to copy many useful resources from other programs and edit them to suit your needs.

Associating YourApp.rsrc with Your Project

Somehow you need to associate your new resource file with the application source code files you're compiling. After compiling and linking your application, your Pascal system copies the resources in your file into the resource fork of your application's compiled file. (Remember that a Macintosh file has two parts—a data fork and a resource fork. It's all one file to the Mac, but the forks store different kinds of data.)

In MPW Pascal, as we noted earlier in this chapter, the resource file is listed as one of the files in your dependency file, which lists the files in an order based on their inter-file dependencies. The Make command then builds the program, doing all necessary compiling, linking, and resource copying.

In THINK Pascal, you associate your resource file with your application by setting up your application's project and then choosing Run Options from the Run menu. In the dialog box that comes up, you click the box labeled "Use resource file:" and select your resource file from the standard file dialog box that appears.

The project needs many of the resources before your program can run at all, so ensure this association at the beginning.

Subclassing *TPicoApp*

Writing a subclass of *TPicoApp* defines your application object, which inherits a great deal from *TPicoApp* but which also overrides some methods, adds others, and so on. Let's look at what you must override, what you should not override, and what you can override.

What Your Application Subclass Must Override

Besides adding an initialization method, your application subclass must override the methods *MakeMenus* and *MakeDocument*. In addition, you'll most likely need to override *FixMenus* (and possibly *FixFileMenu* and *FixEditMenu*), *DoMenuCommand*, and *UnloadSegments*. See the Crapgame and PicoSketch applications on the code disk for override examples.

Override *MakeMenus* to get handles to your application's additional menus besides Apple, File, and Edit. We've already seen how to do this in Crapgame and PicoSketch.

Override *MakeDocument* to create a new document object of your document type. The override should create the document with *New*, have it initialize itself (and call *IPicoDoc*), and return the document object as the function result.

Override *FixMenus* if your application object needs to enable any items in a menu other than File or Edit. *FixMenus* calls *FixFileMenu*, *FixEditMenu*, and additional *Fix[Your]Menu* methods.

Override *FixFileMenu* if you need to enable File menu items other than New, Open, Close, and Quit, or if you need to enable those items differently than PicoApp now does. See Crapgame for an example.

Override *FixEditMenu* for similar reasons. Neither Crapgame nor PicoSketch implements Cut and Paste, so neither has to override this method.

Override *DoMenuCommand* to add code that takes care of selections in the standard menus (other than New, Open, Close, and Quit in the File menu and any item in the Apple and Edit menus) and any item in other menus.

Override *UnloadSegments* to call *UnloadSeg* on any CODE segments you've added to your project besides those that PicoApp already uses. See Crapgame for an example, and see the discussion later in this chapter.

What Your Application Subclass Must Not Override

Actually, you can override any method of *TPicoApp*, but there are some that you'd hardly ever want to:

- *MainEventLoop*
- *DispatchEvent*
- *DoMouseCommand* and other application-level event-handling methods (But you might want to override some methods of the same names in your document and view subclasses.)
- *Run*
- Chain-handling methods such as *RemoveDoc*, *RegisterEvtHandler*, *RemoveEvtHandler*, *RegisterIdleHandler*, *RemoveIdleHandler*, *RegisterMenuFixer*, and *RemoveMenuFixer*
- Special utility methods, such as *UniqueDocID*, which are used only by PicoApp

In Crapgame, we did override *Run*, but we could just as easily have overridden the hook method *DoApplicationSetup* instead—which is what we did in PicoSketch.

What Your Application Subclass Can Override

TPicoApp has a great many hook methods. You'll need to override some of them in order to get special behavior—these in particular:

- Various *DoWith*[Action] methods, such as *DoWithGrow*
- Other hooks, such as *DoEachLoop*
- *DoAbout* so that you can put up a custom About dialog box, as in Crapgame
- *ShowIntro* so that you can put up a special introductory dialog box when the application starts up

You'll find a number of examples of such overrides in the Crapgame and PicoSketch code on the code disk.

Introductory and About dialog boxes

Incidentally, by default PicoApp doesn't put up an introductory dialog box at application startup. It does, however, provide a *ShowIntro* method, which, by default, puts up the About dialog box at startup by calling *DoAbout*. But to get *ShowIntro* to do its thing, you must override *DoApplicationSetup* or *Run* to call *ShowIntro*. That gives you the default behavior and is how we handled startup in PicoSketch. If you want a

custom introductory dialog box, one like Crapgame's, you can handle it by overriding *ShowIntro*. Then also override *DoApplicationSetup* (or *Run*, as we did in Crapgame) to call *ShowIntro*.

In Crapgame we followed the introductory dialog box with a dialog box that let users select various game options. That dialog box was managed by the *SetGameOptions* method, which we called from *Run*. You could call something similar from *DoApplicationSetup*.

Example: *TPicoApp* subclass *TCrapsApp*

Class *TCrapsApp* is Crapgame's application subclass. It overrides only a few methods aside from those it is required to override.

In particular, *TCrapsApp* overrides *TPicoApp.DoAbout* in order to put up its custom About dialog box. And similarly, *TCrapsApp* overrides *TPicoApp.ShowIntro* to put up a custom introductory dialog box rather than defaulting to the About dialog box.

TCrapsApp also overrides *TPicoApp.Run* so that it can do some setup work before starting the main event loop. We did this mainly to show that you can override almost any method—under the right circumstances. A better approach would be to override the hook method *DoApplicationSetup*—which is what we did in PicoSketch.

TCrapsApp adds several methods of its own, including *ICrapsApp* and *SetGameOptions*.

You can see these methods in the Crapgame source code on the disk. *TCrapsApp* is defined and implemented in the file UCrapsApp.p.

Writing a *New*[MyAppname] Function

To make complying with PicoApp's memory management rules easier, write a simple function whose task is to create an instance of your application class. This function should use the *Preflight* method of class *TMemory* to preflight the object allocation. Refer back to Chapter 14 for details. See the code for Crapgame and PicoSketch for examples.

Subclassing *TPicoDoc*

Writing a subclass of *TPicoDoc* defines your document object class. Some applications might need more than one document subclass. A word processor, for example, might create both formatted documents and text-only documents without fonts or styles. That could be handled either by one document class that simply writes the two different file types or by two document classes. But most applications will need only one *TPicoDoc* subclass.

In Crapgame, for example, we created a document subclass called *TCrapGame*. The game object is not really much of a document in most ways: It doesn't read from or write to a file, and it doesn't print. But its position in the PicoApp architecture is equivalent to that of a document.

As we did for the application subclass of *TPicoApp*, we'll look at what the document subclass of *TPicoDoc* must, must not, and can override.

What Your Document Subclass Must Override

Your document subclass must override only two methods: *MakeWindows* and *MakeViews*. You might well need to override others, including *SwitchViews*, *FreeViews*, and *FreeAlternateViews*, as we did in Crapgame. See the Crapgame and PicoSketch source code on the code disk for examples.

Override *MakeWindows* to call *NewSimpleWindow* or *NewSpecialWindow* to create your document's window objects. Then call *inherited MakeWindows* to cause your views to be created. Finally, rename your window(s) if you need to by calling the Toolbox procedure *SetWTitle*. See the Crapgame and PicoSketch source code for different examples.

Override *MakeViews* to specify display rectangles for your views, create the views, string them together in a simple linked list, and install the head of the view list in the instance variable *fCurrentViews*. (See the projects at the end of this chapter for a way to simplify *MakeViews*.)

Override methods such as *SwitchViews*, *FreeViews*, and *FreeAlternateViews* in special circumstances. For example, Crapgame uses two different view lists, so it needs to override these methods to be able to switch those lists and to free them. PicoSketch doesn't use this facility, so it doesn't override *SwitchViews*.

What Your Document Subclass Must Not Override

TPicoDoc contains several utility methods that you should never need to override:

- *SetDocID*
- *DocIDOf*
- *CurrentWindowOf*
- *CurrentWindowObjectOf*

What Your Document Subclass Can Override

TPicoDoc contains several hook methods you might want to override in special cases:

- The methods that have to do with Cut and Paste commands so that you can implement cutting and pasting
- *SwitchViews* (if you have alternate sets of views)

- *DoBeforeClosing*
- *FreeAlternateViews*

You might also want to override the *ReadDoc* method. So far, its actions are not implemented in PicoApp, and the method is a stub. You could also create methods such as *PrintDoc*, *SaveDoc*, and *RevertDoc* to work with the document.

Example: *TPicoDoc* subclass *TCrapGame*

Class *TCrapGame* is Crapgame's document subclass. It adds a good many methods for dealing with the player list, but its overrides are fairly few.

Among the overridden methods are *MakeWindows*, *MakeViews*, *SwitchViews*, *FreeViews*, *FreeAlternateViews*, and *DoBeforeClosing*. As we've observed, most of these overrides involve Crapgame's alternate view list, which is the mechanism we've used to implement a "statistics view" for the current player, to show how the dice are performing for that player.

Ordinarily we wouldn't need to override *FreeViews*, but Crapgame does because it needs to remove the current player object from the main view list before freeing the list. *FreeAlternateViews*, which is called by *FreeViews*, frees the player list and all of its players (once the current player is out of the main view list).

We override *DoBeforeClosing* to post a final message to the user as a Crapgame document closes. It simply calls another method, *GoodbyeMessage*, to do the work.

Most of *TCrapGame*'s added methods manage the player list object: getting the current player, setting the next one, removing a player from the list, and so on.

Many of these methods, such as *LastPlayerOf* and *SetNextShooter*, are also listed as overrides. They are overrides from class *TGame*, *TCrapGame*'s abstract ancestor, itself descended from *TPicoDoc*. The methods are declared as stubs in *TGame* so that they'll be known polymorphically when a *TCrapGame* object is installed in a *TGame* instance variable. This solves a scoping problem in which certain classes needed instance variables of a game type but had to be declared ahead of *TCrapGame*. The overrides in *TCrapGame*, then, simply flesh out the stubs.

Writing a *New*[MyDocName] Function

As for *TPicoApp*, write a simple function whose task is to create an instance of your document class.

Subclassing *TPicoView*

Subclassing *TPicoView* creates all the visual aspects of your program. Each distinct visual element should be a separate view object, especially if it must interact with the user by means of a mouse and the keyboard. (Of course, what's distinct is a matter of interpretation.)

We'll again look at what you must, must not, and can override.

What Your View Subclasses Must Override

Your view subclasses must override *TPicoView*'s *Display* method. They might also need to override *DoClick* and possibly *DoDoubleClick* and *DoTripleClick*. The Crapgame and PicoSketch source code provides examples.

Override *Display* if your view is to display anything—usually a given. Only two of Crapgame's view subclasses don't override *Display*—*TTitleView* and *TStatsButton*. That's because *TTitleView* and *TStatsButton* are subclasses of other *TPicoView* subclasses, *TStatText* and *TTransparentButton*, which have already taken care of the *Display* override. Your *Display* override for a view subclass can do whatever it needs to. The view might create several of its own subviews, for instance, dividing up the display task among them. The view's *Display* method would call its subordinates' *Display* methods. *Display* can call QuickDraw routines or even higher-level graphics routines of your own. It can draw text with *DrawString*, or it can call upon TextEdit to render the text. Crapgame and PicoSketch provide numerous *Display* examples.

If your view needs to interact with the user and the mouse, override *DoClick* and perhaps *DoDoubleClick* and *DoTripleClick*. *DoClick* defines what the view does when it's single clicked. The other two define responses to multiple clicks. You can also use these methods to detect modified clicks—Option key down, Command key down, and so on. Use PicoApp's utility routines *OptionKeyDown*, *CmdKeyDown*, *ShiftKeyDown*, *CapsLockKeyDown*, and *ControlKeyDown*. The source code for Crapgame's *TTitleView* class provides examples of using these utility routines. If your view needs to take keyboard input, you might also need to override *DoKeyCommand* from *TEvtHandler* to process the keys. See the discussion of key handling in Chapter 4.

What Your View Subclasses Must Not Override

Some of *TPicoView*'s methods are meant never to be overridden:

- *SetViewRect*, *SetSub*, *SetWindow*
- *RectOf*, *SubviewOf*
- *DoCursor*
- *TrackMouse*, *Clicked*, *DoMouseCommand*

The first two groups are utility functions. *DoCursor* takes care of changing the cursor shape for your view if you've loaded a cursor resource. The mouse click methods do what all views need and have plenty of hooks for customizing. *TrackMouse*, for instance, calls the hook methods *Hilite*, *DoBeforeTrackingMouse*, *DoWhileTrackingMouse*, and *DoAfterTrackingMouse*.

What Your View Subclasses Can Override

TPicoView provides a number of hook methods that you might want to override:

- *Grow*—if your view must change size as the window does
- *DoBeforeGrow*, *DoAfterGrow*—if your view must perform some tasks before or after the window grows
- *DoActivate*—if your view must highlight or unhighlight a selection when its window activates or deactivates

Views can also override *FixMenus*, *FixFileMenu*, and *FixEditMenu* if they need to enable menu items.

Class *TButton*, which is a subclass of *TPicoView*, demonstrates various ways in which *Hilite* can be overridden. Rounded-rectangle buttons, check boxes, and radio buttons are each highlighted differently when the mouse button is down in their areas. You might want other views to be highlighted when they are clicked.

PicoSketch shows some of the uses for overrides of *DoBeforeTrackingMouse*, *DoWhileTrackingMouse*, and *DoAfterTrackingMouse*. In PicoSketch, those methods are used to implement mouse-driven drawing. Some views might use overrides of these methods for selecting text, graphics, or other items with the mouse. You don't have to worry about code for detecting clicks—your views inherit *TrackMouse*, *Clicked*, and *DoMouseCommand*.

Example: *TPicoView* subclasses

For Crapgame, we created subclasses *TCrapsPlayer* and *TDie*. *TButton* is already a subclass of *TPicoView*, so its subclass *TThrowButton* is also a view object. We also needed the *TPicoView* subclass *TTitleView* to draw the "Crapgame" title in our window. *TPicoView* already has a subclass called *TStatText*, which stores a *Str255* string and provides methods for manipulating and displaying the string. We simply made *TTitleView* a subclass of *TStatText*. We subclassed *TPicoView* directly for *TBorderView*, which draws a border around the other views in the window. We also created a special view class for displaying dice statistics (more on that in Chapter 17). And *TPicoView*'s *TWindow* subclass is the basic, unchanging portion of our Crapgame window. Figure 16-2 on the next page shows the *TPicoView* class hierarchy. The Crapgame-specific classes are highlighted. They inherit basic view functionality from *TPicoView*: the initialization of fields with *IPicoView* and the ability to chain several views in a view list.

The Crapgame-specific classes override several methods, including *TPicoView*'s *Display* method (a stub) so that they can specify what drawing they do.

And they add a variety of methods of their own—methods for button hiding and dimming, button response to mouse clicks, calculation, field access, and the like. We'll explore the most important of these view classes—and complete the implementation of Crapgame—in the next chapter.

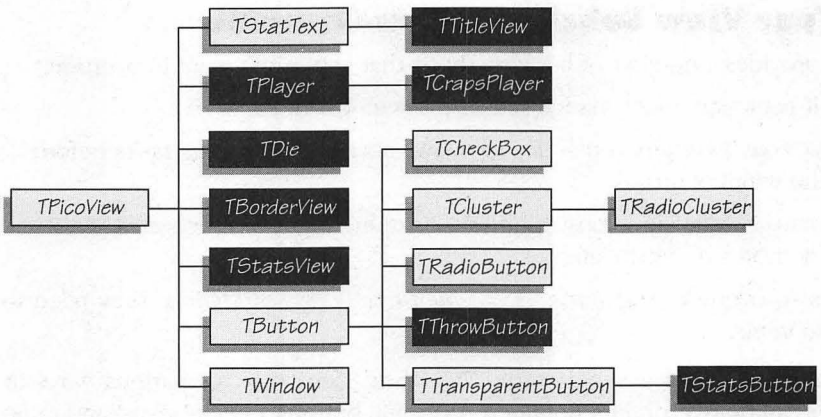


Figure 16-2.
The view class hierarchy.

Writing *New*[MyViewname] Functions

For each view class, write a simple function whose task is to create an instance of your view class.

Writing Additional Classes

The PicoApp classes plus your subclasses of them define only part of what your application does. Those classes handle menus, windows, events, documents, and display. But if your application does calculations, file handling, data communications, and so on, those additional parts of the application have to be added. Writing them as classes is a good approach, and these classes make up the extended class library (PicoApp plus your application's classes) that constitutes your application.

For example, in Crapgame we need the class *TPlayerList* to manage our list of player objects. The player objects, dice, and buttons are views, but they carry out other tasks, too. For instance, in addition to being views, player objects manage throwing the dice and interpreting their results, and dice objects generate random numbers. Those functions are above and beyond the objects' functions as views.

Writing *New*[MyClassname] Functions

For each of your classes, write a simple function whose task is to create an instance of your class.

Preparing Your Version of *UMyGlobals*

You must copy the file UMyGlobals.p/PA to UMyGlobals.p and then edit your copy of UMyGlobals.p.

The unit *UMyGlobals* defines global constants (and perhaps other elements) needed throughout your application. PicoApp expects you to provide such a unit, with precisely that name, containing menu-related, dialog box–related, and other global information.

To copy the file UMyGlobals.p/PA to UMyGlobals.p, in the Finder, select the file UMyGlobals.p/PA in folder PicoApp, Part 2. Then choose Duplicate from the File menu. This creates a copy of the file called Copy of UMyGlobals.p/PA. Move the copy to your project's folder and rename the copy UMyGlobals.p.

Then you must edit your new UMyGlobals.p. The chief set of definitions in *UMyGlobals* is a set of constants that give names to various menu items. Here's an abbreviated view of the constants in UMyGlobals.p/PA when you copy it:

```
const
  kAboutItem = 1;  { Apple menu--standard item }

  kNewItem = 1;    { File menu--standard items; note that numbers can change }
  kOpenItem = 0;
  kCloseItem = 2;
  kSaveItem = 0;   { Unused items get the value 0 }
  kQuitItem = 3;

  kUndoItem = 1;   { Edit menu--standard items }
  kCutItem = 3;
```

The Apple and Edit menu items are mostly fine as they are. If you added any extra items to your Edit menu, such as a Show Clipboard command, you need to add named constants for the extra items in your UMyGlobals.p. The numbers you use correspond to the items' positions in the menu. Lines separating sections of the menu count as positions, too.

The File menu probably needs more work before it will be appropriate for your application. Many of the standard File menu items are given the value *0* in the original file UMyGlobals.p/PA. Our examples, Crapgame and PicoSketch, needed only the most minimal File menus—with New, Close, and Quit commands—so that's all we implemented. Only those items have nonzero numbers. However, the MENU resource for PicoApp's File menu contains all the items listed. You'll need to be sure that the list in your edited copy of that resource and the list of constants in your file UMyGlobals.p match. Change any numbers you need to—in your UMyGlobals.p—to match the positions of those items in your File MENU resource. Don't remove any names from the list in your file UMyGlobals.p. PicoApp might refer to those names, and it won't compile if you delete them. If you don't have the items in your application's File menu, set their values to *0*.

If your application has menus besides Apple, File, and Edit, you need to add constants for their items in your file UMyGlobals.p (or somewhere). And, if you added more dialog boxes, you'll need to include their resource ID numbers in your file UMyGlobals.p.

You can add anything else that needs to be declared globally. You might need additional constants, variables, types, and utility procedures or functions. Write the bodies of any procedures or functions in the implementation part of unit *UMyGlobals*. (Any globals you declare in *UMyGlobals* supplement PicoApp's globals in *UPAGlobals*.)

Crapgame's *UMyGlobals*

For Crapgame, *UMyGlobals* resembles Listing 16-1. You'll find the full version on the code disk.

```
unit UMyGlobals; { File is UMyGlobals.p }
⋮
    kAboutItem = 1;   { Apple menu--only item number needed }

    kNewItem = 1;     { PA: File menu--standard item }
    kOptionsItem = 2; { CG: New Options item in File menu }
    kCloseItem = 3;   { PA: Standard item, but number can change }
    kQuitItem = 4;    { PA: Standard item, but number can change }
    kOpenItem = 0;    { PA: Other items not used in Crapgame }

    { Edit menu--standard items  }

    { Dialog box resource IDs }
    kRemovePlayerDialogID = 2;
    kAnnounceWinnerDialogID = 3;

    { Buttons in Crapgame's custom About dialog box }
    kScoringButton = 2;
    kRulesButton = 3;
    kTechInfoButton = 4;

    { Options dialog box items }
    kPlay = 1;          { OK button in Options dialog box }

    { Radio buttons }
    kThree = 3;    { Means one player: Number of players equals itemNo - 2 }
    kFour = 4;
```

Listing 16-1. *(continued)*
Unit UMyGlobals.

Listing 16-1. *continued*

```
    { Other dialog box items }
    kTitleItem = 1;      { For introductory dialog box }
    kDieItem1 = 2;
    kDieItem2 = 3;

    { Crapgame error codes }
    kUnableToAllocateDie = 80;
    kUnableToAllocatePlayer = 81;

    { Sizes of Crapgame objects }
    kSizeOfDie = kSizeOfViewObject + 18;          { = 82 bytes }
    kSizeOfTitleView = kSizeOfStatText;           { = 73 }
    kSizeOfBorderView = kSizeOfViewObject + 2;    { = 66 }
    kSizeOfThrowButton = kSizeOfButton + 4;       { = 80 }
    kSizeNeededForDoc = kSizeOfCrapgame + kSizeOfWinObj +
        kSizeOfTitleView + kSizeOfBorderView + kSizeOfThrowButton +
        kSizeOfMacWindow +  kSizeOfStatsButton + kSizeOfStatsView + 5 *
        kSizeOfCrapsPlayer + kSizeOfPlayerList;   { = 5546 bytes }

  type

    { Messages is an enumerated type used to simplify printing messages }
    { See method DisplayMsg in TCrapsPlayer }
    Messages = (natural, craps, winner, pointToMake, gameOver, shootEm);

    Rect8Array = array[1..8] of Rect;  { To pass display rectangles to players }

implementation
end.  { UMyGlobals }
```

Creating Your Main Program

Once all the other pieces are in place, you need to write a simple main program.

All the little main program really does is to create a *TMemory* object and install it in the global variable *gMemory* and then, if that's successful, create and initialize your application object.

To create the memory object, you should use the *NewMemoryObject* function provided by PicoApp. And you should use your own *New*[AppName] function to create the application object. This function should follow the memory allocation rules outlined in Chapter 14 (as should all of your memory allocations).

Once your application object is initialized, send it a Run message.

For an example, see the main program for Crapgame, Listing 10-1, in Chapter 10.

Compiling Your Application

Now let's look at the steps you'll go through in the process of compiling your own PicoApplication: segmenting your application, compiling and debugging your application, optimizing your application, setting the "bundle bit" of your application in ResEdit, checking out your application.

Segmenting Your Application

Before you can compile your code, you need to segment it because the Macintosh segment loader handles chunks of code no larger than 32 kilobytes at a time.

When you compile a program on the Mac, the program's compiled code is put into a resource called a CODE resource. CODE resources contain executable code, and an application can have one or more such resources. As procedures are encountered during execution, the Mac switches CODE resources in and out of memory. While it's in use, a CODE resource is locked in the heap, but when it's not in use, it's eligible to be "purged," or unloaded, from memory. If you're familiar with the use of overlays in other programming systems, you will find this memory purging process similar. The catch is the 32-kilobyte limit on the size of CODE resources.

You can use whatever mechanisms your system provides to tell your compiler which CODE resource (or segment) to put the code for a particular procedure or function into. In MPW, you use the *$S* compiler directive in your code to designate segmentation. In THINK, you can use the project window to select which files go into which segment, or you can use a segmentation compiler directive (also *$S*) as in MPW. The THINK default segmentation practice is to put the whole of a unit into the same segment. But, by using the directive, you can put specific procedures and functions into different segments even though they're in the same unit physically.

Check the documentation for your compiler to see how to do segmentation for it.

Objects and segmentation

Traditionally, different modules of code are put into different segments (or overlays). For instance, all program initialization code might go into one segment so that that segment can be loaded into memory when it's needed but purged from memory when it's not. Likewise, all program termination code, all printing code, and so on.

This scheme allows for optimum use of scarce RAM at runtime because code not immediately needed is kept out of memory until it is.

Objects make this approach a bit trickier. An object is, at least textually, a separate module in its own right. It "contains" not only its instance variables but also its methods. Because fields and procedures are in the same basket, it seems only logical to put everything pertaining to one object class into the same segment. If you're using THINK Pascal's usual mode of specifying segmentation, that's precisely what you do. Each unit—which contains the whole of a class—goes into one segment.

That makes it hard to put all initialization code, all printing code, and so forth together, and that reduces the value of segmentation as an overlay mechanism.

PicoApp uses the *$S* segmentation directive to put specific methods in different segments. The *$S* directive also controls which segments are unloaded and which are kept resident. For a list of the PicoApp segments, see *TPicoApp.UnloadSegments* in file UPicoApp.p. Besides the segments in that list, PicoApp has several segments that are kept resident; the principal ones are *Main* and *UtilFreqRes*. You can put your methods into any of the PicoApp segments or put them into new segments—wherever each method seems to fit most logically.

The problem we've raised concerning objects and segmentation is somewhat illusory, though. When an object class is compiled, its method code gets stored in a CODE resource, naturally. But when instances of the class are created dynamically at runtime, the "objects" in the heap contain only the objects' variable data, not their method code. All the method code is somewhere else, accessed by means of a method dispatch table. (See Chapter 9.) All the method code for an object doesn't necessarily have to be in the same CODE resource. The code for some methods could be in one CODE resource, the code for others in another resource.

Since release 3.0, THINK Pascal has allowed the use of the *$S* directive for more precise control over segmentation—procedure by procedure, if you like, rather than unit by unit. You can put different methods in different segments—hence, in different CODE resources. (By the way, version 2 of THINK reportedly had some problems with intersegment method calls, but version 3 should have fixed them.)

For details about controlling segmentation in your application, see the file "PicoApp Segment Notes" on the code disk.

Compiling and Debugging Your Application

Once all the pieces are in place, you compile them. In THINK Pascal, you choose Go from the Run menu to compile. In MPW Pascal or TML Pascal, you issue an MPW command in one of MPW's open document windows: Just type the command into the window and press Enter (or, if the command has already been typed before, select the command and press Enter).

You might want—or need—to set up some compiler directives and compilation options to control how the compilation goes. In MPW Pascal and TML Pascal, directives are inserted in the source code at strategic locations—directives for things like conditional compilation, inserting routine names for debugger use, checking safe use of object references, checking arithmetic overflow, checking ranges on arrays, strings, and sets, and so on.

In THINK Pascal, you set the most common directives in the project window by means of the mouse: *D* (generate information for the debugger), *N* (insert routine names for the debugger), *V* (check for overflow), *R* (check ranges). You can also insert directives where you need them in the source code, including directives not

available in the project window—*$I* for turning automatic Toolbox initialization on and off, conditional compilation directives, and so on. The Compile Options command in the Project menu opens a dialog box that allows you to set other compiler options—whether the compiler should generate code for an MC68020 or 68030 processor, whether a floating point processor is available, and so on.

Of course, compiling never goes completely smoothly. You're bound to encounter errors, and that means debugging.

Debugging your application

Your application isn't finished until you've tried your best to break it—and fixed it when you inevitably do.

PicoApp is moderately reliable now, so any errors you encounter are more likely to stem from Crapgame's code—from the classes and other items we added to PicoApp. Remember, though, that PicoApp is supplied as demonstration software to illustrate the concepts of object-oriented application design and development. It's intentionally much simpler than TCL or MacApp, its older and more mature brethren of the application framework fraternity. Use it, but keep its limits and possible flaws in mind.

Under MPW Pascal, you can use Apple's SADE debugger or a low-level debugger like MacsBug or TMON. Under THINK Pascal, you'll use the built-in LightsBug and other debugging facilities.

Optimizing Your Application

You can do many things to optimize an application to improve its speed, its compiled code size, its use of space in the heap, its user interface, and so on. For now, we'll look at one performance issue.

The performance of PicoApp's memory management mechanism depends on how you choose to size the two memory reserves—the temporary reserve and the emergency reserve we looked at in Chapter 14. PicoApp gives the reserves default values, which you can use as is. That's what we did in PicoSketch. You can supply your own, more finely tuned, sizes and thereby improve performance.

How do you arrive at the best sizes for your application? The only answer at this stage: Experiment.

Try reducing the reserve sizes to see how the application performs. How many documents can you open? Do you detect any problems? Does the application fail?

If reducing the reserve sizes creates problems, try gradually increasing the sizes until you hit upon a combination of sizes that works well. Experiment with the application, putting various kinds of loads on it. See what works and what breaks.

The Crapgame source code provides an example of how to override the default reserve sizes, although it really doesn't need to—you can open only one document in Crapgame. See the file CrapgameMain.p. PicoSketch is a more appropriate vehicle for understanding the system because it lets you open as many documents as memory allows.

One useful modification of PicoApp that I contemplated but did not implement is a mechanism for monitoring memory usage during actual executions of an application. The output would be a guide to sizing the reserves. See the projects at the end of this chapter.

Setting the Bundle Bit of Your Application

Preparing your application's BNDL, FREF, ICN#, and signature resources is only part of the process of giving your application a double-clickable desktop icon. You also need to set the "bundle bit" for your application, which tells the Finder that the application has a BNDL resource and the other necessary resources.

To set the bundle bit, run ResEdit. Select your application's resource file. Choose Get Info from the File menu. In the dialog box, click the check box for the Bundle option. Then close the dialog box.

After you've edited and saved your resource work, compiled your program, and set its bundle bit, the program's icon should appear on the desktop. If it doesn't, try "rebuilding" the hidden Desktop file on your disk: Hold down the Command and Option keys as you boot up with your application on a floppy disk. Then copy the application back to your hard disk if you want to.

Checking Out Your Application

Much of what you need to check is the behavior of the non-PicoApp parts of the application. But you also need to pay attention to some of the PicoApp aspects:

- Menus: Are they all there? Are they enabled and disabled under the right circumstances?
- Behavior of menu items: Do they produce correct results? Are they available (enabled) when you need them?
- Windows: Do they have the right parts (close box, zoom box, size box, scrollbars, title)? Do they behave correctly (zooming, growing, closing, and so on)? Are they correctly positioned on the screen?
- About dialog box: Does the menu item work? Does the dialog box look right? Is what it says correct? Is it correctly positioned on the screen?
- Introductory dialog box: Is it working correctly at application startup? Is it positioned correctly?

- Buttons and other controls: Do they get events when clicked or when a key is pressed? Do they respond correctly? Are they dimmed when appropriate and enabled when appropriate?
- Displays: Is everything correctly drawn? Is everything correctly located? Is everything correctly updated? Do any problems arise if the window is resized or zoomed?
- Icons: Does the application icon display in the Finder? Does it respond correctly when you double-click it? If you create documents and provide them with icons, do the icons display correctly and respond correctly when double clicked? (Currently, PicoApp doesn't support opening documents by double-clicking their icons in the Finder.)
- Quitting: Does the application quit correctly?

Answering these questions should assure you that you've interfaced your application's elements correctly with PicoApp's.

Summary

In this chapter, we've looked at the steps you take to use PicoApp in developing your own PicoApplication—project setup, resources, subclasses of the PicoApp classes, other classes you need, compiling, testing, and debugging, and providing an icon for your application.

Of course, in order to build an application with PicoApp, you need to study its structure and its classes so that you know what methods and instance variables are available, what parameters are required for superclass initialization methods, and so on. And deciding exactly which methods to override is more an art than a science. You've seen some guidelines and examples, and you can look at the source code for Crapgame and PicoSketch on the code disk. And, of course, you can examine the source code for all of PicoApp itself.

It's a good idea to start with something small for your first application. In the course of creating a small project, you can accustom yourself to using objects in an application setting and get the hang of developing from a framework such as PicoApp.

In the final chapter of Part 2, we'll look at the specifics of Crapgame's player, dice, and other special classes.

Projects

- Complete the PicoApp-based application you've been working on all through Part 2.
- Convert an application you've been working on the old-fashioned, non-object way into a PicoApplication.

- Currently, *TPicoDoc.MakeViews* requires you to do a lot of rather low-level work. You have to create your view objects and, object by object, string them together in a simple linked list to pass to the window object.

  Add methods to *TPicoDoc* to add view objects to the view list and to remove them from it. The add method can simply add the view object at the head of the list. (If your documents use multiple view lists, as Crapgame's do, you might want to create extra methods for adding objects to and removing objects from those alternative lists.) Then modify *TPicoView.IPicoView*, which is called by the initialization methods of all view objects, to make a call like

  ```
  itsDoc.AddView(self);
  ```

  This is a way of automatically asking the document to add "self" to the document's view list. Such an approach should simplify *MakeViews* considerably.

- Build some useful instrumentation into PicoApp—things to help you improve the application's performance during development. A mechanism for monitoring memory usage during runtime and then reporting statistics that can help you size the memory reserves would be a good idea.

  Periodically during the main event loop, build a list of CODE and other common resource types, such as PACK, WDEF, CDEF, and so on. You can get handles to resources by calling Resource Manager routines (*Inside Macintosh,* I-103). You might want to look at the THINK Class Library's *CList* class for an example of a convenient way to store handles.

  Each time you build the resource list, scan through it, using *SizeResource* to determine how big each resource is and summing their sizes. If the current size of the list is greater than its previous maximum size, replace the old maximum with the current size. Make a copy of the list for later use.

  At the end of the run, you'll have the peak usage figure (for CODE and related resource types) plus a list of the actual resources in memory at that time. You can scan the list to determine which resources were in use.

  This provides useful information for estimating the sizes your temporary and emergency memory reserves need to be. Display the results in a window before the application quits. You might want to create a special document class to do this. Its methods would manage the resource lists, scan them for sizes, and display the results.

  Bracket all the code for this facility inside conditional compilation directives so that you don't have to include this instrumentation code in your final application.

- Currently, PicoApp provides little support for the Macintosh's Clipboard. *TPicoApp* does provide hook methods for copying the Clipboard (scrap) to your application's private scrap, if it has one, and for copying the private scrap to the Clipboard. If you override these methods, PicoApp does know enough to

call them in most of the right places. *TPicoDoc* also provides hooks for handling the Undo, Cut, Copy, Paste, and Clear commands in the Edit menu. To implement a clipboard in your application, you need to override these methods. See *Inside Macintosh,* I-453 for information on clipboards and their contents.

Consider whether you can provide more general clipboard support than PicoApp currently does. How much of the clipboard apparatus can be general, and how much is application specific? Can you at least add support for cutting and pasting the two most common data types, text and pictures, and for a Show/Hide Clipboard command in the Edit menu?

17

THE MODEL

Now that we've taken the grand tour through PicoApp—and thus to some extent through any object-oriented application framework such as MacApp or TCL—let's finish up our main example program, Crapgame.

We've seen the generic aspects of our application: event handling, menus, windows, and so on. Now we'll look at the specifics that make Crapgame a game of craps:

- The idea of object modeling
- The control structure of the game
- Crapgame's *TButton* subclasses
- Crapgame's *TDie* class
- Crapgame's *TCrapsPlayer* class

The Model

The creators of the Smalltalk language developed the model-view-controller (MVC) approach to software architecture. Their idea was to separate the several aspects of an application's functionality into distinct components. The controller component, for instance, is responsible for user interaction with the application—as the event-getting and event-handling apparatus in PicoApp is. The view component is responsible for the application's displays—as the view objects in PicoApp are. And the model component is responsible for the elements that make the application specifically what it is: a word processor, a game, or something else. Figure 17-1 on the next page suggests the relationship between the MVC components and PicoApp's categories of objects.

PicoApp's is not a pure MVC architecture. In PicoApp, the functions of the MVC view and controller components are intermixed within a library of various object classes. PicoApp views, for instance, have controller aspects. But the MVC idea is nonetheless useful to us. Because PicoApp provides so much of the view and controller components for us, most of PicoApp programming is a matter of designing and implementing the model component.

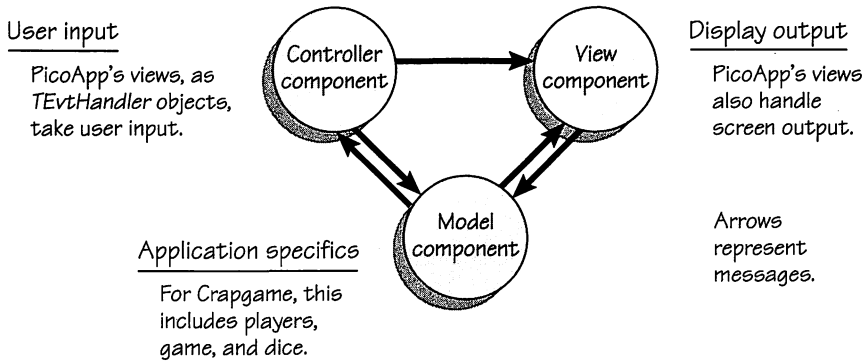


Figure 17-1.
The model-view-controller (MVC) approach to software architecture developed by the creators of Smalltalk. Note the correspondences with PicoApp's "architecture."

Models

Most programming is a process of modeling real-world entities, events, processes, and systems in software. The goal of design is to create a model that can be turned into software.

Object-oriented techniques are particularly useful for such modeling because we can often make a one-to-one mapping of real-world objects to software objects. Naturally, our software objects are, to a considerable extent, abstractions. They represent the real-world objects and mimic much of their behavior and characteristics. But, of course, they also leave out a lot, focusing instead on essence.

The Crapgame Model

In our case, the model derives from the game of craps, a multi-player game that uses dice. Several elements in the game provide bases for building our model, and thus our application.

Crapgame models a group of players who take turns throwing a pair of dice. Hence, our goal, and the basis of our design, has been to develop a group of interacting objects that embody such a model.

We've chosen to model the players and dice with objects. Player objects know how—on command—to throw the dice and how to evaluate the results and act accordingly. Die objects know how to roll on command and produce pseudorandom integers within the correct range.

These objects are controlled by a button, which the real-world game player clicks with the mouse to signal the current player object to roll the dice.

Control Structure of the Game

Crapgame's control structure is simple, and it models the real-world game pretty faithfully.

A real-world user clicks the Throw button with the mouse. The button object's *DoClick* method queries the game (document) object to get a reference to the current player object, which the game object gets from its player list object. Then the button sends that player object a Play message.

When the player object receives a Play message, its *Play* method sends Roll messages to its two die objects, which obligingly return pseudorandom numbers in the range 1 through 6. The rest of the *Play* method interprets the rolls according to the player object's current state.

The player object adjusts its score based on what it has just rolled. If the player object has lost all of its money, it sends a message to the game object, asking to be removed from the player list. If the player object simply "craps out," ending its turn, it sets its final state appropriately and sends the game a message to set up the next player object. Then, when a user clicks the Throw button again, that next player object is "handed over" to the button for play. And so on.

Player States

In craps, the current shooter is either trying to establish a "point" or trying to match a point already established, the two states an active player can be in; a player object is a "state machine." Figure 17-2 shows the states the player object can be in.

When a player's turn starts, he or she is in State 1. The player might stay in that state for several rolls if he or she manages to roll one or more "naturals" (7's or 11's). That state ends when the player either craps out (with a roll of 2, 3, or 12) or establishes a number as the current point to match (by rolling 4, 5, 6, 8, 9, or 10).

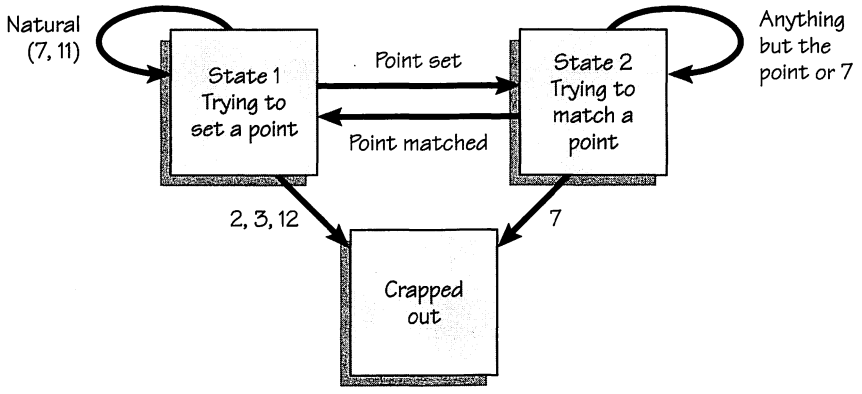


Figure 17-2.
The player object as state machine.

State 2 begins when the player establishes ("sets") a point. All subsequent rolls are attempts to match the point. The state ends when the player either matches the point and wins (and then starts over at State 1) or craps out by rolling a 7.

The player oscillates between the two states. We'll see these states implemented in the method *TCrapsPlayer.Play* a little later in this chapter.

Crapgame's *TButton* Subclasses

Crapgame actually uses two buttons, both subclasses of *TButton*. The main button is the Throw button, an instance of *TThrowButton*. The lesser button is the statistics button, an instance of *TStatsButton*. (It's actually a subclass of *TTransparentButton*.) We'll look at the essences of those subclasses.

The Heritage of the Buttons

The button classes are subclasses of *TButton*, so they're subclasses, in turn, of *TPicoView* and *TEvtHandler*. They thus inherit the ability to display themselves and to interact with the mouse and the keyboard. We've already looked at those aspects of button behavior.

What the buttons override

Both buttons must override the *DoClick* method that defines what the button does.

TThrowButton's *DoClick* method gets the current player object and sends it a Play message, as we've said. *TStatsButton*'s *DoClick* method calls the game object's *SwitchViews* method to display the statistics view object associated with the current player. Then the statistics button grows to fill the entire window—when the user clicks anywhere in the window, he or she is clicking the statistics button again. The statistics button is a toggle—clicking the button switches one list of views (the normal play views) to another view (the statistics view) and back again.

The Button Subclasses

Here are the declarations for *TThrowButton* and *TStatsButton*. We'll look at their *DoClick* methods shortly.

```
type
  TThrowButton = object(TButton)

      { kSizeOfThrowButton = kSizeOfButton + 4 = 80 bytes }
      fGame: TGame;     { Store reference to game object }

      procedure IThrowButton (game: TGame; displayRect: Rect);
      procedure DoClick (event: EventRecord);
      override;
        { Do what throw button does--ask player to roll dice }
```

```
    function TThrowButton.TypeOf: Str30;
    override;
      { Return the string 'TTHROWBUTTON' }
  end;  { Class TThrowButton }

TStatsButton = object(TTransparentButton)
  { Is-a cursor adjustor }

    { kSizeOfStatsButton = kSizeOfTransparentButton + 4 = 82 bytes }
    fGame: TPicoDoc;     { Store reference to game object }

    procedure IStatsButton (game: TPicoDoc; title: Str255; active: Boolean;
        displayRect: Rect; showTitle: Boolean);
    procedure DoClick (event: EventRecord);
    override;
      { Do what the button does--display current player's dice statistics }
    function TStatsButton.TypeOf: Str30;
    override;
      { Return the string 'TSTATSBUTTON' }
  end;  { Class TStatsButton }
```

TThrowButton's DoClick method

Here's *TThrowButton.DoClick*:

```
procedure TThrowButton.DoClick (event: EventRecord);
  var
    thePlayer: TPlayer;
begin
  thePlayer := self.fGame.CurrentShooterOf;
  if thePlayer <> nil then
    thePlayer.Play;
end;  { TThrowButton.DoClick }
```

Class *TCrapGame* has a *CurrentShooterOf* method that asks the player list for the current player. *CurrentShooterOf* returns a reference to an object of type *TPlayer*, but it's really a *TCrapsPlayer* object in disguise. *TPlayer* is an abstract ancestor class of *TCrapsPlayer*. Real player objects are of type *TCrapsPlayer*, but for reasons of scope we've invented *TPlayer*. Recall that a *TPlayer* variable can reference ("contain") an instance of its descendant, *TCrapsPlayer*. We don't need a typecast in *DoClick*. Although we have a *TCrapsPlayer* object in a *TPlayer* variable, we can safely send it a Play message. This works because class *TPlayer* has a stub *Play* method. Runtime binding will see to it that *TCrapsPlayer.Play* is the method actually called.

It's good practice to check object references to be sure they're non-*nil* before trying to send them messages, so we do. Then, if all is well, we send the current player object a Play message.

TStatsButton's DoClick method

The statistics button is transparent, overlying but not obscuring the images of the dice objects in the window. Its purpose is to replace the views in the window with a single view showing statistics on how the dice are falling for the current player. Clicking the statistics button switches views. You can see how *TStatsButton* works in its *DoClick* method and in *TCrapGame*'s *SwitchViews* method. *TStatsButton* is in unit *UCrapsViews* on the code disk. *TCrapGame* is in unit *UCrapGame.*

Crapgame's Die Class

A die object has two critical functions: displaying itself and generating a pseudo-random number within the correct range. We'll explore our implementations for both functions.

Here's the class declaration for *TDie*, which implements one die object. (We won't create an object representing two or more dice at the same time.)

```
type
  TDie = object(TPicoView)

    { kSizeOfDie = kSizeOfView + 18 = 82 bytes }
    fOutlineRect: Rect;          { Outlines display rectangle }
    fFaces: Integer;             { Number of faces on this die }
    fCurrent: Integer;           { Current upper face }
    fBaseID: Integer;            { Base ID for ICON resources }
    fGenerator: TRandom;         { Die's random number generator }

    procedure TDie.IDie (NumberOfFaces: Integer; itsDoc: TDocument;
        displayRect: Rect);
      { Initialize the die }
    procedure TDie.SetBaseID (baseID: Integer);
      { Set new base ID if resources with different IDs are used }
    procedure TDie.SetFaces (NumberOfFaces: Integer);
      { Set the number of faces of this die }
    procedure TDie.SetGenerator (g: TRandom);
      { Install a random number generator other than the default }
    function TDie.NumFacesOf: Integer;
      { Get the number of faces of this die }
    procedure TDie.Display;
    override;
      { Display the die on the screen }
    function TDie.Roll: Integer;
      { Roll the die--returns a value in range 1..fNumFaces }
    procedure TDie.Hide;
      { Hide the displayed die }
```

```
    procedure TDie.Reset;
      { Randomize the number generator }
    procedure TDie.Free;
    override;
    function TDie.TypeOf: Str30;
    override;
      { Return the string 'TDIE' }

    { Also inherits methods of classes TPicoView }
    {  and TEvtHandler, but doesn't do anything with TEvtHandler }

  end;  { Class TDie }
```

We designed *TDie* to work smoothly with PicoApp. The class is descended from *TPicoView* so that die objects can be placed in a window's view list and receive Display messages. But *TDie* is also the kind of software component that could be used in many programs—not all of them necessarily PicoApplications. What about using the class in MacApp or TCL, for instance? But a general *TDie* wouldn't descend from *TPicoView* and couldn't be stored in a view list. In Chapter 19, we'll look at a way to have the best of both worlds: a class with complete generality and independence as well as the ability to be integrated with application frameworks like PicoApp. We'll discover a way to treat non-*TPicoView*s as if they did descend from *TPicoView*.

The most important *TDie* methods, of course, are *IDie*, *Display*, *Roll*, *Hide*, and *Reset*.

Dice Initialization

To set up a die for use, we tell it how many faces it has and where it is to display itself. We also have to call the initialization method of *TDie*'s ancestor class, *TPicoView*; set up the outline rectangle, based on the display rectangle; establish initial characteristics of the die, such as what base ID it uses to gets its ICON resources and what face it shows initially; and create its random number generator. Here's the *IDie* method:

```
procedure TDie.IDie (numberOfFaces: Integer; itsDoc: TDocument; displayRect: Rect);
begin
  self.fFaces := numberOfFaces;
  self.IPicoView(displayRect, itsDoc, false);
  self.fOutlineRect.top := self.fViewRect.top - 4;
  self.fOutlineRect.left := self.fViewRect.left - 4;
  self.fOutlineRect.bottom := self.fViewRect.bottom + 4;
  self.fOutlineRect.right := self.fViewRect.right + 4;
  self.fBaseID := 800;   { Base resource ID for die icons }
  self.fCurrent := 6;    { Default face to show initially }
  self.fGenerator := NewRandom(self.fViewRect);  { Create random }
                                                 {  number generator }
end;  { TDie.IDie }
```

Dice Display

There are lots of possible ways to have a die display itself. We could have it draw a number on the screen or draw a picture of its top face or do some kind of animated drawing. The last approach could get pretty elaborate, of course.

We'll use a simple animated approach. The idea is to simulate a tumbling die by drawing, erasing, and redrawing the die over and over again, displaying a random face each time, and finally drawing the face that's up when the die stops rolling.

Again, there are lots of ways to do that, but we'll use Macintosh icons. They're a suitable size, and they're easy to "stamp" onto the screen. And because they're stored as ICON resources, it's easy to use ResEdit to pretty them up if we want to, even after Crapgame has been compiled.

Crapgame's resource fork contains six ICON resources, one for each face of the die. To obtain an icon for drawing, we call *GetIcon*. To draw it, we call *PlotIcon*. And to erase it, we call the QuickDraw procedure *EraseRoundRect*. All of these facilities are in the Mac Toolbox, ready to use.

To get the appropriate icon for a particular roll of the die, we start with the base resource ID, arbitrarily set at 800, and add the number just rolled to the base:

```
ID to use = baseID + current roll
```

This computes the ID of the appropriate icon. Here's our die display method:

```
procedure TDie.Display;
  var
    faceID: Integer;
    theIcon: Handle;
    oldPen: PenState;
begin
  faceID := self.fBaseID + self.fCurrent;     { Compute icon ID }
  theIcon := GetIcon(faceID);                 { Get ICON resource }
  PlotIcon(self.fViewRect, theIcon);          { Draw the icon }
  GetPenState(oldPen);
  Pensize(2, 2);
  FrameRoundRect(self.fOutlineRect, 8, 8);    { Frame the icon }
  SetPenState(oldPen);
end;  { TDie.Display }
```

After plotting the icon in its designated display rectangle, we draw a framing line around it, saving and restoring the drawing environment as we go. Hiding the die is simple:

```
procedure TDie.Hide;
begin
  EraseRoundRect(self.fOutlineRect, 8, 8);
end;  { TDie.Hide }
```

This erases the outline and everything inside it.

Random Numbers

Our two dice need to generate pseudorandom numbers, so we design a random number–generating object class. *TRandom* provides several methods for generating different sorts of random numbers. It also supplies methods for examining the generator's performance and a method to reset the generator so that it starts over.

We design *TRandom* to be as generally useful as possible because many applications need such a facility. Actually, there are two ways we could use the class in Crapgame.

We could have subclassed *TRandom* to create *TDie*. Then we would have been able to say that every die object *is-a* random number generator. That's an appealing way to look at it. But the alternative is to have *TDie* use *TRandom* instead of inheriting from it. A *TDie* object simply creates a *TRandom* object, installing it in one of the die's instance variables. Then, to use the generator, we can make calls like

```
num := fGenerator.Get(upperBound);
```

which sends a Get message to the *fGenerator* object, which returns a number from *1* through *upperBound*. This is the approach we chose for *TDie*, as borne out by the *IDie* method we looked at earlier in this chapter. You can see class *TRandom* in unit *URandomStream* on the code disk.

Random Number Alternatives

TRandom uses the *Random* function built into the Mac's Toolbox. If you should find that *Random*'s algorithm doesn't suit your needs, you can always use a different algorithm. (There are many—see a good algorithms text.)

Here's how.

First, subclass *TRandom* and override any or all of its three random number–generating methods: *Get*, *GetRandom*, and *GetRandomFrac*. (Note: *Get* calls *GetRandom*; *GetRandom* calls *Random*; *GetRandomFrac* has its own independent algorithm.) Reimplement these methods with your algorithm. Remember that you still have to satisfy the semantics of the methods. *Get* delivers a pseudorandom integer in the range *1* through *upperBound*. *GetRandom* delivers a pseudorandom integer in the range *–32767* through *32767*. And *GetRandomFrac* delivers a pseudorandom real in the range *0.0* through *1.0*.

After creating your die object (or whatever), replace its default generator with your new one by

- Creating a new generator object of your type
- Sending the die object a SetGenerator message, passing your generator as a parameter

Keep in mind that you can declare your die, card deck, or other object class as a subclass of *TRandom* as an alternative to having it contain a *TRandom* instance variable.

Reseeding the Random Number Generator

TRandom also provides the *Reset* method to reseed the random number generator. This starts it over from scratch and restarts the statistical information available through *TRandom*'s statistical methods. *Reset* uses the current time from the system clock, a long integer obtained by calling the Toolbox procedure *GetDateTime*, as its seed. This results in improved randomization if you call *Reset* fairly often, as Crapgame does before every roll of the dice in the *TCrapsPlayer.Play* method, which we'll look at later in this chapter.

The *Roll* Method

Let's get back to *TDie*. Now we can see how it generates its random numbers. First, we create a *TRandom* object in *TDie.IDie* and install it in the instance variable *fGenerator*. Then we send the generator object Get messages when we need to roll the die.

Here's *Roll*:

```
function TDie.Roll;
  var
    result: Integer;
begin
    result := self.fGenerator.Get(self.fFaces);  { Get a random number 1..6 }
    self.fCurrent := result;                     { Store it }
    self.Display;                                { Show it to players }
    Roll := result;                              { Return it for checking }
end;  { TDie.Roll }
```

First, we send a Get message to *fGenerator*, passing the number of faces on the die as our upper bound. For *TDie*, that will usually be six.

After storing the number in *fCurrent* for use with the *GetLast* method, we call *Display* to show the rolled number. Then we return it so that the player object can evaluate it.

Animating the Dice

Our animation in Crapgame is simple. We can't just put a method into *TDie* to simulate rolling the die over and over. With two dice, that would have the effect of rolling one and then the other. We want the illusion that the dice are bounding across the table simultaneously.

The solution is to write a non-method procedure somewhere in Crapgame to repeatedly roll both dice. Procedure *AnimateDice* is declared in the implementation part of unit *UCrapsPlayer*.

```
procedure AnimateDice (die1, die2: TDie);
  var
    i: Integer;
    junk: Integer;          { No need to keep values--merely displaying }
begin
  for i := 1 to 10 do
    begin
      junk := die1.Roll;    { Returns a random value and displays die face }
      junk := die2.Roll;
    end;  { for loop }
end;  { AnimateDice--not a method }
```

Crapgame's Player Class

TCrapsPlayer is a class of objects that "know" how to play craps. They keep track of what state they're in and what their score is. They roll dice. And they know how to pass the dice when a turn ends and how to get themselves out of the game if they lose all their money. They pretty fairly model human craps players—though without tattoos, cigars, and the ability to blow on the dice and intone "Baby needs a new pair of shoes."

TCrapsPlayer's display is rather elaborate, so we'll take a quick look at that. But its most important method is *Play*. Listing 17-1 gives the *PlayerStates* and *TCrapsPlayer* declarations.

```
type
  PlayerStates = (needToSetPoint, needToMatchPoint);

  TCrapsPlayer = object(TPlayer)

      { kSizeOfCrapsPlayer = kSizeOfPlayer + 589 = 659 bytes }
      { fViewRect, inherited from TPicoView, holds player number }
      fPlayerTitleRect: Rect;       { Display rectangle for player title }
      fState: PlayerStates;         { State the player is in }
      fPointToMatch: 0..12;         { Point that must be matched to win }
      fPointRect: Rect;             { Where to display the point }
      fPointTitleRect: Rect;        { Display rectangle for point title }
      fDie1, fDie2: TDie;           { Dice the player rolls }
      fScore: Integer;              { This player's score }
      fScoreRect: Rect;             { Where to display the score }
      fScoreTitleRect: Rect;        { Display rectangle for score title }
      fTheGame: TGame;              { The game object }
      fMessageRect: Rect;           { Where to display messages }
```

Listing 17-1. *(continued)*
The declaration of class TCrapsPlayer.

Listing 17-1. *continued*

```
    fLastMessage: Str255;          { Last message displayed }
    fNewMessage: Str255;           { New message to display }
    fStatsRect: Rect;              { Display rectangle for statistics view }
    fStatsObj: TStatsView;         { Object to keep track of dice statistics }
  procedure ICrapsPlayer (playerNumber: Integer; game: TGame;
      rects: Rect8Array; die1, die2: TDie; initialStakes: Integer);
  function StatsObjectOf: TStatsView;
  procedure Display;
  override;
    { Display player number, score, point }
  procedure SetNewMsg (msg: messages);
  override;
  procedure DisplayMsg;
  procedure DisplayValues;
  procedure DisplayPlayerID;
  procedure DisplayScore;
  procedure DisplayPoint;
  procedure Play;
  override;
    { Throw the dice and test results }
  procedure Free;
  override;
  function TypeOf: Str30;
  override;
    { Return the string 'TCRAPSPLAYER' }
end;  { Class TCrapsPlayer }
```

TCrapsPlayer Display

In practice, many of the display methods are nested—that is, although they're declared as separate methods, most are called by higher-order methods. *Display* draws the title text for the player number, score, and point, calls *DisplayValues* to fill in the current values, and then displays the current message. *DisplayValues* calls *DisplayPlayerID*, *DisplayScore*, and *DisplayPoint*. *TCrapsPlayer* also includes the method *DisplayMsg*, which can be called at any time to display a message. *Display*'s approach gets everything called at the same time when necessary, as in updates, but it also leaves the smaller methods available to be called individually for special purposes.

You can see the display code in the folder Crapgame, Part 2 on the code disk. Figure 17-3 shows what the player display looks like and shows the methods that produce it.

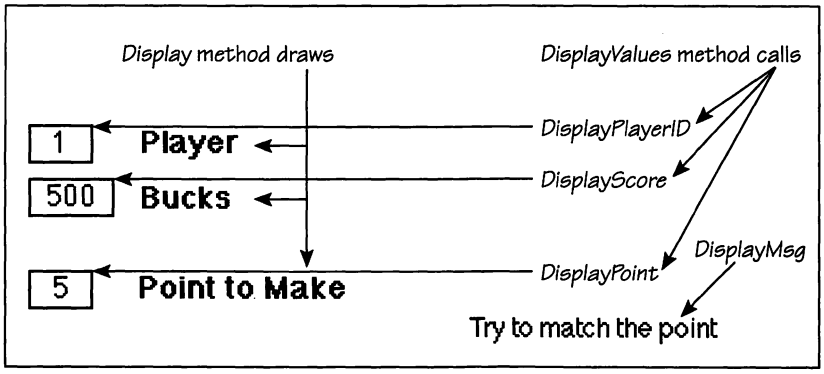


Figure 17-3.
The player display and the methods that produce it.

TCrapsPlayer's Play Method

The *Play* method, called by the Throw button, is the heart of a player object.

Listing 17-2 is a skeleton version of the method, a fairly large method simplified to highlight its structure. You can see the full code listing in file UCrapsPlayer.p on the code disk.

```
procedure TCrapsPlayer.Play;
  override;
  const
    kSmallLoss = 50;              { Dollars }
    kBigLoss = 100;
    kSmallWin = 50;
    kBigWin = 100;
  var
    throwResult: Integer;
    winObj: TWindow;

  function Broke: Boolean;        { Not a method }
    { Tests for no money; removes player from game if true }
    :
begin { Play }
  { Make the dice appear to "roll" on screen }
  AnimateDice(self.fDie1, self.fDie2);
```

Listing 17-2. *(continued)*
A skeleton version of TCrapsPlayer's Play *method.*

Listing 17-2. *continued*

```
{ Randomize the dice random number generators }
self.fDie1.Reset;
self.fDie2.Reset;
{ Make the actual roll of the dice }
throwResult := self.fDie1.Roll + self.fDie2.Roll;

{ State 1:  Player needs to set a point }
if self.fState = needToSetPoint then
  case throwResult of
    2, 3, 12:  { Craps }
      begin
        { Increment statistics }
        { Adjust score for a small loss }
        { Display the message 'Craps!...Next shooter' }
        { Cause an update of the display }
        if not Broke then
          if self.fTheGame.LastPlayerOf then
            { Display dialog box to announce winner }
            { Tell game to close window }
          else
            self.fTheGame.SetNextShooter;
      end;  { 2, 3, 12 }

    7, 11:  { Winner by a "natural"--keep throwing!! }
      begin
        { Increment statistics }
        { Adjust score for a big win }
        { Display the message 'Natural!...Shoot again' }
        { Cause an update of the display }
      end;  { 7, 11 }

    4, 5, 6, 8, 9, 10:  { Point set—save the point }
      begin
        { Set as new point to match }
        self.fPointToMatch := throwResult;
        { Display the message 'Try to match the point' }
        { Cause an update of the display }
        { Increment statistics }
      end;  { 4, 5, 6, 8, 9, 10 }
```

(continued)

Listing 17-2. *continued*

```
      otherwise
        ;
    end  { case }

  { State 2:  Player needs to match an existing point }
  else if self.fState = needToMatchPoint then
    if throwResult = 7 then
      begin
        { Increment statistics }
        { Adjust score for a big loss }
        { Display the message 'Craps!...Next shooter' }
        { Cause an update of the display }
        { Zero out the point to match }
        { Toggle state to needToSetPoint for next time }
        if not Broke then
          if self.fTheGame.LastPlayerOf then
            { Display dialog box to announce winner }
            { Tell game to close window }
          else
            self.fTheGame.SetNextShooter;
      end  { If throw = 7 }

    else if throwResult = self.fPointToMatch then { Player matched point }
      begin
        { Increment statistics }
        { Adjust score for a small win }
        { Zero out the point to match }
        { Display the message 'Winner!...Shoot again' }
        { Cause an update of the display }
        { Toggle state to needToSetPoint for next time }
      end

      { Else we throw again--state and point to match stay the same }
    else
      self.fStatsObj.IncFrequency(throwResult);
end;  { TCrapsPlayer.Play }
```

The method is complicated, but all it really does is capture the logic of craps, embodying the rules.

As we've seen, the player object is a state machine that can be in one of two states: *needToSetPoint* or *needToMatchPoint*. The *Play* method code uses *case* and *if* statements to handle the conditions under which the player object gets a transition from one state to the other. As long as the transition conditions aren't met, the player stays in the same state and keeps rolling the dice.

The *Play* method toggles from one state to the other and handles special cases:

- When a player's turn ends
- When a player runs out of money and must leave the game to the other players

A player ends his or her turn when he or she craps out, either by rolling a 2, 3, or 12 in State 1 (*needToSetPoint*) or by rolling a 7 in State 2 (*needToMatchPoint*). When that happens, either there are no more players and we announce the last player as the winner—by calling the utility procedure *AnnounceWinnerDialog*—or there are other players and we ask the game object to set up the next shooter.

A player leaves the game when he or she goes broke, as determined by function *Broke*, which is embedded in *Play*. *Broke* checks for a 0 or a negative score. If *Broke* finds either, it calls a utility procedure, *RemovePlayerDialog*, to alert the user and then asks the game object to remove the player object from its player list.

You can see the code for *AnnounceWinnerDialog* and *RemovePlayerDialog*, and the rest of *TCrapsPlayer*, on the code disk.

Last Thoughts on Crapgame

I originally designed Crapgame as a collection of cooperating objects without much of a Macintosh interface. Then I developed more of a Mac interface for it. Next, I abstracted the basic classes of PicoApp out of Crapgame and set them up on their own as an application framework. Finally, I wrote the version of Crapgame presented here—as a PicoApplication built on top of the PicoApp framework.

The game is relatively simple as applications go, yet it is large enough to illustrate a good many things about Mac programming, about writing applications with lots of objects, and about using application frameworks. Of course, now a great deal of Crapgame's original design is found in PicoApp instead of in the game application derived from PicoApp.

Under the heading of Macintosh programming, you can see in Crapgame

- Dialog boxes and alert boxes, including user items such as the dice in the introductory dialog box
- Event handling
- Menus
- Windows
- MultiFinder programming
- Desk accessory handling
- Schemes for functions like Undo, cursor handling, and so on
- And a good deal more

Under the heading of programming applications with objects, you can see a wide variety of object classes for

- Running a Mac application
- Operating a window
- Recording and displaying statistics
- Simulating buttons, dice, and other real-world objects
- Managing memory
- Handling events
- Managing displays
- Generating random numbers
- And more

And the objects in Crapgame and PicoApp (and PicoSketch) demonstrate a variety of ways to make objects interact. Some objects send other objects messages; sometimes the messaging is two-way. Some objects contain other objects in instance variables. Some objects create other objects.

Under the heading of working with application frameworks such as MacApp, TCL, and PicoApp, you can see

- The nature and structure of an application framework
- How the classes of a framework interact
- How to subclass and override parts of a framework
- How to build your own classes on top of the framework's
- How to gain leverage from all that the framework does for you
- How to use resources in a PicoApplication

To get the most from the wealth of information in PicoApp and our two PicoApplications, I recommend that you study the code thoroughly and try writing your own small PicoApplications.

As you study the code, you'll learn some techniques, see other or better ways to do things, and, no doubt, find some bugs, design flaws, and weak approaches. As you write your own PicoApplications, you'll come to appreciate the great expressive power of objects, the ease and efficiency of component reuse, and the benefits of using application frameworks.

Summary

Earlier chapters in Part 2 went a bit far afield from the simple game program we started out to build. In this chapter, we returned to Crapgame. In particular, we used the "working" parts of Crapgame to demonstrate the idea of modeling the real world with software objects—button subclasses, our die class, our random number–generator class, and our player class.

Projects

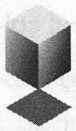
- Expand on Crapgame to give it more features, better performance, and anything else you think it needs.
- Expand on PicoSketch to turn it into a fairly powerful painting or drawing program, à la MacPaint or MacDraw.

PART

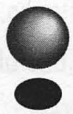
3

POLYMORPHIC SOFTWARE COMPONENTS

Part 3 will broaden and deepen your understanding of the object-oriented programming concepts discussed in Part 1 and exemplified in Part 2, especially of polymorphism.



Chapters in this part of the book take up the topic of software components, what Brad Cox (1986a) calls "software ICs." Like the hardware engineer's chips and integrated circuits (ICs), software components are designed to be general, robust, reliable, and easy to take off the shelf and drop into a new application. They are modular, plug-in components such as linked lists, trees, stacks, streams, filters, search and sort modules, and even higher-level constructs such as spelling checkers, spreadsheet cells, graphing modules, and text editors. Cox and others predict a software component industry in the near future, with some companies as component suppliers and others as component consumers.



We'll begin by considering, in Chapter 18, how to think about data structures like lists in a much more object-oriented way. We'll explore three different list models: traditional linked lists whose nodes are record types with object-type data fields (call this Model 1); linked lists whose nodes are themselves objects, linked to other nodes by object references (Model 2); and array-based lists whose elements are object types (Model 3). The Model 3 list will be the focus of most of our attention throughout Part 3, but we'll implement enough of the Model 2 list to learn how to work with object-oriented linked structures.

In Chapter 19, we'll take up several key design issues for list components: generality, safety, and reusability. Then, in Chapter 20, we'll partially implement a highly reusable linked list software component based on the second model. In Chapter 21, we'll implement another list component, this one based on the third model and using a dynamically allocated array as its underpinning. Chapters 20 and 21 leave a few special problems for us to take up in Chapters 22 and 23, where we'll deal with searching and traversing a general list and complete the remaining Model 3 list methods. In Chapter 24 we'll use the list as the basis for stack and queue components and then develop a really general collection class. Finally, in Chapter 25, we'll develop a binary search tree component based on the ideas used for the Model 2 list in Chapter 20. We'll also generalize "linkability" so that the same objects can be used in lists, stacks, queues, trees, and other data structures. And we'll consider several other useful components, including a dynamic array class, input/output streams, and filters.

Taken together, these chapters are a pretty thoroughgoing tour of the techniques of object-oriented data structures, reusable software components, and polymorphism.

18

LIST OBJECTS AND LISTS OF OBJECTS

Lists—and other structures derived from or related to lists—are a mainstay of programming. In this chapter, we'll look at the relationship between objects and lists:

- Standard linked lists with object-type instance variables in their record-type nodes
- Linked lists of object-type nodes rather than of record-type nodes
- "Linkability" as an inheritable trait of objects
- List manager objects
- The problem of searching lists (especially polymorphic ones)

We'll explore several strategies for building lists of and with objects. In particular, we'll develop classes *TLinkable* and *TDoublyLinkable*. These classes make it possible to create singly linked and doubly linked lists of any objects descended from the classes.

Then we'll work out a higher-level list abstraction: a manager object that encapsulates a linked list, hiding its details behind a layer of more abstract methods. Such a list lets you think at a higher level of abstraction—in terms of items rather than nodes and links. You add items, remove items, and perform other high-level actions on the list as a whole.

Finally, when a list is encapsulated and polymorphic, certain problems arise. We'll look at the problem of finding data in such a list, based on a key.

Let's begin our exploration of object-oriented data structures that will continue throughout Part 3. In this chapter, we'll lay the groundwork, developing a complete, highly general, but very basic, list software component and dealing with a number of problems associated with polymorphism as we go. We'll extend that work in later chapters to cover more data structures, such as stacks, queues, and trees.

Software Components

A software component is a highly reusable object that can be plugged into any application with a minimum of difficulty. It happens that OOP is particularly useful in building such components, largely because of object encapsulation and the reusability of classes. And polymorphism makes it possible to produce highly general components, such as general lists that can hold multiple types of items simultaneously. Our focus in Part 3 is on the design and implementation of such components. We'll design two list components.

To begin development, we need first to explore how lists are different in an object-oriented environment.

Lists

Lists are among the most fundamental of programming structures. Abstractly, a list is a sequence of items, possibly containing no items. You can add or insert items in the list, remove items, traverse the list to visit each item, and so on. Loosely speaking, an array is a list. But more complex "linked list" structures can be built, based on either pointers or arrays. We'll look at both pointer-based lists and array-based lists.

Pointer-Based Linked Lists

In their simplest form, pointer-based linked lists consist of a number of nodes linked together, either singly (forward links only) or doubly (forward and backward links). Figure 18-1 shows a singly linked list schematically. Each node contains

- Some data
- One or more pointers to other nodes (the "links")

On the Macintosh, of course, we use handles instead of pointers. If we didn't, list items would get lost when the Mac's Memory Manager shuffled blocks of storage around in RAM.

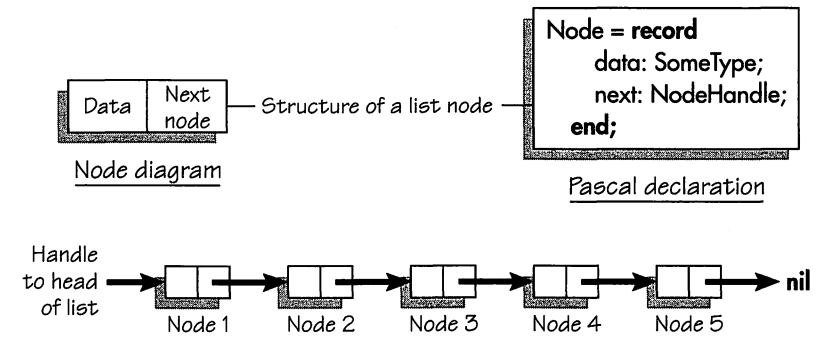


Figure 18-1.
Pointer-based singly linked list.

Linked list declarations typically look like this:

```
type
  NodePtr = ^Node;                { Indirect reference }
  NodeHandle = ^NodePtr;          { Doubly indirect reference }

  Node = record
      data: SomeType;             { One or more fields }
      :
      nextNode: NodeHandle;       { Link to next node }
    end;  { Record Node }
```

Note that *Node* is a record type.

To access a particular node, you usually begin at the head of the list and traverse the list from node to node, using code something like this:

```
var
  currentNode: NodeHandle;
:
currentNode := listHead;          { listHead is a NodeHandle, too }
while currentNode <> nil do
  begin
    { Do something with current node--display, for example }
    { Then move to the next node in the list }
    currentNode := currentNode^^.nextNode;
  end;  { while }
```

This code will move down the list, operating on each node, until the code "runs off the end" of the list. (The last item's *nextNode* handle will be *nil*.)

Objects in Lists

Linked list nodes can hold any sort of data—as fields in the node record. In this section, we'll look at two ways of building lists in which the data consists of objects.

Objects as Node Data

The most obvious way to make a linked list of objects is to put objects in the list nodes as their data:

```
type
  Node = record
      data: TSomething;           { Object type as data }
      :
    end;  { Record Node }
```

where *TSomething* is an object type (the name of some class). For an example, see the list of *TButton* objects on the next page.

```
type
  Node = record
      button: TButton;              { Button as a data field }
      nextNode: NodeHandle;
    end;  { Record Node }
```

You might use such a list in a program that needs to send events to a number of button objects one after another (as *TEvtHandler* methods do). Polymorphically, you can put any subclass of *TButton* into the *button* field in such a node.

> **NOTE:** *We're discussing some hypothetical button class, not necessarily the button class we developed in Chapter 4 and redeveloped as a descendant of* TPicoView *in Part 2.*

It's possible, in fact, to use even a very general object type as the data in a node—even *TObject*—but not without complications, as we'll see later.

Linkable Objects

You can use any object type as the data component of a traditional linked list node, and such data components can also be polymorphic.

But remember that objects are related to records, in a way, and that, like traditional records, objects are dynamically allocated blocks in the heap to which we have a handle or reference. Also, an object can contain fields that are references to other objects, including objects of its own type.

So why not eliminate the middleman and use a linked list of objects themselves rather than a linked list of nodes? Figure 18-2 shows such a list.

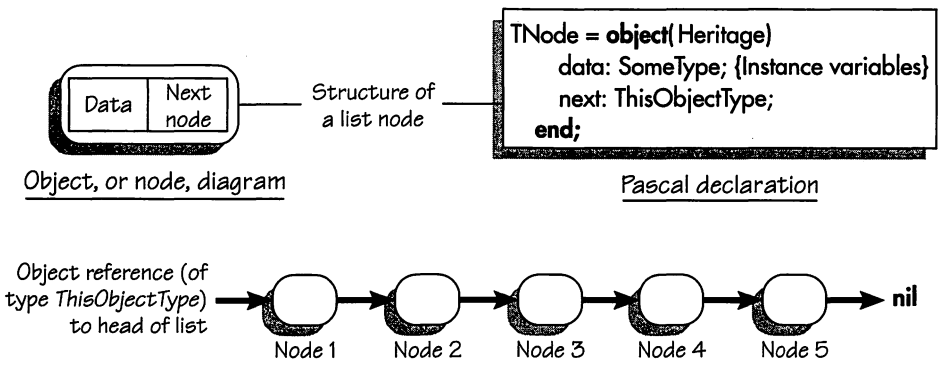


Figure 18-2.
Linked list of objects.

We've already seen this idea in *TEvtHandler* objects. Here's the declaration for a class of button objects that can be linked:

```
type
  TButton = object(TObject)
      { Button fields... }
      fNextButton: TButton;          { Link to next in list }
      { and button methods... }
      procedure TButton.SetNext(button: TButton);
      function NextButtonOf: TButton;
    end;  { Class TButton--with links }
```

The *fNextButton* field can contain a reference to the next button in the list. To link the current button to the next one, send the current button a SetNext message, passing it a reference to the next button. The list itself (its head) is a variable of type *TButton*—a reference to the first button in the list:

```
var
  theList: TButton;                  { First button }
```

To traverse the list, use the *NextButtonOf* method:

```
var
  currentButton: TButton;
:
currentButton := theList;            { Refers to the first button }
while currentButton <> nil do
  begin
    { Do something with the current button }
    currentButton := currentButton.NextButtonOf;  { Move to next button }
  end;  { while }
```

One key difference between this code and the similar code for ordinary linked-list nodes we've looked at is that this "linkable" version has no explicit pointer or handle dereferencing—no use of double carets (^^). Because the next pointer of a button node is an object reference (*TButton*), the implicit dereference is done for you by the compiler. The code for linkables otherwise greatly resembles the linked list code you're familiar with.

Linkability

So far, linkability is a characteristic we've built right into our button objects. To link other kinds of objects—say, windows or documents or game players—we'd have to build the same capability into each of those classes.

You should see it coming by now. Given a capability we'd like lots of different object types to share, the natural solution is to build the special ability into a separate class. We'll design an abstract ancestor class called *TLinkable* that carries the linkability characteristic. Then any other class can inherit the ability, just as all sorts of classes now inherit from *TObject* the ability to free themselves. Figure 18-3 diagrams the relationship between *TButton* and its new ancestor, *TLinkable*.

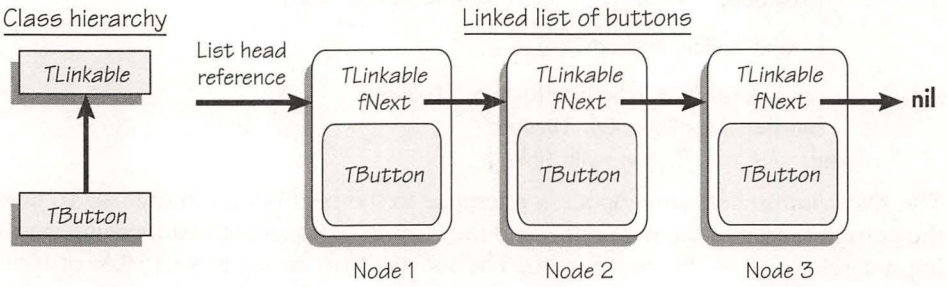


Figure 18-3.
Linkability. TButton *'s ancestor provides the linkability.* TButton *provides the button functionality.* TLinkable *has an instance variable* fNext: TLinkable.

So that this new ability doesn't interfere with objects inheriting from *TObject*, class *TLinkable* will list *TObject* as its ancestor. As we learned to do in Chapter 8, we're inserting an abstract capability into the hierarchy at a high level, where it's easily accessible to any lower class. Figure 18-4 illustrates the new hierarchical relationships.

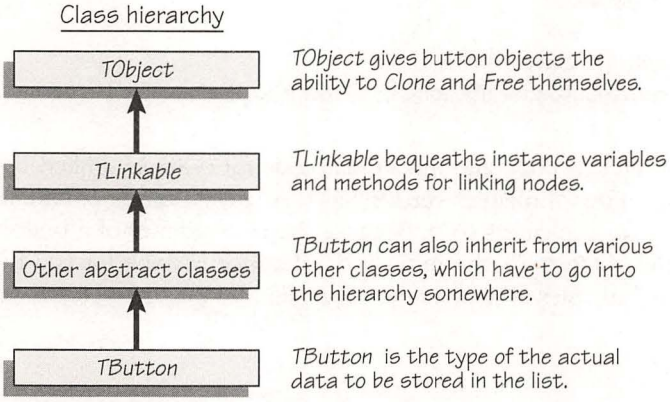


Figure 18-4.
A hierarchy of linkables.

Class TLinkable

TLinkable is small and simple. To let its descendants participate in singly linked lists of objects, it uses one instance variable and two methods. (We'll add a few more useful things later.) Here's the declaration for *TLinkable*:

```
type
  TLinkable = object(TObject)

      fNext: TLinkable;

      procedure TLinkable.SetNext(obj: TLinkable);
      function TLinkable.NextOf: TLinkable;
    end;  { Class TLinkable }
```

Any class inheriting from *TLinkable* gains the ability to use its inherited *SetNext* method to install a reference to some other *TLinkable* object in its own inherited *fNext* field. Such a class can also report what *TLinkable* object it's linked to by means of the *NextOf* function. *TLinkable.NextOf* can be used to traverse a list of *TLinkable* objects.

The other objects in such a list can be any of type *TLinkable*—objects of any classes that also inherit from *TLinkable*—hence, this kind of list is polymorphic (and subject to some special considerations, which we'll go into shortly). Figure 18-5 shows a polymorphic linked list containing instances of several classes.

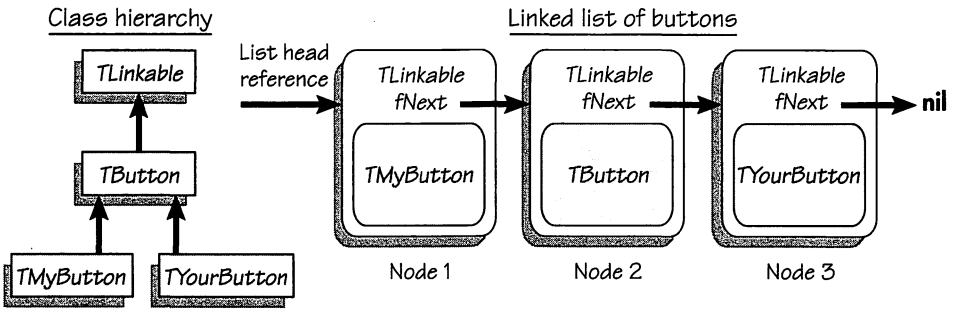


Figure 18-5.
Polymorphic linkables. TMyButton *and* TYourButton *descend from* TButton, *so they can be assigned to* TButton *or* TLinkable *variables (types above them in the hierarchy).*

NOTE: *Look again at PicoApp's class* TEvtHandler. *The mechanism it uses to link event-handler objects is the same one we use here.*

TDoublyLinkable

We can also easily set up a class for making doubly linked lists. Figure 18-6 illustrates such a list.

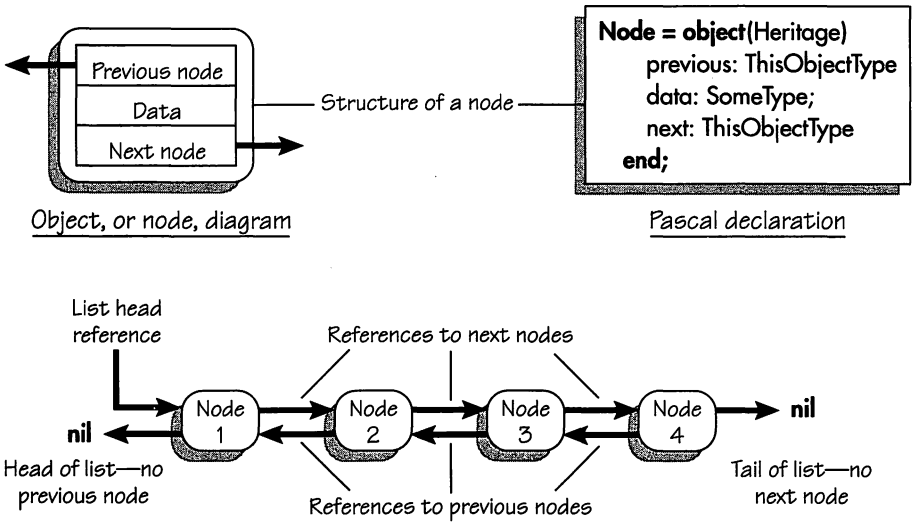


Figure 18-6.
Doubly linked objects.

A descendant of *TLinkable*, *TDoublyLinkable* adds another field and two more methods:

```
type
  TDoublyLinkable = object(TLinkable)

      { Inherits TLinkable's instance variable }
      fPrevious: TDoublyLinkable;

      { Inherits TLinkable's methods }
      procedure TDoublyLinkable.SetPrev(obj: TDoublyLinkable);
      function TDoublyLinkable.PreviousOf: TDoublyLinkable;
    end;  { Class TDoublyLinkable }
```

Any object of type *TDoublyLinkable* is also of type *TLinkable*.

TLinkable Methods

Here's the code for *TLinkable*'s (and *TDoublyLinkable*'s) two methods:

```
procedure TLinkable.SetNext(obj: TLinkable);
begin
  self.fNext := obj;
end;  { TLinkable.SetNext }
```

```
function TLinkable.NextOf: TLinkable;
begin
  NextOf := self.fNext;
end;  { TLinkable.NextOf }
```

So Far

In rapid succession, we've looked at three progressively more object-oriented ways to make lists:

- Using an ordinary linked list whose record-type nodes contain objects in their instance variables and whose "next" links are ordinary Macintosh handles (Nodes are records, and links are handles.)
- Using custom linkable objects that themselves are nodes and that contain links to similar nodes in the form of object references, not ordinary handles (Nodes are objects, and links are object references.)
- Using an abstract class of "linkables" from which any class can inherit the ability for its objects to be nodes in linked lists of similar linkable objects (Nodes are objects, and links are object references.)

All of these approaches are perfectly useful and essentially equivalent from the point of view of the linked list abstraction. But it's the last one, using *TLinkable*, that we'll pursue further.

A *TLinkable* Example

Let's develop a quick example using class *TLinkable*. We'll stick with our linked buttons example.

Suppose that we want to build a linked list of buttons. The list might be attached to a particular window (somewhat like a dialog box item list), or we might, with some adjustments, incorporate it into PicoApp as part of a view. We can then traverse the list, asking the buttons to display themselves, asking them whether a particular event belongs to them, and so on. This kind of traversal technique to send the same message to all objects in a list is a common OOP activity.

Because buttons are *TEvtHandlers*, we can add *TLinkable* to the hierarchy above *TEvtHandler*, making all event handlers potentially linkable. *TButton* thus inherits *TLinkable*'s instance variable and its methods through *TEvtHandler*.

Now let's build the list of buttons:

```
var
  theList: TLinkable;     { Head of list }
  aButton: TButton;       { Dummy for making button objects }
⋮
aButton := NewButton(title, active, r);
theList := aButton;
aButton.SetNext(nil);
```

So far, the list contains one button. We've set its *fNext* field (which, remember, it inherits from *TLinkable*) to *nil* by means of *SetNext* to show that it's the last button in the list at this point.

We have to decide where to add new buttons to the list: at the front (head) of the list, or at the tail. We have a reference to the object at the head, and the order of the buttons in the list is unimportant for this application, so we'll just add new buttons at the head. Figure 18-7 shows how this works.

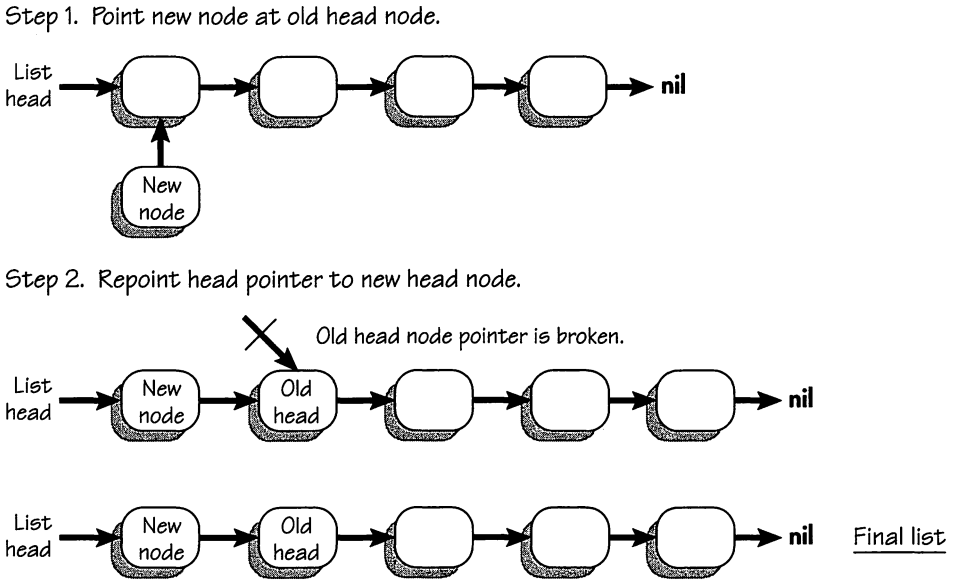


Figure 18-7.
Adding a node at the head of a simple linked list.

Now we'll add several new buttons, each in front of the previous one added:

```
var
  i: Integer;
:
for i := 1 to 9 do
  begin
    aButton := NewButton(title, active, r);
    aButton.SetNext(theList);          { Point to head of list }
    theList := aButton;                { Becomes new head }
  end;  { for }
```

Now the list holds 10 buttons. On each pass through the *for* loop, we created a new button, set the new button's *fNext* field (using *SetNext*) to refer to the former head

object in the list, and made the head of the list refer to the new button instead of the old. The button at the head of the list is the last one the program added.

Next, let's ask all buttons in the list to display themselves. We rely on the fact that all *TButton* objects have *Display* methods, but at first we forget that *TLinkable* objects don't have *Display* methods:

```
var
  currentLinkable: TLinkable;
:
currentLinkable := theList;              { Start with head }
while currentLinkable <> nil do
  begin
    currentLinkable.Display;             { Oops! }
    currentLinkable := currentLinkable.NextOf;
  end;  { while }
```

Why does this code produce a compiler error at the italicized line? Because *currentLinkable* is of type *TLinkable*, and *TLinkable* doesn't have a *Display* method. You and I know that these linkable objects are buttons, but the compiler doesn't.

What to do? Here are two very different approaches, briefly sketched. We'll discuss the problem of polymorphism in lists shortly.

Promotion by Typecasting

Our first solution involves typecasting the linkable object to a *TButton* before sending the Display message. This works because *TButton* is a subclass of *TLinkable* and any ancestor can be typecast to any of its descendant types and because *TButton* objects do have a *Display* method. Figure 18-8 shows the promotion process.

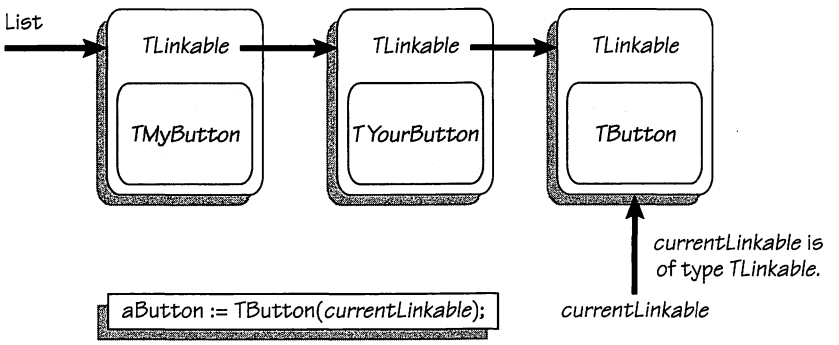


Figure 18-8.
Promoting an object. We know that currentLinkable *really contains a* TButton *object, but we have to promote the* TLinkable *type of* currentLinkable *to the* TButton *we know is there. The typecast* TButton(currentLinkable) *does the job.*

Here's the code that uses the typecast:

```
var
  currentLinkable: TLinkable;
  currentButton: TButton;              { Extra utility variable }
⋮
currentLinkable := theList;            { Start with head }
while currentLinkable <> nil do
  begin
    currentButton := TButton(currentLinkable);  { Typecast }
    currentButton.Display;             { This works }
    currentLinkable := currentLinkable.NextOf;
  end;  { while }
```

We could abbreviate that code a bit, as follows:

```
TButton(currentLinkable).Display;
```

This combines the typecast with the message send, all in one line of code. We eliminate any need for the extra *currentButton* variable. By the way, because the list will contain only buttons anyway and because *TButton is-a TLinkable*, we can make the whole list of type *TButton* in the first place:

```
var
  theList: TButton;                    { Has Linkable methods, recall }
  currentButton: TButton;
⋮
```

We'll call *TButton* the "base class" of the list. By defining it as the base class, we limit what we can put into the list to button objects, just as we limited our parts bin in Chapter 6 to miscellaneous parts by declaring *TMiscPart* as the base class. We still have to make the typecast, though, because the *NextOf* function returns a *TLinkable*, even though the object is certainly a button. If we don't typecast, the compiler will return an error:

```
currentButton := theList;
while currentButton <> nil do
  begin
    currentButton.Display;
    currentButton := currentButton.NextOf;      { This won't work }
  end;  { while }
```

Instead, we could make the typecast:

```
currentButton := TButton(currentButton.NextOf);  { This will work }
```

Adding a *Display* Method to *TLinkable*

The simplest approach of all is to add a *Display* method to class *TLinkable*. It would be a stub, of course, because we'll never instantiate *TLinkable* itself and so don't need to display it. Subclasses of *TLinkable* must override *Display* to provide their

own display code. Now our problem code on page 442 will work fine without typecasting or other tricks. At runtime, the correct button's *Display* method will be called.

Our first solution is good for heterogeneous (polymorphic) lists—for a list that mixes buttons, scrollbars, and text fields, say. But to simplify things, our second solution gives *TLinkable* a *Display* stub. We can be sure that almost anything we can put in a list will need to be displayed at some point. And we can expect almost any sort of object to be able to display itself in some fashion, so let's add *Display* to *TObject*, as we added *TypeOf* in Chapter 8. True, some objects don't have a meaningful display capacity, but the vast majority do. Now the *Display* method in a *TLinkable* descendant will be an override of *TObject.Display*. This won't work for every conceivable method, of course. We can't predict what methods will be needed, and certainly *TLinkable* (or *TObject*) can't stub every method that every possible *TLinkable* subclass might need. This works for *Display* because *Display* is such a common operation.

TLinkable and Base Classes

TLinkable permits us to string diverse types together in one list, but using it is really not such an undisciplined proposition. Let's look at what we need to do.

Establishing a Base Class for the List

TLinkable provides linkability to objects, but it isn't the real base class of the list. The list, after all, is a list of buttons or players or integers, and it's those we want to work with. Consider the two mail hierarchies in Figure 18-9.

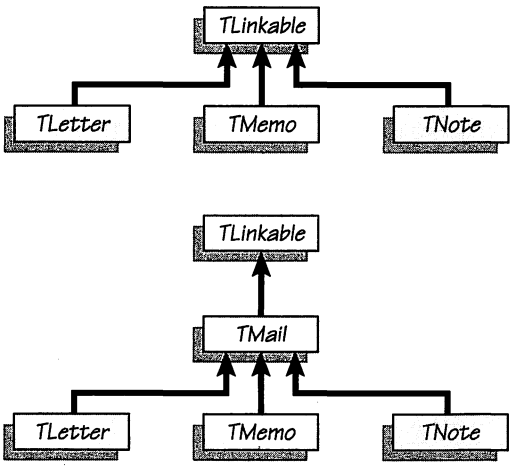


Figure 18-9.
Two mail hierarchies. TLinkable *is a poor base class because it lacks mail-handling methods.* TMail *is a better base class because it still inherits linkability but it also supplies the mail-handling methods needed by its descendants.*

The second hierarchy's intermediate class, *TMail*, should be the base class of the list. We're storing mail, not nodes, and *TLinkable* has no mail-handling methods to call. A *TLinkable* node can't tell you the date of a piece of mail stored in it or who the sender is or what the subject is. Class *TLinkable* has instance variables and methods pertaining only to linkability (and now display). Its subclasses supply all other functionality.

Suppose we use a list with a base class *TLinkable* and that *TMail* and its descendants have a method *DateOf* that returns the date on which the piece of mail was sent. If we traverse the list to some *TLinkable* node and make a call like this:

```
var
  currentNode: TLinkable;
  date: DateType;
:
currentNode := ...                    { Move currentNode to some node }
date := currentNode.DateOf;           { Send node a DateOf message }
```

the DateOf message will result in a compiler error because the compiler looks at the type of *currentNode*, sees that it's a *TLinkable*, and can't find a *DateOf* method for *TLinkable*. Never mind that it's a polymorphic node predictably containing a *TMail* object at runtime.

If we use code like this, every class actually stored in the list must have a *DateOf* method. There are two ways we can achieve that:

- We can use a variety of classes if they all descend from some ancestor intermediate between them and *TLinkable*. The intermediate class—*TMail* in this case—supplies the method stubs the classes need, such as *DateOf* (even if the subclasses override them). Usually, the lower this intermediate class is in the hierarchy, the better.
- All objects stored in the list can be of exactly the same type—all *TLetter*, say.

Either way establishes a true base class that we can use in working with the nodes. We'll still have to typecast the return values of *TLinkable*'s methods, but there will be no ambiguity about what to cast to.

Even if our list of pieces of mail contains many different *TMail* subclasses, we can still use the *TLinkable* descendant *TMail* as the base class. It has all the methods that the classes need. And if we want, say, a list of integer objects, we can use *TInteger* as the base class.

Choosing the Right Base Class

The base class is usually the lowest class in a hierarchy that is still above all classes a list must contain and that has the methods and instance variables the objects in the list will need. The base class is usually an abstract class.

The ideal base class has the right methods, even if many or all are stubs. But later, in designing a reusable list component, we'll look at more general lists in which

the base class doesn't have all the right methods but we use it anyway for its generality and in which we find alternative ways to manipulate the list despite our choice of base class. In one case, the base class will be *TLinkable*'s ambidextrous child, *TDoublyLinkable*. In another, the base class will be an immediate descendant of *TObject* called *TLockable*. We want our general list to be as polymorphic as possible.

Highly Polymorphic Lists

Lists based on polymorphic classes are, of course, polymorphic. Different nodes in the list can be of different classes. The list is a collection of diverse types.

But even a highly polymorphic list must have a base class, just as a polymorphic array of *TApples*, *TOranges*, and *TPomegranates* must have a base type that is some superclass of all those types, such as *TFruit*. It's possible to use even something as general as *TLinkable* or even *TObject* as the base type of a list. However, when we put objects belonging to subclasses into such a list, we have to use typecasting to get at them again, as in our first solution to the problem code on page 442. Because a base class like *TLinkable* or *TObject* doesn't have the necessary methods, we must typecast the nodes to a descendant type before we can call their methods.

Thus, the more general our base class, the more likely that we'll have to typecast to get at the objects stored in the list.

What if all the objects are of different types—but happen to have all the correct methods? True, no base class is established, and there will be no way to typecast. We can certainly send the same message to each node, but list traversal will turn out to be a problem—how can we advance our node pointer if it has to point to objects of varying types?

Polymorphism becomes even trickier when the list is itself encapsulated within another object. That's our next topic.

List Objects

So far, we've been looking at lists of objects, taking a low-level view in which what we see are nodes and the links connecting them and in which we do every painstaking little linkup ourselves.

But there's a high-level way to look at lists. Abstractly, a list is a black box object into which we put things and take them back out. Figure 18-10 on the next page shows the structure of a list object.

We're speaking of the list as an entity in itself—an object. The data of such an object consists of our low-level list of nodes and links, but we are above managing the details ourselves now. We let the list object do it. We simply add things and remove things, using *Add* and *Remove* methods. The list manages the pointer shuffling for us.

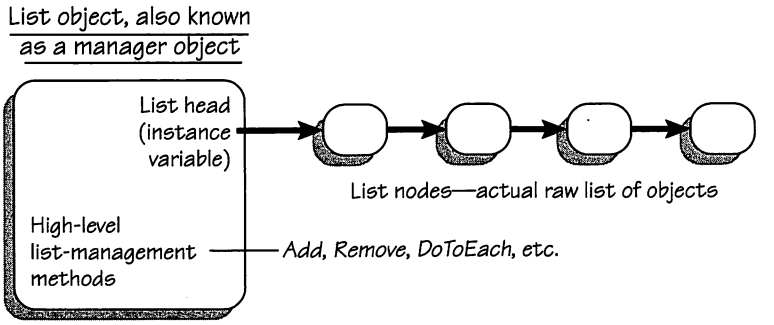


Figure 18-10.
List object, or manager object, and its underlying list.

Such an "encapsulated," or "managed," list has great advantages. It's a complete, tested, reliable abstract data type (ADT). It can even be polymorphic, so it's highly general in what it can contain—even multiple types at the same time. Hiding the details, it does a lot of grubby (and error-prone) work that we'd otherwise have to do over and over again ourselves. Having one or more lists with various capabilities in our software library would be handy indeed, and this is one aspect of the reusability for which proponents praise OOP.

Of course, we'll find that creating and working with such encapsulated list objects can be tricky. In particular, it's not easy to search such a list for data already in it, especially if the list is polymorphic. We'll get to some solutions for that as well.

A List Object Example

Here's a list class, a manager class that encapsulates a raw *TLinkable* list in just the high-level way we've been considering:

```
type
  TSimpleList = object(TObject)  { Could even be TLinkable }

    fList: TLinkable;

    procedure ISimpleList;
    procedure Add(obj: TLinkable);
    function Remove: TLinkable;
    function ItemOf: TLinkable;
  end;  { Class TSimpleList }
```

Now we need to sort out which of several possible list models to use. In the most basic sort of list, order is unimportant, so we merely add items at one place (say, the head or the tail) and likewise remove them from one place. Such a list is really either a stack or a queue, but that doesn't matter to the list abstraction.

Lists ordered in some special way, such as numerically or alphabetically, require an ability to insert new items and remove existing ones anywhere in the list. And that

ability implies the ability to search the list for the right place to insert or for the piece of data to remove. Even then, we'd like to shove such details down into the list object and simply say add, counting on the list to add in the right place.

For starters, let's keep it simple. We'll add and remove at the head (front) of the list. Later, we'll tackle searching and general insertion and deletion, particularly in polymorphic lists.

The *Add* method, then, adds a new list node at the head. The *fList* field is a reference to the head, so this will be easy. If we wanted to add at the tail, we'd need to maintain a reference to the last node in the list.

The *Remove* method extracts and returns a reference to the head node. If we want to inspect the head node without removing it, we can use *ItemOf* to inspect its value or call one of its methods.

TSimpleList methods

Here's code for the *ISimpleList*, *Add*, *Remove*, and *ItemOf* methods:

```
procedure TSimpleList.ISimpleList;
begin
  fList := nil;                    { Start out with an empty list }
end;  { TSimpleList.ISimpleList }

procedure TSimpleList.Add(obj: TLinkable);
begin
  if self.fList = nil then         { Start a new list }
    obj.SetNext(nil)
  else                             { Add to a nonempty list }
    obj.SetNext(self.fList);
  self.fList := obj;               { Assign object as new list head }
end;  { TSimpleList.Add }

function TSimpleList.Remove: TLinkable;
  var
    temp: TLinkable;
begin
  if self.fList <> nil then        { Remove only from a nonempty list }
    begin
      temp := self.fList;
      self.fList := self.fList.NextOf;
      Remove := temp;
    end;
end;  { TSimpleList.Remove }

function TSimpleList.ItemOf: TLinkable;
begin
  ItemOf := self.fList;            { Works even for an empty list }
end;  { TSimpleList.ItemOf }
```

TSimpleList example

First we create and build a list of type *TSimpleList*:

```
var
  theList: TSimpleList;
  aRef: TLinkable;                    { Auxiliary object reference }
  anObj, anotherObj, aThirdObj: TLinkable;  { Buttons, etc.}
  ⋮
  theList.Add(anObj);                 { Build list by adding nodes }
  theList.Add(anotherObj);
  theList.Add(aThirdObj);
  ⋮
  aRef := theList.ItemOf;             { Reference to head item }
  ⋮                                   {  = aThirdObj }
```

Suppose, though, that we subsequently call *ItemOf* and then *Free* to release the referenced item; then the *aRef* auxiliary reference is to an object that has been freed. We'll get a runtime error if we try to call any of *aRef*'s methods:

```
aRef := theList.Remove;
aRef.Free;                            { Remove old head }
⋮
if aRef <> nil then                   { Who knows what aRef points to? }
  aRef.Display;                       { Error }
```

Worse, now the list is broken. The head no longer exists. Any further attempts to traverse the list or access its head node will yield a runtime error. We'll look at a solution to these problems in Chapter 19. (The danger of multiple references to the same block of memory is no doubt familiar to any Pascal programmer who's worked with pointers. But, with objects, a fix is available.)

Consequences of our *TSimpleList* design

We now have a simple encapsulated list object. But in designing it, we had to make some important choices. For example, we wanted as general a list object as possible so that we could put all sorts of things in it and reuse it over and over again for different applications. That's why the base class of the underlying list is *TLinkable*, not, say, *TButton*. Likewise, *Add* and *ItemOf* use *TLinkable* for parameter and function return types.

Because of that decision, we can't resort to an impromptu base class choice as we did earlier, using *TButton* as the base class for a particular list and deciding to do that on the spur of the moment. The base class is already wired into the guts of our list object. Now we'd have to write a whole new list class to make the *fList* field of type *TButton*. That freedom—with its simplicity—is gone.

We'll feel the effects of our decision most strongly when we want to search the list for some piece of data, or if we want to implement a new *TLinkable*-based list with general insertion, or if we want to traverse the list sending some arbitrary message to

all objects in it. Those are common operations to want to perform on a list, but they'll be harder because we've made the list so general and encapsulated it. Inevitably, *TLinkable* objects won't have the methods we need for some purpose.

Because the underlying raw list of nodes and links is now encapsulated, we can't (at least we aren't supposed to) get at its internals directly. We're supposed to use the list at a distance, through its methods only. Doing so is really the only acceptable course if we're to be able to rely on the software we write. Directly accessing an object's data sooner or later leads to inconsistencies in the object's state and ultimately to corrupted data or program errors. In a very good cause, we've encapsulated ourselves into a sticky situation. After another detour or two, we'll see how sticky it is.

Constraining Lists

Is there some more precise way to constrain what type(s) an essentially polymorphic list can hold? Suppose, for example, that we have a list whose base class is *TMail* but that we'd like to store only *TLetter* objects in it. Ideally, we'd have some mechanism for screening out unwanted types, a mechanism more reliable than merely trusting to our good intentions.

Several mechanisms are available to us.

Recall from Chapter 8 the *TypeOf* method that we added to class *TObject*. Any descendant can override *TypeOf* to have it return the descendant's class name—for example, 'TLetter'. A descendant that doesn't override *TypeOf* simply inherits *TObject*'s version of the method, which, by default, returns the string 'TObject'.

The first approach we'll consider is appropriate for use with a list object, a single object that encapsulates an entire list, providing a high-level interface to it. Our solution is to parameterize the list when we initialize it, passing it a type-name string, such as 'TLetter', which the list stores in an instance variable. In the method used to add objects to the list, we

- Ask the type of the object-to-add: by calling *obj.TypeOf*
- Compare that with the type passed as the list's parameter ('TLetter'):

```
if obj.TypeOf = fTheType then
:
```

- Add the object only if it matches
- Signal some kind of error condition if it doesn't match

This approach does at runtime something like what type checking does at compile time. Consumers using the list are constrained to adding only objects of the approved type to the list.

But suppose we want to be able to add objects of multiple classes: *A*, *B*, and *C*—and no other types. One way to do that is to provide a method with which we can "register" a new type with the list. The list keeps a secondary list of acceptable type

names, and when we later try to add an object to the list, it first scans the type-name list to see whether the type of the object to be added is acceptable.

A second approach would use a hook method like those we embedded in PicoApp. The hook method, *TypeMatches*, would be called from the list object's addition method(s). To use it, we'd subclass the list class to override the hook method. Inside our overridden hook, we could directly code a type check for any types we wanted to allow:

```
function TSimpleList.TypeMatches(theType: Str30): Boolean;
  override;
begin
  TypeMatches := ((theType = 'TLetter') or (theType = 'TMemo'));
end;  { TSimpleList.TypeMatches }
```

Inside all of the list's *Add* methods, we'd screen the type of the current object-to-add. Here's how *TSimpleList.Add* would look:

```
procedure TSimpleList.Add(obj: TLinkable);
begin
  if self.TypeMatches(obj.TypeOf) then
    begin
      if self.fList = nil then
        obj.SetNext(nil)
      else
        obj.SetNext(self.fList);
      self.fList := obj;
    end
  else
    { Error-handling code }
end;  { TSimpleList.Add }
```

By default, *TSimpleList.TypeMatches* would return *true*, permitting all types to be added. We'd override it to supply our own tests.

Now suppose that we want to constrain the list so that it accepts all descendants of a base type—all *TMail* descendants, for instance. Neither scheme that we've looked at does that, but we can modify the hook method approach to do the job. Instead of asking an object whether it's a specific type, we ask it whether it's a member of the base class:

```
function TSimpleList.TypeMatches(obj: TLinkable): Boolean;
  override;
begin
  TypeMatches := Member(obj, TMail);
end;  { TSimpleList.TypeMatches }
```

This *TypeMatches* method returns *true* if *obj* is a member of any subclass of *TMail*. It will work, not only for all known *TMail* subclasses, but also for all future *TMail* subclasses added to the program. This is considerably more general.

Along with the concept of a base class, any of these techniques gives us a fair amount of control over what goes into our highly polymorphic data structures.

Array-Based Lists

In Chapter 21, we'll look at a different kind of list—one based on an array (dynamically allocated at that) instead of on nodes linked by pointers.

Searching a List

We really have two major categories of lists now: encapsulated and raw (unencapsulated) lists. And a list from either category can be polymorphic or not. If not polymorphic, the list in question holds one type of data, such as *TInteger*, and we have little difficulty in doing whatever we like with the data. If the list is polymorphic, we have the problem of what the underlying type in a given node really is and whether the base type has the methods we need.

Let's take a quick survey of searching under the four possible list categories: unencapsulated and not polymorphic, unencapsulated and polymorphic, encapsulated and not polymorphic, and encapsulated and polymorphic. Figure 18-11 shows a schematic view of these categories.

	Unencapsulated	Encapsulated
Nonpolymorphic	One type of data. Programmer accesses data directly.	One type of data. Programmer accesses data indirectly by means of list object methods.
Polymorphic	Multiple data types at the same time. Programmer accesses data directly.	Multiple data types at the same time. Programmer accesses data indirectly by means of list object methods.

Figure 18-11.
Four categories of linked object lists.

To search any kind of list, we need

- A key value of a type contained by the objects in the list
- The ability to move down the list, comparing the key with the data inside each object—called "traversing," "enumerating," or "iterating"

The search key is the value we want to find in the list. Of course, we might sometimes be looking for exactly that value, sometimes for any value greater than the key, sometimes for any value less than or equal to the key, and so on. Different searches require different combinations of relational tests.

Moving down a list is not hard. We've shown several examples using raw, unencapsulated lists, and it wouldn't be hard to build traversal into a method of an encapsulated list.

What is complicated, though, is comparing the key with the data in each object, at least in polymorphic lists, especially encapsulated ones. Let's see why by examining the situation in all of our list categories.

Searching an Unencapsulated, Nonpolymorphic List

We can always use any descendant of *TLinkable* as the base class of our unencapsulated (raw) list. Suppose that we have a list of *TInteger* objects, where *TInteger* is declared as

```
type
  TInteger = object(TLinkable)
      fTheInteger: Integer;
      procedure Put(i: Integer);
      function Get: Integer;
    end;  { Class TInteger }
```

Every node is a *TInteger* object, and, as *TLinkable* descendants, these objects all have the *TLinkable* methods needed for traversing the list:

```
var
  theList: TInteger;
  currentItem: TInteger;
  key: Integer;
  found: Boolean;
  ⋮
  key := 4;                          { Use a key value of 4 }
  currentItem := theList;            { Start at head }
  found := false;
  while (not found) and (currentItem <> nil) do
    begin
      if key = currentItem.Get then
        found := true
      else
        currentItem := currentItem.NextOf;
    end;  { while }
  { CurrentItem is now nil or a reference to the found item }
```

Because *key* is of the same type as that returned by *currentItem.Get*, comparing is easy. And we can be sure that all objects in the list have a *Get* method because the list contains only *TIntegers*. Because the list can contain nodes of only one type,

TInteger, the list isn't polymorphic and we don't have to do any typecasting to get at the stored objects.

Searching an Unencapsulated, Polymorphic List

Now suppose that the list is polymorphic, based on *TLinkable*, but still with only *TIntegers* stored:

```
var
  theList: TLinkable;
  currentItem: TLinkable;
  currentInteger: TInteger;                { We need a "pointer" to }
                                           {    traverse the list }
  key: Integer;
  found: Boolean;
  :
  key := 4;
  currentItem := theList;                  { Start at head }
  found := false;
  while (not found) and (currentItem <> nil) do
    begin
      currentInteger := TInteger(currentItem);  { Typecast so that we can call method}
      if key = currentInteger.Get then     { Here's the method call }
        found := true
      else
        { Not found, so move the pointer along to next node }
        currentItem := currentItem.NextOf;
    end;  { while }
  { CurrentItem is now nil or a reference to the found item }
```

We have to typecast the current node before sending it the Get message: *TLinkable* doesn't have a *Get* method.

And what if several types are stored in the same unencapsulated, polymorphic list? Then they must all have common properties: at least one key field that we can key the search on and a method to return the value of that field for comparison. If the base type of the list were *TMail*, for example, and we wanted to search a list full of various *TMail* descendants for date, all *TMail* descendants that were in the list would have to share a date field of the same type and a method to return the date, such as *DateOf*. If we wanted to search for something else, say, sender, all *TMail* descendants would need a sender field and a *SenderOf* method. This implies that *DateOf* or *SenderOf* would have to be a method of *TMail* itself. The main way we can tell what *TMail* descendant is in a given list node is to use the *Member* function.

```
if Member(TLetter, currentItem) then
  begin
    currentLetter := TLetter(currentItem);
    { Do something with currentLetter }
  end
else if Member(TMemo, currentItem) then
  begin
    currentMemo.TMemo(currentItem);
    :
```

But if *TMail* has the right key instance variable(s) and method(s), we don't need *Member* and we don't have to promote anything by typecasting. We can just send the *currentItem*, whatever it is, a message:

```
x := currentItem.Get;
```

Searching an Encapsulated, Nonpolymorphic List

An encapsulated, nonpolymorphic list declaration might go something like this:

```
type
  TNonPolyList = object(TObject)

      fList: TInteger;

      procedure INonPolyList;
      procedure Add(i: TInteger);
      :
      function Search(i: Integer): TInteger;
    end;  { Class TNonPolyList }
```

Such a list is confined to one type of data, *TInteger* objects, say, that has no descendants. We can simply write methods based on our knowledge of the underlying data type. Because *TInteger* is a descendant of *TLinkable*, it has the necessary methods for list traversing. Our *currentItem* pointer can be of type *TInteger*. Let's look at a *Search* function:

```
function TNonPolyList.Search(i: Integer): TInteger;
  var
    currentItem : TInteger;
    found : Boolean;
begin
  found := false;
  currentItem := self.fList;
  while (not found) and (currentItem <> nil) do
    begin
      if i = currentItem.Get then
        found := true
      else
        currentItem := currentItem.NextOf;
```

```
    end;  { while }
  { CurrentItem is nil or a reference to the found item }
  Search := currentItem;
end;  { TNonPolyList.Search }
```

Encapsulation is no problem here. The list "understands" the data it holds.

Searching an Encapsulated, Polymorphic List

Let's encapsulate a *TMail* list, perhaps to manage a mailbox for an e-mail system. The base type is *TMail*, a subclass of *TLinkable*, and the list might contain objects of type *TLetter*, *TMemo*, *TReport*, *TNote*, and *TEndorsement*—all descendants of *TMail* and thus also of *TLinkable*:

```
type
  TMailList = object(TObject)

      fList: TMail;

      procedure IMailList;
      procedure Add(m: TMail);
      procedure Remove;
      function ItemOf: TMail;
      function Search(date: DateType): TMail;
    end;  { Class TMailList }
```

All objects in the list must inherit instance variables and methods from *TMail* to retrieve the instance variable values.

To search the list, use a method with a key as parameter. The *Search* method returns an object reference of type *TMail*. Why *TMail* rather than *TLetter*, *TMemo*, or whatever? Because in a polymorphic list like this we can't "see" what's in a node. But we can send it messages, such as *DateOf*, in full confidence that the node can respond.

But suppose we want to search the list based on other keys, such as sender, and/or other conditions, such as "less than" or "greater than or equal to"? Obviously, the current *Search* method in our *TMailList* declaration is good only for date and for "equal to." Do we then need dozens of search methods, one for each combination of key type and search condition? That's one viable but bulky approach, of course. But is that the only way?

Remember that this is an encapsulated list. In a raw list, we could move a pointer along the list and do anything we liked with the node pointed to—using any search condition and any key (or any combination, such as "date greater than *x* and sender equal to *JBlow*"). But in an encapsulated list, we no longer have access to the list's internals. We have to rely on methods that know how to do what we want.

And what method knows how to do all of those things—every possible combination of key and condition we might ever dream up? Writing such a general and extensible method turns out to be hard. In Chapter 22 we'll look at several alternatives.

Summary

In this chapter we progressed through several levels of sophistication as we explored five approaches to building list data structures:

- Simple linked lists with record-type nodes containing object-type data in record fields, with ordinary handles forming the links
- Custom linked lists in which the nodes themselves are objects and the links are object references
- *TLinkable* objects, in which "linkability" becomes an inheritable trait, built into a single class from which many classes can inherit the ability
- Encapsulated lists, in which a list object hides and manages an underlying raw list, built with either *TLinkable* objects or an array

We also developed the idea of a list's base class, the class that defines the type of the list's nodes. And we explored the nature of base classes and how to choose one.

Finally, we looked at polymorphic lists and made a preliminary survey of the problems we face when a list is polymorphic, especially if it is also encapsulated in a list object. The main problem we've considered so far is searching the list for a key based on a conditional expression of some kind.

We left a number of these issues hanging for the next chapters, in which we'll develop a reusable software component—a list class so general that it can hold any subclass of *TLockable*. The class will be usable almost anywhere for almost any kind of data. And it will provide a strong basis for other useful classes, such as stacks and queues.

We'll explore several solutions to the most general list problems: searching and traversing a list to send arbitrary messages.

Projects

- Develop class *TShape*, which has subclasses *TCircle*, *TRectangle*, *TPolygon*, and so forth, all in a hierarchy. Then develop a raw, do-it-yourself linked list that can contain any object of type *TShape*. Outside the class declaration, write non-method utility procedures to fill the list with an arbitrary assortment of shapes, to traverse the list sending a Display message to each shape, and to traverse the list sending a Free message to each shape. (Be careful with this one.)
- Write a *TSimpleList*-style list object to contain *TInteger* objects only. Include methods that traverse the list sending a Display message to each object, traverse the list summing the integer values in the objects, and search the list for a key integer value. For fun, make your *TSimpleList* a descendant of *TLinkable*. Then write a raw list of *TSimpleList* objects. Write a non-method procedure to traverse this list-of-lists, summing the sums of the integer lists and reporting the result.

19

DESIGNING A GENERAL LIST SOFTWARE COMPONENT

In the last chapter, we explored several ways to implement object-oriented list data structures: linked lists of records containing objects, linked lists using objects as nodes, and list manager objects.

In this chapter, we'll focus on a particular abstract list model, one we'll develop further in two ways in coming chapters. Then we'll explore a number of design issues and make some broad decisions about the lists we plan to implement.

Among our concerns are

- Making a list general enough to hold all kinds of data
- Making a list safe to use
- Making a list highly reusable
- Endowing a list with "data knowledge"

We'll begin with a bit more about reusable software components, a topic we've taken up briefly before.

Reusable Software Components

When we promise to build a reusable list object, what we're really promising is a reusable software component—a piece of software general enough and reliable enough to use over and over again in many different kinds of programs.

Brad Cox (Cox 1986a), one of the chief proponents of using OOP to develop reusable software components, calls them "software ICs" (or "software integrated circuits"). Software ICs are the software equivalent of the many standardized chips and circuit boards that hardware engineers can take off the shelf and plug in to build more complex pieces of hardware. By embedding the right collection of chips into a board, the engineer can work at a high level of structure, far above the nitty-gritty details of specific circuits, gates, adders, and so forth.

A software component is a piece of self-contained software that performs a function and can be plugged into any application without serious modification. Examples include data structures such as lists, trees, queues, and stacks; sort and search modules; windowing packages on PCs; and buttons, scrollbars, and other controls.

Components can be written without OOP, but OOP components offer major advantages. OOP software components are extensible. If a component implemented as an object class doesn't quite meet your needs, you can subclass it to get something that does. OOP software components are often based on polymorphism, which means that they can handle nearly any kind of object-oriented data. And any data can be made object oriented.

When you use a software component (written by you or someone else), you should be able to rely on its robustness because it has a track record of testing and use in real programs. It's one part of your program that you shouldn't have to debug.

Sometimes you might have to arrange a suitable interface to the component, of course. If your application uses raw (non-object) data, you might need to wrap the component up in a manager object that can translate your raw data into objects. But you won't have to write your own list, tree, or stack, and you won't have to tinker with the component's inner workings.

You can keep a library of components, drawing on them as you need to. A first-class object library might have several different kinds of list components, for example. Some would be array based rather than pointer based. Some would be optimized for speed, some for space. Some might also be optimized for safety, with extra precautions built in (as we'll see), and others would sacrifice the added security for speed, space, and ease of use. Some list components would be singly linked, like *TSimpleList* in the last chapter, and others, like the *TLinkedList* we'll develop in the next chapter, would be doubly linked. A good component library should provide for the needs of the moment, with a component possessing just the right characteristics—a component close enough that you can get what you need by subclassing it.

Today Object Pascal doesn't have a first-class library of components, as some OOP languages (such as Smalltalk) do. MacApp, TCL, and, to some extent, PicoApp provide a library of powerful Macintosh user interface components (application objects, document objects, and so forth). But none of these libraries contains the general-purpose programming components such as data structures and data streams that it should. That kind of library remains to be built.

Is Object Pascal a good language for building reusable software components? Well, it's not bad. You can do a lot with it, as we'll see. But it could be much better. For one thing, Object Pascal lacks multiple inheritance. There are strong arguments against multiple inheritance, in which a subclass has two or more immediate ancestors and can inherit from all of them at once. For a language to support multiple inheritance requires a lot of overhead and the resolution of some serious difficulties. For example, what should the language do if two immediate ancestors have methods with the same name but different parameter profiles or different semantics? Multiple inheritance does sometimes have its uses, and if the mechanism for it in a language isn't overly complex, it might be a good tool to have.

But even more troublesome is Object Pascal's lack of real encapsulation. You can urge programmers to access an object's instance variables only through the interface of its methods. But programmers can easily refer directly to any instance variable from outside and can change its value—perhaps in an unorthodox or even dangerous way. Some languages allow a class to have both public and private parts. The private items can be accessed only from inside an instance of the class. Anything you don't want outsiders tampering with would be declared as private.

Perhaps a Pascal++ will come along someday, adding valuable tools like real encapsulation, and perhaps even multiple inheritance, to make building components, and libraries of components, easier.

Our List Model

In the next two chapters, we'll develop two different lists, both of which will be polymorphic and encapsulated. Here, though, we'll look at the high-level model on which both list implementations will be based.

List Abstractions

Computer scientists like to talk about lists, queues, trees, and other data structures in abstract terms. An abstract list, for example, is the concept of a list without any details of how it might be implemented. From the abstract point of view, a list is a linear collection of items, possibly ordered, possibly not. It has a head item, a tail item, and items in between. We can add new items to the list, remove existing items from it, get its length, see whether it's empty (or possibly full), and so on.

That's not the only possible list abstraction. For example, in the LISP programming language, a list consists of a "car" (head) plus a "cdr" (all the rest of the list besides the car).

We'll use the first abstraction, but note all the things we're not saying about the list. We don't say what its items are. We don't say what sort of receptacle the items are kept in. We don't discuss whether the items are linked in some way. And we don't say anything about the details of adding or deleting items. In other words, the same list abstraction might be implemented in a number of very different ways—some of which we saw in Chapter 18.

Our Particular List Abstraction

Without leaving the abstract level, let's now be a bit more precise about the list model we're going to use.

We'll call it a Mark List. A Mark List maintains a movable "mark," like a cursor or like the mark the Macintosh File Manager uses to pinpoint a particular location in a file. Unless the list is empty, the mark always marks one element of the list. It could be at the head element, at the tail element, or anywhere in between.

We'll provide ways to move the mark around in the list. This ability is important, because most actions in the list take place at the mark. Except for a few special cases, we always add items at the mark, remove the item currently at the mark, and so on.

To use the list, we first move the mark to the list element we want. Then we perform some action on that element. For example, to add an element before element number 10, we move the mark to the tenth element and then insert there.

Besides specific ways to move the mark, we'll provide a search capability so that we can locate items in the list. Searching will move the mark to the first item that matches the search criteria. And we'll provide ways to traverse the list, doing something useful to each element.

A Profile of the Mark List

Now let's see what fundamental operations the Mark List abstraction supports. Here's a pseudocode version of the list, outlining the class declarations we'll see in Chapters 20 and 21.

The class for a Mark List is

- A list (without specifying its implementation)
- Methods to
  - Initialize the list
    - Specify what class or classes the list can hold
    - Specify whether the list is "locked" or "unlocked"
  - Move a mark to any element of the list
  - Find an element containing a given key value
  - Report whether the list is empty
  - Report the list's current length
  - Report the current position of the mark
  - Report whether the mark is at a list end or off the end
  - Add data at the head, tail, or anywhere in between
  - Extract the data item currently at the mark
  - Extract data from the list head
  - Delete the data item currently at the mark

Peek at the data item currently at the mark
Display the whole list
Traverse the list, doing arbitrary things to each element
Clone the list
Free the list

We'll get around to breaking those specifications down into methods later. Right now we need to discuss a number of design issues.

Design Issues

Among the issues we want to think about before attempting either of our list implementations are these:

- *Generality.* We know that we'll use polymorphism to make the list general enough to hold any kind of data. That means we'll select a base class for the list, from which all objects to be stored in the list must descend. But what base class? And what are the consequences of our choice?
- *Safety.* Building a highly general data structure involves some risks. When we put data into the structure, will it be safe from corruption or misuse? In particular, we'll use dynamically allocated objects in the heap. There's considerable danger of inadvertently freeing an object in a list through some outside reference to the object. This would effectively destroy the list. Can we prevent such a catastrophe? Do we need to?
- *Reusability.* How easy will it be to integrate our list components with consuming programs? In particular, how well will our lists mesh with PicoApp, MacApp, or TCL?
- *Primitiveness.* Exactly what operations should the list include? Some would argue that only the most primitive operations should be included, the ones that absolutely require direct access to the inner state of the list. Nonprimitive operations can be built by combining various primitive operations. For example, extracting an element at the head of the list is merely a special case of extracting an element from anywhere in the list. We can achieve the same effect by moving the mark to the head and then extracting the element at the mark. But we can dispose of this issue right now. We'll include some nonprimitive operations. We're building practical software components designed to be directly useful, so we'll include all the operations we think useful enough, primitive or not.
- *Data Knowledge.* Another problem for a polymorphic, encapsulated list such as the ones we'll build is data knowledge. Data knowledge refers to the list's knowledge of types and other characteristics of a piece of its data, including what methods are available for manipulating the piece of data. The issue is where the data knowledge is and who's responsible for what.

In a really general data structure, such as the lists we'll design, there's a limit to what the structure can know about its data. The base class of the structure can have an unlimited number and variety of descendants, each with different instance variables and methods besides those inherited from the base class. A *TButton*, a *TInteger*, and a *TPlayer* have almost nothing in common, yet they could conceivably all be put into our Mark List at the same time. Our Mark List can't tell what's there, what types of fields the data has, or what methods are available.

The solution to this limitation is to require the consuming programmer to supply the data knowledge for tasks like searching, sorting, and inserting. The list provides a general framework. The client supplies the data knowledge. This is also a requirement in the Ada language's generics, for example, where the client must provide data types and even functions to define what, say, "=" means for some data type, such as complex numbers or salaried employees.

How can the consumer using an encapsulated component supply data knowledge to it? Among the approaches we'll explore are Pascal's functional parameters; an invention of my own called search key objects; another invention of mine called traversal action objects; and an idea borrowed from Objective-C, sequence classes.

Generality

We'll define a general data structure as one that can hold a variety of data types, perhaps all at the same time. We've already looked briefly at such structures, calling them containers, or collections. For examples, refer to the miscellaneous auto parts bin in Chapter 6 and the general collection class we'll develop in Chapter 24.

We could build our list on any of a number of base classes. We might base it on *TButton*, for instance, or *TPlayer*, or *TInteger*. But then the only data we could store in the list would be buttons, players, or integers. We'd have to write a new list for each new type of data we wanted to list. Doing that would be easy enough with a text editor: We'd make a copy of the source code for a *TButton* list, say, and then change all occurrences of the word *TButton* to the new type.

But we'd like something more useful, something we won't have to rebuild for each new data type. We'd like one list general enough to hold objects of type *TButton*, *TPlayer*, *TInteger*, *TInventoryItem*, or whatever—anything, so long as it's a subclass of whatever base class we build the list around. Such a general list is like an extension to the Pascal language. It's there; like *Writeln*, it's multipurpose; and it's easy.

Of course, there are some negatives too. Such a list is so general that discipline is relaxed. Type checking doesn't mean a lot when the structure can hold apples and planets at the same time (although we discussed *parameterizing* lists in the previous chapter). We can run into problems getting things back out of such a list. Once we hide a *TApple* in a *TDoublyLinkable* list, it's hard to tell what's in there—maybe it's a *TRattlesnake*.

Is It Generic?

It might be tempting to call our general list "generic." But would that be the same notion of generic that software engineer types mean when they talk about the Ada programming language, or even the "Son of Pascal" (Modula-2)?

As in Ada, we can instantiate a list object that will hold whatever type we have in mind—*TIntegers* today, *TApples* tomorrow. But in Object Pascal we can instantiate a list that can hold both integers and apples. Our list object is always generic—any instance can hold any type, anytime. In Ada, a generic "package" or a generic procedure is, like our classes, a template, from which you create instances. But any given instance of an Ada generic entity can handle one and only one type. You instantiate it for that type. In a way, although this is an oversimplification, Ada does for you what you used to do with your text editor. You write a package containing list code, say, identifying the type of data to go into the list as *element* or something equally vague. The compiler, in effect, makes a copy of the code, substituting, say, *integer* for every occurrence of *element.* (It's more complex and impressive than that, but that's the essence.)

Our list classes can be instantiated as is, resulting in list objects that can hold any type descended from the base class. But we'll also make it possible to subclass the list classes, overriding a method called *TypeMatches* that lets us constrain the list to any given type or selection of types.

By comparison with Ada's, our Object Pascal approach is big and happy and sloppy. We have to take considerable pains to control how the list is used, and, in the abstract, our all-embracing list looks rather easy to abuse. But in the context of a given application, we can certainly control what types go into a list, and we can guarantee that they all have the necessary methods.

In *Object-Oriented Software Construction* (Meyer 1988), Bertrand Meyer discusses genericity vs. inheritance in great detail; he is worth reading on this and many other OOP topics.

Base Classes for Our Lists

We'll build two different lists, so they might have different base classes. The base class, recall, is the class declared as the type of each list element. In Part 1, for example, we declared an array of *TFruit* capable of holding *TApples*, *TOranges*, and so on—any descendant of *TFruit. TFruit* was the array's base class. We discussed base classes at some length in Chapters 6 and 18.

TLinkedList will be implemented as a doubly linked list of objects. Its base class, then, will be *TDoublyLinkable.* All objects storable in this list must be descendants of *TDoublyLinkable*, directly or indirectly.

TList will be implemented inside a dynamically allocated array. Each node will be an element in the array. The array's elements don't need instance variables pointing to their next or previous elements in the list, so the base class doesn't need to be either *TDoublyLinkable* or *TLinkable*. We don't have to support the 8-byte overhead of those classes. In fact, we could, as in MacApp and TCL, implement the list with a base class of *TObject*. That's as general as you can get. But in the long run we'll decide to use a new class called *TLockable* as *TList*'s base class. That's because we'll decide to make it possible for the list to "lock" all of its elements so that they can't be freed through outside object references.

We're going to insert the same *TLockable* class into the *TDoublyLinkable* hierarchy, above *TLinkable*, so that *TLinkedList* can also lock its elements.

Implications of Our Choices

Figure 19-1 shows the positions of the base classes *TLockable* and *TDoublyLinkable* in the object hierarchy. Because *TLockable*, the base class for *TList*, is also in *TDoublyLinkable*'s ancestry, we need only one hierarchy.

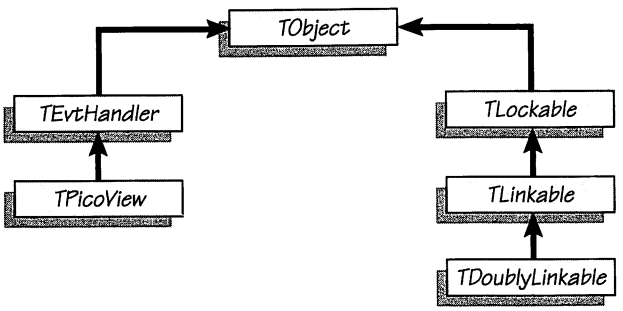


Figure 19-1.
TLockable *and* TDoublyLinkable *in the class hierarchy.*

This means that any subclass of *TDoublyLinkable* can be stored in either kind of list. But subclasses of *TLockable* that aren't also subclasses of *TDoublyLinkable* can be stored only in *TList*. We'll take up the issue of compatibility among different kinds of data structures in Chapter 25.

And we'll have more to say about the consequences of choosing these base classes later in this chapter, under the heading of reusability.

Using the Lists

To use the two lists, we must be sure that our data object classes are subclasses of *TLockable* (for *TList*) or *TDoublyLinkable* (for *TLinkedList*). Classes can intervene in the hierarchy between the base class and the class of our data—the data's class doesn't have to be directly descended from the base class. Indirectly will do.

Safety

A good software component should, if possible, protect its data. If a list's methods can be bypassed to access its innards, it won't be long before the list's data becomes corrupted or unreliable. We'll address two related safety issues: back doors into a list and access to a list's elements through outside object references.

Encapsulation

Encapsulation means that outsiders can't tinker with an object's innards. An encapsulated list should let consumers access it only through the interface of its methods.

As it stands now, our Mark List is vulnerable to misuse and abuse of its encapsulation, especially in its *TLinkedList* implementation. Of course, in Pascal the underlying list isn't really encapsulated. You can get at it and traverse it as a raw list any time you like, as in this *TLinkedList* example:

```
currentNode := aList.fList;  { Direct field access }
while currentNode <> nil do
  begin
    { Do something with currentNode }
    currentNode := currentNode.fNext;
  end;
```

This code directly accesses an instance variable, *fList*, of the object, which is, as we've noted, considered bad form. But Object Pascal allows it, and we can only urge consumers to be careful.

Back Doors into a List

We'll encounter other difficulties with the encapsulation of Mark List. Any method that returns a reference to a list node is potentially hazardous. A programmer willing to violate encapsulation could use such a method as a "back door" to the whole list, with some potential for corrupting the list's state. Several proposed methods of our doubly linked list *TLinkedList* would allow backdoor access. Specifically dangerous are methods that return references to nodes—we'll name them *Extract*, *ExtractHead*, *Peek*, and *NextItem*. In *TLinkedList*, each of these methods returns a *TDoublyLinkable* reference to a node in the list. This potentially provides access to the list's innards. Figure 19-2 on the next page illustrates getting into a list by means of an external reference and following previous and next links in order to access nodes. This isn't a problem for *TList*, however—list elements in *TList* don't have instance variables storing object references to the next or previous element. There's nothing to follow, although a programmer could access the list's underlying array.

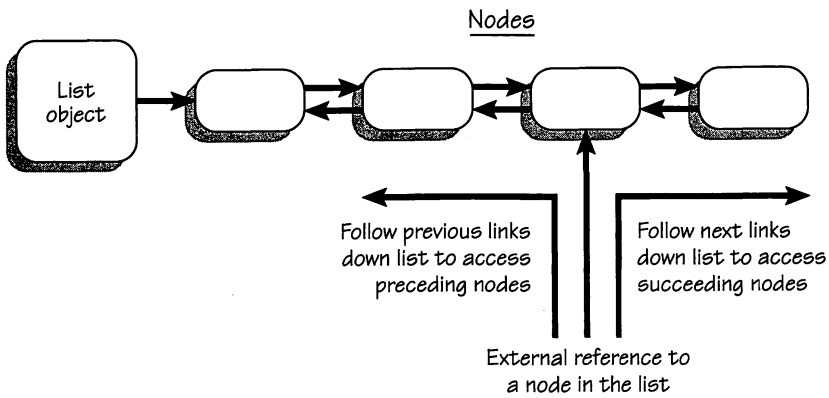


Figure 19-2.
Back doors to a list. Once you have access to a node, you can send it messages, such as Free.

Extract and *ExtractHead* are actually no problem in *TLinkedList* because they remove the node from the list before returning a reference to it. You can't access the list through the node. But *TLinkedList.Peek* returns a reference to a node still residing in the list. You could easily follow that node:

```
item := aList.Peek;  { Item refers to node at the mark }
item := item.NextOf;
```

or

```
item := item.PreviousOf;
```

In the long run, we could choose to dispense with *Peek*, replacing it with some safer mechanism. We'll get to *NextItem* later, after we've dealt with sequence classes.

Multiple Object References

But there's an even bigger problem. Pascal and its relatives have always carried the hazard of multiple pointers to a single heap block. You can easily shoot off a toe by using one pointer to dispose of the block in the heap and then forgetfully trying to access the same block at the end of another pointer. Because the block is gone, you get a runtime error.

If anything, multiple references are an even bigger problem in Object Pascal because OOP programs use, in effect, many more pointer-based entities than standard Pascal programs do. With lots of object reference variables around, it's easy to assign one to another and forget about the multiple reference. What happens if you mistakenly use one reference to call the object's *Free* method and then later try to access it through the other reference? Bang. Figure 19-3 illustrates the problem.

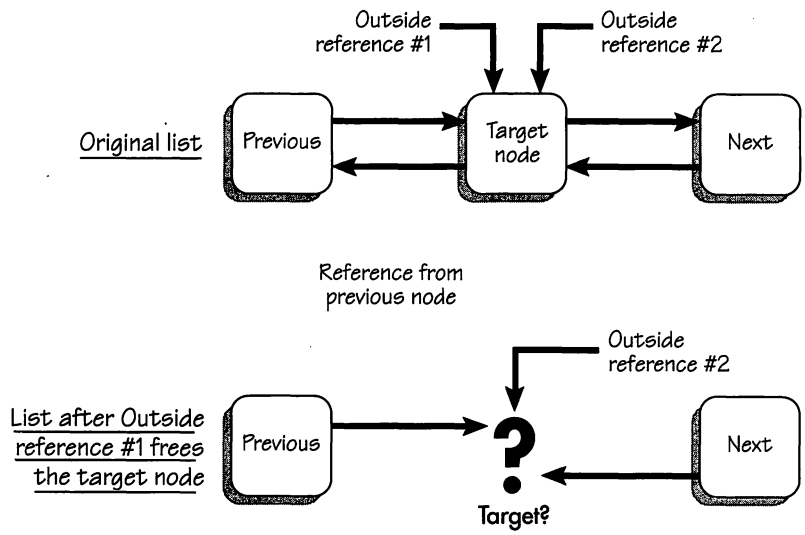


Figure 19-3.
Multiple references to a node.

The problem is even more acute if the object is part of a data structure, such as a *TLinkedList* object. If you use an external reference to a node to free it, the state of the whole list is totaled. When you then try to traverse the list—bang. Again, see Figure 19-3. And freeing list nodes from outside isn't the only danger. You can also corrupt the list by altering its data from outside.

Sometimes your design might require access to the same object through multiple references, but be advised that it's dangerous. It's far better to design the list so that all actions on its nodes must go through the list as intermediary, using its methods.

Both *TLinkedList* and *TList* are susceptible to the multiple reference problem. In *TList*, a freed element causes no problem if you're simply traversing the list. But if you send a message to an element that's no longer there, you get a runtime error.

Lockability

Is there any solution to the multiple reference problem? As it happens, yes, at least for objects.

If you have two references (call them *ref1* and *ref2*) to the same object, how can you keep from accidentally freeing the object by means of *ref1* and then trying to access the nonexistent object through *ref2*?

You can lock the object through one reference so that only that reference (and no others) can free it. If you like, locking can extend to other actions on the object, but we'll implement a limit on freeing only. (Actually, the technique that follows could be implemented for ordinary dynamically allocated records as well.)

How to lock

To make an object "lockable," we give it two instance variables and two methods for locking and unlocking:

```
fLocked: Boolean;    { True if self is locked }
fOwner: Integer;     { ID of owning reference }
function Lock (owner: Integer): Boolean;
function Unlock (owner: Integer): Boolean;
```

To prevent unauthorized or accidental freeing, we override *TObject.Free* so that it works only if the object (*self*) is not locked:

```
procedure TSomeClass.Free;
  override;
begin
  if not self.fLocked then
    inherited Free
  else
    ;  { Error handling }
end;  { TSomeClass.Free }
```

Who's the owner?

The trick, of course, is to identify the owning object uniquely, so that it and only it can unlock and free the owned object.

Any approach that yields a unique value will do, as long as we can associate it somehow with a particular reference to the owned object. We'll make use of the following fact: The most certainly unique thing about an object reference is the physical address of the reference variable in memory. When you declare a reference variable:

```
var
  aRef: TSomeClass;
```

the compiler stores that variable's value at a specific address, which you can obtain with THINK Pascal's @ operator:

```
addressOfARef := @aRef;
```

This yields a pointer value 4 bytes long. That's a lot to store for each object you want to lock, so let's use some magic to shorten it:

```
signatureOfRef := LoWord(Ord(@aRef));
```

The @ operator, as we said, returns a pointer value. We use the *Ord* function to convert that pointer to a *Longint* value (still 4 bytes). Then, to get a shorter integer value, we extract the low-order 2 bytes of the *Longint*. *signatureOfRef* will be an integer value, costing us only half as much to store.

Signatures

We'll call the magic value we just obtained for the object reference variable *aRef* its "signature." The signature is this particular object reference's unique ID, which it uses to lock and unlock other objects.

On a call to some object's *Lock* method, you pass the locker's signature as a parameter. *Lock* stores the signature in the locked object's *fOwner* field and sets the locked object's *fLocked* field to *true*. After that, no object can successfully free the locked object without first getting it unlocked. And only the signature of the original locking object can open it up. Of course, you could cheat. Suppose you have two references:

```
var
  ref1, ref2: TSomeClass;
```

both set to refer to the same *TSomeClass* object. If you used *ref1*, and its signature, *LoWord(Ord(@ref1))*, to lock the object, you could cheat by sending this message:

```
success := ref2.Unlock(signatureOfRef1);
```

We can't prevent that. Our goal is to prevent accidents, not malicious intent.

Danger! Quicksand

The @ operator returns a variable's address—where it is in the Mac's address space. But many object references are variables located inside objects (instance variables), and objects are dynamically allocated blocks in the heap. If you have an instance variable *fField* inside an object and you apply @ to it:

```
success := fField.Lock(LoWord(Ord(@fField)));
```

you're walking on quicksand. The Mac's Memory Manager shifts relocatable blocks (including objects) around as needed. The absolute address provided by *@fField* can quickly have no connection to *fField* at all. If you then try something like this:

```
success := fField.Unlock(LoWord(Ord(@fField)));
```

the new value @ returns might be different from the old. Down you'd go. Figure 19-4 illustrates the block move and the consequent loss of reference.

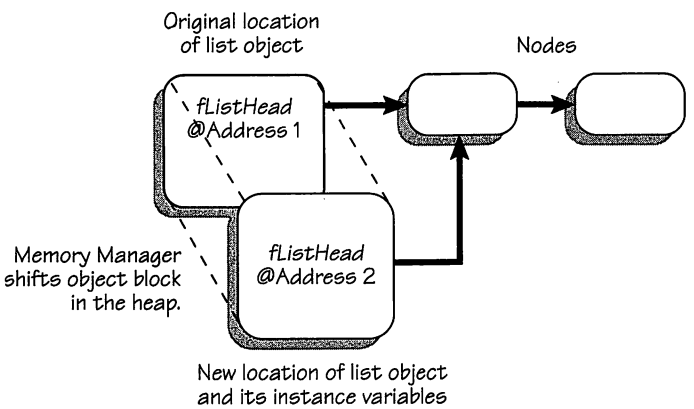


Figure 19-4.
Shifting memory blocks. If the current address of fListHead *is now used to unlock the first node, the attempt will fail. Address 2 is not the same as Address 1, which was used to lock the node.*

Can we get around this potentially deadly problem? Yes. When the locking object is first created, we get the signature of some reference-type instance variable in the object with *LoWord(Ord(@fTheVariable))*. We store that signature in an integer-type *fSignature* field. Then, throughout its life, the locking object can manipulate its locked objects through that reference using the same, original signature value.

Does it matter that many objects can be locked through that same reference, using that same signature? Not really, as long as no other reference to the locked objects has the same signature.

The Cost of locking

In the locking object, locking costs 2 bytes for the *fSignature* value used to lock other objects. In locked objects, the cost is 4 bytes: 2 for the *fLocked* field and 2 for the *fOwner* field. Any such overhead is expensive, of course, but safety from multiple references can be worth it.

Class *TLockable*

One way to give all lockable objects (such as those that can be stored in a list) the same reliable locking mechanism (the fields and the methods) is to make the lockable objects descend from a *TLockable* class from which they inherit the necessary instance variables and methods. Here's a sketchy version of such a class:

```
type
  TLockable = object(TObject)

      fLocked: Boolean;   { True if object is locked }
      fOwner: Integer;    { Signature of locking object or reference }

      procedure lLockable;
      function Lock (owner: Integer): Boolean;
      function Unlock (owner: Integer): Boolean;
      function Locked: Boolean;
      procedure Free;
      override;
    end;  { Class TLockable }
```

You can see the full implementation of this class on the code disk.

TLockable and the Mark List Implementations

Here's how we'll implement locking in *TLinkedList*, *TList*, and other structures, using *TLockable*.

When we add an item of data to a list, the *Add* method locks it. In both of our list implementations, we'll arbitrarily lock the item with the instance variable that refers to the underlying list. The *Add* method contains code like this:

```
if not theNode.Locked then
  if not theNode.Lock(self.fSignature) then
    { Handle the error }
else
  { Handle the error }
```

Figure 19-5 illustrates list node ownership.

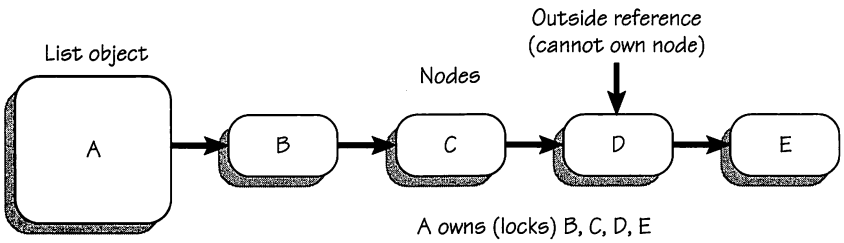


Figure 19-5.
A node's owner. Only the owner can unlock or free the owned node.

When we add an object to the list, it arrives unlocked; otherwise, the new owner inside the list wouldn't be able to lock it for the list's use. If an object is locked, we must explicitly unlock it through the owning reference before we call any of the list's add or insert methods for the object.

Before we can extract a node, extract the head, or delete a node from the list, the node to be removed (the target) must be unlocked by means of the owning reference, as this code from inside an *Extract* method, for example, shows:

```
if not self.fList.Unlock(self.fSignature) then
  { Handle an error }
else
  { Node was successfully unlocked, so go ahead }
```

Fortunately, we can hide the details inside the list code.

We'll typically free a node object only after clipping it out of the list with *Extract* or *Delete* or when we traverse the list freeing all nodes at once. Before a node still linked into the list can be freed, we normally have to unlock it. In our list components, we have to traverse the list, sending each node an Unlock message followed by a Free message. In *TList*, we then dispose of the array handle, which now contains undefined references for the freed node objects.

Optional Locking

What we've finally chosen to do, in both list implementations, is to make locking optional. By default, the list will lock all objects as it adds them and unlock them as it removes them. But, right after initializing the list, the using programmer can override the default by sending the list a SetLocking message. *SetLocking(false)* turns off the default locking action.

Note that once we start out in one mode—locking or not—we can't switch in midstream. *SetLocking(false)* works only if the list is empty. The list locks either all of its nodes or none of them. Of course, if the list doesn't lock a particular object, some other reference to the object can.

Can We Keep *Peek*?

Peek, you'll recall, returns a reference to the object at the mark. Can that reference be used for mischief now?

At this point, the reference returned by *Peek* can't be used to free the object it refers to. This is because the object is still locked by the list. However, in *TLinkedList*, the lock doesn't prevent us from accessing the list's innards through *Peek* for other purposes. We can still call

```
aReference := aList.Peek;
anItem := aReference.NextOf;
```

to trace out the chain of nodes using *TLinkable*'s *NextOf* method (or *TDoublyLinkable*'s *PreviousOf* method). But, because all the following nodes are locked, at least we can't mischievously free any of them.

We could let *NextOf* check for a lock before we proceed, but that won't work for *PreviousOf*. A node can be locked by only one reference, so we can't lock it from both the previous node and the succeeding node.

Peek still looks somewhat dangerous, although we'd have to be foolishly malicious, going out of our way, to abuse it. Do we really need it? What about situations in which we want to look at a node's value without removing it from the list? Because a node is the data object, and because the list is polymorphic and highly general, a way to peek at a node without disturbing it would be handy. Returning a reference to it is the only way.

On balance, we'll probably keep *Peek*, especially because the lock mechanism does at least prevent freeing a node through the reference returned by *Peek*. If paranoia's your game, you might want to extend the locking mechanism, building in a requirement to unlock a node before any of its methods will work, especially data update methods.

Reusability

Finally, we have to ask: How reusable are *TLinkedList* and *TList*? Are there any limitations on the kinds of data they can hold? Are there any programming situations in which they don't fit?

Reusability turns out to be a considerable stumbling block for *TLinkedList* (but not so much a one for *TList*). The problem occurs when we try to use a *TLinkedList* in PicoApp or other application frameworks, specifically when we try to store view objects in the list. If you think about it, the major part of all the data we're likely to

store in a *TLinkedList* instance in PicoApp will be views. Nearly all kinds of data will have a need to display themselves, and views are the medium of display in PicoApp.

Here's why we'll have a problem. Figure 19-6 shows the class branch for *TLinkedList*'s nodes beside the branch for PicoApp's views.

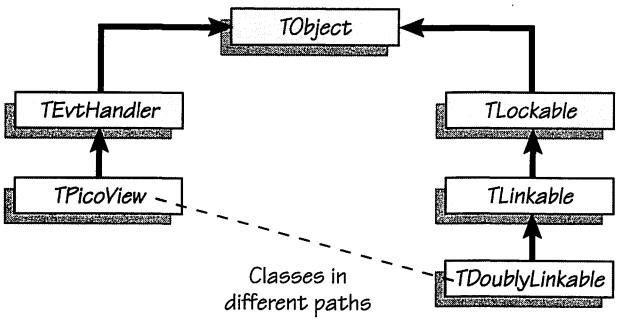


Figure 19-6.
The difficulty of merging two branches of a hierarchy.

If a view object is to be stored in a list, how can we make it a *TDoublyLinkable* descendant? *TPicoView* isn't a *TDoublyLinkable* object now, and *TLinkedList* can't contain objects that aren't *TDoublyLinkable*s.

The problem of integrating *TLinkedList* into the fabric of PicoApp becomes a matter of deciding how to change one or both class hierarchies to make them mesh. Should we insert the entire *TDoublyLinkable* hierarchy into PicoApp's class hierarchy above *TPicoView*? Below it? Somewhere else? Or not at all?

TDoublyLinkable* above *TPicoView Figure 19-7 on the next page shows *TDoublyLinkable* inserted into PicoApp's class hierarchy above *TPicoView*. If we mesh the hierarchies this way, every view object automatically has all the apparatus of a list node: lockability, linkability, and double linkability. That's an overhead of 16 extra bytes of instance variables per view object. But how often will view objects actually be used in a list (other than the view list)? Thirty percent of the time? Forty? Less? That's quite a burden of extra bytes to impose just so that some views can be put into a list. Suppose we're dealing with 10,000 data objects, each with that extra 16 bytes; that's 160,000 extra bytes.

Another question about this strategy arises. Might we also want to list other kinds of objects in PicoApp? Documents? Windows? How high in the hierarchy should we install *TDoublyLinkable*—and with what ramifications?

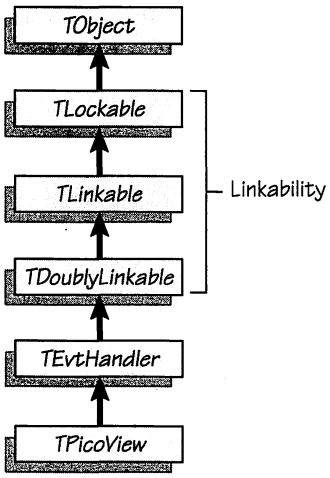


Figure 19-7.
TDoublyLinkable *inserted into the hierarchy above* TPicoView *to make all views linkable.*

TDoublyLinkable* below *TPicoView Figure 19-8 shows *TDoublyLinkable* inserted into PicoApp's class hierarchy below *TPicoView*. In this case, only some views would be linkable—those descended from *TPicoView* through *TDoublyLinkable*. We'd

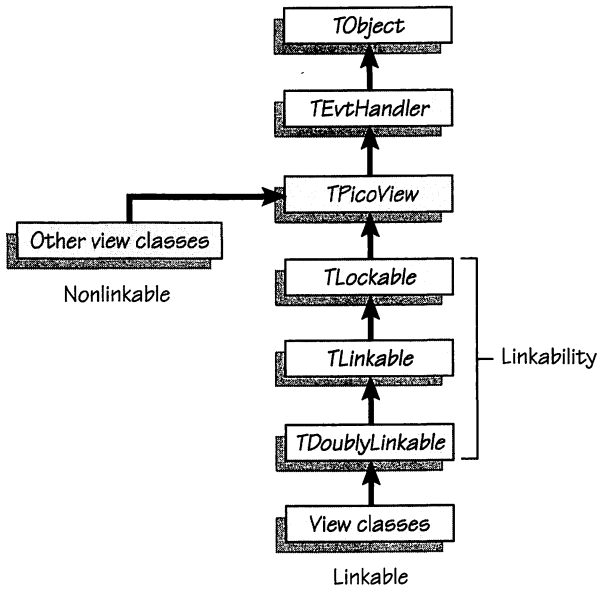


Figure 19-8.
TDoublyLinkable *inserted into the hierarchy below* TPicoView. *Only some views are linkable, and all linkables are views.*

have a choice of which to subclass in making our view classes. And only those views that really needed to be in the list would have to carry the extra 16-byte overhead. The main drawback of this strategy is that other kinds of objects wouldn't be storable in the list: documents or windows, say, or any other class not descended from both *TPicoView* and *TDoublyLinkable*.

Is there a better place to insert the property of linkability into the PicoApp hierarchy? Unfortunately, no. Should we simply forget it? The answer is both yes and no.

For the purposes of this book, we won't pursue *TLinkedList* beyond the next chapter. That way we don't have to face completely the issue of how to integrate it with PicoApp. Instead, we'll move on to a different kind of list, *TList*, whose requirements impose only a small overhead on PicoApp's views and other objects and which otherwise meshes smoothly with PicoApp. We'll start developing *TList* in Chapter 21.

Is that the end of *TLinkedList*? Not quite. One of the projects at the end of this chapter asks you to implement *TLinkedList*. Reading the next few chapters will help you to do that much more easily.

And we'll explore *TLinkedList*'s implementation briefly in Chapter 20 before we move on to *TList*.

Finally, we're about to look at one perhaps surprising way to integrate *TLinkedList* with PicoApp.

Integrating Our Lists with PicoApp

We've looked at the difficulties of using *TLinkedList* to store view objects in PicoApp. Now we'll briefly consider an interesting workaround for the problem. (By the way, *TLinkedList* will work fine with PicoApp as long as you don't want to store instances of PicoApp's classes in the list.) Our solution is the one alluded to in Chapter 16 as a way to make *TDie* completely general but still storable in PicoApp's view lists.

We'll also compare *TList*'s ability to mesh with PicoApp with *TLinkedList*'s. *TList*'s has considerably more ability to mesh with PicoApp.

A Trojan horse

The first part of our integration strategy is a way to get *TLinkedList* into PicoApp without changing PicoApp's class hierarchy.

What we want is the ability to use a *TLinkedList* object in PicoApp to store viewable data items. The items must be able to respond to Display messages and perhaps to mouse clicks or keypresses.

Let's first solve the problem of getting the list into PicoApp. Then we'll see how to make data items of the views.

The trick is to think of the list not as many views but as one view. We create a special *TPicoView* subclass, called *TListView*. One of its instance variables is of type *TLinkedList*, so it can hold a *TLinkedList* instance. *TListView* also has methods to

manage the list. Those methods call *TLinkedList*'s methods to get things done. Figure 19-9 shows how *TListView* and the linked list classes fit into PicoApp's hierarchy.

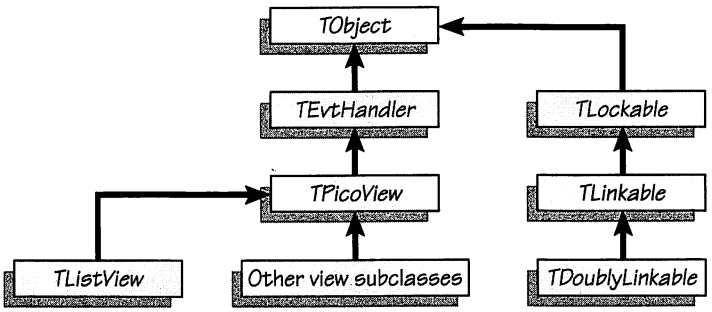


Figure 19-9.
TListView *in the PicoApp class hierarchy.*

And here's what *TListView* might look like:

```
type
  TListView = object(TPicoView)

      fList: TLinkedList;

      procedure IListView;
      procedure Add (aView: TLinkableView);
      function Remove: TLinkableView;

      function DoMouseCommand (event: EventRecord): Boolean;
      override;
      procedure Display;
      override;
      procedure Free;
      override;
    end;  { Class TListView }
```

This is simply another manager object, like many we've already seen. But it's also a view subclass. When PicoApp sends a DoUpdate message to the front window, the window object sends Display messages to all its views—including, perhaps, this one. When mouse-downs are passed along the event-handler chain by means of DoMouseCommand messages, a *TListView* object can be made to receive them, too, by registering itself as an event handler. Figure 19-10 shows the structure of a PicoApplication window's view list that uses a subclass of *TListView*, *TIconView*.

What the *TListView* instance does then depends on how we override *Display* and *DoMouseCommand*. An overridden *Display* method should send the *TLinkedList* instance stored in *fList* a Display message. The list, in turn, sends each of its nodes a Display message. Processing mouse commands is similar. The *TListView* object traverses its underlying list to send each list node a DoMouseCommand message.

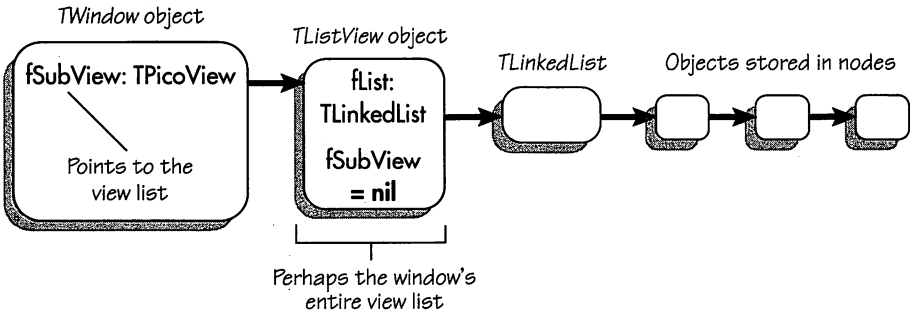


Figure 19-10.
How TListView *fits into PicoApp's view mechanism.*

Example: Icon List

To see a concrete example, let's take a quick look at a simple application. (We won't develop it in detail.) Icon List displays various icons in a neat grid in a window. The user can select icons with the mouse and move them around in the grid by means of various menu commands. The commands translate into list movement messages, such as BringCloser, SendFarther, BringToFront, and so forth. The user can also add new icons and delete existing ones, again by choosing from a menu. Figure 19-11 suggests what the screen for Icon List might look like.

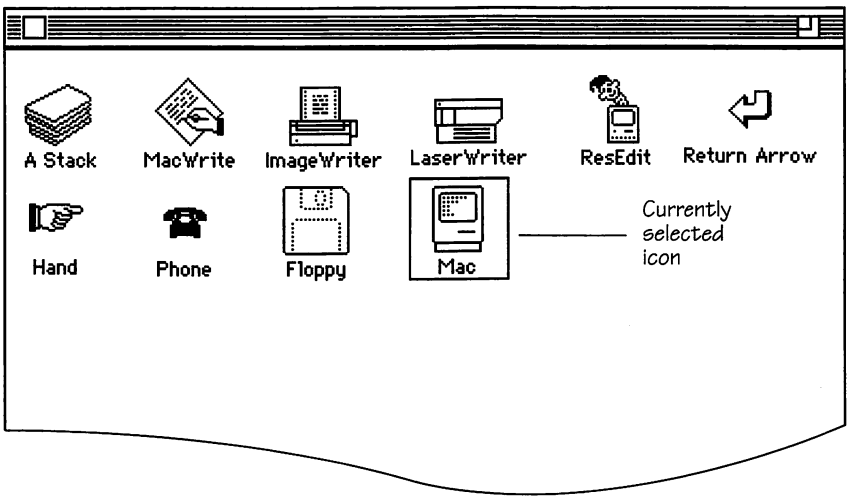


Figure 19-11.
The hypothetical Icon List's interface screen. We won't write the entire program.

The icons are managed by a class called *TIcon*, which can get an ICON resource from the application's resource fork and plot the icon in a designated rectangle in the window. *TIcon* is declared as a descendant of *TDoublyLinkable*, so we can store *TIcon* objects in a *TLinkedList*.

A second *TLinkedList* is used to store *TRectangle* objects. *TRectangle*, also a descendant of *TDoublyLinkable*, manages a QuickDraw *Rect* data item. The rectangle list stores rectangles in the same order in which the icon list stores icons. When we need to draw an icon, we get its index position in the icon list and use that to find the icon's display rectangle at the same index position in the rectangle list. Figure 19-12 shows how the icon and rectangle lists work together.

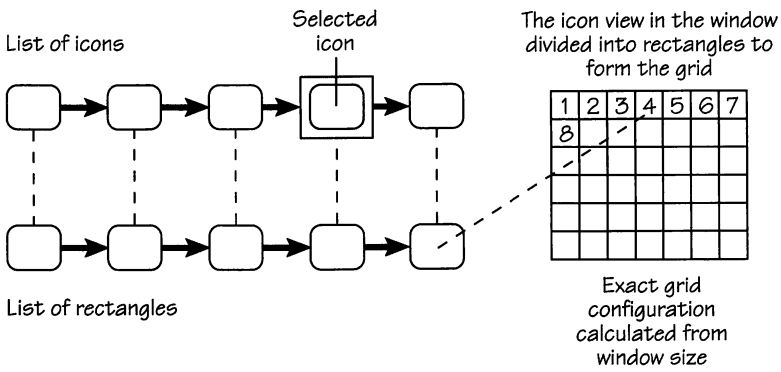


Figure 19-12.
Icon List's icon and rectangle lists. The selected icon is in position 4 in the icon list, so we get rectangle 4 from the rectangle list when it's time to display the icon.

The extra commands to move icons around in the window can be implemented as methods in a *TListView* subclass:

```
type
  TIconListView = object(TListView)

      { Inherits from TListView }
      {  and adds... }
      fRectList: TLinkedList;  { Holds view rectangles }

      procedure BringCloser;
      procedure SendFarther;
      procedure BringToFront;
      procedure SendToBack;
      :
      function Select (event: EventRecord): TIcon;
      :
    end;  { Class TIconListView }
```

We can also implement an internal scrap (a private clipboard for the application's use) by adding a global variable, *gClipboard*, to the application. It stores an icon object that the user has cut or copied. A *Select* method is called by the *DoMouseCommand* method of an icon object that the user has clicked, to install *self* in the scrap. We can implement the Edit menu's commands in our application's document subclass to work with the same scrap. Here's a sample *Cut* method:

```
procedure TIconListDoc.Cut;
  override;
begin
  gClipboard := TIcon(fList.Extract);  { Remove item and store it }
  gClipboard.Hide;                     { Hide the icon at old location }
end;  { TIconListDoc.Cut }
```

Suppose that the user clicks an icon and chooses Delete from a menu. PicoApp responds by sending a DoMouseCommand message down the event-handler chain. Here's the icon list view's *DoMouseCommand* method:

```
function TIconListView.DoMouseCommand(event: EventRecord): Boolean;
  override;
begin
  gSelection := self.Select(event);
end;  { TIconListView.DoMouseCommand }
```

And here's *Select*:

```
function TIconListView.Select(event: EventRecord): TIcon;
  var
    aSequence: TDoublyLinkedSequence;   { See Chapter 22 }
    obj, foundItem: TIcon;
    ok: Boolean;
begin
  ok := false;
  foundItem := nil;
  { We use a "sequence object"--discussed in Chapter 22--to }
  {  traverse the list and act on each of its nodes }
  aSequence := self.fList.EachItem;
  obj := TIcon(aSequence.NextItemOf);
  while (not ok) and (obj <> nil) do      { Loop through list }
    begin
      { At each node, we send object a DoMouseCommand message }
      ok := obj.DoMouseCommand(event);
      if ok then
        foundItem := obj
      else
        begin
          obj.Hilite(false);              { Turn off its hilite }
          obj := TIcon(aSequence.NextItemOf);
        end;
    end;  { while }
  aSequence.Free;
  Select := foundItem;
end;  { TIconListView.Select }
```

When the icon list view object receives the message, it sends the same message along its *TLinkedList* of icons. When the clicked icon responds to the message, the icon object's *DoMouseCommand* moves the mark in the list to the icon's node. This selects the icon object, and the icon list view object returns a reference to that icon. The icon list view object stores the reference in the variable *gSelection*, which thereby points into the list. When the list is sent a Delete message, the icon at the mark is deleted and another icon in the list is set as the currently selected icon.

Similarly, when the user selects an icon and chooses Bring To Front from the menu, the icon is selected (the list's mark moves to it), and the icon list view object's *BringToFront* method sends Extract and AddHead messages to the list. *Extract* removes the marked list node. *AddHead* attaches the list node to the front of the list. Here's a sample *BringToFront* method:

```
procedure TIconListView.BringToFront;    { Bring up item at mark }
  var
    item: TIcon;
begin
  item := TIcon(fList.Extract);          { Remove item at the mark at old location }
  item.Hide
  fItsList.AddHead(item);                { Re-add item at the head }
  self.Display;                          { Redisplay the icons }
end;  { TIconListView.BringToFront }
```

Of course, a Bring To Front command requires that the grid of icons in the window be redrawn, so an update message is generated. As each icon is about to display itself, it asks the icon list view object to give it a rectangle for display. The icon list view uses the icon's list index position to find the right rectangle on the screen.

Remember, of course, that what we're considering in this example is how a *TLinkedList* can be smuggled into PicoApp, inside a special view object that manages the list.

Making the list nodes viewable

So far, we've talked about the *TIcon* objects in our *TLinkedList* object as if they were legitimate heirs of *TPicoView*. But, in fact, they're still stuck in the *TDoublyLinkable* hierarchy. So how do we make them into views?

Look at the problem this way: The icon objects are necessarily storable in the list, and the list itself is hidden inside a real view object. The icon objects don't have to be descendants of *TPicoView*—but they do have to be viewable. They also have to be clickable.

Our solution is to declare a class called *TLinkableView*. It's basically a copy of *TPicoView*, with a few modifications to reflect the fact that it doesn't live inside PicoApp. *TLinkableView* is a subclass of *TDoublyLinkable*, so its instances can be stored in a *TLinkedList*. Here's the declaration for *TLinkableView*:

```
type
  TLinkableView = object(TDoublyLinkable)

      fViewRect: Rect;
      fClickRect: Rect;
      fWantsClicks: Boolean;
      fWantsDoubleClicks: Boolean;
      fWantsTripleClicks: Boolean;
      fCloseEnough: Integer;
      fActive: Boolean;

      procedure ILinkableView (r: Rect; active: Boolean; wantsClicks, wantsDouble,
          wantsTriple: Boolean; closeEnough: Integer);
      procedure SetViewRect(r: Rect);
      function RectOf: Rect;
      procedure Display;
      override; { From TLinkable }
      procedure Hide;
      procedure Hilite (hiliteYes: Boolean);
      function TrackMouse (var time: Longint; var place: Point): Boolean;
      function Clicked (event: EventRecord; var clicks: Integer): Boolean;
      function DoMouseCommand (event: EventRecord): Boolean;
      override;
      procedure DoClick (event: EventRecord);
      procedure DoDoubleClick (event: EventRecord);
      procedure DoTripleClick (event: EventRecord);
end;  { Class TLinkableView }
```

The declaration is roughly parallel to *TPicoView*'s.

The moral

TLinkableView necessarily duplicates much of *TPicoView*'s functionality. Normally, we'd subclass, letting *TLinkableView* inherit from *TPicoView*. But, in this case, we have to copy *TPicoView*. *TLinkableView* can't inherit from *TPicoView*.

Recall that we started this section by talking about a workaround. Largely because we need such a workaround (not to mention *TList*'s greater efficiency), we won't implement *TLinkedList* in Part 3. We'll work instead with *TList*, which is much simpler to integrate with PicoApp.

But we shouldn't abandon *TLinkedList* entirely—and we won't. It's worth spending Chapter 20 studying how *TLinkedList* works internally, although we'll look at only a few sketchy methods. Besides, the list will work fine with many other kinds of data, and it has no problem at all coexisting with an application that isn't based on an application framework. It's quite suitable in ordinary Pascal programs.

One final conclusion to draw from this discussion is that even when a component doesn't fit well into PicoApp (or MacApp or TCL), there's usually a way to smuggle it aboard. We've only just begun to delve into OOP's bag of tricks.

Integrating *TList* with PicoApp

That's how we'd proceed with *TLinkedList*. *TList* is easier, although we still face a little work before the list will fit neatly into PicoApp. Recall that *TList*'s nodes will be descendants of class *TLockable*, not of *TDoublyLinkable*. We could have made the nodes descendants of *TObject* if we hadn't wanted to lock the nodes to protect them from damage through outside object references. Then any object, practically speaking, could have been put into the list. *TList*'s hierarchy would mesh with PicoApp's class hierarchy because *TObject* is already in that hierarchy.

But we did decide to lock nodes, so we do have to put up with *TLockable*, which is not part of PicoApp's hierarchy. It looks like the *TLinkedList* problem all over again—except that at least the overhead from *TLockable* is much smaller than that from *TDoublyLinkable*, *TLinkable*, and *TLockable* combined.

Let's consider two options for integrating *TList* with PicoApp.

First, we could choose to push *TLockable*'s capabilities up into *TObject*, dispensing with the *TLockable* class altogether. *TObject* would have the instance variables and methods formerly located in *TLockable*. This isn't a terrible option, really. There's some logic to it. Lockability is a capability we might well want all objects to have. Putting a capability that general into *TObject* might be the best solution. Of course, then all objects descended from *TObject* would inherit the bytes for the old *TLockable* fields. The overhead from each object would be 4 bytes. For 10,000 objects, that would be 40,000 bytes.

The second option is to modify PicoApp, putting *TLockable* into the class hierarchy there. Then the question is where to put it. Should only views be lockable? Or views, documents, and windows? Or all event handlers? We'll add *TLockable* to the PicoApp hierarchy just below *TEvtHandler*. That way, all views, documents, and windows can be lockables. At the same time, though, some event handlers, such as application objects, could descend directly from *TEvtHandler*, bypassing *TLockable*. And that sums up the changes we could make. (Note that we haven't made these changes to PicoApp in the version on the code disk. We'll leave the actual job as a project.)

One further option would be to use a modified version of *TList*, one that does use *TObject* rather than *TLockable* as its base class. This version would be a lot like the *TList* class now used quite successfully in MacApp and the similar *CList* class used in TCL. We won't choose this alternative, though, because we are designing a component to be used in all sorts of programs, perhaps even in ordinary Pascal programs rather than programs based on application frameworks. We'll let the added security of lockability outweigh the extra integration hassles. We might decide not to at another juncture, of course.

One Other Approach to List Making

At the beginning of Chapter 18, we briefly took up how to store object-type data in a traditional linked list—one in which nodes are records and links are ordinary

handles, not object references. We'll shortly look at how such a list compares with *TLinkedList* and *TList* in terms of cost; the traditional list turns out to be more expensive in its storage requirements than either of our more object-oriented lists.

But consider this: The traditional list has no problem at all integrating with PicoApp, MacApp, TCL, or any other program structure. Its record-type nodes can contain fields to hold object-type data, and that data can even be of class *TObject*. From the standpoint of integration, the data's type doesn't matter at all.

As it turns out, lockability is still an issue with such a list. We can't destroy the whole list by freeing one of its nodes, but we can cause runtime errors by freeing the data object stored in a node. As soon as we try to send that object a message, we get an error. We have the same relatively minor integration problem we did with *TList*.

For some kinds of data structures, then, we might end up preferring traditional building blocks and hiding our object-type data inside them. We'd want to keep that option open for any structure built from linked nodes. If a data structure component's reusability depends on how easily we can integrate the component with our programs, integration becomes a high priority.

Last Thoughts

There's more to reusability, of course, than how well our components integrate with consumers' programs.

We should take great pains to give a component a clean, easy-to-use interface. Consumers will use the component only to the extent that its interface is informative and indicates flexibility.

We should provide methods for any valid internal access that consumers might need. If methods are available, consumers will be less likely to violate encapsulation.

We should document a component clearly enough—and concisely enough—that consumers will have no trouble learning its capacities and limitations.

We should make it easy to extend or modify a component by subclassing. In some cases, providing hook methods improves the ease of overriding some parts of a component's behavior while leaving the heart of it intact. Some of PicoApp's classes provide extensive examples of this technique.

Finally, and probably most important, we should take every possible step to insure that a component is correct and robust. This means extensive testing, user testing, error checking, and, if possible, error recovery. No component is perfect, but we should strive to make ours as certifiably reliable as possible. In the long run, the future of a reusable software component industry rests on our ability to deliver on that guarantee.

Summary

In this chapter, we've proposed an abstract version of the two list classes we'll implement in detail in the next chapters. We've also examined several important design issues for these and any reusable software components. We've looked at proposed base classes for our two lists and studied the consequences of our choices. We've foreseen safety hazards likely to be found in our list components and proposed ways to prevent or minimize the hazards. And we've considered how reusable our components will be—how easy they will be to integrate with the programs that need such components, including application frameworks such as PicoApp.

In Chapter 20, we'll develop at least a few of the more interesting methods of class *TLinkedList*, and then, in Chapter 21, we'll move on to a full implementation of *TList*. In the chapters that follow, we'll deal with some of the more difficult *TList* implementation issues.

Projects

- We won't develop a full working version of *TLinkedList*, so try your hand at it. Model your class declaration on the pseudocode on pages 460 and 461. Chapter 18 goes into many of the techniques you'll need. And you can peek ahead to Chapter 20 and borrow from what we do with *TLinkedList* there.
- Try your hand at implementing our locking mechanism in your *TLinkedList*. Most of what you'll need is available in this chapter.
- Implement at least one list component based on an ordinary statically allocated array. Declare an array of some appropriate type as an instance variable in your list class. Then develop methods to manage a list within the array. Think carefully about how to implement deletions and insertions in the list. Keep the array's finite size in mind, too.
- Make the changes in PicoApp that are necessary to integrate *TList* with PicoApp. That means inserting *TLockable* into the class hierarchy just below *TEvtHandler*. *TLockable*'s immediate superclass will be *TEvtHandler*. *TLockable* itself becomes the immediate superclass of *TPicoView*, *TPicoApp*, and *TPicoDoc*. You'll need to change the appropriate heritages and find a good place in PicoApp's units to put *TLockable*'s declaration and its implementation. Look at units *UPAGlobals* and *UPAUtilClasses*. Test your work by writing a small PicoApplication that uses the list. You might even try your hand at the Icon List application we sketched out in this chapter.
- As an alternative to *TLinkedList*, design a component encapsulating a traditional linked list with record-type nodes. Give each node forward and backward links of some handle type. And give each node a data field of type *TObject* (or *TLockable*). Your component should supply the methods suggested in the pseudocode on pages 460 and 461. It should implement a "mark" (a handle) that always points to a current node and that can be moved around in the list.

20

IMPLEMENTING THE *TLINKEDLIST* COMPONENT

In this chapter, we'll complete enough of our *TLinkedList* software component to see how it works. We'll implement only a few of its methods, but you should be able to write the others from what you'll see here. We won't implement the search and traversal methods for *TLinkedList*. The similar methods for *TList* in later chapters should lend themselves to easy adaptation for *TLinkedList*. But we'll cover several other *TLinkedList* methods here. Even general insertion and deletion are doable at this point because they take place at the mark. We might need our *Find* method to move the mark to the right place, but that's independent of insertion or deletion. We'll work with code for

- Linkables, revised
- Moving the mark in the list
- Performing various utility actions
- Cloning and freeing the list
- Adding nodes to the list
- Removing nodes from the list
- Accessing data in the list

We'll also compare the speed and space costs of using each kind of list. We'll look at how *TLinkedList* does against a traditional linked list we'll call *TOldList* and at how it does against *TList*.

Updates of *TLinkable* and *TDoublyLinkable*

During the development of *TLinkedList*, our sketchy version of *TLinkable* from Chapter 18 filled out a bit. By means of inheritance, so did *TDoublyLinkable*. Linkables are now descendants of *TLockable* rather than only of *TObject*. And they participate in the operations of cloning the list.

TLinkable Methods

Listing 20-1 is the new, improved declaration for *TLinkable*:

```
type
  TLinkable = object(TLockable)

    fNext: TLinkable;                { "Pointer" to next node in a list }

    procedure TLinkable.ILinkable (eh: TErrorHandler);
      { Initialize node's fields }
    procedure TLinkable.SetNext (obj: TLinkable);
      { Link up to obj as next node }
    function TLinkable.NextOf: TLinkable;
      { Return reference to next node }
    function TLinkable.ErrHandlerOf: TErrorHandler;
      { Return the node's error handler object or nil }
    function TLinkable.Clone: TObject;
    override;
      { Clone this object and those that follow it }
  end;  { Class TLinkable }
```

Listing 20-1.
The revised class TLinkable.

To the original *SetNext* and *NextOf* methods, we've added a *Clone* method. *ILinkable* is added to initialize a node's inherited *fLocked* and *fOwner* fields and install an error handler in the node, if one is wanted. *Clone* overrides *TObject*'s *Clone*. By definition we're dealing with linkables—objects strung together in lists—so *TLinkable.Clone* is designed to do its part in cloning the whole list.

Clone uses recursion, relying on the fact that lists are recursive structures: Not only is the whole string of nodes a list, but any substring of nodes is also a list. Even a single node is a list. Even an empty list is a list. Figure 20-1 diagrams the recursive process of cloning.

The purpose of *TLinkedList.Clone* is to copy all nodes in a list, returning a reference to the first one. Our strategy is to have each node send the node after it a TLinkable.Clone message. It's as if the first node tells the rest of the list to clone itself.

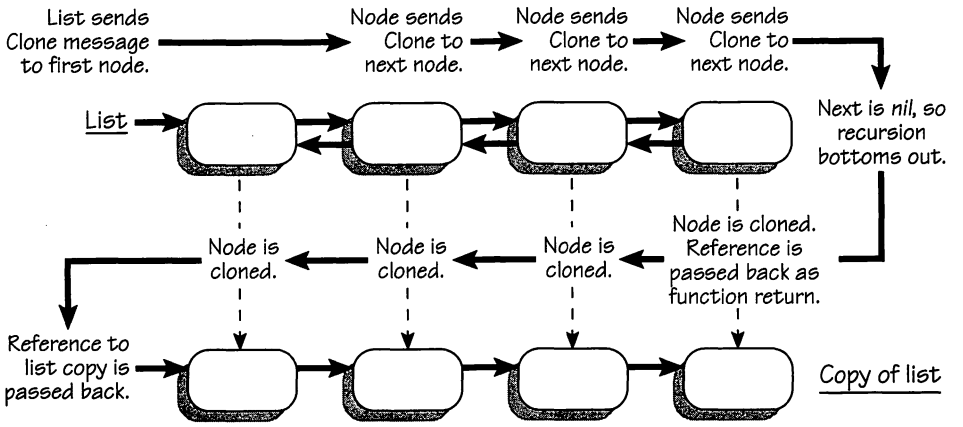


Figure 20-1.
Cloning list nodes. The copy is incomplete—back links go into the original list.

Then the second node tells the rest of the list to clone itself, and so on. As those Clone message sends (function calls) reach the end of the list, they start to unwind back up the list. As each node finds that its next node has been cloned, it finishes up the *Clone* call from its own predecessor. Each node thus plays a small role in the overall strategy.

When the recursive calls return, each node installs a reference to the "rest of the list" (the next node) in its *fNext* field. Thus, as the recursion unwinds back up the list, all of the *fNext* links are correctly set up. Here's *TLinkable.Clone*:

```
function TLinkable.Clone: TObject;
  override;
  var
    temp: TLinkable;
begin
  { Clone self }
  temp := TLinkable(inherited Clone);
  if self.fNext <> nil then
    { Clone rest of list, that is, the next node }
    temp.fNext := TLinkable(self.fNext.Clone);
  Clone := temp;
end;  { TLinkable.Clone }
```

Later in this chapter, we'll look at cloning from the higher perspective of the whole list. We hand the *temp* reference back to the previous node; the *temp* reference the first node hands back will be a reference to the whole list. A call to *inherited Clone* (*TObject*'s *Clone*) produces a copy of *self* (the node), storing a reference to the copy in *temp*. Because *TObject.Clone* is a function that returns a *TObject*, we have to typecast the returned object to a *TLinkable*.

We'll discuss some other issues related to cloning when we get to *TLinkedList.Clone*. Here we're concerned only with each node's part in the process.

TDoublyLinkable hasn't really changed from our original specification a few chapters back—except that it now inherits more from *TLinkable*.

The *TLinkedList* Class Declaration

Listing 20-2 is the full class declaration for *TLinkedList*. It's quite a large class, with some 30 methods.

```
unit ULinkedList;                    { Not available on the code disk }

interface

  uses
    ObjIntf, UListGlobals, UErrorHandler, ULockable, ULinkable,
    UDoublyLinkable, USearchable, USequence;

  { Some declarations for other, related classes given later; others }
  {  left as a project }

  type
    TLinkedList = object(TDoublyLinkable)

        fList: TDoublyLinkable;          { Head of underlying raw list }
        fTail: TDoublyLinkable;          { Tail of underlying raw list }
        fMark: TDoublyLinkable;          { Current list position of mark }
        fPosition: Integer;              { Number of current node }
        fLength: Integer;                { Number of nodes in list }
        fSignature: Integer;             { ID for locking nodes }
        fErrHandler: TErrorHandler;      { An error-handler object }

        { Initializing the list }
        procedure ILinkedList;           { Sets the list to empty, initially }
        procedure Clear;                 { Frees all nodes and sets list to empty }

        { Moving the mark around in the list }
        procedure Move (toNode: Integer);  { Moves fMark to the }
                                           {  given node number }
        procedure MoveToNode(node: TDoublyLinkable);  { Moves fMark to the }
                                                      {  given node reference }
        procedure Previous;              { Moves fMark to previous node }
        procedure Next;                  { Moves fMark to next node }
        procedure Head;                  { Moves fMark to first node }
        procedure Tail;                  { Moves fMark to last node }
```

Listing 20-2. *(continued)*
The class TLinkedList.

Listing 20-2. *continued*

```
    function Find (obj: TSearchKey): Boolean;  { Traverses the list, using obj }
                                               {  to compare each node's }
                                               {  data with obj's key(s) }
                                     { Starts at current mark, so move as needed }
                                     { Moves fMark to found node if found }

    { Utility functions }
    function IsEmpty: Boolean;       { True if there are no nodes in the list }
    function LengthOf: Integer;      { Returns number of nodes in the list }
                                     { Based on fLength, which is updated by }
                                     {  all add, delete, and insert methods }
    function PositionOf: Integer;    { Returns node number of fMark }
    function HeadOf: Boolean;        { True if mark is currently at head node }
    function TailOf: Boolean;        { True if mark is currently at tail node }
    function OffEnd: Boolean;        { True if a move has tried to go past }
                                     {  either end of list }
    function Clone: TObject;         { Copies self plus all nodes in list }
    override;
    procedure Free;                  { Frees nodes in list and then self }
    override;
    procedure Display;
    override; { From TObject }       { Traverses the list, displaying all items }
    function EachItem: TSequence;    { Creates and initializes a sequence }
                                     {  object }

    { Adding and deleting nodes in various ways }
    procedure AddHead (obj: TDoublyLinkable);  { Adds obj at the head }
                                               {  of the list }
    procedure AddTail(obj: TDoublyLinkable);   { Adds obj at the tail of }
                                               {  the list }
                                               { fMark moves to this node }
    procedure Insert(obj: TDoublyLinkable);    { Inserts obj before node }
                                               {  fMark }
    procedure Append(obj: TDoublyLinkable);  { Adds obj after node fMark }

    procedure Delete;  { Deletes obj at node fMark; throws it away }

    { Access and traversal methods }
    function Extract: TDoublyLinkable;         { Removes and returns node }
                                               {  at fMark }
    function ExtractHead: TDoublyLinkable;     { Removes and returns node }
                                               {  at the head of the list }
```

(continued)

Listing 20-2. *continued*

```
    function Peek: TDoublyLinkable;     { Returns reference to node }
                                        {  at fMark but does not }
                                        {  remove the node }
    function ListError: Integer;        { Returns latest value of error handler }
    function TypeOf: Str30;             { Returns the string 'TLINKEDLIST' }
  end;  { Class TLinkedList }

function NewLinkedList: TLinkedList;    { Not a method }

implementation
  :
```

TLinkedList's Heritage

We've declared *TLinkedList*—which is designed to hold a list of *TDoublyLinkables*—as itself a subclass of *TDoublyLinkable*, so the list class is also descended from *TLinkable*, *TLockable*, and *TObject*. We've given the list itself the properties of a node, so we can build compound list structures: lists of lists, or lists of lists of lists. This is similar to the way in which we can set up types in standard Pascal as arrays of records, records that contain array fields, and so on.

TLinkedList's Instance Variables

TLinkedList comes with seven instance variables (besides the several it inherits from *TDoublyLinkable*). The first, *fList*, is a *TDoublyLinkable* reference to the first node in the list object's underlying list.

We keep two other references to nodes: *fTail* and *fMark*. The *fTail* reference makes it easy to perform operations at the tail end of the list. The *fMark* reference is moved around in the list, from node to node, by the operations of various methods. The *fPosition* instance variable holds the index into the list of the mark's current position.

The *fLength* instance variable provides a count of nodes currently in the list. The initialization method *ILinkedList* initializes the count to 0, and all methods that add or remove nodes update the count.

The *fSignature* variable stores an integer used as the list's signature, which it "stamps" on nodes as it locks them. Then the nodes can't be unlocked unless the same signature is provided as a key.

The *fErrHandler* variable holds a reference to the list's error-handler object, which is created by the initialization method. A reference to the error-handler object is also passed to each node so that the node can post error messages to the list.

TLinkedList's Methods

In this section, we'll look at most of the interesting *TLinkedList* methods except the search and traversal methods.

Moving the Mark

Most of the action takes place at the current mark, so we provide many methods for moving the mark to some desired node. If a node's numeric position in the list is known, we can use *Move* to move the mark according to the node number. If we have a reference to a node, we can use it to send a MoveToNode message. Or we can move in small jumps with *Previous* and *Next* or in large leaps with *Head* and *Tail*. Figure 20-2 diagrams the various movements of the mark within the list.

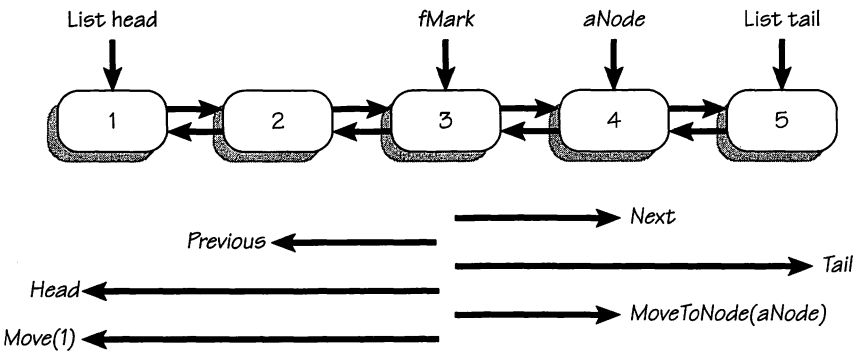


Figure 20-2.
Methods for moving the mark in the list.

Although it makes no sense to move within an empty list, we allow that, but we also post an error message. Likewise, moving past either end of the list makes no sense, so we simply stop the mark at the last (or first) node and post an error message. Trying to move to a node with an index number that is out of range also just stops the mark at the end (or beginning) and posts a message. *MoveToNode* can present the biggest error headaches. We might inadvertently pass in a *nil* node, or we might pass a reference to some node that isn't in the list. Either error is detected, resulting in no movement of the mark and an error message.

Move and *MoveToNode* are the most complicated methods, but they use simple "Data Structures 101" techniques. We'll look at *MoveToNode* here.

MoveToNode

MoveToNode moves the mark to the node whose reference is passed to it—provided the parameter isn't *nil* or a reference to some node outside the list. A good deal of its code checks for these conditions. We need the *while* loop to move the value of *fPosition* to match the new position of the mark.

```
function TLinkedList.MoveToNode (node: TDoublyLinkable): Boolean;
  { Node must be a node in the list; if not, we don't want to move }
  {  the mark to it, so we do nothing and return false }
  var
    current: TDoublyLinkable;
    pos: Integer;
begin
  if (self.fList <> nil) and (node <> nil) then
    begin
      self.fMark := node;              { Move the mark }
      current := self.fList;           { Set up to move fPosition }
      pos := 1;
      while (current <> nil) and (current <> self.fMark) do
        begin
          pos := pos + 1;              { Count the node }
          current := TDoublyLinkable(current.fNext);
        end;  { while }
      { Got there, so keep the position }
        if current = self.fMark then
        begin
          self.fPosition := pos;
          MoveToNode := true;          { Moved it successfully }
        end
      else                             { Current = nil--we didn't find the node }
        { Weird node, so halt }
        self.fErrHandler.FatalError(kBadListNode);
    end  { If fList <> nil }
  else
    { Error if list nil or node nil--something fishy about the node }
    begin
      if self.fList = nil then
        self.fErrHandler.SetError(kMoveInEmptyList)
      else if node = nil then
        self.fErrHandler.FatalError(kBadListNode);
      MoveToNode := false;
    end;
end;  { TLinkedList.MoveToNode }
```

Setting the mark to the desired node is easy. But then we have to scan the list to see whether the node is really there and post an error message if it isn't. If the node is there, we use our node count to reset *fPosition.* Moving within an empty list is no crime, but we do post an error message. Passing a *nil* node as the node to move to is considered fatal. Something is seriously wrong with the list.

Simple moves: *Previous, Next, Head,* and *Tail*

Moving to the previous node, or to the next node, is simply a matter of setting the mark equal to the current node's *fPrevious*, or *fNext*. The only thing to check is whether there is in fact a previous or next node to go to. If not, we issue an "offleft" (head) or "offright" (tail) error message. *Previous* and *Next* are simple and similar, so we'll look at *Next* only:

```
procedure TLinkedList.Next;
  var
    oldMark: TDoublyLinkable;
begin
  if self.fList = nil then
    { Error--tried to call Next in empty list }
    self.fErrHandler.SetError(kMoveInEmptyList);
  else
    begin
      self.fMark := TDoublyLinkable(fMark.fNext);
      if self.fMark <> nil then
        { Advance position, too }
        self.fPosition := self.fPosition + 1
      else
        self.fErrHandler.SetError(kOffRight);
    end  { If fList <> nil }
end;  { TLinkedList.Next }
```

If the list isn't empty, we move the mark to the next node and advance *fPosition* as well. If the mark would go off the end, we post an error message.

Moving the mark to the first node or the last is even simpler. We maintain references to the head of the list (*fList*) and the tail (*fTail*), and it's easy to tell what the new position value should be. If we're moving the mark to the head, *fPosition* becomes 1; if to the tail, *fPosition* equals the current value of *fLength*.

Using List Utility Methods

Most of the list utility methods are very simple. *IsEmpty* returns *true* if *fList* is *nil* or if *fLength* is 0. *LengthOf* returns the number of nodes, which is kept in *fLength*. *PositionOf* returns the node number of the current position of the mark, which is kept in *fPosition*. *HeadOf* returns *true* if the mark is at the first node in the list. *TailOf* returns *true* if the mark is at the last node in the list. *OffEnd* returns *true* if an attempt has been made to move past either end of the list ("offleft" or "offright"). These methods are so simple that we won't need to look at them here.

Cloning and Freeing the List

Two common tasks for any list are making a duplicate of it for some purpose and disposing of the whole thing when we've finished with it. We duplicate a list by using an override of *TObject*'s *Clone* method. Our method for disposing of a list is an override of *TObject*'s *Free* method.

Copying a list: *Clone*

Clone is a good deal more complicated than *Free*. To make an exact copy, *listB* of *listA*, we'd send

```
listB := TLinkedList(listA.Clone);
```

We need to do the typecast to *TLinkedList* because *Clone*, overridden from *TObject*, is a function that returns a *TObject*. We typecast the *TObject* to the *TLinkedList* we know it really refers to.

Here's the code for *TLinkedList.Clone*:

```
function TLinkedList.Clone: TObject;
  override;
  var
    temp: TLinkedList;
    current, rear: TDoublyLinkable;
begin
  { First, duplicate the list object and typecast it }
  temp := TLinkedList(inherited Clone);    { Nil if self is nil }
  { Then duplicate the underlying list }
  if self.fList = nil then
    self.fErrHandler.SetError(kCloneEmptyList)
  else
    begin
      { Clone the nodes, hooking them up to temp's fList }
      { Hook up the fNext references, cloning as we go }
      { This one call copies the whole list of nodes }
      { Nodes are copied locks and all }
      { We replace their fNext fields as we clone }

      temp.fList := TDoublyLinkable(self.fList.Clone);
      { But the copied nodes still have copied fPrevious references }
      { Move through new list and hook up fPrevious references }
      current := temp.fList;                { Current = the list head }
      current.fPrevious := nil;             {  whose previous is nil }
      rear := current;                      { Set a trailing pointer }
      { Advance current }
      current := TDoublyLinkable(temp.fList.fNext);
```

```
      while current <> nil do
        begin
          current.fPrevious := rear;        { Set current's previous }
          rear := current;                  { Advance pointers }
          current := TDoublyLinkable(current.fNext);
        end;  { while }
      { Then set internal references }
      if (self.fList <> nil) and (temp <> nil) then
        begin
          temp.Move(temp.fLength);          { Set the tail }
          temp.fTail := temp.fMark;
          temp.Move(fList.fPosition);       { Position the mark }
        end;  { if }
    end  { if fList <> nil }
  Clone := temp;
end;  { TLinkedList.Clone }
```

Cloning requires several steps:

1. Clone the list object.
2. Clone each of the list's nodes and hook a copy of the nodes to a copy of the list object. This leaves the *fPrevious* references in the nodes pointing back into the original list.
3. Fix the *fPrevious* references so that they point within the list copy. The *fMark* and *fTail* pointers are still duplicates of those in the original list, pointing to the original list's *fMark* node and *fTail* node.
4. Make the list copy's *fMark* and *fTail* references point to the marked node and the tail node.
5. Return the list copy, typecast to a *TLinkedList*.

First, we make a copy of the list object, assigning it to a *TLinkedList*-type variable called *temp*:

```
{ First, duplicate the list object and typecast it }
temp := TLinkedList(inherited Clone);
```

This code calls *TObject.Clone*, which copies *listA* (*self*). Again, we have to typecast the result. What exactly was copied by this call? The *TLinkedList* object and all of its instance variables, including the exact references in its *fList*, *fMark*, and *fTail* fields—so the copy, *temp*, has pointers into *listA*'s nodes. But *listA*'s nodes were not copied. We're saying that two separate *TLinkedList* objects reference exactly the same underlying list of nodes. We'll have to copy the nodes and install this duplicate list in the new *temp* object's *fList* field. Then the new list object will point to its own nodes, not those of *listA*.

We can clone the list of nodes by means of one statement, relying on recursion to do the work:

```
temp.fList := TDoublyLinkable(self.fList.Clone);
```

To temp's *fList* field, we assign the result of calling *TLinkable.Clone* (which we looked at earlier in this chapter) for *listA*'s (*self*'s) *fList* field. The result returned will be a whole chain of nodes. Recall our description of the recursive cloning process as the calls move down the list and then unwind back up it again, creating cloned nodes as they go. It smacks of hocus-pocus, but it works.

When everything unwinds, *temp*'s *fList* points to its own (nearly) exact copy of the list of nodes in *listA*. The contents of all fields in each node are the same, except for the *fNext* links, which have been reset. That includes each node's *TLockable* fields, *fLocked* and *fOwner*. All the copied nodes are still locked and still owned. Does it matter that the signature of *listB* will duplicate the one in *listA*? Not really. As long as all the references are correct, the signature is arbitrary—that the owner in each node match the signature of its parent list object is the only requirement.

What's different about the *temp* list? The *fNext* fields in all the nodes have been correctly set by the recursive process. But *TLinkable*'s *Clone* knows nothing about *fPrevious* fields. It doesn't even know that it's working on *TDoublyLinkable* nodes. So the *fPrevious* fields in the new node list still point off into *listA*. We'll have to go through *temp* and reset all the *fPrevious* fields, which is what this loop does:

```
current := temp.fList;                  { Current = the list head }
current.fPrevious := nil;               {  whose previous is nil }
rear := current;                        { Set a trailing pointer }
{ Advance current }
current := TDoublyLinkable(temp.fList.fNext);
while current <> nil do
  begin
    current.fPrevious := rear;          { Set current's previous }
    rear := current;                    { Advance pointers }
    current := TDoublyLinkable(current.fNext);
  end;  { while }
```

Using two auxiliary reference variables, *current* and *rear*, we move through *temp*'s nodes—*current* moves ahead, and *rear* trails it by one node. At each current node, we set its *fPrevious* to point to the node that *rear* points to.

Now *temp* has copies of all of *listA*'s instance variables, including a correct copy of the nodes referred to by *fList*. But *temp*'s *fMark* and *fTail*, as duplicates of the same fields in *listA*, still point off into *listA*, not into *temp*'s nodes. We have to find where to put the mark and tail references in *temp*. The original list's *fPosition* and *fLength* fields tell us where the mark and tail should be in *temp*. And the *Move* method allows us to put them in the right places in *temp*:

```
{ Then set internal references }
if (self.fList <> nil) and (temp <> nil) then
  begin
    temp.Move(temp.fLength);          { Set the tail }
    temp.fTail := temp.fMark;
    temp.Move(fList.fPosition);       { Position the mark }
  end;  { if }
```

Finally, with a complete copy of *listA* referenced by *temp*, we simply return *temp* as the function result—which takes us back to the beginning of this *Clone* discussion:

```
listB := TLinkedList(listA.Clone);
```

Disposing of a list: *Free*

TObject.Free simply frees the object calling it (*self*). It calls *DisposHandle* on the handle underlying the object reference, and the result is a now-undefined object reference. The Memory Manager reclaims the object's storage space.

But if *TLinkedList* simply inherits *TObject.Free*, a Free message will free only the list object, not its nodes. The nodes will be stranded, carving up space in the heap, causing fragmentation, and continuing to use up valuable master pointers. Figure 20-3 shows how not to free a list.

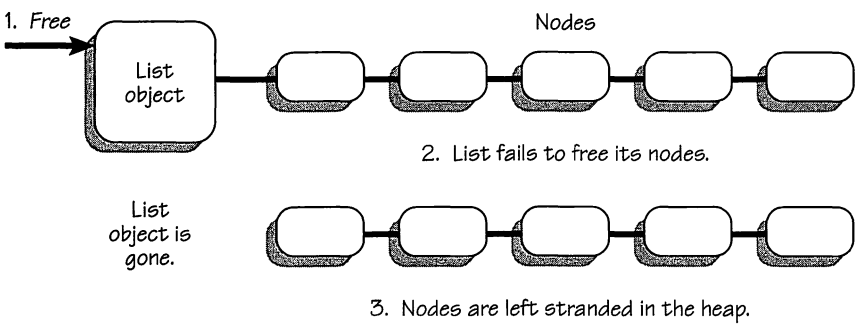


Figure 20-3.
The wrong way to free a list—freeing the list object without freeing the nodes.

We therefore have *TLinkedList* override *Free*. As we've seen before, this is a typical OOP practice for objects like *TLinkedList* that allocate their own subordinate objects and data structures. The overridden version of *Free* first frees up all the subordinate allocations and then calls *inherited Free* to dispose of *self*.

TLinkable.Free starts at the list's tail node and "walks" backward up the list to the head. The auxiliary object reference variable *prev*, of type *TDoublyLinkable*, walks ahead of the *fTail* reference, allowing us to reset the tail reference after each node is freed. At each node, operating through the *fTail* reference, we first unlock the node (passing it a copy of the list's signature) and then send the node a Free message.

Then we reset the *fTail* reference to the new tail node. When *fTail* becomes *nil*, all nodes are gone and we can call *inherited Free* to free the list object. Here's the code for *TLinkedList.Free*:

```
procedure TLinkedList.Free;
  override;
  var
    prev: TDoublyLinkable;
begin
  while self.fTail <> nil do
    begin
      prev := self.fTail.fPrevious;
      if not self.fTail.Unlock(self.fSignature) then
        self.fErrHandler.FatalError(kCantUnlockNode);
        self.fTail.Free;
        self.fTail := prev;
    end;  { while }
  inherited Free;
end;  { TLinkedList.Free }
```

Adding Nodes to the List

We have a variety of ways to add new nodes to or insert new nodes in the list. We can add at the head, add at the tail, insert before any given node (including the first), or add after (append to) any given node (including the last). The main concern is doing each step of the addition in the right order so that no part of the list is left dangling.

All the addition methods take a *TDoublyLinkable* object as a parameter and perform the necessary surgery on the list to break old ties between nodes and relink them to incorporate the new object in the right place. The list's *fPosition* and *fLength* values are adjusted, and if the addition takes place at the tail, the *fTail* reference is moved. In all cases, by convention, *fMark* is moved to refer to the newly added node.

All of our addition methods (*AddHead*, *AddTail*, *Insert*, and *Append*) basically have to reset the *fNext* and *fPrevious* fields of

- The node preceding the added node (if any—none at head)
- The added node
- The node following the added node (if any—none at tail)

Before adding a node, the list locks it, this way:

```
if not newNode.Lock(self.fSignature) then
  { Error-handling code }
```

We'll look at only one addition method. The others are quite similar.

Two problems with adding nodes

A problem occurs if you pass a locked node object to an addition method. You must unlock nodes before the list can add them—so that the list can lock them itself. All addition methods check for and reject a locked input object. Passing unlocked objects to the list is the using programmer's responsibility.

A second problem occurs if you fail to call *New* on every single object passed in. If you do something like this:

```
New(anObj);
anObj.Init;
for i := 1 to 100 do
  begin
    aList.AddHead(anObj);
    { Should call New(anObj) again here }
    {  and initialize the object, but we forget }
  end;  { for }
```

the same object gets added over and over again. So *fList* points to node 1, node 1's *fNext* also points to node 1, and nothing else happens. Without an error check, no matter how many times through the loop and how many AddHead messages, only one node gets added. If you try to traverse this list, the *fNext* references keep leading you back to the same object. A test,

```
if obj = fList
```

at the beginning of each addition method is designed to catch this error. It tests for whether the object is already in the list. Figure 20-4 shows the problem graphically.

Figure 20-5 on the next page shows the link-up process for the *AddHead* method. The other addition operations would have similar diagrams.

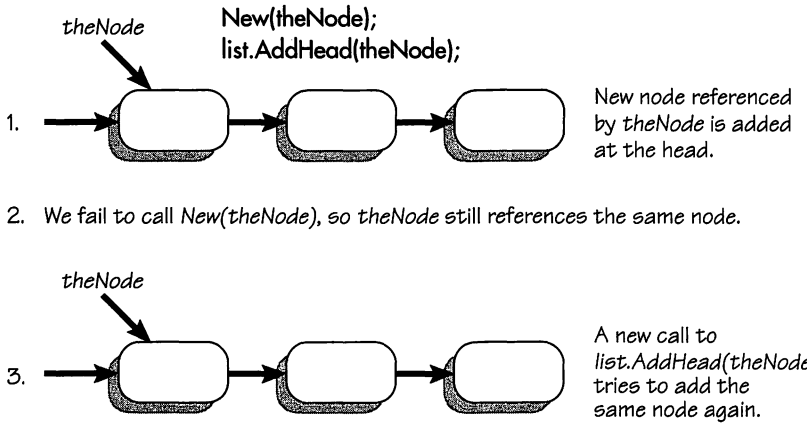


Figure 20-4.
Erroneous adding at the head—adding the same object over and over again.

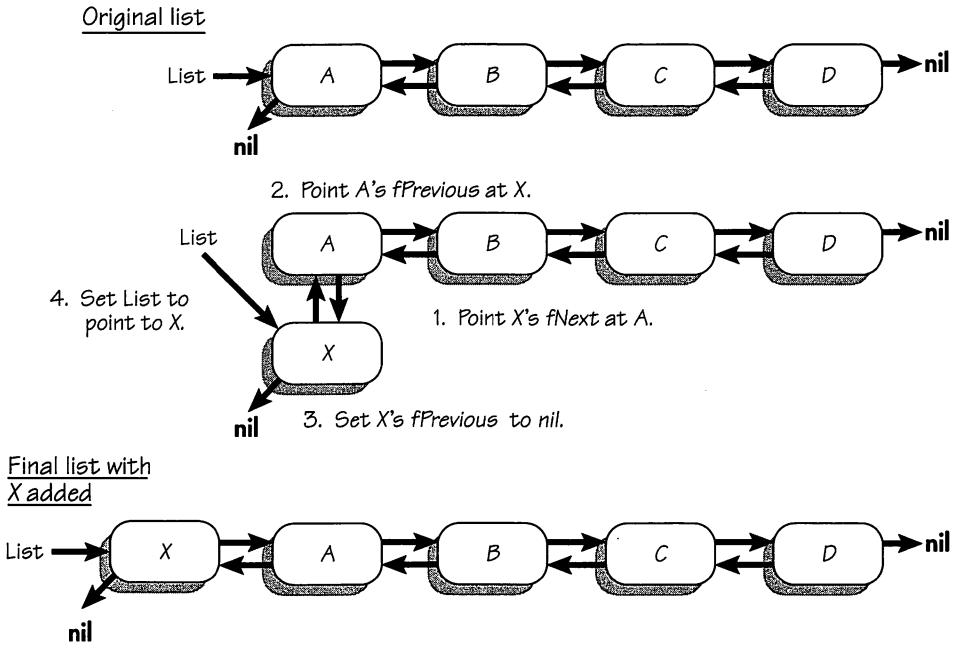


Figure 20-5.
Adding at the head.

Specialized addition methods: *AddHead* and *AddTail*

We provide separate *AddHead* and *AddTail* methods even though these aren't primitive operations. We could do *AddHead* by moving the mark to the head and inserting. We could do *AddTail* by moving to the tail and appending. But because these two spots are so frequently the targets of additions, we include methods especially for those cases. Their code is similar to the code for the *Insert* method.

Adding anywhere in the list: *Insert*

General insertion can occur at the head of the list or before any existing node. Inserting at the head of the list is a special variant of inserting before any existing node. A third case, inserting into an empty list, is either an *AddHead* or an *AddTail*. We call *AddHead*.

In *TLinkedList*, the mark must be positioned on the node following the position at which we want to insert. In other words, we insert at the position before the mark. We can move the mark to that position by several means, including using *Find*.

Inserting a new node before the old head node is just a case of *AddHead*, so we handle it by calling *AddHead*.

In the general case of inserting before any existing node, there's always a node before the new node and a node after the new node, so we have to handle lots of linking up. Here's the implementation of *TLinkedList.Insert*:

```
procedure TLinkedList.Insert (obj: TDoublyLinkable);
  { Insert object before the currently marked node, if any }
  var
    eh: TErrorHandler;
begin
  eh := self.fErrHandler;  { For convenience }

  { Case 1:  empty list }
  if (self.fList = nil) then
    self.AddHead(obj)      { Call self's method to do job }

  { Case 2:  mark at first item in list--insert new head node }
  else if (fMark.fPrevious = nil) then
    self.AddHead(obj)      { Call self's method to do job }

  { Case 3:  mark anywhere else--general insertion }
  else
    begin
      if obj = nil then
        { Passing nil object }
        eh.FatalError(kBadAddNode);
      else
        if obj.locked then
          eh.SetError(kCantAddLockedNode)
        else { OK to add }
          begin
            { Lock the object so that list can protect it }
            if not obj.Lock(fSignature) then
              eh.FatalError(kCantLockListNode);
            { Set all the necessary links }
            { Object's next and previous: }
            obj.fNext := self.fMark;
            obj.fPrevious := fMark.fPrevious;
            { Relink existing nodes to object: }
            fMark.fPrevious.fNext := obj;
            fMark.fPrevious := obj;
            { By convention, move mark to added node }
            self.fMark := obj;
            { fPosition stays the same }
            { fTail stays the same }
            self.fLength := self.fLength + 1;
          end;  { if obj <> nil }
    end;  { Case 3 }
end;  { TLinkedList.Insert }
```

The *Insert* method handles the same error conditions in the same ways as the other addition methods. In particular, the *AddHead* method checks whether you keep trying to insert the same node (caused by failing to call *New* each time) and, if you do, rejects the node and posts an error code. Then the program is halted. The list isn't damaged, but the node doesn't get inserted the second time (or a third, etc.). This is sufficiently serious to warrant stopping the program. It's also an error to pass in a locked node, but in that case, only an error warning is posted. Using programmers should check the results of an addition by calling the list's *ListError* method. That method calls the error handler's *Error* method. It's also an error to pass in a *nil* object reference to be added. That error is considered fatal.

Adding anywhere in the list: *Append*

Whereas *Insert* can add a node anywhere in the list except at the tail, *Append* can add one anywhere except at the head. *Insert* puts the new node before the marked node; *Append* puts the new node after the marked node.

We can append to an empty list, after the head or some intermediate node, or at the tail. We call *AddHead* to take care of appending to an empty list. We call *AddTail* to take care of appending to the tail.

The same locking and unlocking rules, and the same error conditions, apply as for *Insert*, and the code for *Append* is very similar to *Insert*'s code.

Removing and Accessing Nodes from the List

We've provided one way to remove nodes from a list and simply throw them away (*Delete*) and three ways to get at the data in nodes, two of which also remove a node from the list (*Extract* and *ExtractHead*). The third access method, *Peek*, lets us work with the node but leaves it in the list.

If the node at the mark is no longer needed at all, we can use the *Delete* method, which throws the node away without bothering to retrieve anything from it. If we want the node and its data for purposes outside the list, we can use the *Extract* method or a special case of *Extract*, *ExtractHead*. *Extract* clips the node at the mark out of the list and returns a reference to the node, thus handing it over. *ExtractHead* clips off the head node and gives it to us. Both methods also unlock the node, so we're entitled to free it if we like. Figure 20-6 shows the node removal process.

Peek returns a reference to a node without clipping the node out of the list. We can peek at the node's data while leaving the node in the list. We've already considered the philosophy and dangers of using *Peek*.

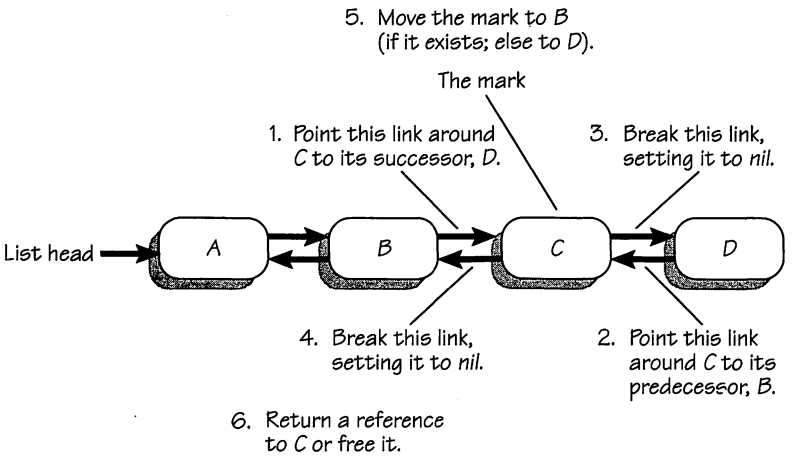


Figure 20-6.
List node removal—deleting the node at the mark.

In general, we can say that removing a node requires

- Unlocking the target node
- Resetting the previous node's *fNext* to point around the target to its successor
- Resetting the successor node's *fPrevious* to point around the target to its predecessor
- Resetting the *fNext* and *fPrevious* fields of the target node to *nil* so that no access into the list is possible from the removed node
- Resetting the mark—and, if necessary, the tail—to the preceding node if there is one, or else to the following node if there is one
- If the node is to be thrown away, calling its *Free* method; otherwise, returning a reference to it

Throwing away a node: *Delete*

Here's the implementation of *Delete*, the most general removal method:

```
procedure TLinkedList.Delete;
  var
    temp: TDoublyLinkable;
    eh: TErrorHandler;
begin
  eh := self.fErrHandler;
  { Case 0:  empty list--do nothing }
  if self.fList = nil then
    eh.SetError(kRemoveFromEmptyList)
```

```
    else
      begin
        { Case 1:  mark at head node }
        if self.fPosition = 1 then
          { Clip off head node }
          { Mark moves to next node; fPosition remains set to 1 }
          {  unless this is only node in list }
          begin
            temp := self.ExtractHead;      { Use a method }
            temp.Free;                     { Throw node away }
          end  { If case 1 }

        { Case 2:  mark at tail node }
        else if self.fMark = self.fTail then
          begin
            { Clip off last node and sew up loose ends }
            { Mark moves to previous node unless this is only node in list }
            temp := self.fMark;
            self.fMark := temp.fPrevious;
            fMark.fNext := nil;
            self.fTail := self.fMark;
            if not temp.Unlock(fSignature) then
              eh.FatalError(kCantUnlockListNode);
            temp.Free;
            self.fPosition := self.fPosition - 1;
            self.fLength := self.fLength - 1;
          end  { If fMark }

        { Case 3:  mark anywhere else  }
        else
          begin
            { Clip out marked node, bypassing it fore and aft }
            { Throw away node; mark moves to previous node }
            temp := TDoublyLinkable(fMark.fNext);
            if not fMark.Unlock(self.fSignature) then
              eh.FatalError(kCantUnlockListNode);
            temp.fPrevious := fMark.fPrevious;
            temp := self.fMark;
            self.fMark := fMark.fPrevious;
            fMark.fNext := temp.fNext;
            temp.Free;                     { Throw it away }
            self.fPosition := self.fPosition - 1;
            self.fLength := self.fLength - 1;
          end;  { If case 3 }
      end;  { If fList }
    end;  { TLinkedList.Delete }
```

General deletion calls for a big method because there are four cases to deal with:

1. If the list is empty, we do nothing (except to post an error code so that the caller can learn that an attempt was made to delete from an empty list); otherwise, we could be deleting the first node, the last node, or some node in between.
2. To get rid of the head node, we can call an *ExtractHead* method, which returns a reference to the node after clipping it off the list; then, because the node has been safely unlocked in the process, we can send the node a Free message.
3. To get rid of the tail node, we unlock it, "cauterize" the preceding node's *fNext* by setting it to *nil*, and send the node a Free message.
4. To get rid of an intermediate node, we make the *fNext* and *fPrevious* references of its predecessor and successor point around it, thus clipping it out. Then we send the node a Free message.

Retrieving a node: *Extract* and *ExtractHead*

We provide two methods for removing a node from the list so that we can use it for some other purpose. *Extract* removes the node currently marked. And *ExtractHead*, of course, removes the head node.

Extraction is the same as deletion except that, instead of freeing the removed node, we return a reference to it. We also have to take more care about "cauterizing" the node's *fNext* and *fPrevious* references so that they won't give unexpected and possibly dangerous access into the list. Once the node is out, it should have no hidden connections to the list.

Examining a node without removing it: *Peek*

Compared to the removal methods, *Peek* is simple. It returns a reference to the node pointed to by *fMark*. If *fMark* happens to be *nil* because the list is empty, *Peek* returns *nil*.

The node being peeked at is not removed from the list. It is still locked, though, unlike the nodes returned by *Extract* and *ExtractHead*. Thus, we can't send Free to a node using the reference we've gotten with *Peek*. But we can do anything else to the node—update its fields, display it, and so on. (We could write a variation of *TLinkedList* to put lock protection on other node methods, to prevent their use if the node is locked. We might not want a node to be updated by means of *Peek*.)

Peek doesn't prevent our accessing other nodes in the list by using the *Peek* reference to call the node's *NextOf* or *PreviousOf* methods. But those nodes are locked, too, so we won't be able to free them, either, although we can update them. But updating other nodes by means of a reference returned by *Peek* is a little perverse. We'd surely want to have a good reason for updating that way.

We'll see some other uses for *Peek* later on.

Costs of Using *TLinkedList*

Before we leave *TLinkedList* and move on to *TList*, let's briefly consider the costs of using a *TLinkedList* versus the costs of using a *TList* or a traditional linked list with object-type fields. We'll call the traditional list *TOldList*. A *TOldList* object is a manager object that operates an underlying linked list of record-type nodes that are linked with ordinary Macintosh handles. Each record-type node has a data field of type *TObject* (or *TLockable*, to make the comparison really fair). *TOldList* is functionally equivalent to *TLinkedList*. Here are the declarations for *TOldList*'s underlying list of record-type nodes:

```
type
  Node = record
      data: TLockable;{ Object-type data here }
      nextNode, prevNode: NodeHandle;
    end;  { Record Node }
  NodePtr = ^Node;
  NodeHandle = ^NodePtr;
```

Speed

List traversal in methods such as *Find* and *Display* would be done differently in each of our three lists. In *TOldList*, to move to the next node we would doubly dereference the *nextNode* handle:

```
currentNode := currentNode^^.nextNode;
```

In *TLinkedList*, the same maneuver requires a method call:

```
currentNode := currentNode.NextOf;
```

although we can optimize that a bit by intentionally violating encapsulation:

```
currentNode := currentNode.fNext;
```

Traversing with method calls is slower, of course. The double dereference is made efficient by the compiler. But the second version for *TLinkedList*, which violates encapsulation, is equivalent to the *TOldList* version because a double dereference of the object reference is done for us, behind the curtain.

Traversing a *TList*, on the other hand, is done with array-access techniques. The method *TList.Display*, for instance, might contain code such as this:

```
for current := 1 to self.fLength do
  self.fArray^^[current].Display;
```

This code can be optimized a bit by putting the double dereference outside the *for* loop as in the following code.

```
var
  list: HArray;
  current: Integer;
:
list := self.fArray^^;
for current := 1 to self.fLength do
  list[current].Display;
```

To get at the elements (nodes) of a *TList*, we have to doubly dereference the handle that points to the start of our dynamically allocated array. Then we can access individual elements with array-access notation. Once the dereferencing is done, the rest of the traversal is very fast. Other list-access operations, such as adding or deleting nodes, are similar.

Size

Surprisingly, *TLinkedList* fares very well against *TOldList* when it comes to storage requirements per node.

TOldList

We'll first add up the storage costs of a *TOldList* node. Figure 20-7 shows the structure of a *TOldList* node beside the structures of *TLinkedList* and *TList* nodes.

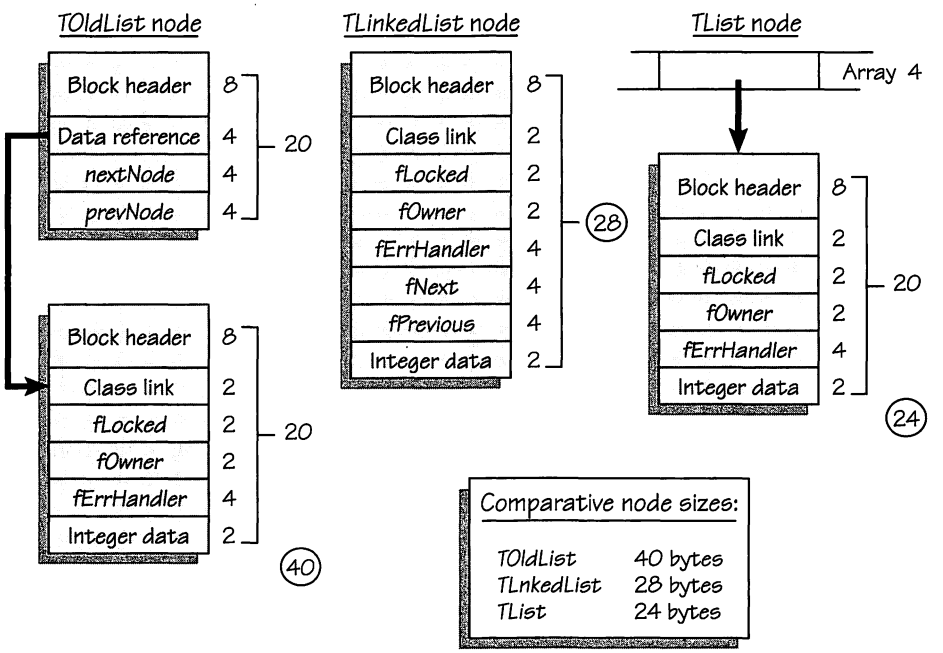


Figure 20-7.
Node structures and the relative storage requirements for TOldList, TLinkedList, *and* TList *nodes.*

Each *TOldList* node actually takes up two blocks of storage in the application heap. The node block has an 8-byte block header, a 4-byte data field, and two 4-byte handles for the next and previous links. So far, the node needs 20 bytes. But the data field is an object reference variable, so it points to the data object—let's say a *TInteger*. The *TInteger* object pointed to by the node also has an 8-byte block header, plus a 2-byte class link and 2 bytes for the integer data. That's an extra 12 bytes. So far, 32 bytes. But each piece of data stored in a node should really be lockable. If it isn't, freeing the data item will cause runtime errors when we try to traverse the list, sending messages to the nodes' data objects. *TLockable* requires another 4 bytes, 2 for *fLocked* and 2 for *fOwner*, plus 4 more bytes for an error-handler object. The grand total cost per *TOldList* node is 40 bytes.

TLinkedList

TLinkedList needs only 28 bytes per node. That's right—it's 12 bytes cheaper per node than *TOldList*. If we had a list containing 1000 *TInteger* items, the difference would amount to 12,000 bytes—hardly insignificant.

The main space saver is the fact that a *TLinkedList* node requires only one block in the heap. Not only do we save space, but the heap is less cluttered and fragmented and we save master pointers.

Each *TLinkedList* node has a block header of 8 bytes, a 2-byte class link, 8 bytes for locking and error handling, 8 bytes for forward and backward links, and 2 bytes for the integer data. That's a total of 28 bytes per *TLinkedList* node.

TList

How does *TLinkedList* compare with the *TList* class we'll develop in the next chapter? *TList* is cheaper still: 4 bytes better than *TLinkedList*.

Each *TList* node needs an 8-byte block header, a 2-byte class link, 8 bytes for locking and error handling, and 2 bytes for the integer data. In addition, each node is referenced by an element of the array, which is a 4-byte object reference. That's 24 bytes per *TList* node, a full 16 bytes fewer than *TOldList*.

The main reason for this low overhead is that *TList* doesn't link objects together in a list. *TList* stores objects in an array and doesn't need forward and backward references, saving 8 bytes per node.

TList's economy is one of the main reasons for choosing it as our main list component. But that's not to say that a good class library shouldn't have all three lists—*TList*, *TLinkedList*, and *TOldList*—and more.

Before we abandon our consideration of *TLinkedList*, we should entertain a good reason for including all three lists in a library: With either *TLinkedList* or *TOldList*, you can build lists as long as you like, subject only to limits on available memory. *TList*, our dynamic array-based list, despite its ability to grow and shrink as needed, faces a much more constrained upper limit of about 2000 nodes. *TList* could be

modified to hold more nodes than that, but then it would only face a new limit, even if there were lots of RAM available. *TList* is faster and smaller, making it suitable for many listing needs. But for some kinds and volumes of data, *TLinkedList* or even *TOldList* might be the better choice.

Summary

This completes our implementation of *TLinkedList* as a reusable software component—except for the search, traversal, and error methods. We'll leave those as projects to be done after you've read the discussion and seen the fairly similar code for *TList* in the next few chapters.

Projects

- Complete the implementation of *TLinkedList*. You might want to study Chapters 22 and 23 before attempting the *EachItem* method.
- Design and implement a general stack component by developing its methods along the lines of *TLinkedList*'s. Try to incorporate lockability because a stack is no less vulnerable to outside access than a list is.
- Design and implement a general queue component with locking.

21

IMPLEMENTING THE *TLIST* COMPONENT

TList, the list component we'll implement fully, is based on a dynamic array.

When a list is array based, its elements reside, not in linked nodes, but in the elements of an array. No forward or backward links are needed. A dynamic array itself resides inside a dynamically allocated block of storage in the heap, referenced by a Macintosh handle.

On our way to implementing our *TList* component, we'll explore

- Designing a hierarchy rather than a class
- Implementing an array in a heap block
- Implementing a dynamic array class
- Implementing a dynamic list class based on the array class

We'll postpone implementing the *Find*, *DoToEach*, and *EachItem* methods of class *TList*. We'll get to those in Chapters 22 and 23.

The *TList* Demonstration Program

You can see a thoroughgoing *TList* demonstration program on the code disk. To compile the program, set up your files in the order in which they appear in the project window shown in Figure 21-1 on the next page.

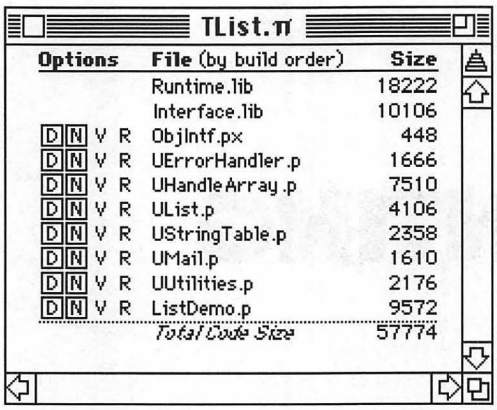


Figure 21-1.
The TList *demonstration project window.*

Files

As usual, the demonstration program project includes the files Runtime.lib, Interface.lib, and ObjIntf.px. The first two provide Pascal and Macintosh routines. ObjIntf.px contains our enhanced version of the *ObjIntf* unit, providing class *TObject*. The project also includes the unit *UErrorHandler*, which defines the same class *TErrorHandler* that we used in PicoApp.

The two key files are UHandleArray.p and UList.p. The unit *UHandleArray* defines a class *THandleArray*, which implements a dynamic array. *UList* defines the *TList* class built atop the dynamic array.

The remaining files contain parts of the demonstration code. The code in UString-Table.p demonstrates a way to use the list class: Class *TStringTable* is descended from *TList* to provide a lookup table for strings. From *TStringTable*, we derive a *TMonthTable* class, which lists strings that name the months of the year. The user can look up a month by index number (1..12) and get a string containing its name or look up a month by name and get its index number. These tools were easy to build from the *TList* class.

UMail.p contains mail classes—*TMail*, *TLetter*, and *TMemo*—to provide data types for the list demonstration program. UUtilities.p provides several utility classes: a search key class, "traversal action" classes, a mail list derived from *TList*. (We won't get to some of those classes until Chapters 22 and 23.) ListDemo.p contains the main program, which creates instances of lists and mail and puts the lists through their paces. The results appear in THINK Pascal's Text window.

TList Design Goals

TList is the equivalent of a linked list—but without the links—implemented inside an array. The array can grow as needed, up to a maximum.

We'll first need to develop a resizable array, and then we'll need to put a list inside the array.

To implement the *TList* array, we have to figure out its dynamic (listlike) aspects and its array aspects. We'll look at how to allocate an array dynamically, at runtime, and at how to access its elements.

To implement the list, we have to work out the details of using array elements to store list elements. We'll look at how the logical list fits into the physical array.

Using a Hierarchy

The list will be based on the array, but something should be nagging at your awareness along about now: We keep talking about two different (but related) entities—the array and the list.

Our initial plan might have been to hide the array mechanism inside the list mechanism. The using programmer would see a list interface much like *TLinkedList*'s, but he or she wouldn't have to be aware of the array details underneath.

But recall the advice in Chapter 7 to design hierarchies, not simply classes. Why not think in terms of two classes—an array class and a list class—related by inheritance? That's exactly what we'll do.

Class *TList* descends from an immediate ancestor called *THandleArray*, which implements the array. *TList* inherits all the array capabilities it needs. It overrides methods that are more array oriented than list oriented. *THandleArray* is really an abstract class, despite the amount of functionality in its methods. It's abstract because we don't provide any methods to put data into or get data out of the array. Each subclass provides its own input and output methods. We'll derive another class from *THandleArray*—besides *TList*—that will be a true dynamic array. Figure 21-2 on the next page shows the class hierarchy for the list demonstration program, including the *THandleArray-TList* hierarchy.

This approach offers several advantages:

- *TList*'s interface stays cleaner. The *TList* class declaration includes only list methods. *THandleArray*'s methods are inherited but "out of sight and out of mind." This isn't exactly information hiding, but it does look better.
- *THandleArray* is available to support other list implementations or other array-based classes. We'll look at one such class, a true dynamic array called *TDynamicArray*, written as a subclass of *THandleArray*.

- *THandleArray* provides many services but doesn't tie a subclass's hands. *THandleArray* takes care of expanding and constraining the type(s) the array can hold, freeing, cloning, and numerous other helpful matters. But it leaves each subclass free to use the array in its own way.

We'll develop *THandleArray*, the dynamic array class, first, and then we'll develop *TList* from it. We'll look at the most interesting methods for each class. The full code is available on the code disk in the folder Lists, Part 3. In passing, we'll also look at class *TDynamicArray*.

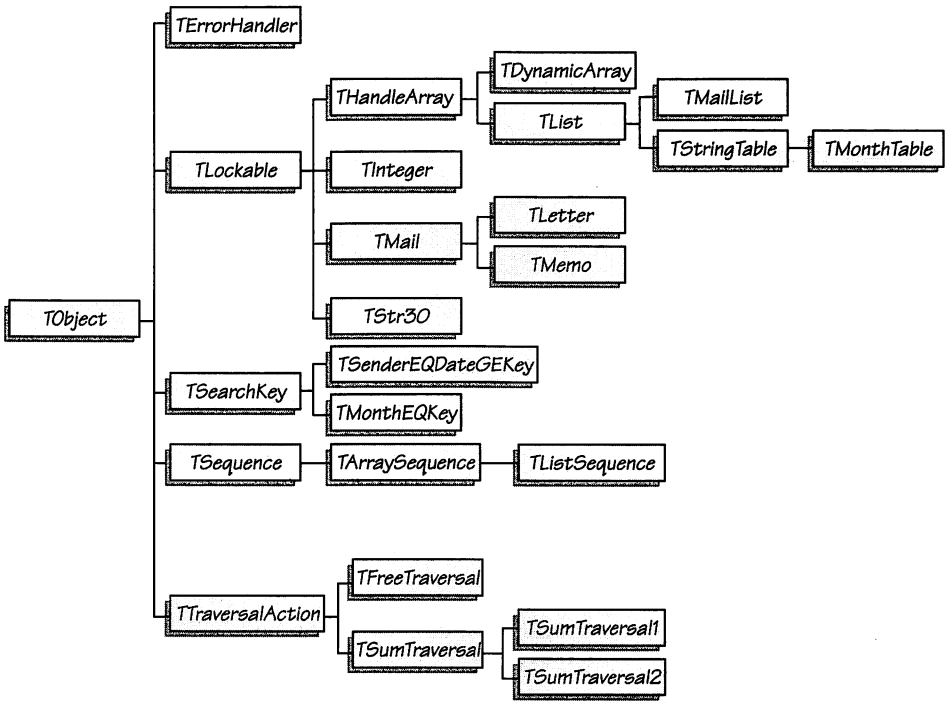


Figure 21-2.
List demonstration hierarchy, with THandleArray *and* TList.

Building a Dynamic Array

When we use a dynamic array, we access data elements in it with array-access notation something like this:

```
dynamicArray[i] := x;
y := dynamicArray[j];
```

But that code is incomplete because *dynamicArray* is not an ordinary array at all. It sits at the end of a handle, so, to access it, we first have to doubly dereference the handle—and then use array-access notation:

```
dynamicArray^^[i] := x;
y := dynamicArray^^[j];
```

The variable *dynamicArray* is the name of a handle to an array, not the name of the array itself. The actual array has no name at all, except *dynamicArray*^^.

Keep in mind that we're dealing with handles, not object references; hence, the explicit double dereference.

Allocating the Array

To allocate the array initially, we use a Toolbox function called *NewHandle*:

```
fArray := ArrayH(NewHandle(Size(fSlots * kSlotSize)));
```

In this statement, *fArray* is the handle to the array and *fSlots* is the number of slots or elements to be allocated. The constant *kSlotSize* gives the size, in bytes, of an array element. Our elements are to hold object references, so they're 4 bytes long. *NewHandle* returns a value of the generic *Handle* type, so the code must typecast the returned handle to *fArray*'s actual type, *ArrayH*. The value of *(fSlots * kSlotSize)* is an integer, which we typecast to type *Size* (a Mac type equivalent to a long integer) to meet *NewHandle*'s parameter requirements.

If *fSlots* is *20*, a handle pointing to a heap block of 20 * 4, or 80, bytes will be allocated. The block has an 8-byte block header in addition to the 80 bytes, but that's transparent to us.

Figure 21-3 shows what a handle-based array looks like schematically.

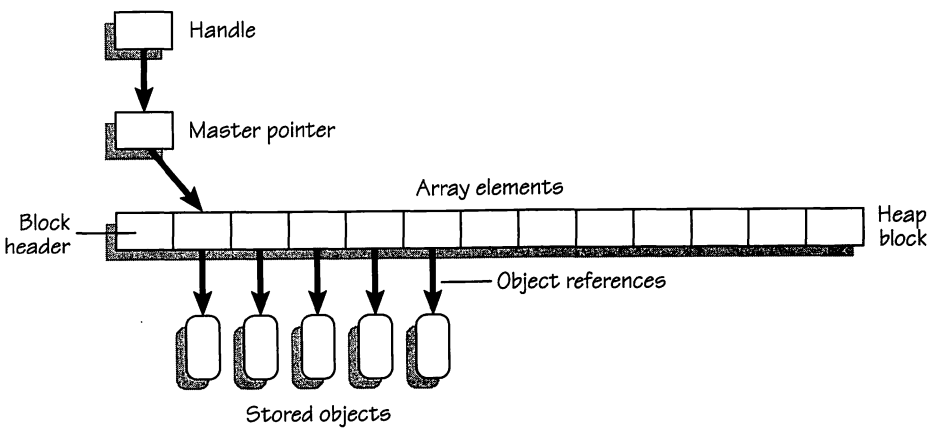


Figure 21-3.
A handle-based array in the heap.

Expanding the Array

To expand the array so that it can hold more data, we resize the handle with a Toolbox procedure called *SetHandleSize*:

```
SetHandleSize(Handle(fArray), Size(fSlots * kSlotSize));
```

Again, we must typecast the parameters. The amount of growth is determined by the value of *fSlots*. To increase an array of 20 elements by adding 10 more elements, for example, we'd give *fSlots* the value

```
fSlots := fSlots + 10;
```

and call the *SetHandleSize* Toolbox procedure. Now the array's heap block takes up 30 * 4, or 120, bytes, plus its block header.

We can also use *SetHandleSize* to shrink the array. *THandleArray* subclasses can use the *Grow* and *Shrink* methods to keep the allocated size of the handle in tune with the array's current contents. In *TList*, for example, addition methods automatically expand the array as it gets filled up and shrink it as items are deleted. To make expanding and shrinking more efficient, the *Grow* and *Shrink* methods both work with blocks of array elements. The *PrivateAdd* utility procedure, which calls *Grow*, allocates *kGrowAmount* elements, not just one. *TList.Extract*, which uses *Shrink*, keeps track of deleted elements and shrinks the array only after *kGrowAmount* elements have been deleted. Then it starts counting deletions again from scratch. In the unit *UHandleArray*, we set *kGrowAmount* to *20*.

Making a Heap Block into an Array

We've seen how to work with the handle to our array. Now let's figure out how to declare the necessary variables to make a dynamic array—when we can't predict the array's size.

We begin with declarations something like these:

```
type
  MyArray = array[kLowerLimit..kUpperLimit] of BaseType;
  MyArrayPtr = ^MyArray;
  MyArrayHdl = ^MyArrayPtr;
```

The trick is to specify a large limit for the array's upper bound. The upper limit is the maximum size of the array—if we expand it as far as possible. We'll set the value of *kUpperLimit* to the result of the expression

```
(maxint div SizeOf(Longint)) - 1
```

which on the Macintosh equals approximately *8191*. The constant *maxint* on the Mac is *32,767*. A *Longint* is 4 bytes wide and is the same size as the object references contained in the array. The base type of our array is *TObject*.

Figure 21-4 shows the relationships among the array's actual size at some point, the size for which space is allocated, and the theoretical, or potential, array size.

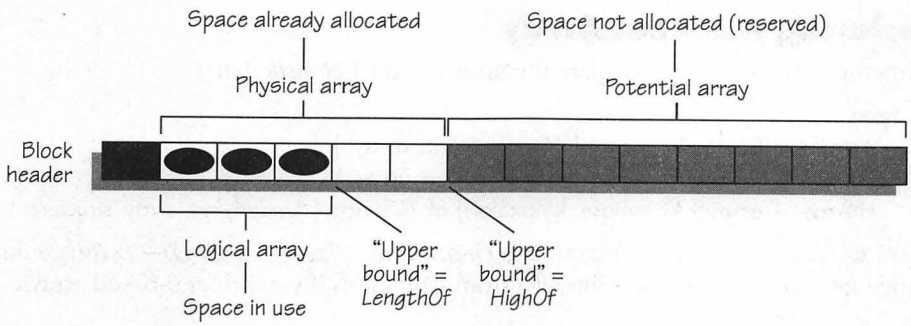


Figure 21-4.
Structure of a dynamic array.

It's helpful to think of the portion of the array structure currently in use (up to the highest occupied slot) as the logical array, the space allocated for the array as the physical array, and the possible maximum size that could be allocated as the potential array. The logical array might fill the entire physical array, or it might use only part of it. The upper limit of the potential array is *kUpperLimit*; we'll provide instance variables and methods (*LengthOf* and *HighOf*) for working with both logical and physical arrays.

Most of the time, the physical array will have far fewer than the potential maximum number of elements—and the logical array might or might not use the elements available in the physical array. At full capacity, the array can hold fewer elements than a linked list, which is limited only by the amount of free memory available to allocate nodes and by the amount of space available to store master pointers. But even though the array has a lower maximum capacity than, say, a *TLinkedList*, 8191 elements should be suitable for a great many programming needs.

Our value *kUpperLimit* is somewhat arbitrary. THINK Pascal limits the size of all statically declared variables in a block (program or procedure) to 32 kilobytes. The dynamic array isn't really limited by that figure, however—if we declare a handle to a dynamic array, only the size of the handle variable itself (4 bytes) counts against the 32-kilobyte limit for that block. We could actually set *kUpperLimit* to something like *99999 div 4*, for instance, allowing approximately 25,000 4-byte elements. The 99,999 bytes, even if the whole array capacity were used, would not contribute to the 32-kilobyte limit for the block in which the array handle is declared. But we'll select an upper limit ample for most purposes. A complete class library might include handle-based arrays with larger (and perhaps smaller) capacities, as well as arrays based on other base types, such as *Integer* or *Char*. (A dynamic *Char* array would be useful for implementing dynamic strings. See the projects at the end of the chapter.)

Declaring the Real Array

Here's the way we actually declare the array in unit *UHandleArray*:

```
type
  ArrayH = ^ArrayP;          { Handle to an array }
  ArrayP = ^HArray;          { Pointer to an array }
  HArray = array[kArrayBase..kMaxSlots] of TObject; { Underlying array structure }
```

where *kArrayBase* is *1* and *kMaxSlots* is *(maxint div SizeOf(Longint)) – 1*. You could change *kArrayBase* to 0 and subtract 1 from *kMaxSlots* if you prefer 0-based arrays.

Locking the Array Elements

We've gotten a bit tricky with the base type of our array—we can have our cake and eat it too. By declaring a base type *TObject* and making locking optional instead of mandatory, we increase the array's flexibility. Sometimes we might not mind letting our array or list elements stay unlocked (and endangered by Free calls through outside references). On those occasions, we can create an array object and call its *SetLocking* method to turn off element locking. Then we can store any object descended from *TObject* in the array, regardless of whether it's a *TLockable*. By default, the array locks its elements—in which case, the objects we try to put into the array must be descended from *TLockable* so that they'll have the right locking and unlocking methods. The example list program on the code disk shows *TList* instances using both approaches.

Managing the Array

The array itself is fairly complicated, given that we have to doubly dereference a handle to reach the array before we can start using array index values in brackets. So we'll encapsulate the raw dynamic array in a manager object class, called, appropriately, *THandleArray*. A *THandleArray* object manages an underlying dynamic array, as shown in Figure 21-5.

To do its job, a *THandleArray* object has instance variables for tracking the array's current size and state and methods for manipulating the array. Consumers using the dynamic array use it only through *THandleArray*'s methods.

What methods does *THandleArray* need? Consumers need to be able to expand, shrink, clone, and free the array. Because the array itself is encapsulated, consumers may also need methods to search the array for an element with a given value and to traverse the array sending messages to its elements. We'll postpone searching and traversing until Chapters 22 and 23, but we'll develop *Grow*, *Shrink*, *Clone*, *Free*, and other useful methods now.

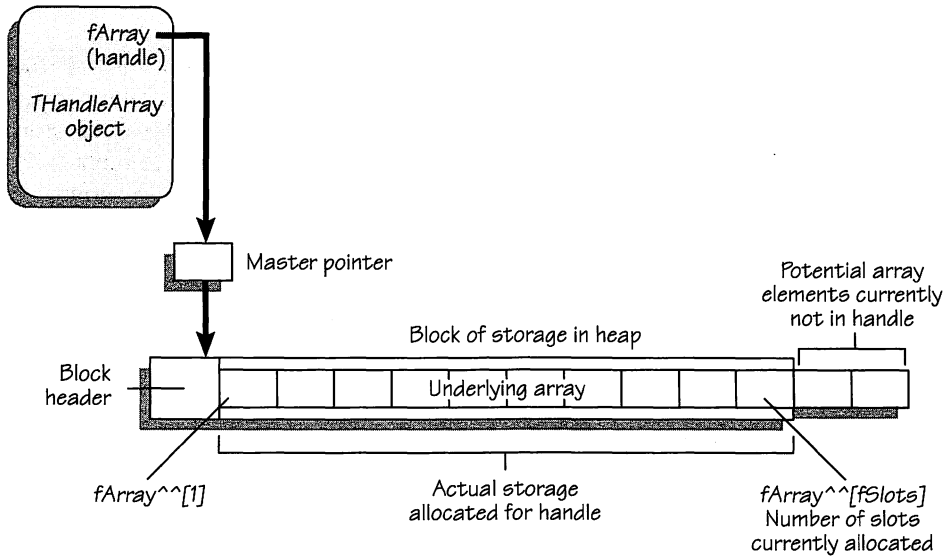


Figure 21-5.
A THandleArray *object.*

The THandleArray Class

Class THandleArray is too long to list here, but you can see it in its full glory on the code disk.

The most important of its 27 methods are *LengthOf* and *HighOf*, which give different versions of the array's size, and *Grow* and *Shrink*, which allow the array to change its space allocation as conditions require.

Initializing the Array

The *IHandleArray* method does a great deal, but we'll focus on how it creates the underlying array. The method receives two parameters, *canGrow* and *canShrink*, that govern its ability to resize. The array is created initially with this code:

```
self.fArray := ArrayH(NewHandle(Size(self.fSlots * kSlotSize)));
```

The number of slots to allocate depends both on the *wantSlots* parameter to *IHandleArray* and on the maximum size allowed. Once the method determines how many slots it can really allocate, it calls *NewHandle* to allocate that many slots of the right size—in this case 4 bytes each (to hold object references). The resulting handle is cast to the *ArrayH* type and stored in the array object's *fArray* instance variable. The array elements are initially empty (*nil*).

Getting the Array's Size

The array's "length" is defined as the index value of the highest array slot in use (that is, the extent of the logical array). We might have 50 elements allocated (the physical array) but use only 40 of them. *LengthOf* would return 40, and *HighOf* would return 50—the number of elements allocated. Each value has its uses. Array elements allocated but not in use store the value *nil*. Figure 21-6 illustrates the meanings of *LengthOf* and *HighOf*.

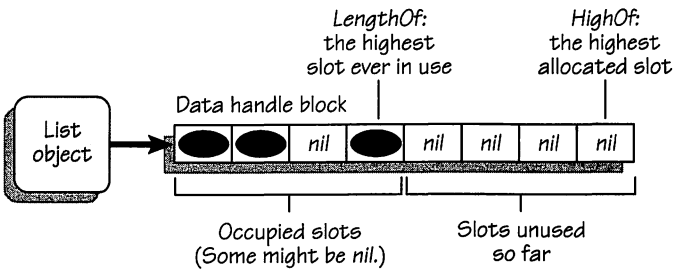


Figure 21-6.
LengthOf *and* HighOf *in an array.*

An empty array is defined as one with no elements allocated. The *fArray* instance variable is *nil*, and *fSlots* is *0*. *IsEmpty* returns *true* in this case. *IsFull* returns *true* if all possible space has been allocated (regardless of whether the elements are actually in use). A full array can't grow anymore.

Free and FreeAll

Objects stored in the array might be referred to by outside reference variables as well as by the list. Or they might have no such outside links. In the latter case, it's safe to free the data objects. But if there are valuable references to them from outside, you might prefer to destroy the list but not its data. *THandleArray* therefore provides two *Free* methods.

Free frees only the list object, not the data. *FreeAll* frees everything—the list object and the stored data items. Figure 21-7 illustrates the difference between the two methods.

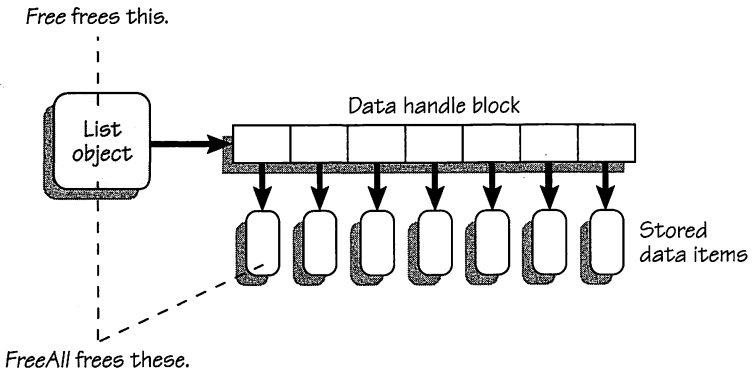


Figure 21-7.
Free *and* FreeAll.

Putting Data In and Getting It Out

Short of violating the array object's encapsulation, there's no way in *THandleArray* to put data into the array or get data out. *THandleArray* is really an abstract class, not meant to be instantiated (although it's a pretty detailed and full-functioned abstract class). One subclass of *THandleArray*, of course, will be the *TList* class that we develop later in the chapter. Another subclass might be an actual dynamic array object, with methods to store data in the array and to retrieve it again. Figure 21-8 illustrates this basic *THandleArray* hierarchy:

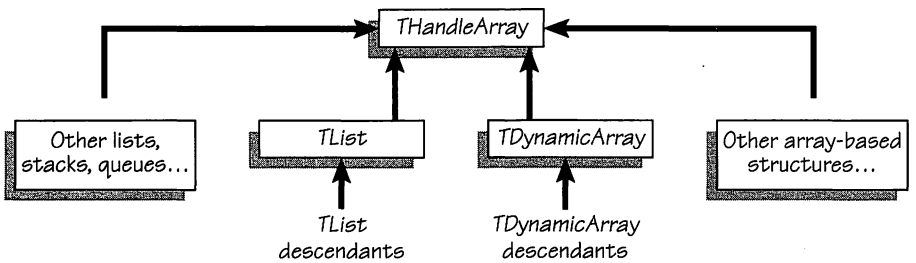


Figure 21-8.
THandleArray *and its descendants.*

Here's a simple *TDynamicArray* class declaration:

```
type
  TDynamicArray = object(THandleArray)

      procedure TDynamicArray.IDynamicArray(wantSlots: Integer;
          canGrow, canShrink: Boolean; eh: TErrorHandler);
        { Calls IHandleArray }
```

```
      procedure Put (elem: TObject; atIndex: Integer);
        { Puts element in the slot atIndex if atIndex is not out }
        {  of range }
        { If out of range, posts an error message }
      function Get (fromIndex: Integer): TObject;
        { Returns reference to element at fromIndex if not out }
        {  of range }
        { If out of range, returns nil and posts an error message }
        { You must typecast the result to what you expect }
      function TypeOf: Str30;
      override;
        { Returns the string 'TDYNAMICARRAY' }
    end;  { Class TDynamicArray }
```

The added methods are simple:

```
procedure TDynamicArray.Put (elem: TObject; atIndex: Integer);
begin
  if self.fErrorHandler <> nil then
    begin
      self.fErrorHandler.SetError(kNoErr);
      if not self.TypeMatches(elem) then
        self.fErrorHandler.SetError(kTypeNotAllowed)
      else if ((atIndex < kArrayBase) or (atIndex > self.fSlots)) then
        self.fErrorHandler.SetError(kOutOfRange)
      else
        self.fArray^^[atIndex] := elem;
    end;
end;  { TDynamicArray.Put }
```

and

```
function TDynamicArray.Get (fromIndex: Integer): TObject;
begin
  if self.fErrorHandler <> nil then
    begin
      if (fromIndex < kArrayBase) or (fromIndex > self.fSlots) then
        begin
          Get := nil;
          self.fErrorHandler.SetError(kOutOfRange);
        end
      else
        Get := self.fArray^^[fromIndex];
    end
  else
    Get := nil
end;  { TDynamicArray.Get }
```

We won't look at *IDynamicArray* or *TypeOf*.

Let's briefly see how we might use a *TDynamicArray*. The following code fragment shows traversing the array to do something useful at each element, putting data into the array at a particular index position, and getting data out of the array from a particular index position.

```
var
  anArray: TDynamicArray;
  data: TInteger;
  i: Integer;
⋮
{ Create an array object for 20 objects }
{ The array can grow and shrink, and we accept the default }
{  error handler }
anArray := NewDynamicArray(20, true, true, nil);
New(data);                              { Create a TInteger data object }
data.IInteger(3);
anArray.Put(data, 14);                  { Put object in array }
New(data);
data.IInteger(5);
anArray.Put(data, 15);
for i := kArrayBase to anArray.HighOf do { Traverse array }
  begin
    data := TInteger(anArray.Get(i));   { Get data at i }
    if data <> nil then                 { Use data's version of Display }
      data.Display;                     { Could call anArray.Display instead }
  end;  { for }
anArray.FreeAll;                        { Free array and data }
⋮
```

We'll have more to say about the uses of *TDynamicArray* in Chapter 25.

Searching and Traversing

We'll postpone implementing the searching and traversing methods until Chapters 22 and 23. The traversal part should be easy to do, though, in an array.

Storing Non-Object Data and Screening Data Types

THandleArray provides a method that lets you admit only the data types you choose. You can subclass *THandleArray* (or one of its descendants) and override the *TypeMatches* method. *TypeMatches* is unimplemented in *THandleArray*, but you could define it to screen out all types except, say, *TInteger*.

```
function TMyList.TypeMatches(elem: TObject): Boolean;
begin
  TypeMatches := Member(elem, TInteger);
end;  { TMyList.TypeMatches }
```

Note that *TypeMatches* could screen for multiple data types as easily as for one—it's all in how you code the test.

For an array or list class to take advantage of *TypeMatches*, it must call *TypeMatches* from all methods that add items. An item should be added only if the call to *TypeMatches* returns *true*. By default, the *THandleArray* version of *TypeMatches* (which *TDynamicArray* and *TList* inherit unchanged) returns *true*—so it admits all data. See the discussion of constraining lists in Chapter 18.

Cloning a *THandleArray*

To duplicate a *THandleArray* object, call its *Clone* method, as we do here for a *TList* object:

```
aCopy := TList(aList.Clone);
```

Be sure to typecast the result to the type of the object you're cloning: Inherited all the way from *TObject*, *Clone* returns a *TObject* value.

THandleArray overrides *Clone* to duplicate more than just the object being cloned. If we simply called *TObject.Clone*, the result would duplicate the handle array object but not its data.

The following *Clone* code first copies the array object and then copies the block pointed to by the object's data handle and creates a new error handler for the object:

```
function THandleArray.Clone: TObject;
  override;
  var
    copy: THandleArray;
    eh: TErrorHandler;
    i: Integer
begin
  if self.fErrorHandler <> nil then
    begin
      self.fErrorHandler.SetError(kNoErr);
      { Copy the THandleArray object, self }
      copy := THandleArray(inherited Clone);  {Copy self, a THandleArray object }
      { Copy THandleArray's data }
      self.fErrorHandler.FailOSErr(HandToHand(Handle(copy.fArray)),
          kUnableToClone);
      for i := kArrayBase to copy.LengthOf do
        copy.fArray^^[i] := copy.fArray^^[i].Clone
```

```
        New(eh);              { Give new object its own error handler }
        eh.IErrorHandler;
        copy.fErrorHandler := eh;
      end
    else
      copy := nil;
    Clone := copy;            { Return the copy }
  end;  { THandleArray.Clone }
```

Clone is simpler for *THandleArray* (and, through inheritance, for *TList*) than it was for *TLinkedList* in Chapter 20. Instead of tracing a linked list of nodes, copying as we go, we have only to duplicate the block at the end of a single handle and duplicate the objects stored in the array's slots. The Toolbox function *HandToHand* does most of the job for us.

The following, rather complex, line does most of the copying, along with some typecasting, and also checks the result for operating system errors:

```
self.fErrorHandler.FailOSErr(HandToHand(Handle(copy.fArray)), kUnableToClone);
```

That's a lot of code in one line, so let's look at it as if it were set out in several lines:

```
handleToCopy := Handle(copy.fArray);
OSErrResult := HandToHand(handleToCopy);
self.fErrorHandler.FailOSErr(OSErrResult, kUnableToClone);
```

In the first line, the *fArray* data handle in the copy of the *THandleArray* object, *copy*, is typecast to a generic handle. Then we pass the typecast result to *HandToHand*, which returns an *OSErr* value. In its parameter, *HandToHand* returns a duplicate of the original handle array's heap block, pointed to by the *fArray* handle in the *THandleArray* object, *copy*. The resulting *OSErr* code is then passed to a method of the error-handler object, along with one of *TErrorHandler*'s own codes to be used if the *OSErr* value is anything other than *noErr*. If *OSErrResult* doesn't equal *noErr*, *FailOSErr* calls *TErrorHandler*'s *FatalError* method to terminate the program with an alert for the user; otherwise, *FailOSErr* does nothing and we continue.

This style of error handling is similar to that used in MacApp, which has the non-method procedures *FailNil*, *FailOSErr*, *FailMemErr*, and others.

The cloning situation is analogous to the *Free* vs. *FreeAll* distinction illustrated in Figure 21-7 on page 521, where we differentiate between freeing the data handle and freeing the data stored in the array.

At the end of *Clone*'s code, *copy* is a duplicate of the cloned object, with its own copy of the cloned object's data handle and duplicates of the cloned object's instance variables.

Resizing the Array

A *THandleArray* object can be resized—as you add data to it, you can make it grow to accommodate more, and as you remove data from it, you can make it shrink to fit what's left.

The *Grow* method expands the object's data handle by a specified number of slots. The *Shrink* method shrinks the data handle by a specified number of slots. Both are Boolean functions, so you can see immediately whether the resizing was successful. Here are *Grow* and *Shrink*:

```
function THandleArray.Grow (numSlots: Integer): Boolean;
  var
    i, oldSlots: Integer;
begin
  if not self.fCanGrow then
    begin
      if self.fErrorHandler <> nil then
        self.fErrorHandler.SetError(kCantGrowArray);
      Grow := false;
    end
  else
    begin
      oldSlots := self.fSlots;                    { Save for reversion }
      if self.PRIVATEResize(numSlots) then        { Try to grow }
        begin
          { Mark all new slots empty--nil }
          for i := (oldSlots + 1) to self.fSlots do
            self.fArray^^[i] := nil;
          Grow := true;
        end
      else
        begin
          self.fSlots := oldSlots
          Grow := false;
        end;
    end;
end;  { THandleArray.Grow }
```

and

```
function THandleArray.Shrink (numSlots: Integer): Boolean;
  var
    oldSlots: Integer
    result: Boolean
```

```
begin
  if not self.fCanShrink then
    begin
      if self.fErrorHandler <> nil then
        self.fErrorHandler.SetError(kCantShrinkArray);
      Shrink := false;
    end
  else
    begin
      oldSlots := self.fSlots
      result := self.PRIVATEResize( - (numSlots));
      if not result then
        self.fSlots := oldSlots;
      Shrink := result;
    end;
end;  { THandleArray.Shrink }
```

Both methods call a private method, *PRIVATEResize*, to do the dirty work. The method's name illustrates another common OOP naming practice. The method must be declared in the class interface and is thus available to consumers, but its name warns them not to use it. They should call *Grow* or *Shrink* instead. (The alternative would be to make *PRIVATEResize* a non-method function hidden inside the unit's implementation part, passing it a reference to *self* so that it can access the array object's instance variables and methods. In *TList*, we do that with the *PrivateAdd* procedure.

Here's the *PRIVATEResize* method that serves both *Grow* and *Shrink*:

```
function THandleArray.PRIVATEResize (byNumSlots: Integer): Boolean;
  var
    oldSlots: Integer;
begin
  if self.fErrorHandler = nil then
    PRIVATEResize := false
  else
    begin
      self.fErrorHandler.SetError(kNoErr);
      oldSlots := self.fSlots;                    { Save for reversion }
      self.fSlots := self.fSlots + byNumSlots;    { Could be negative }
      { See whether list is really expandable }
      if self.fSlots > kMaxSlots then             { Too much? }
        begin
          { List not expandable, so revert to old number of slots; alert caller }
          self.fSlots := oldSlots;
          PRIVATEResize := false;
```

```
            { Post error message }
            self.fErrorHandler.SetError(kCantGrowArray);
          end
        else if self.fSlots <= kArrayBase then  { Too little? }
          begin
            self.fSlots := oldSlots;
            PRIVATEResize := false;
            { Post error message }
            self.fErrorHandler.SetError(kCantShrinkArray);
          end
        else
          begin
            { Expand or shrink the array }
            SetHandleSize(Handle(self.fArray), Size(self.fSlots * kSlotSize));
            if MemError <> noErr then          { See whether successful }
              begin
                { If call failed, low-memory situation }
                { List didn't resize }
                self.fSlots := oldSlots;
                PRIVATEResize := false;
                { Post error message }
                self.fErrorHandler.SetError(kCantGrowArray);
              end
            else
              PRIVATEResize := true;           { All's well }
          end;  { else }
      end;  { If error handler exists }
  end;  { THandleArray.PRIVATEResize }
```

The code is mostly for error checking. We don't want to resize the array handle if it would end up with a negative length or outgrow its maximum size. In case those problems occur, we save the previous value of *fSlots* in *oldSlots* so that we can revert to it. If all goes well, we call

```
SetHandleSize(Handle(self.fArray), Size(self.fSlots * kSlotSize));
```

to do the actual resizing. If *fSlots* has become smaller, the array handle will shrink. If *fSlots* has grown, so will the array handle. If *fSlots* doesn't change, the array handle won't change either.

Growing and shrinking are options. Subclasses of *THandleArray* can allow either, both, or neither. *TList* allows both. That keeps the allocated space in the heap fairly consistent with the actual amount of data in the list. But, for efficiency when we expand the array, we prefer to expand it by more than one slot. And we don't want to shrink the array until a certain number of slots has been deleted. *THandleArray* doesn't enforce a particular amount of growing and shrinking, but it's easy for

subclasses to specify the amounts by which an array can grow and shrink. See the code for *TList*'s *PrivateAdd* utility procedure and its *Extract* method later in this chapter for examples.

Now let's apply *THandleArray* in *TList*.

Building *TList*

Recall that *TList* will be a subclass of *THandleArray*. As such, it inherits all the capabilities of *THandleArray*: optional locking; error reporting; optional data type screening; optional growing and shrinking; cloning and freeing; searching; and traversing to perform actions on all elements. That's a lot of capability to inherit, and that makes implementing a list inside the array fairly easy. It also saves code. Figure 21-9 shows a *TList* sitting inside a *THandleArray*.

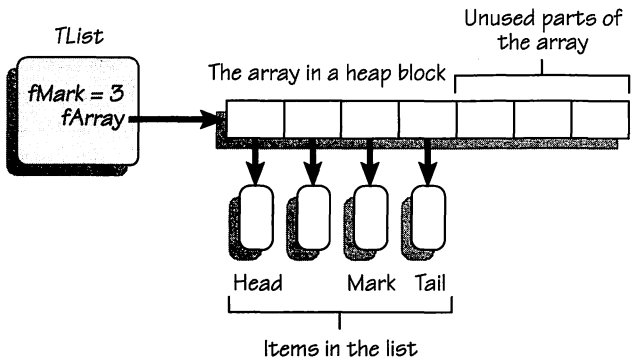


Figure 21-9.
TList *and* THandleArray.

Differences Between *TList* and *THandleArray*

A list is not the same thing as an array, of course, so *TList* does some things differently than *THandleArray*. *TList* also selects from among *THandleArray*'s numerous options differently than other *THandleArray* subclasses might.

TList allows optional locking. You can disable the default locking by calling *TList*'s inherited *SetLocking(false)* immediately after initializing a list instance. That allows you to store any descendant of *TObject*. If you don't disable locking, you must store descendants of *TLockable* only.

TList doesn't implement any data type screening. It inherits *THandleArray*'s stub *TypeMatches* method unchanged. To implement type screening, you must subclass *TList* and override *TypeMatches*, as we demonstrated earlier in this chapter. *TList* does accommodate type screening, however, by calling *TypeMatches* from all its addition methods. If you instantiate a *TList* rather than a subclass, the call to *TypeMatches* simply returns *true*, admitting any type of data whatsoever.

TList enables both expanding and shrinking. This keeps the allocated space in line with the current amount of data. But *TList* also makes expanding and shrinking more efficient than if we always added or deleted new elements one at a time. The utility procedure *PrivateAdd*, called by the addition methods, calls *Grow* with a parameter of *kGrowAmount*, which in *TList* equals *20*. This allocates blocks of 20 new elements at a time. The *Extract* method tracks the number of elements deleted in an instance variable, *fDeleteCount*. When *fDeleteCount* reaches *kGrowAmount*, the next deletion will cause the array to shrink. You can adjust these actions by changing *kGrowAmount*.

TList overrides five *THandleArray* methods—*IsEmpty*, *IsFull*, *LengthOf*, *HighOf*, and *TypeOf*—to implement them in its own way. Whereas the length of a *THandleArray* is the index number of the highest occupied (non-*nil*) slot, the length of a *TList* is based on a count of elements as they are added to the list and is kept in *fLength*. All addition and deletion methods adjust the count so that *fLength* always gives the current number of items stored in the list. *TList.LengthOf* returns the value of *fLength*. A *THandleArray* is full if all possible slots (*fMaxSlots*—the potential array) have been allocated. But a *TList* is full if the number of elements in the list (*fLength*) equals the number of slots currently allocated (*fSlots*—the physical array). A *THandleArray* is empty if *fSlots* is *0*, meaning that no slots have been allocated or that all have been deallocated. A *TList* is empty if *fLength* is *0*. An override of *THandleArray.HighOf* returns the number of items currently in the list, exactly as *LengthOf* does.

TList implements a *Find* method that takes care of some overhead before it calls the *Search* method inherited from *THandleArray*.

And *TList* implements its own methods for putting data into the list and taking it out. This is done differently, of course, than it was done in *TDynamicArray*.

Some Cautions

TList inherits some features it doesn't need—and even a few that might be dangerous. For example, the list doesn't need the *PRIVATEResize* method, at least not directly, although *TList* does use *Grow* and *Shrink*, which call *PRIVATEResize*. And the list doesn't need *THandleArray.Search*, except from within *TList.Find*. Calling either of those methods directly for a *TList* might yield unpredictable results.

These are small cases of a larger problem that we'll take up again in Chapter 24. Sometimes inheritance brings problems—just as biological inheritance sometimes brings useless or debilitating genes.

For now, we simply caution consumers of the list not to call the *PRIVATEResize* and *Search* methods it inherits from *THandleArray*.

The *TList* Class Declaration

Listing 21-1 is the class declaration for *TList*. Notice how similar the *TList* interface is to that for *TLinkedList* in the previous chapter, despite their very different implementations.

```
type
  TList = object(THandleArray)
      { Actual list is inherited from THandleArray, along with other }
      {  THandleArray instance variables and methods }

      fMark: 0..kMaxSlots;       { Current position of mark in list }
      fLength: 0..kMaxSlots;     { Current number of items }
      fDeleteCount: Integer;     { Number of items deleted; used in shrinking list }
      { Initializing the list }
      procedure TList.IList (wantSlots: Integer; eh: TErrorHandler);
        { Initializes a new, empty list with wantSlots slots for }

      { Moving in the list }
      procedure TList.Move (toIndex: Integer);
        { Moves mark to list item at toIndex }
      procedure TList.MoveToNode (node: TObject);
        { Moves mark to item equal to node; node must be a reference }
        {  to an item already in the list }
      procedure TList.Head;
        { Moves mark to head item of list }
      procedure TList.Tail;
        { Moves mark to tail item of list }
        { If mark already at tail, generates error message and doesn't }
        {  move mark }
      procedure TList.Previous;
        { Moves mark to previous item of list }
        { If mark already at head, generates error message and doesn't }
        {  move mark }
      procedure TList.Next;
        { Moves mark to next item of list, following current location }
      function TList.Find (s: TSearchKey): Boolean;
        { Uses s to find a matching item in list }
        { If match found, moves mark to found item; if not, returns false and }
        {  leaves mark where it is }
        { Starts from previous mark, so call Head first to start there }
```

Listing 21-1. *(continued)*
Declaration for the class TList, *descended from* THandleArray.

Listing 21-1. *continued*

```
{ Getting the state of the list }
function TList.IsFull: Boolean;
override;
  { Returns true if all currently allocated list slots are full }
function TList.IsEmpty: Boolean;
override;
  { Returns true if list has no items }
  function TList.LengthOf: Integer;
  override;
  { Returns the number of items in list }
function TList.HighOf; Integer;
override;
  { Returns the number of items in list--same as LengthOf }
function TList.PositionOf: Integer;
  { Returns current index value of the mark }
function TList.AtHead: Boolean;
  { Returns true if mark is at the head item}
function TList.AtTail: Boolean;
  { Returns true if mark is at the tail item }

{ Adding to the list }
procedure TList.AddHead (elem: TObject);
  { Adds item at the head of the list }
  { Calls TypeMatches first--subclass and override TypeMatches to }
  {  screen out unwanted data types }
procedure TList.AddTail (elem: TObject);
  { Adds item at the tail of the list }
  { Calls TypeMatches first--see AddHead }
procedure TList.Insert (elem: TObject);
  { Inserts item before the current position of the mark }
  { Calls TypeMatches first--see AddHead }
procedure TList.Append (elem: TObject);
  { Adds item after the current position of the mark }
  { Calls TypeMatches first--see AddHead }

{ Removing from the list }
function TList.Extract: TObject;
  { Removes item at the mark and returns a reference to it }
  { Callers must typecast the reference to the item to the expected type }
function TList.ExtractHead: TObject;
  { Removes item at the head and returns a reference to it }
  { Callers must typecast the reference to the item to the expected type }
```

(continued)

Listing 21-1. *continued*

```
    procedure TList.Delete;
      { Removes item at the mark and throws it away }
    function TList.Peek: TObject;
      { Returns a reference to item at the mark }
      { Doesn't disturb the list }
      { Callers must typecast the reference to the item to the expected type }
    function TList.TypeOf: Str30;
    override;
      { Returns the string 'TLIST' }
    { Inherits SetErrorHandler, ErrorHandlerOf, Error, Clear, Grow, Shrink, }
    {  Clone, Free, FreeAll, DoToEach, and other methods from THandleArray }
    { Free frees only the underlying handled array, not its contents }
    { FreeAll frees the data in the list, too }
  end;  { Class TList }

{ Non-method function to create and initialize a new TList object }
function NewList (wantSlots: Integer; eh: TErrorHandler): TList;
```

TList's Instance Variables

TList inherits all of *THandleArray*'s instance variables, including *fArray*, which is the handle to the actual list—where the data is stored.

The *fLength* instance variable stores the current number of items in the list. All addition and deletion methods increment or decrement *fLength*.

The *fMark* instance variable stores the index number of the mark in the underlying array. (For *TLinkedList*, *fMark* stored an object reference instead of an integer.)

The *fDeleteCount* instance variable is used by *Extract* to make shrinking the list more efficient.

TList's Methods

Many of *TList*'s methods are straightforward because they work in an array. Let's look at some of the more interesting methods. You can see full code for all of them on the code disk.

Initialization

IList is simple. It calls *IHandleArray* and sets its three instance variables to 0:

```
procedure TList.IList (wantSlots: Integer; eh: TErrorHandler);
  const
    kCanGrow = true;
    kCanShrink = true;
begin
  self.IHandleArray(wantSlots, kCanGrow, kCanShrink, eh);
  self.fLength := 0;                    { Empty list }
  self.fMark := 0;                      { No mark for empty list }
  self.fDeleteCount := 0;               { No deletions yet }
end;  { TList.IList }
```

To take advantage of growing and shrinking, we pass *true* for the *canGrow* and *canShrink* parameters in the *IHandleArray* call.

Moving in the list

We'll look at code for *Move*, *MoveToNode*, *Tail*, and *Next*. Most of these methods and the counterparts *Head* and *Previous* are simple—but complicated a bit by error-handling code.

Move first checks that the index number passed is in range. If it is, *Move* sets the mark to the index number. Otherwise, it posts an error message, which we can retrieve with *TList.Error*, inherited from *THandleArray*.

```
procedure TList.Move (toIndex: Integer);
begin
  if (toIndex >= kArrayBase) and (toIndex <= self.fLength) then
    self.fMark := toIndex
  else
    self.fErrorHandler.SetError(kOutOfRange);
end;  { TList.Move }
```

MoveToNode must search the array for an object reference value equal to the node value passed in. Not finding such a matching value is an error. If the value is found, *MoveToNode* sets the mark to the index number of the position at which the value was found.

```
procedure TList.MoveToNode (node: TObject);
  var
    found: Boolean;
    current: Integer;
begin
  found := false;
  for current := kArrayBase to self.fLength do
    begin
```

```
      if (node = self.fArray^^[current]) then
        begin
          self.fMark := current;
          found := true;
          Leave;  { The for loop }
        end;  { if }
    end;  { for }
  if not found then
    self.fErrorHandler.SetError(kNotFound);
end;  { TList.MoveToNode }
```

Head and *Tail* simply set the mark to *kArrayBase* (head) or *self.fLength* (tail). Here's *Tail*:

```
procedure TList.Tail;
begin
  self.fMark := self.fLength;
end;  { TList.Tail }
```

Previous and *Next* decrement and increment the mark's value and then test to see whether the mark hasn't gone off the front or rear ends of the list. Unlike *OffEnd* in *TLinkedList*, *TList*'s *OffEnd* is a private function hidden away in the implementation part of *TList*'s definition. The *OffEnd* function returns *true* if the mark's value is less than *kArrayBase* or greater than the list's length. Here's *Next*:

```
procedure TList.Next;
begin
  self.fMark := self.fMark + 1;
  if OffEnd(self) then
    begin
      self.fErrorHandler.SetError(kOutOfRange);
      self.fMark := self.fMark - 1;
    end;  { if }
end;  { TList.Next }
```

Finding items in the list

Although we won't go into the find mechanism fully until Chapter 22, we can see most of what it does. *TList.Find* does some error checking and setup and then calls its inherited *THandleArray.Search*.

Search positions the mark at the index number of the item found, if any. If none is found, *Search* leaves the mark alone and returns *false*.

Find-Search is set up to work with multiple finds in the list. To find the first occurrence of an item, first call *Head* and then call *Find*. Subsequent finds begin searching from the current position of the mark.

Here's *TList.Find*:

```
function TList.Find (s: TSearchKey): Boolean;
  var
    found: Boolean;
    index: Integer;
begin
  { Error trying to find item in empty list }
  if self.fLength < 1 then
    begin
      found := false;
      self.fErrorHandler.SetError(kEmptyArray);
    end
  { OK to find }
  else
    begin
      { Position the mark for next search }
      { Assume search begins at head }
      if self.fMark = kArrayBase then
        { Ensure looking at first item }
        index := self.fMark
      else
        { Assume repeat from last find }
        index := self.fMark + 1;      { Start at position just after mark }
      { Check for index number out of range }
      if index < kArrayBase then
        begin
          index := kArrayBase;
          { Assume head intended but post error message }
          self.fErrorHandler.SetError(kOutOfRange);
        end
      else if index > self.fLength then
        begin
          index := self.fLength;
          { Assume tail intended but post error message }
          self.fErrorHandler.SetError(kOutOfRange);
        end;
      { Do the actual search from object at the index number onward }
      found := Search(s, index);      { Call THandleArray method }
      if found then
        self.fMark := index;
      Find := found;
    end;  { else }
end;  { TList.Find }
```

Among the error conditions that *Find* checks for are an empty list and an index value that's out of range. It also has to set the value of *index* that specifies the position at which the search begins. If the mark is at *kArrayBase*, the assumption is that the using programmer has called *Head*, so the search starts at the mark's location. If the mark is somewhere else, the assumption is either that the using programmer wants to search from that point on or that the using programmer has already made at least one previous find. The search proceeds from the next element rather than from the position of the mark.

We'll see what a *TSearchKey* object is and how *Search* uses it in Chapter 22.

Utility methods

We've already looked at *TList*'s *IsEmpty* and *LengthOf* methods in our consideration of *THandleArray*.

PositionOf, *AtHead*, and *AtTail* are simple, so we won't examine them in any detail. *PositionOf* returns *fMark*'s value. *AtHead* returns *true* if *fMark* is equal to *kArrayBase*. *AtTail* returns *true* if *fMark* is equal to *fLength*.

Adding objects to the list

Adding objects in *TList* is considerably easier than it was in *TLinkedList*. There are no links to break and relink. We simply find the right array element and assign the new object to it. The only complication is that data stored in any array elements following the selected element has to be moved down a space.

To reduce code bulk and simplify the addition methods, we've written a private workhorse method—hidden in the implementation part of the unit that contains the class definition—and called it from all addition methods. We'll look first at the addition methods and then at the utility procedure, *PrivateAdd*. The addition methods pass *PrivateAdd* a reference to *self* so that it can access the list's instance variables and methods. We've seen that technique before.

AddHead simply passes *PrivateAdd* the object to add, along with the index value *kArrayBase* (signifying the position of the first array element, the head).

```
procedure TList.AddHead (elem: TObject);
begin
  PrivateAdd(self, elem, kArrayBase);
end;  { TList.AddHead }
```

AddTail passes an index value one place beyond the current tail element's position. It resets the mark's value to the new length value set by *PrivateAdd*.

```
procedure TList.AddTail (elem: TObject);
begin
  PrivateAdd(self, elem, self.fLength + 1);
  self.fMark := self.fLength;
end;  { TList.AddTail }
```

Insert calls *PrivateAdd* with an index value equal to the current position of the mark. *PrivateAdd* makes room for the new item just before the mark and assigns the object to that slot.

```
procedure TList.Insert (elem: TObject);
begin
  PrivateAdd(self, elem, self.fMark);
end;  { TList.Insert }
```

Insert adds the item before the mark. *Append* adds it after. *Append* passes an index value one greater than the mark's position, effectively turning an append at the mark to an insert at the item one past the mark. *Append* also resets the mark to reference the position of the item added.

```
procedure TList.Append (elem: TObject);
begin
  PrivateAdd(self, elem, self.fMark + 1);
  self.fMark := self.fMark + 1;
end;  { TList.Append }
```

All the simplicity of these methods relies on the complexity hidden by *PrivateAdd*. *PrivateAdd* performs several services:

- It calls *TypeMatches* to screen out unwanted data types and adds only if *TypeMatches* returns *true* (which it does by default, unless we subclass *TList* and override *TypeMatches*).
- It posts error messages if the object passed (*elem*) is *nil*, if the list must grow but can't, or if the list can't lock the new object (probably because it was already locked).
- It also attempts to lock the new object if *fWithLocking* is *true*. To lock the object, *PrivateAdd* must typecast the object to *TLockable* so that it can call the object's *Lock* method.

After everything is in order, *PrivateAdd* uses the Toolbox procedure *BlockMove* to shuffle bytes upward in the array, if necessary, making a hole for the added object. We'll look at *BlockMove* after we see the *PrivateAdd* procedure:

```
procedure PrivateAdd (list: TList; elem: TObject; index: Integer);
  var
    elemP: ArrayP;
    elems: Integer;
    ok: Boolean;
    eh: TErrorHandler;
begin
  eh := list.fErrorHandler;              { For convenience }
  if not list.TypeMatches(elem) then
    begin
      if eh <> nil then
        eh.SetError(kTypeNotAllowed);
    end
```

```
    else                                    { Type matches }
      begin
        if elem = nil then
          eh.SetError(kAddingNilElem);    { Advisory message }
        if (index < kArrayBase) or (index > (list.fLength + 1)) then
         begin
            eh.SetError(kOutOfRange);
            Exit(PrivateAdd);
          end;  { if }
        { Expand the list space if necessary }
        if list.fLength >= list.fSlots then
          ok := list.Grow(kGrowAmount);  { Use the Grow method }
        { If list can't grow, get out }
        if eh.Error = kCantGrowList then
          begin
            { But restore error message before leaving }
            {  so that the error can be checked outside this procedure }
            eh.SetError(kCantGrowList);
            Exit(PrivateAdd);
          end;
        if list.fWithLocking then
          if (list.fTypeStored = objects) and (elem <> nil) then
            { Lock the element }
            eh.FailFalse(TLockable(elem).Lock(list.fSignature), kUnableToLock);
        { Point to the list head }
        elemP := list.fArray^;
        { If list is not empty }
        if list.fLength > 0 then
          begin
            elems := list.fLength - index + 1;
            if elems > 0 then
              BlockMove(@elemP^[index], @elemP^[index + 1], elems * kSlotSize);
          end
        else
          { If list is empty }
          index := kArrayBase;
        elemP^[index] := elem;
        list.fLength := list.fLength + 1;
        { Mark keeps the same number }
      end;  { Else type matches }
  end;  { PrivateAdd--not a method }
```

If the new list length would exceed the current number of slots allocated, *PrivateAdd* calls *THandleArray.Grow*, asking for *kGrowAmount* new slots.

The most interesting aspect of *PrivateAdd* is its call to *BlockMove*. *BlockMove* takes three parameters: a pointer to the start of the bytes we want to move, a pointer to the location we want to move them to, and a count of the number of bytes to move. Our first step is to dereference the *fArray* handle to obtain a pointer to the start of the array:

```
elemP := list.fArray^;
```

Then we use the @ operator to obtain pointers to the bytes at the positions in the array from which and to which we want to move the bytes:

```
@elemP^[index]          { Pointer to byte at array[index] }
@elemP^[index + 1]      { Pointer to next element }
```

To give the count of bytes to move, we calculate the number of slots remaining in the list beyond our start index value:

```
elems := list.fLength - index + 1;
```

Multiplied by *kSlotSize*, *elems* gives us the number of bytes to move upward in memory. *BlockMove* is a pretty safe operation, even if the two ranges of bytes overlap. See *Inside Macintosh*, II-44. Figure 21-10 shows the effect of *BlockMove* if we're inserting a new item in the middle of the list.

If we pass *PrivateAdd* the correct index value, it works for *AddHead*, *AddTail*, *Insert*, and *Append*.

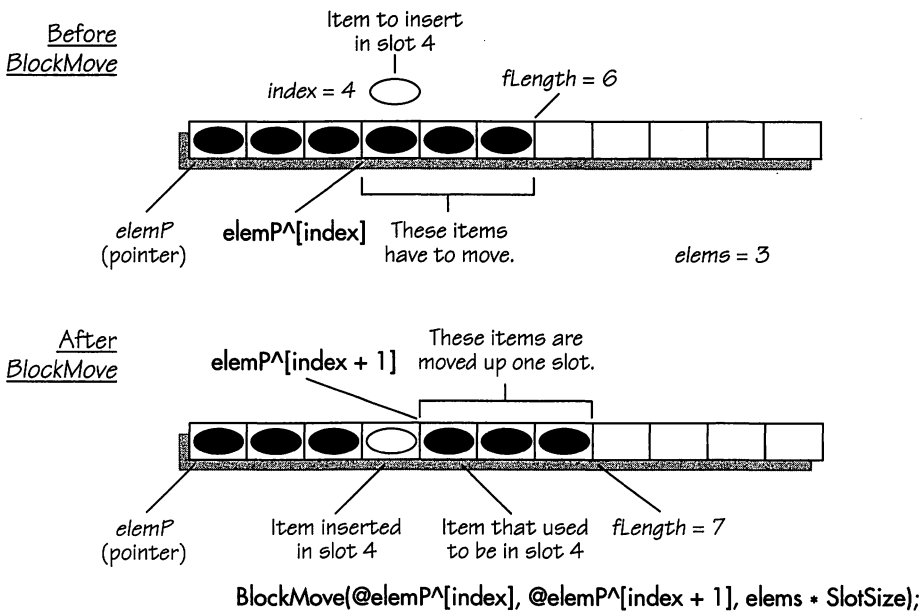


Figure 21-10.
Using BlockMove *to insert an item in the middle of a list.*

Removing objects from the list

All of *TList*'s removal methods are based on *Extract*, which removes the item at the mark, fixes up the array, and returns a reference to the extracted object. We'll look first at the methods that call *Extract* and then at *Extract* itself.

ExtractHead simply sets the mark at the first element and then calls *Extract* and returns the result:

```
function TList.ExtractHead: TObject;
begin
  self.fMark := kArrayBase;          { Position mark at head of list }
  ExtractHead := self.Extract;       { Extract the element there }
end;  { TList.ExtractHead }
```

Delete removes the element at the mark but doesn't return it. *Delete* simply calls *Extract* without returning its result:

```
procedure TList.Delete;
  var
    unwanted: TObject;
begin
  unwanted := self.Extract;
end;  { TList.Delete }
```

Now for *Extract*, which naturally does a lot more than either *ExtractHead* or *Delete*. *Extract* first gets a reference, *elem*, to the object stored at the mark. Then *Extract* tries to unlock the object, posting an error message if it can't. If it successfully unlocks the object, *Extract* gets a pointer to the start of the array by dereferencing *fArray*. If the mark is currently on any element but the tail element, we have to move the bytes of the array down one element, toward the head, to fill up the vacated space in the array. We use *BlockMove* to do that. Then we reset the mark and the length and check to see whether the array should be shrunk. We shrink the array and reset *fDeleteCount* to *0* if *fDeleteCount* has reached *kGrowAmount*.

```
function TList.Extract: TObject;
  var
    elemP: ArrayP;
    index, elems: Integer;
    elem: TObject;
    ok: Boolean;
begin
  { Save item to return }
  elem := self.fArray^^[self.fMark];
  Extract := elem;
  if self.fWithLocking then
    { Unlock it for outside use }
```

```
        if elem <> nil) then
          self.fErrorHandler.FailFalse(TLockable(elem).Unlock(self.fSignature),
              kUnableToUnlock);
        { Point to head of list }
        elemP := self.fArray^;
        { Mark not on last element }
        if self.fMark < self.fLength then
          begin
            index := self.fMark;
            elems := self.fLength - index;
            { Move rest of list down one element }
            BlockMove(@elemP^[index + 1], @elemP^[index], elems * kSlotSize);
            { Mark the vacated element as empty }
            elemP^[self.fLength] := nil;
          end;
        { Reset mark and length }
        if self.fMark = self.fLength then
          self.fMark := self.fMark - 1;
        { Else mark keeps same number }
        self.fLength := self.fLength - 1;
        self.fDeleteCount := self.fDeleteCount + 1;
        if self.fDeleteCount >= kGrowAmount then
          if self.Shrink(kGrowAmount) then
            self.fDeleteCount := 0;
    end;  { TList.Extract }
```

Notice the *TErrorHandler.FailFalse* method called to test successful unlocking. *FailFalse* is similar to the *FailOSErr* method we looked at earlier. Here's the line in which we call *FailFalse*:

```
self.fErrorHandler.FailFalse(TLockable(elem).Unlock(self.fSignature), kUnableToUnlock);
```

FailFalse takes two parameters: a Boolean value and an error code to pass to *TErrorHandler.FatalError*. The code

```
TLockable(elem).Unlock(self.fSignature)
```

typecasts *elem* to a *TLockable* and then sends *elem* an Unlock message, passing the list's signature as the key. The result of the *Unlock* call is *true* or *false*.

FailFalse uses that returned value, checking it for false and failing if it is; otherwise, it does nothing, and the element is unlocked.

One related access method, *Peek*, has its own way of returning information about the list, without calling *Extract*. *Peek* simply returns a reference to the item currently at the mark, which it obtains by doubly dereferencing *fArray* and then using array-access notation to get the item at the index value contained in *fMark*. It takes longer to describe a *Peek* than to do one.

```
function TList.Peek: TObject;
begin
  Peek := self.fArray^^[self.fMark];     { Refer to element at mark }
end;  { TList.Peek }
```

Recall our earlier warnings about *TLinkedList*'s *Peek*. Because you can't use the value returned by *TList.Peek* to traverse the list's links forward or backward, you can't use *Peek* to reach other list elements. They're safe from backdoor tampering, especially freeing. *TList.Peek* is safer than *TLinkedList.Peek*.

Using the List

To see how to use the *TList* list in a variety of ways, see the example program List Demo on the code disk. The program illustrates storing *TObject*s in an unlocked list and performing many operations on locked lists. List Demo is hardly a comprehensive test suite, but it shows off much of what you can do with *TList*.

Summary

We've developed an interesting class hierarchy, with an abstract dynamic array class, a true, instantiable dynamic array class, and an array-based list class.

We've dealt with almost all of the list's more interesting features. All that's left is to work out the secrets of searching and traversing polymorphic encapsulated lists. We'll get to those problems next.

Projects

- Think about other classes in addition to *TList* and *TDynamicArray* that you might derive from *THandleArray*. Add them to your class library.
- Look at the example classes we derived from *TList* in the List Demo program: *TStringTable*, *TMonthTable*, and *TMailList*. Develop other *TList* descendants that might be generally useful.
- Try your hand at implementing *THandleArray*'s *Search* and *DoToEach* methods. We'll cover these methods in Chapters 22 and 23, respectively.
- Use the same techniques we used for *THandleArray* to write a dynamic array class for type *Char* data. (This will not be an array of objects.) You'd want to provide a full interface, including the ability to grow and shrink. Keep in mind that a packed array of type *Char* stores characters one per byte in THINK Pascal and that a single *Char* variable is 2 bytes wide, with the *Char* value in the low byte and *0* in the high byte. However, assigning a *Char* to an element of a packed array of *Char* works as you'd expect.
- Having completed the dynamic array class for type *Char* data, derive a dynamic string class from it. Make the string capable of interacting with the world of static strings: assigning strings of any length up through 255 characters

to a dynamic string, returning the first 255 characters of a dynamic string as a *Str255*, assigning a single character to a particular index value in a dynamic string, and reporting a character from any position in the dynamic string. The class should interact with other dynamic string objects. In addition, write basic string-handling operations such as *Concat*, *Length*, *Copy*, *Pos*, *Delete*, *Omit*, *Insert*, and *Include*. Base the dynamic string's ability to grow on the dynamic *Char* array class you developed in the previous project.

22

POLYMORPHISM: THE FIND PROBLEM

I've deliberately postponed our consideration of some problems that come up in dealing with object-oriented lists such as *TList*. These problems stem from our use of polymorphism to make a list general.

The problem we'll tackle in this chapter involves searching *TList*, a highly polymorphic (and encapsulated) list: If there might be different kinds of objects in the list, how can we search on a variety of keys? We'll look at several solutions, some more object oriented than others. The most OOP-like solution uses a "helper" object of class *TSearchKey*. Along the way, we'll consider

- Searching a polymorphic, encapsulated list
- Comparisons done with procedural parameters
- Search key objects
- Using *TList.Peek*

Searching

We've already previewed searching various kinds of lists, including encapsulated, polymorphic ones like *TList*. Encapsulation prevents us from getting directly at the innards of the list. The list's generality prevents it from "knowing" anything about particular data types that might be put into it. And polymorphism makes it hard to tell exactly what's in a list element and whether it has some particular method needed for the search.

The problem is to develop a search mechanism as general as the list and still simple enough to be practical. One time, we might want to search for integer data equal to a key, and another time, for string data less than or equal to a key. A third time, we might want to search with a compound condition and several different key types.

We'll develop four different approaches to the search problem.

Goals

Our specific goal in this chapter is to write a general *Search* method for class *THandleArray*. More generally, we'll undertake an exploration of the intricacies of searching polymorphic, encapsulated data structures of all sorts. On the way to writing *Search*, we'll sometimes use *TList* or one of its descendants to illustrate a point. Eventually, we'll arrive at *Search* itself.

While we're focused on *Search*, let's take care of the business of naming our search methods. Recall that *TList.Find* does some setup and then calls *THandleArray.Search* for the actual searching. Because the two classes are in the same hierarchy—*TList* descending from *THandleArray*—we have to use two different method names: *Search* and *Find*. We could simply override *Search* in *TList*, but our design calls for different parameter lists, and Object Pascal won't allow us to change the parameter list (or the name) of an overridden method. Here are the two method headings:

```
function THandleArray.Search (s: TSearchKey; var index: Integer): Boolean;
  { True if an item matching s's data is found; starts at index and }
  {  returns index number of find or 0 if not found }
```

and

```
function TList.Find (s: TSearchKey): Boolean;
  { Uses s to find a matching item in list }
  { If match found, moves mark to found item; if not, }
  {  returns false and leaves mark where it is }
```

Find's interface is simpler, relying on the list's mark to keep track of the index value, whereas *Search* must track the current location of the mark in the array explicitly. This choice is a judgment call, of course, and you might prefer to keep the *Search* interface in all of *THandleArray*'s descendants. If that's the case, you could override *THandleArray.Search* in *TList*, write the same setup code as *Find*'s, and then call *inherited Search* to do the searching. We've worked around the limitations on overrides by using the *Find* method.

Lost in the Mail: What It Takes to Be Found

Let's look at searching with a concrete example, this one using *TList.Find*.

Suppose we need to store various kinds of mail in a *TList*—as we did, in fact, in the List Demo program for Chapter 21. Class *TMail* has two subclasses: *TLetter* and *TMemo*, so we're dealing with polymorphic data. A given list element might be a *TLetter* object or a *TMemo* object.

We want to search the list for any mail since a given date, any mail from a given sender, and any mail from a given sender since a given date. (We might want to perform other kinds of searches too, but these will do for illustration.)

First, let's take up some class design considerations. Searching requires a key (or keys) of some type (or types), so every piece of mail must have certain instance variables in common (and methods to return those fields' values, of course). So our design of *TMail* and its subclasses has to ensure the common features. The best way to make the features common to all classes descended from *TMail* is to put them in *TMail* itself so that the subclasses can inherit them. *TMail* will have, at a minimum, a date field, a sender field, and a contents field. It will also have methods to set and get those fields' values. Here's our simple *TMail* class:

```
type
  TMail = object(TLockable)

      { Instance variables common to all mail types }
      fDate: DateType;
      fSender: Str255;
      fContents: Str255;

      procedure TMail.IMail (date: DateType; sender: Str255; contents: Str255);
        { Initializes a new piece of mail }
      function TMail.DateOf: DateType;
        { Returns the date of a piece of mail }
      function TMail.SenderOf: Str255;
        { Returns the sender's name on a piece of mail }
      function TMail.ContentsOf: Str255;
        { Returns the contents of a piece of mail-- }
        {  to "read" it }
      procedure TMail.Display;
      override;
        { Displays the piece of mail }
    end;  {  Class TMail }
```

In order to be usable in a list, *TMail* objects must descend from *TLockable*, if we want locking, or from *TObject* if we don't. We'll usually prefer locking, so *TMail* is a descendant of *TLockable*.

Somewhere above the *TMail* declaration, we need a type declaration for dates:

```
type
  DateType = record
      month, day, year: Integer;
    end;  { DateType record }
```

(We might want to develop a *TDate* object class instead, which can store and process a date in several useful ways. But the record will do for our needs here.)

We've made the sender and contents instance variables *Str255* strings, although, in a real mail system, it might be better to use a smaller string type for the sender and a handle to some arbitrary text for the contents.

At this point, all descendants of *TMail* will inherit the fields and methods we need for searching the list for particular kinds of mail.

Now let's declare a couple of sample subclasses:

```
type
  TLetter = object(TMail)              { Overrides only one method }

    function TLetter.TypeOf: Str30;
    override;
      { Returns the string 'TLETTER' }
  end;  { Class TLetter }
```

and

```
type
  TMemo = object(TMail)                { Adds subject line }

    fSubject: Str255;

    procedure TMemo.IMemo (date: DateType; sender: Str255;
        contents: Str255; subject: Str255);
    function TMemo.SubjectOf: Str255;
    procedure TMemo.Display;
    override;
    function TMemo.TypeOf: Str30;
    override;
      { Returns the string 'TMEMO' }
  end;  { Class TMemo }
```

A *TLetter* object has only what it inherits—it adds no new instance variables and overrides only the trivial *TypeOf* method. A *TMemo* object adds a new field, to contain a subject line for the memo, along with a new and improved initialization method, a method to get the subject field's value, an overridden *Display* method to handle the special aspects of displaying a memo, and an overridden *TypeOf* method.

Despite differences, *TLetter* and *TMemo* do have a lot in common: *fDate*, *fSender*, and *fContents* fields and the methods to get those fields' values. This is enough to support searching along the lines we want. Figure 22-1 shows schematically what our *TMail* list looks like.

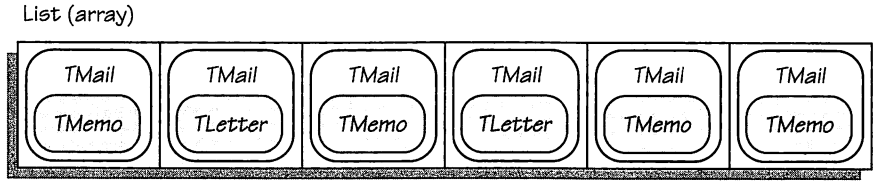


Figure 22-1.
A polymorphic TMail *list. Each node contains a* TMail *subclass instance.*

Searching a *TList* Full of Mail

As the list stands, searching a list full of mail objects is easy. Although the list is polymorphic, by our design we're guaranteed that all list elements will have the right fields and methods.

And, because the searches we want to do are few and specific, we could simply subclass a *TList* that has no built-in search methods and add our own custom-designed searches. With only three searches to worry about, we could dispense with a general search method altogether. But don't forget that our real quest is for a general search method.

Let's look at one or two of the methods we'd want to add to a *TMailList* class descended from our *Find*-less *TList*. Function *FindDate* would find the first date meeting our criteria (date >= a key date) and put the mark on that date. *FindDate* would return *true* if there were a find; *false*, otherwise. Sending FindDate again would start a search from the present mark and would find subsequent dates that meet the criteria, one per FindDate message, returning *true* for finds and putting the mark on found items. (If you wanted to search the whole list, you would need to send a Head message to the list to position the mark at the head of the list before sending your first FindDate message. Otherwise, *FindDate* would start looking from *fMark + 1*.) Function *FindDateAndSender* would find the first piece of mail from our key sender with a date greater than or equal to a key date. Sending FindDateAndSender again would find the next such match. (We'd want similar methods for finding only a sender, to round out the method list.) We'll look at *FindDate* and *FindDateAndSender* to illustrate. Here's FindDate:

```
function TMailList.FindDate(key: DateType): Boolean;
  var
    lastFind: Integer;
    mail: TMail;
    date: DateType;
    found: Boolean;
begin
  found := false;
  { Set the starting point }
  if self.fMark = kArrayBase then
    lastFind := self.fMark          { Starting from head }
  else
    lastFind := self.fMark + 1;     { One place beyond last find }
  { Loop through the list }
  for i := lastFind to self.fLength do
    begin
      { Retrieve current mail item and get its date }
      mail := TMail(self.fArray^^[i]);
      date := mail.DateOf;
```

```
        { Compare mail's date with key's date }
        if date.year > self.fKeydate.year then
          found := true
        else if date.year = self.fKeydate.year then
          if date.month > self.fKeydate.month then
            found := true
          else if date.month = self.fKeydate.month then
            if date.day > self.fKeydate.day then
              found := true;
        if found then
          begin
            self.fMark := i;          { Mark the find }
            Leave;                    { Exit the for loop }
          end;
      end;  { for }
    FindDate := found;
  end;  { TMailList.FindDate }
```

FindDateAndSender would be similar. We'd move through the list, one element at a time, stopping at each element to typecast it to *TMail*. We'd retrieve the sender and date and compare them. The function would take two parameters, *keySender* and *keyDate*. Here's just the retrieval and comparison part—the body of the *for* loop:

```
  :
  begin
    { Retrieve current mail item and get its sender and date }
    mail := TMail(self.fArray^^[i]);
    if (mail.SenderOf = self.fKeySender) then
      begin
        date := mail.DateOf;
        { Check the mail's date against key }
        if date.year > self.fKeydate.year then
          found := true
        else if date.year = self.fKeydate.year then
          if date.month > self.fKeydate.month then
            found := true
          else if date.month = self.fKeydate.month then
            if date.day > self.fKeydate.day then
              found := true;
        if found then
          begin
            self.fMark := i;          { Mark the find }
            Leave;                    { Exit the for loop }
          end;
      end;  { if sender }
  end;  { for }
```

We'd speed things a bit by checking for the sender first. There would be no sense in checking the date if the sender were wrong. If the sender test failed, we'd skip the date tests.

The rest of *TMailList*'s search methods would operate on the same principles. We could build a list with as many search methods as we needed.

How good a solution would this be? Well, if you think about it, we're building specific data knowledge into our highly general list. For example, we use a *TMail* typecast in the *for* loop. That seems wrongheaded. Ideally, we'd like *TList* to contain a general and adaptable search mechanism. The using programmer should supply the data knowledge by some other means.

General Searching

Because *THandleArray* and its descendants are highly general structures, they could contain many different data types at the same time, and they might have many and varied instance variables. We'd prefer to have a more general solution. We'd like exactly one search method, which *THandleArray*'s descendants could inherit or override as needed.

Remember, though, that with general data structures such as *THandleArray*, *TList*, or *TMailList*, the using programmer has to supply the data knowledge. There is no way to build it into the list. The using programmer will have to write some code to support searches. That's the price of generality. Figure 22-2 shows the division of labor between the list object and the using programmer.

List	Using programmer
Provides:	Provides:
• list	• data knowledge
• management methods	• data functions: =, >=, etc.
	• search functions and methods

Figure 22-2.
Division of labor in TList.

In this chapter, we'll look at three different ways to provide data knowledge to the list in support of searches. The first uses Pascal's procedural parameters. The second uses a special object class called *TSearchKey*. The third involves two ways of getting the data out of the list before testing it. In this chapter, we'll see procedural parameters, search keys, and one way to get data out. We'll look at the second way to search by getting data out of the list when we develop sequence classes in Chapter 23.

The Using Programmer Provides Search Functions

We've faced up to the fact that the polymorphic list code can't do the whole job by itself. This is a problem in which the designer of the polymorphic list contributes part of the solution and the consumer using the list must do the rest.

What can the consumer do? He or she can tell the list how to do comparisons for the kind of data being stored. The list supplies the polymorphism, and the consumer supplies the data knowledge.

But how? By passing the *Search* method a function as a parameter. The function handles the comparison and returns *true* or *false*.

Let's look at an implementation of *THandleArray.Search* that uses a procedural parameter:

```
function THandleArray.Search(key: TObject; function Compare (KeyObj,
    TargetObj: TObject): Boolean): Boolean;
  { Requires consumer to pass in a function defining the desired }
  {  kind of comparison of target objects, e.g. =, >=, <=, etc. }
  var
    lastFind, i: Integer;
    found: Boolean;
    elem: TObject
begin
  { Set the starting point }
  if self.fMark = kArrayBase then
    lastFind := self.fMark          { Starting from head }
  else
    lastFind := self.fMark + 1;     { One place beyond last find }
  { Loop through the list }
  for i := lastFind to self.fLength do
    begin
      { Retrieve current list element }
      elem := self.fArray^^[i];
      if Compare(elem, key) then
        begin
          found := true;
          self.fMark := i;          { Mark the find }
          Leave;                    { Exit the for loop }
        end;  { if }
    end;  { for }
  Search := found;
end;  { THandleArray.Search }
```

This *Search* function specifies a formal procedural parameter called *Compare*, which itself takes two *TObject* parameters. *Compare* is a Boolean function, returning *true* if the comparison is true, *false* if it isn't. Here's the function declaration again, with the procedural parameter in italics:

```
function THandleArray.Search(key: TObject; function Compare (KeyObj,
    TargetObj: TObject): Boolean): Boolean;
```

Inside the *Search* method, *Compare* is used this way:

```
if Compare(elem, key) then ...
```

The *if* statement calls the actual function that was passed in, passing it the current element's data and the key parameter that was passed to *Search*. (Each is some descendant of *TObject*.) Depending on how the comparison comes out, the body of the *if* statement is executed or bypassed.

Function *Compare* is a pretty anonymous creature. What does it really do? That depends entirely on what function is passed in as the actual parameter. Whatever that function is, though, it has to obey certain rules:

- It can have any name, not necessarily *Compare*.
- It must have two *TObject* parameters, and only two.
- It must return a Boolean value.

Any function that meets those requirements can be passed in and used within the *Search* method.

Compare's virtue is that it's so generic that we can define it any way we like. We can define it to see whether

- The two *TObject*s are equal (=)
- The first *TObject* is less than the second (<)
- The first *TObject* is greater than the second (>)
- The first *TObject* is greater than or equal to the second (>=)
- The first *TObject* is less than or equal to the second (<=)
- The two *TObject*s are unequal (<>)

We can even use *Compare* for compound Boolean expressions that involve more than one data key, as in the *FindDateAndSender* method we developed earlier in the chapter.

Pascal's Procedural Parameters

Procedural parameters are nothing new. They're a fixture of standard Pascal. A programmer can set up a procedure or function one of whose formal parameters itself calls a procedure or function. Then, inside the main procedure, the code calls whatever procedure or function is passed in as an actual parameter. If the parameter is set up as a procedure, we call it a "procedural parameter." If the parameter is set up as a function, we call it a "functional parameter." Either term can be used generically to include the other. The programmer can supply as an actual parameter any procedure or function whose own parameter list (and function return type, if a function) matches the profile of the formal procedural parameter. Unfortunately, the actual procedure or function can't be a method. See your Pascal's documentation.

How does *Compare* actually compare the two *TObjects*? First, it must know what underlying data type is expected. *TObject* conceals the true type stored. Second, it must convert the two *TObjects* to the expected type by typecasting. Then, third, it can compare their data values.

Here's an example, with some actual *Compare* functions of various kinds. The base data type this time is *TInteger*, which we've seen before.

The several function examples provide tests for =, for >=, and for <=, on data objects of type *TInteger*.

```
{ A set of functions for comparing TIntegers }
{  stored in TObjects }
{ Pass a function to THandleArray.Search or TList.Find }
{  to define a comparison relation }

function Equals (obj1, obj2: TObject): Boolean;
  { Pass to list.Find/Search }
  var
    elem, key: TInteger;
begin
  elem := TInteger(obj1);              { Data conversion }
  key := TInteger(obj2);
  Equals := (elem.Get = key.Get);  { Here's the test }
end;  { Equals--not a method }

function GreaterEqual (obj1, obj2: TObject): Boolean;
  { Pass to list.Find/Search }
  var
    elem, key: TInteger;
begin
  elem := TInteger(obj1);
  key := TInteger(obj2);
  GreaterEqual := (elem.Get >= key.Get);
end;  { GreaterEqual--not a method }

function LessEqual (obj1, obj2: TObject): Boolean;
  { Pass to list.Find/Search }
begin
  LessEqual := (TInteger(obj1).Get <= TInteger(obj2).Get);
end;  { LessEqual--not a method }
```

Each function converts its two *TObject* parameters. Then it does the comparison with a line like

```
GreaterEqual := (elem.Get >= key.Get);
```

This line assigns the value *true* if *elem*'s data value is greater than or equal to *key*'s data value; *false*, otherwise. The actual values for the comparison are obtained by sending Get messages to the two *TIntegers* being compared. *TInteger.Get* returns an integer value.

The last function uses a shortcut, eliminating the two auxiliary variables. The typecasts of *TObject* parameters to *TIntegers* are combined with the message sends and the test.

Depending on what we want to search the list for, we can use a variety of such functions in different calls to *Search*. For many purposes, an *Equals* function is the right one: It gives us the item that has the same data as the key we pass in. But we might sometimes want to search for the first element in the list less than our key value, or greater than, or not equal to, and so on. Such variations are useful for creating ordered lists based on the polymorphic *TList* or *THandleArray*. Given a new element to add, for instance, to an ascending list, we'd find the first element greater than the element's key and insert before that element.

Now we have a highly polymorphic list with considerable flexibility, as long as we provide it with the necessary comparison functions. We know how to make data elements findable by ensuring that each element has the necessary key fields (and methods to retrieve their values). And we can write any kind of *Compare* function we need.

This approach resembles the subclass-and-add-custom-search-methods approach we tried with *TMailList*, but with two main differences. First, the code to search the list is general, built into a *Find* or *Search* method, where it belongs. Second, the consumer supplies the data knowledge from outside the list—here in the form of comparison functions. The comparison functions are simple Pascal functions, not methods—you can't pass a method in as a procedural parameter.

The comparison functions have less to do than the search methods we added earlier, so they're simpler and easier to write. They have a pretty standard format, so it's easy to add a new function for any new search conditions we want to use.

But is there a more object-oriented way?

The Using Programmer Provides Search Key Objects

In our *Search* method using procedural parameters, we passed in two parameters: an object containing key information (*key*) and a comparison function (*Compare*). The one passes in data, and the other passes in functionality. Hmmm. That sounds familiar. Isn't there some other way to pass around both data and functionality?

You guessed it. Remember that an object carries both data and the methods to operate on that data. Comparison is certainly an operation on the data. So how about doing the whole job with an object—a search key object? Figure 22-3 on the next page shows schematically what a search key object is like.

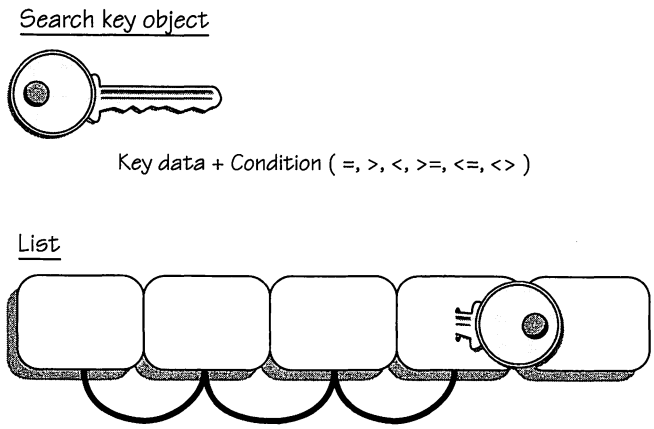


Figure 22-3.
The notion of a search key object. It compares the key data to each node.

At a high level, here's how it works. We pass the *Search* method a search key object as a parameter. This object parameter contains any data keys needed (in its instance variables) plus a *Condition* method that knows how to convert a *TObject* into, say, a *TInteger* and how to make the comparison, returning *true* or *false*. Inside *Search*, the code looks a lot like the code we used for the procedural parameter approach:

```
function THandleArray.Search (s: TSearchKey): Boolean;
  { Traverse array, using s's Condition method to compare data }
  var
    found: Boolean;
    lastFind, i: Integer;
    elem: TObject;
begin
  { Only bad search key we can detect is a nil }
  if s = nil then
    begin
      Search := false;              { Error--bad search key }
      self.fErrorHandler.SetError(kBadSearchKey);
    end
  else if self.fArray = nil then    { Do nothing for empty list }
    begin
      Search := false;              { Error--do nothing }
      self.fErrorHandler.SetError(kSearchEmptyArray);
    end
  else
    begin
      { Set the starting point }
```

```
      if self.fMark = kArrayBase then
        lastFind := self.fMark      { Starting from head }
      else
        lastFind := self.fMark + 1; { One place beyond last find }
      { Loop through the list }
      for i := lastFind to self.fLength do
        begin
          { Retrieve current list element }
          elem := self.fArray^^[i];
          if s.Condition(elem) then
            begin
              found := true;
              self.fMark := i;      { Mark the find }
              Leave;                { Exit the for loop }
            end;
        end; { for }
    end; { else }
  Search := found;
end; { THandleArray.Search }
```

The essential line of this code is

```
if s.Condition(elem) then ...
```

which is the OOP equivalent of what we did with the procedural parameter. *Search* calls the search key object's *Condition* method at the appropriate moment, passing a reference to the current element as a parameter. *Condition* returns *true* if the current element met the search criteria. If it did, *Search* puts the mark on the found element. Note that *Search* also starts at the mark (really one element beyond it), so if you want to search the whole list, you should first send the list a Head message to put the mark on the first element. Calling *Search* from there searches the whole list—until an element meets the search criteria.

TSearchKey

The first question is, Can *TInteger* be a search key object by itself? Probably not. To provide the necessary comparison functions inside *TInteger* (or any other *TLockable* class), we'd have to add multiple condition methods, each with a different name. But then what name would we call from inside *Search*?

This wasn't a problem with procedural parameters. We could pass in a function of any name, as long as it had the right parameter profile.

We need a new kind of object. With each data type, such as *TInteger*, we can associate one or more "helper" classes. These will be descendants of a class *TSearchKey*. For each kind of search, we declare a new subclass of *TSearchKey*, one that has the necessary key fields and that has a *Condition* method that can convert a *TObject* into the appropriate data type and then compare its data with the key data in some fashion. *Condition* takes one parameter of type *TObject* (or a descendant) and returns a Boolean value.

For example, the three mail searches we specified earlier in this chapter:

- sender = key sender
- date >= key date
- (sender = key sender) and (date >= key date)

would each be embodied in a new subclass of *TSearchKey*. Let's call the classes *TSenderEQKey*, *TDateGEKey*, and *TSenderEQDateGEKey*. Our convention for naming search key classes is to specify a key and the condition that applies to it; for multiple conditions (and/or keys), we specify all the conditions in sequence. This makes for long names, but that's helpful when we need to understand at a glance which search key object is which.

The following example uses *TList* and its *Find* method, which calls *Search*.

Given declarations for the search key classes (which we'll get to shortly), we can use them this way:

```
var
  aSenderEQ: TSenderEQKey;
  aDateGE: TDateGEKey;
  aSenderEQDateGE: TSenderEQDateGEKey;
⋮
New(aSenderEQ);                    { Create search key object }
aSenderEQ.Init('Buji');            { Initialize its key field }
if mailList.Find(aSenderEQ) then   { Pass the search key object to Find }
  elem := mailList.Extract;
⋮
New(aDateGE);
aDateGE.Init(yesterday);           { A variable with a date }
if mailList.Find(aDateGE) then
  elem := mailList.Peek;
⋮
New(aSenderEQDateGE);
aSenderEQDateGE.Init(theSender, theDate);
if mailList.Find(aSenderEQDateGE) then
  mailList.Delete;
⋮
aSenderEQ.Free;
aDateGE.Free;
aSenderEQDateGE.Free;
⋮
```

(By the way, this code is practically identical for *TList* and *TLinkedList*, thanks to the similarity of their interfaces.)

A particular application would declare any subclasses of *TSearchKey* that it needed.

Here's *TSearchKey*, an abstract class that does little but serve as a template for subclassing (and as a type for parameter lists):

```
type
  TSearchKey = object(TObject)

    function TSearchKey.Condition (target: TObject): Boolean;
  end;  { Class TSearchKey }
```

TSearchKey has no instance variables and its *Condition* method is just a stub.

Sample *TSearchKey* subclasses

Here are the three *TSearchKey* subclasses we've declared for searching a list of mail:

```
type
  { TSenderEQKey is for searching for a mail item }
  {  whose sender equals a key }
  TSenderEQKey = object(TSearchKey)

    fKey: Str255;                { String-type key for sender information }

    procedure TSenderEQKey.Init(sender: Str255);
      { Initializes key values }
    function TSenderEQKey.Condition (target: TObject): Boolean;
    override;
      { Performs data conversion and conditional testing }
      {  between key and current data element in a list }
  end;  { Class TSenderEQKey }

  { TDateGEKey is for searching for a mail item whose }
  {  sender is greater than or equal to a key }
  TDateGEKey = object(TSearchKey)

    fKey: DateType;

    procedure TDateGEKey.Init(date: DateType);
      { Initializes key values }
    function TDateGEKey.Condition (target: TObject): Boolean;
    override;
      { Performs data conversion and conditional testing }
      {  between key and current data element in a list }
  end;  { Class TDateGEKey }

  { TSenderEQDateGEKey is for searching for a mail item whose sender }
  {  equals a key and whose date is greater than or equal to a key }
  { Example of encapsulating a compound condition in an object }
```

```
TSenderEQDateGEKey = object(TSearchKey)
  fKey1: Str255;              { String-type key for sender info }
  fKey2: DateType;            { Date-type key for date info }
  procedure TSenderEQDateGEKey.Init (sender: Str255; date: DateType);
    { Initializes key values }
  function TSenderEQDateGEKey.Condition (target: TObject): Boolean;
  override;
    { Performs data conversion and conditional testing }
    {  between key and current data element in a list }
end;  { Class TSenderEQDateGEKey }
```

As our comparison functions do, these classes follow a pattern: We declare one or more instance variables (of the types needed as keys); we include an initialization procedure with one or more parameters matching the key fields; we override *TSearchKey.Condition*. We didn't declare the initialization procedure in *TSearchKey* because it will take such varied parameters in the subclasses. Likewise, each subclass has to take care of its own instance variables because their number and types will vary greatly. Here's a sample *Init* method:

```
procedure TSenderEQDateGEKey.Init (sender: Str255; date: DateType);
begin
  self.fKey1 := sender;
  self.fKey2 := date;
end;  { TSenderEQDateGEKey.Init }
```

Condition methods

The *Condition* method for each *TSearchKey* subclass also follows a pattern. It checks the current element in the list to be sure it's the right data type (in this case a mail element). It then typecasts the current element to the type expected, extracts the element's data items, and compares them with the key field(s).

```
function TSenderEQDateGEKey.Condition (target: TObject): Boolean;
  override;
  var
    mail: TMail;              { Need two auxiliary variables because date is }
    date: DateType;           {  a record variable }
begin
  if not Member(target, TMail) then
    { Note use of Member function to see whether we can use item }
    Condition := false
  else
    begin
      mail := TMail(target);  { Typecast TObject to TMail }
      date := mail.DateOf;    { Get mail item's date information }
```

```
        { Conditional testing--note compound expression and two keys }
        if (mail.SenderOf <> self.fKeySender) then
          Condition := false
        else
          begin
            { Check the mail's date against key }
            if date.year > self.fKeydate.year then
              Condition := true
            else if date.year = self.fKeydate.year then
              if date.month > self.fKeydate.month then
                Condition := true
            else if date.month = self.fKeydate.month then
              if date.day > self.fKeydate.day then
                Condition := true;
            else
              Condition := false;
          end;  { else is matching sender }
      end;  { else target is TMail }
  end;  { TSenderEQDateGEKey.Condition }
```

This method is smart enough not to try typecasting oddball list elements that aren't mail into *TMail*. Of course, we probably aren't going to keep a list that contains mail and non-mail. But that kind of discrimination could be very important in circumstances in which we do have a mixed list, with objects of several perhaps only loosely related types. For instance, if the current element were of type *TInteger* (again—someone somewhere will mix integers and mail for some as yet unfathomable reason), this code would just skip the element, reporting *false* to indicate that, no indeed, this is not the mail we're looking for. This feature will keep someone from blowing up a list someday. Figure 22-4 shows how a search key object can skip nodes that aren't the right type.

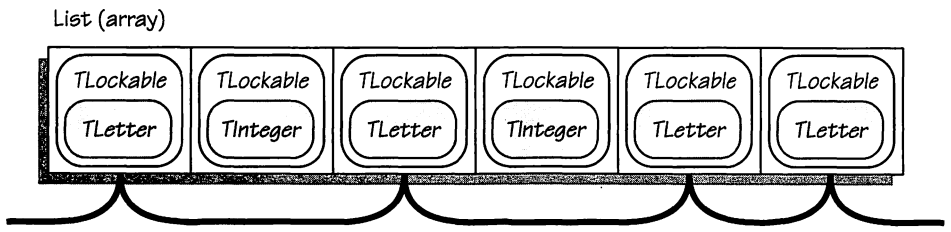


Figure 22-4.
Skipping oddball elements.

After we've used the *Member* function to be sure an element is mail, we can safely typecast it. Then we can extract the pieces of data the method needs—the date and sender. We get the data by sending DateOf and SenderOf messages to the current item, which we now know it has because it's mail.

Then we use a multi-part compound Boolean test to set the value to be returned by *Condition*. If the data passes all tests, the function returns *true*. If the data fails any one of them, the function returns *false*. Note that because *date* is a record type, we use record-style dot notation to access its fields for testing. We've also taken care to ensure that the Boolean expressions are phrased correctly: We're looking for data that is >= the key, so the key goes on the right-hand side of the test.

This model for condition methods will accommodate any number of keys (just put them in your subclass of *TSearchKey*) and any sort of Boolean test expression, from a simple one to a very complex one like our multi-parter.

Each *TSearchKey* subclass contains both the key data and the ability to compare the key data with list data for some set of keys and some condition. We can create as many subclasses as we need.

Searching a List for a Type

While we're on the subject of search key objects, let's consider one more search problem: finding the first object of a given type in a list. In a mail list, for instance, we might want to find the first *TLetter*, which might be the third or the tenth item.

We'll look briefly at two related solutions, both of which use search key objects of a special sort designed to locate a given type.

For the first approach, we'll declare a subclass of *TSearchKey*—say, *TTypeEQLetterKey*—whose condition method contains this line:

```
Condition := Member(target, TLetter);
```

The method uses the *Member* function as its Boolean test for the condition.

The second approach is in some ways more flexible and useful, although it involves larger changes to the object hierarchy. Recall that in Chapter 8 we developed the *TypeOf* method, a new method for *TObject* that reports the name of the class as a 30-character string. Classes that inherit from *TObject* can override *TypeOf* to report their own names; otherwise, they'd report the name 'TObject'.

With *TypeOf*, we can ask an object in a list What type are you? The *Condition* method in our search key object contains a line like

```
Condition := (target.TypeOf = fTheType);
```

where *fTheType* contains a *Str30* string. Now we can pass in any arbitrary type-name string, as a variable, to a more general search key object. So long as *target* is an object whose class overrides *TypeOf*, we're in business.

Using *TSearchKeys*

Let's summarize using a search key object:

- Declare a *TSearchKey* subclass with the right key fields, an initialization method, and an overridden condition method
- Declare a variable of the new type
- Call *New* with the variable as its parameter to create the object
- Send the object an initialization message to set its key fields
- Pass the object as a parameter to the list's find method

You can use search key objects in other places in which you have to search—in trees or other data structures. Chapter 25 will go into some of these other uses.

At Last, *THandleArray.Search*

Now we can look at the final version of *Search*, as it appears in unit *UHandleArray* on the code disk. Then we'll look at *TList.Find*, which calls *Search*.

```
function THandleArray.Search (s: TSearchKey; var index: Integer): Boolean;
  var
    elem: TObject;
    found: Boolean;
    len: Integer;
begin
  if self.fErrorHandler = nil then
    begin
      found := false;                { Something is wrong with list }
      index := 0;
    end
  else
    begin
      self.fErrorHandler.SetError(kNoErr);
      { Error if search key isn't an object }
      if s = nil then
        begin
          found := false;
          self.fErrorHandler.SetError(kBadSearchKey);
        end
      { Error trying to find in empty list }
      else if self.LengthOf = 0 then
        begin
          found := false;
          self.fErrorHandler.SetError(kEmptyArray);
        end
```

```
        { OK to find }
        else
          begin
            len := self.LengthOf;
            found := false;
            while (not found) and (index <= len) do
              begin
                elem := self.fArray^^[index];
                found := s.Condition(elem);
                if found then
                  begin
                    Search := true;
                    Exit(Search);
                  end
                else
                  index := index + 1;
              end;  { while }
          end;  { else }
        if not found then
          index := 0;
      end;
    Search := found;
  end;  { THandleArray.Search }
```

This function starts at the item corresponding to the index value passed in as a *var* parameter. When it finds the target, it returns the target's index value in the same parameter for use by the caller. We'll see shortly that *TList.Find* makes use of this mechanism to implement its ability to make multiple finds in the list. By passing in an appropriate index value, *Find* can control where *Search* looks.

If there's no error-handler object to send error reports to, the *Search* function reports failure. There's something wrong with a list that has no error handler.

The *Search* function reports an error if a *nil* search key object reference has been passed in. If that happens, something is very wrong. At this point, we could have the function fail, but it's more polite to post an error message. It's up to the consumer to check for the error and take appropriate action.

The function also reports an error if the array happens to have no contents. The error report is merely an advisory. The error itself has no bad consequences.

The function gets down to business in the *while* loop, moving through the array, retrieving the next stored object, and passing it to the search key object's *Condition* method for testing.

THandleArray knows nothing about that test, of course. It simply sends the search key object a Condition message and waits for a Boolean response, which controls its further action.

And Now, *TList.Find*

Recall that *TList.Find* sets up the list's mark position as the index value on which to base a search. Then it calls its inherited *THandleArray.Search* method, which we've just seen. When the search is successful, *Find* resets the mark to the index value *Search* returns—marking the found item. Otherwise, the mark doesn't move. Here's *TList.Find*:

```
function TList.Find (s: TSearchKey): Boolean;
  var
    found: Boolean;
    index: Integer;
begin
  { Error trying to find in empty list }
  if self.fLength < 1 then
    begin
      found := false;
      self.fErrorHandler.SetError(kEmptyArray);
    end
  { OK to find }
  else
    begin
      { Position mark for next search }
      { Assume search starts at head }
      if self.fMark = kArrayBase then
        { Ensure looking at first element }
        index := self.fMark
      else
        { Assume repeat from last find }
        index := self.fMark + 1;  { Start at position just after mark }
      { Check for index number out of range }
      if index < kArrayBase then
        begin
          index := kArrayBase;
          { Assume head intended, but post error message }
          self.fErrorHandler.SetError(kOutOfRange);
        end
      else if index > self.fLength then
        begin
          index := self.fLength;
          { Assume tail intended, but post error message }
          self.fErrorHandler.SetError(kOutOfRange);
        end;
```

```
        { Do the actual search from index number onward }
        found := Search(s, index);   { Call THandleArray method }
        if found then
          self.fMark := index;
        Find := found;
      end;  { else }
    end;  { TList.Find }
```

The first few lines set the index value for starting the search. If the consumer called *Head* before calling *Find*, the *Find* function assumes that he or she wants to search every item in the list, starting with the first one. If the mark isn't at the head, the function assumes instead that either the consumer wants to search a sublist or he or she wants to make a subsequent search following an earlier one. In either of those cases, *Find* sets the index value to *fMark + 1*. The value of *fMark* would have been set to the index value of the previous find, and we don't want to keep finding the same item again and again, so we start the next find one element beyond the mark.

Then *Find* checks the index value to verify that it's in range. If it isn't, *Find* assumes that the consumer wanted the mark to be (and the search to start) at the end of the list, so we set the index value to the end value. But *Find* also signals the anomaly with an advisory error message. Our general philosophy of error checking in reusable software components is to call for failure only if the component itself has been corrupted (or if it might have been). Otherwise, we try to do something reasonable and predictable but alert the caller that something strange has occurred. If the calling code always calls the component's error method after calling the component's other methods, the calling code can do something appropriate in response. But that response is left in the consumer's hands.

Finally, *Find* calls the *Search* method inherited from *THandleArray*. If the *Search* function returns *true*, the mark is set to the returned index value.

Search key objects are a pretty good solution: They're flexible, easy to write, and easy to use. They also make good use of OOP principles. We discussed their main alternative earlier in this chapter—both MacApp's *TList* class and TCL's *CList* class pass in procedural parameters. Your choice.

Finding Duplicates

Some lists will allow duplicate items; some won't. One way to screen out duplicates is to search the list for an item equal to the one you're about to add and then to add the item only if no matching item is found.

But when a list does allow duplicates, as *THandleArray* and its descendants do by default, a search that finds only the first occurrence is not enough by itself.

Suppose, for instance, that we wanted to find not only the first piece of mail from our key sender but all pieces from that sender. Or suppose that we used a *TList* to implement the auto parts bin we designed in Chapter 6. We might in some cases want the first three parts of a type, or all of them.

We've said this before, but it bears mentioning again. We've implemented *THandleArray.Search* so that it starts at the current mark. So does *TList.Find*. To search the entire list, we need to send the list a Head message before we send Find. After the first find, we can just send Find again, repeatedly, to advance through the whole list, finding each successive matching element.

Finding Without *Find*, Part I

There are two more ways to search a list, neither of which requires a find method. Both approaches rely on pulling the data out of the list to examine it, which is certainly another way to deal with encapsulation. The first uses the *Head*, *Next*, and *Peek* methods. The second uses a sequence class, which we'll get to in Chapter 23, under "Finding Without *Find*, Part II."

Finding by Peeking

One way to externalize a data item in a list—so that we can do things like typecasting it to *TMail* and extracting its instance variables and comparing them with keys—is to send the list a Peek message. Recall that the *Peek* function returns a reference to the item currently marked.

We also have ways of traversing the list to peek at element after element. We can use Head and then repeated Next messages until we reach the end of the list. Here's code that will do a search for a sender key in a mail list:

```
var
  current: TObject;
  mail: TMail;
  list: TList;
  keySender: Str255;
  found : Boolean;
⋮
keySender := 'Jones';
found := false;
list.Head;                          { Put mark at list head }
current := list.Peek;               { Get the head item }
if current <> nil then              { If list isn't empty }
  begin
    while (not found) and (fErrorHandler.Error <> kOffRight) do
      begin
        mail := TMail(current);     { Typecast item }
        if mail.SenderOf = keySender then
          found := true
        else
```

```
            begin
              list.Next;              { Get next item }
              { If already at end, this produces }
              {  kOffRight error code }
              current := list.Peek;
            end;
        end;  { while }
      if found then
        { Do something with the found item }
      else
      { Do something else }
    end;  { if current }
  :
```

This code is outside the list—perhaps in our main program or in a procedure—although we could indeed write this code as a method in a subclass of *TList*.

We start by moving the mark to the head item and then accessing that item with *Peek*. Then, in the loop, we typecast the item to *TMail* and test its sender against our key. If we have a find, we set *found* to *true* and get out of the loop. If we don't have a find, we use *Next* to move the mark to the next item, access the item with *Peek*, and loop again. The loop terminates if there's a find or if we move past the last element in the list.

Because of the way *Next* behaves—stopping at the last element—we have to get tricky to check for the end of the list. When *Next* brings us to the last element, *TailOf* will be *true*. But if we use that as the stopping condition, we'll leave the last element untested against our sender key. Instead, we rely on the fact that sending a Next message again when the mark is already at the last element produces the error code *kOffRight*. We can retrieve this code with the *Error* method from *TErrorHandler*. If the latest error code is *kOffRight*, we've processed all elements. So we use

```
fErrorHandler.Error <> kOffRight
```

as the stopping condition for end-of-list. (The only other list message that could interrupt the search before we get to examine all elements is the final Peek message at the top of the loop, but it sets an error code only if the list is empty—which by now we know it isn't.)

That this is cumbersome might be an argument for restoring *TList.OffEnd* as a method rather than a private procedure not available to the consumer. Of course, now that we've arrived at a general solution to the search-find problem, using *Peek* to find an item in the list isn't really necessary.

Summary

In this chapter, we completed all but a few details of the list software component *TList* and its array-based ancestor, *THandleArray*. We explored several ways to search a polymorphic, encapsulated list or array.

Along the way, we developed a useful, and rather exotic, object class: *TSearchKey*. And we completed *THandleArray*'s *Search* and *TList*'s *Find* methods.

We have at least five ways to search a polymorphic, encapsulated list:

- With custom-made data-specific methods in a subclass
- With procedural parameters
- With search key objects
- With *Next* and *Peek*
- With sequence objects, which we'll cover in the next chapter

Projects

- Because stacks and queues don't typically require searching, the search issue might seem inapplicable to them. But in a carefully controlled stack or queue, we might well want to know what objects are in the underlying list even if we can't access them except at the top of the stack (front of the queue). Develop a stack or queue based on either *THandleArray* or *TList*. Adapt *Search* and *Find* to fit your new class.
- Consider whether any of *TList*'s methods might be dangerous in a stack or queue. Which of *TList*'s methods would you eliminate from your stack? From your queue?
- Implement a version of *THandleArray* that uses procedural parameters for *Search*. Compare the ease of use and versatility of that interface with the search key object approach.
- Think about uses for *TDynamicArray*. Is its interface complete? Is anything in its ancestor, *THandleArray*, inappropriate in an instantiable dynamic array class? Make any changes you see fit to.
- Can you think of any improvements to *THandleArray*, *TList*, or *TDynamic-Array*? Implement them.

23

POLYMORPHISM: THE DOTOEACH PROBLEM

In Chapter 22, we looked at the find problem in relation to encapsulated, polymorphic lists. A related problem involves traversing an encapsulated list to send arbitrary messages to its elements. In other words, the problem involves wanting to *do* something *to each* element. How do we know that all the elements will have the necessary methods? And how can we reach inside the list's encapsulation to send just any old message—including messages that couldn't have been foreseen when the list was written? We'll look at several solutions:

- Traversing with a procedural parameter
- Traversing with a *DoCommand* method
- Traversing with *Next* and *Peek*
- Traversing with sequence objects
- Traversing with traversal action objects

Enumerating the List

One of the classic moves an OOP program makes is to traverse a list of objects, sending each object the same message. Each object in its turn is supposed to respond in its own way to that message, polymorphism providing the proper response. This process is called "enumeration" or "iteration." We'll use "traversal" as a generic term for the same phenomenon. Figure 23-1 on the next page will give you the idea.

We've built provision for one such traversal right into our lists. We're sure of a need to display all elements, so *TList* has a *Display* method, inherited from *THandleArray*, that traverses the list sending Display messages to all nonempty items.

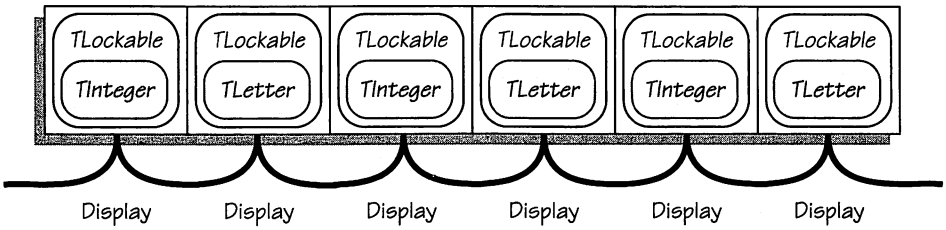


Figure 23-1.
Traversing a list to send each element a message.

But what about all the many other possible messages we could send to the list items? Because the list is highly general, its items could have a huge variety of methods. We might want to send each object in the list a message such as Display (display yourself), Accumulate (add yourself to a running total), or Process (do process X to yourself). It turns out to be a bit hard to loop through the list doing useful—but unpredictable—procedures.

Because the list is encapsulated, we can't have a traversal method that sends any arbitrary message whatever to all elements. As Figure 23-2 suggests, the general code of the list can't know anything about the data—nor about what messages can legally be sent to whatever objects are stored in the list.

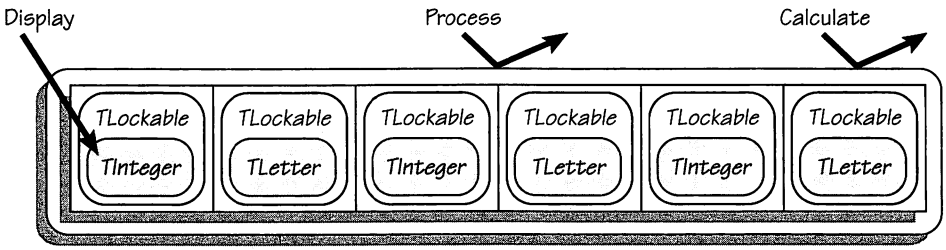


Figure 23-2.
Encapsulation enforcing inaccessibility. You can send only those messages that are known to the list regardless of what messages the elements know.

Let's be more specific about the problem with encapsulation. In our *THandleArray.Display* method, the sending of the Display message to each element of the array is itself encapsulated in the method. We know what message we'll want to send, so we build it right in. We can't, of course, build in all possible messages. But is there a way to "pass in" a message to send, something like what we did with our compare functions and our condition methods?

Passing in the Message to Send?

Let's declare a possible *TList* method called *DoToEach* that knows how to traverse the list. We'll pass *DoToEach* a message to send as a parameter:

```
procedure TList.DoToEach (msg: WhatType);  { Doesn't work }
```

What's the type of *msg*? We have the name of the method, *msg*. But how do we specify that method in order to send a message to the target object? As a string? As something else? This turns out to be a dead end. We can't do it.

Passing in a Procedure or Function?

Let's take a different tack. Suppose we want to send a Process message to each item in the list. Our hypothetical *Process* method is a parameterless procedure, so let's declare *DoToEach* this way:

```
procedure TList.DoToEach (procedure DoIt);
```

Then we pass in the Process message this way:

```
aList.DoToEach(TInteger.Process);     { Doesn't work }
```

Sorry, but that doesn't work, either. The phrasing "TInteger.Process" is unacceptable. The name before the dot must be a variable identifier (an object), not a type identifier (a class). Furthermore, Object Pascal won't accept a method name as a procedural parameter, any more than it will accept any of Pascal's built-in procedures, such as *Writeln*. Too bad, but it won't.

Using a Cloaking Device?

As Figure 23-3 on the next page suggests, a cloaking device offers a kind of Klingon or Trojan horse solution. The idea is to pass a procedure or function as a parameter, but inside the actual procedure passed, to hide a message-send to the object. The *DoToEach* method might look like this:

```
procedure TList.DoToEach(procedure DoIt(element: TObject));
  var
    traverse : Integer;
    elem: TObject
begin
  traverse := kArrayBase;
  while self.fArray^^[traverse] <> nil do
    begin
      elem := self.fArray^^[traverse];
      DoIt(elem);                    { Do it to that element }
      traverse := traverse + 1;
    end;
end;  { TList.DoToEach }
```

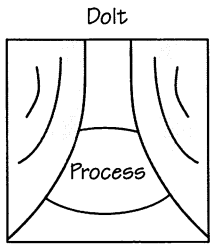


Figure 23-3.
A cloaking device. DoIt *masks* Process.
We say, "DoIt," *and it does* Process.

Then we'd write a *DoIt* procedure (it's not a method) outside *TList*'s unit. We'd have to know in advance what data type *DoIt* is to work on because it must typecast the *TObject* passed to it to the appropriate type. *TObject* doesn't have the *Process* method we want to send to objects in the list. If we try something like this in the body of *DoIt*:

```
procedure DoIt(element: TObject);
begin
  element.Process;                    { Won't work!! }
end;  { DoIt--not a method }
```

we'll get a compiler error. Process is a message to a *TObject*, and *TObject* doesn't have the *Process* method. (Neither does the *TLockable* probably stored in the element.) Never mind that there's undoubtedly a *TInteger* (that has a *Process* method) "inside" the *TObject*. The compiler has no way of knowing that.

So we have to do this instead:

```
procedure DoIt(elem: TObject);
  var
    convertedData: TInteger;          { Or whatever type we expect }
begin
  convertedData := TInteger(elem);    { Conversion }
  convertedData.Process;              { Will work!! }
end;  { DoIt--not a method }
```

We've seen this kind of typecasting often.

Any flaws in this approach? Well, as with the *Find* method, we have to write custom *DoIt* procedures for each data type that might be in the list. That's acceptable. Probably the biggest difficulty is that the *DoToEach* mechanism isn't very general. The current version of *DoToEach* takes one and only one parameter—a procedural parameter that has one *TObject* parameter. What if the method invoked by the DoIt message takes some parameters itself, and what if the method invoked is a function method that needs to return a value? Can we do those things with *DoToEach*?

To some degree we can. The little procedure we pass in to *DoToEach* simply has to do more—such as supplying the necessary parameters inside its body (hard-wired in) or getting, and dealing with, a function return value. We can also use tricks like passing a generic pointer (type *Ptr*) to the actual parameter we want to pass to *DoIt*:

```
procedure TList.DoToEach(procedure DoIt(element: TObject; param: Ptr));
```

Then we can obtain a *Ptr* to whatever the parameter is (using the @ operator), pass it in *param*, and dereference it inside *DoIt* to use it. This will make *DoIt* a bit more flexible. *DoToEach* might also be a function returning a *Ptr*, in the same spirit. This technique for passing parameters is illustrated in the search methods for TCL's *CList* class.

Using a Command System?

At the heart of all the schemes we've seen so far is the fact that "sending a message" means sending some unpredictable message to each element. MacApp uses a completely different approach, a system of commands.

The idea in this scheme is that each element would have a *DoCommand* method that would take an integer command code as its parameter. The command codes could be assignments like

```
kSaveCmd = 17;
kOpenCmd = 19;
kDisplayCmd = 46;
kAccumulateCmd = 97;
⋮
```

When we wanted to have all the elements in the list display themselves, we'd send

```
list.DoToEach(kDisplayCmd);
```

DoToEach would contain the kind of straightforward list-traversal code we've seen over and over again, so we won't look at that code now. See *THandleArray.Display* on the code disk for an example. The heart of *DoToEach* in our context here would be something like this:

```
procedure THandleArray.DoToEach(theCmd: Integer);
begin
  ⋮
  currentElement.DoCommand(theCmd);
  ⋮
end;  { THandleArray.DoToEach }
```

Each element's *DoCommand* method might be inherited, perhaps from *TObject* or *TLockable*, and *DoCommand* would contain a *case* statement itemizing all the command cases the element knows how to handle:

```
procedure T——.DoCommand(whichCmd: Integer);
begin
  case whichCmd of
    kDisplayCmd:   self.Display;
    kAccumulateCmd:   self.Accumulate;
    :
    otherwise
      ;
  end;  { case }
end; { T——.DoCommand }
```

On receiving a DoCommand message, the element would execute *DoCommand*'s *case* statement to check whether it knew the command specified by *whichCmd*. If it knew the command, it could respond to the command appropriately. If the element didn't recognize the command, the element would take the *otherwise* branch in the *case* statement and do nothing. Or it might call *inherited DoCommand*.

This is a pretty successful approach. It's basically the technique MacApp uses to pass commands to objects. Each element would have to have a *DoCommand* method—it might be a stub inherited from some ancestor. We'd need a list of command codes. And we might need to expand *DoToEach* and *DoCommand* to add a *Ptr*-type parameter for passing extra data in with the command. We'll keep this in mind.

We're not quite finished with our examination of approaches to the enumeration problem. Let's look at three completely different—and more object-oriented—solutions, all of which work rather well, although they do require some work on the part of the consumer just as using a cloaking device and using a command system do.

Letting *TList* Enumerate Itself

Our next two solutions to enumerating a list are approaches we've already looked at in the context of searching: pulling the data out of the list so that we can examine it or operate on it—or send it messages.

Peek and *Find*

The first of these approaches, which we'll simply mention without demonstration, uses *Head*, *Next*, and *Peek*, the way we did for a findless find in Chapter 22. This peek mechanism, coupled with a *Find* procedure, could be the basis for building a variety of utility procedures (or methods in a *TList* subclass) in support of lists of specific kinds of data.

But the second approach is more intriguing. That's the one we'll look at in detail.

Using Sequence Classes

The sequence approach, borrowed from the practices of programmers using the Objective-C language (Cox 1986a), is more object oriented.

The idea is that each kind of list implementation (whether array based or pointer based) carries along a sequence class that can provide the enumerating ability to the list. The sequence mechanism has two basic parts: a method in the list class called *EachItem*, which creates a new instance of the sequence class when called; and the resulting sequence object itself, which has the *NextItemOf* method that can be invoked to deliver the next item in the list. The sequence object keeps track of its current place in the list so that it can deliver the next item. A new sequence object is created, used, and freed each time we need to traverse the list; or we can create the sequence object, use it, and then reset it for another pass. Figure 23-4 illustrates the idea of sequencing a list.

You can use this mechanism on an *ad hoc* basis, outside the list object, setting up a loop to traverse the list each time you need to. Or you can write a subclass of your list with new methods that contain traversal loops based on a sequence object.

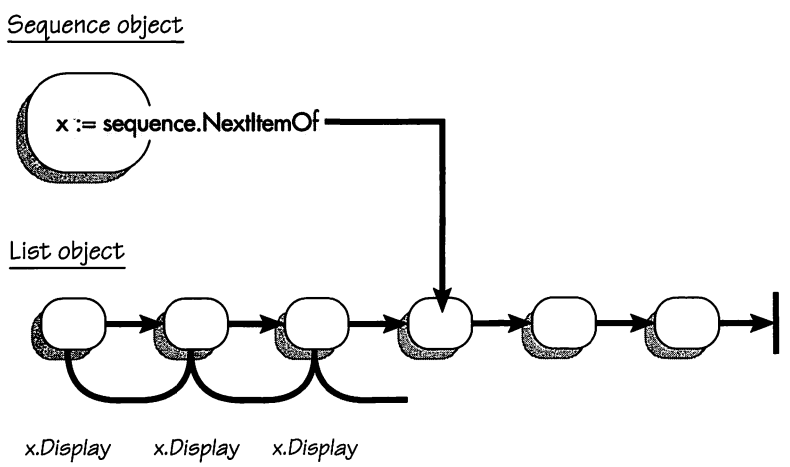


Figure 23-4.
Sequencing a list.

The Sequence Classes

We'll look here at one particular kind of sequence class: one for an array-based list, such as *TList*. A sequence class must be a subclass of *TSequence*, which exists only as an abstract superclass designed to solve a Pascal problem with declaration order. Here's an example to illustrate that need for *TSequence*. In the declaration of *THandleArray*, we declare an *EachItem* method, which returns a *TArraySequence* object tailored to the array's needs. However, we declare *TArraySequence* below the

THandleArray declaration. It's there because it contains an instance variable and a method parameter of type *THandleArray*. The two types have a mutual dependency. Figure 23-5 illustrates the class relationships and the class declaration order.

Because *TArraySequence* is necessarily declared below *THandleArray*, we have a problem with the return for *THandleArray.EachItem*. We can't declare the type of its return value as *TArraySequence* because that name hasn't been declared yet. So we declare

```
function THandleArray.EachItem: TSequence;
```

using *TSequence* as a placeholder for the actual *TArraySequence* value that the method returns. *TSequence* must be declared above *THandleArray*.

We would use a similar arrangement for the sequence classes associated with other kinds of lists, such as *TLinkedList* lists.

When we call an array's or a list's *EachItem* method, we have to typecast the result to the type of *TSequence* descendant we expect. For example, when we call an array's *EachItem* method we have to perform this typecast to *TArraySequence*:

```
aSequence := TArraySequence(list.EochItem);
```

Here's the superclass *TSequence*, which declares no instance variables or methods:

```
type
  TSequence = object(TObject)
    end;  { Class TSequence }
```

Underneath, *TSequence* is really only *TObject*.

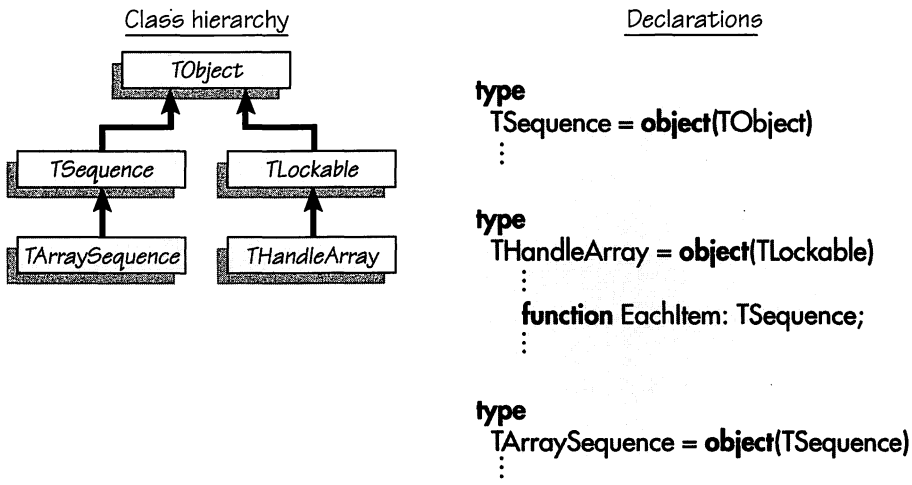


Figure 23-5.
The position of TArraySequence *in the class hierarchy and the order of its appearance in the declarations of classes.*

Class *TArraySequence*

The sequence class for *THandleArray* looks like this:

```
type
  ArrayTops = (arrayLen, arrayHi);
type
  TArraySequence = object(TSequence)

      fList: THandleArray;
      fNextItem: Integer;

      procedure Reset(list: THandleArray; pos: Integer);
      function NextItemOf: TObject;
      function PositionOf: Integer;
      function AtEnd (top: ArrayTops): Boolean;
      { Inherits Free from TObject }
    end;  { Class TArraySequence }
```

The instance variable *fNextItem* is used to store the index value of the next item in the list. After creating a sequence object of the *TArraySequence* class, using code like this:

```
aSequence := TArraySequence(aHandleArray.EachItem);
```

the consumer can repeatedly send the sequence object a NextItemOf message to get the object stored at the next location in the array. *EachItem* initializes *fNextItem* to *kArrayBase*, but *fNextItem* can be reinitialized to any array index value by means of the *Reset* method. (You can use *Reset* to start a sequence anywhere in the list.)

When the value of *fNextItem* goes out of the array's range, the utility method *AtEnd* returns *true*, signifying that there are no more items to process:

```
while not aSequence.AtEnd(arrayLen) do
  ⋮
```

One of two possible parameters, of type *ArrayTops*, can specify what *AtEnd* should consider to be the end of the array. If you pass the value *arrayLen* to *AtEnd*, it returns *true* if *fNextItem* has passed the last occupied slot in the array (the end of the logical array). If you pass *arrayHi* to *AtEnd*, it returns *true* if *fNextItem* is now greater than the number of allocated slots (the end of the physical array). This lets you work either with only the part of the array currently in use (up to the last data item) or with all the allocated space.

AtEnd's versatility also helps when you are working with a *THandleArray* descendant such as *TList*, where only *AtEnd(arrayLen)* is meaningful. In fact, for *TList*, we declare a subclass of *TArraySequence*.

```
type
  TListSequence = object(TArraySequence)

      procedure Reset(list: THandleArray; pos: Integer);
      override;
      function AtEnd(top: ArrayTops): Boolean;
      override;
    end;  { Class TListSequence }
```

The overridden *Reset* checks that *pos* is not beyond the end of the list (the logical array), so you can't use it to start sequencing from an array element beyond the end of the list. The overridden *AtEnd* always responds as if it had received the *arrayLen* parameter. If you sent the message *AtEnd(arrayHi)*, *AtEnd* would treat it as *AtEnd(arrayLen)*. The other *TArraySequence* methods are inherited unchanged.

Using *TArraySequence*

TArraySequence.NextItemOf returns the *TObject* item at position *fNextItem* in the array. Then it increments *fNextItem*. You can use the reference to the item to do something useful, such as sending the item a message. The item remains in the array.

The consumer can traverse the array with something like the following code, which gets a reference to each item in turn and sends the item two messages. The first message is Display. Because we modified *TObject* a few chapters ago, *TObjects* recognize Display and we don't have to typecast the data from *NextItemOf*. The second message is Process. *TObject* doesn't have a *Process* method, so we do have to typecast the data to the type we expect. Here's the sample code:

```
var
  aSequence: TArraySequence;
  obj : TSomeType;
⋮
{ Create and initialize sequence object--note the typecast }
aSequence := TArraySequence(anArray.EachItem);
while not aSequence.AtEnd(arrayLen) do  { Loop through list }
    begin
      obj := aSequence.NextItemOf;       { Get an item }
    if obj <> nil then
      begin
        { Our modified TObject knows Display; no conversion }
        obj.Display;
        { TObject lacks a Process method; must convert }
        TSomeType(obj).Process;
      end;
  end;
{ Release sequence object }
aSequence.Free;
```

Instead of freeing the sequence, we could reset it and use it again:

```
{ Set up for another pass }
list.Head
aSequence.Reset(list, list.PositionOf)
```

In pseudocode, the processing steps are

```
Create a sequence object (and typecast it)
While not at end of sequence, do
  Get the next item                { Starts with first item }
  Do something with it
End while
Free the sequence object
```

At the end of the routine in our example, each item in *anArray* has been sent both the Display message and the Process message and the array is still intact. *Nil* elements are skipped because sending them messages would cause a runtime error. We enumerate only up to the "length" of the underlying logical array, by testing for *AtEnd(arrayLen)* instead of for *AtEnd(arrayHi)*, the end of the physical array.

As a variation, and to make the sequencing more general, we might have *NextItemOf* return an integer value—the index value of the next item in the array—instead of the item itself. Then we could use *NextItemOf* this way:

```
obj := Peek(aSequence.NextItemOf);
```

Here's the code for the version of *NextItemOf* that returns a *TObject*:

```
function TArraySequence.NextItemOf: TObject;
begin
  if self.fNextItem > self.fList.fSlots then
    NextItemOf := nil
  else
    begin
      NextItemOf := self.fList.fArray^^[self.fNextItem];
      self.fNextItem := self.fNextItem + 1;
    end;
end;  { TArraySequence.NextItemOf }
```

The method takes care not to try returning an "object" from beyond the end of the physical array (the currently allocated slots—remember that our declared array type extends for many bytes, almost always beyond the end of the allocated space). If *fNextItem* is an index value beyond the array's physical high bound, *NextItemOf* returns *nil*—a value safe but meaningless. If *fNextItem* is in bounds, *NextItemOf* returns the value stored there—which could also be *nil*, if that element is empty. This *nil* is a meaningful value in the sense that *NextItemOf* has returned the "value" stored at that place in the array. The ambiguity of a *nil* return from *NextItemOf* is one reason we need *AtEnd*.

How *EachItem* Works

Let's examine the code for *TList*'s *EachItem* method, which creates and initializes a sequence object. Actually, *TList* inherits *EachItem* from *THandleArray*.

```
function THandleArray.EachItem: TSequence;
  var
    aSequence: TArraySequence;
begin
  New(aSequence);
  aSequence.Reset(self, kArrayBase);  { Self is list or array object }
  EachItem := aSequence;
end;  { THandleArray.EachItem }
```

The code is straightforward. *EachItem* creates a new *TArraySequence* instance, initializes the object with *Reset*, and returns the object. *Reset* takes a reference to the list or array object whose *EachItem* method this is and a starting index value.

The new sequence object is ready to use, as shown in the sample consumer code on page 580.

What a Sequence Object Knows

A sequence object has to be tailored for the list class it supports. As a "friend" of the list, it needs certain kinds of knowledge in order to work with the list. First, the sequence object must work with the same kind of element reference (or handle) used by the list class. Second, the sequence object has to be told where to start, meaning that it has to have access to the actual list. In *TArraySequence* we give the sequence object the starting element by handing it a reference to the array or list object itself. Sequences are thus intimately related to the list classes they support.

Yet even though a sequence object has a detailed inside view of a list, it encapsulates that knowledge and hides it from the consumer. Unless he or she is prepared to violate encapsulation, the consumer can't really access the actual list except by means of either the list's methods or the sequence's methods. And because list elements are locked (normally) when *NextItemOf* returns a reference to them, the consumer can't accidentally free an element while using the sequence.

Final Thoughts About Sequences

When is a new sequence class needed? When a new list class is created. When we wrote *TList*, for example, we needed to write a corresponding sequence class that knows how to work with a *TList*. The full code for *THandleArray*, *TList*, and their accompanying sequence class is in the folder List, Part 3 on the disk.

Probably the best approach for sequences is to make writing a specialized sequence class part of the process of writing a new list class. But after you see traversal action objects, which we'll discuss next, you might prefer them to sequences.

Find Without *Find*, Part II

One final aside. Chapter 22 hinted at yet another way to search a list. That way is to use a sequence object. The process is much like searching with *Peek*—you pull the element out, test it, and then use the sequence object to move down the list.

Using Traversal Action Objects

Our final traversal solution—in addition to the procedural parameter, *DoCommand*, *Peek*, and sequence techniques—is to develop a "helper" class something like our search key objects. Given such a class, we'd write our own traversal action subclasses and pass instances of them to a *DoToEach* method for the list. This technique puts the traversal process back inside the list's encapsulation instead of pulling elements out to check them. The List Demo program on the code disk includes a few examples of traversal action subclasses.

DoToEach

First, let's see the sort of list method we'd call to traverse the list with a traversal action parameter, performing the same action on each element. This version of *DoToEach* can be inherited from *THandleArray,* so we implement it at that level:

```
procedure THandleArray.DoToEach(a: TTraversalAction);
  var
    current: TObject;
    i: Integer;
begin
  if self.fErrorHandler <> nil then
    begin
      self.fErrorHandler.SetError(kNoErr);
      if (a = nil) then
        self.fErrorHandler.SetError(kBadSearchKey)
      else
        for i := kArrayBase to self.LengthOf do
          begin
            current := self.fArray^^[i];  { Get the current item }
            { Call a's DoIt method on current }
            a.DoIt(current);
          end;  { for }
end;  { THandleArray.DoToEach }
```

This method does some error checking and then traverses the array. At each element, it sends the action object a DoIt message, passing the current element as a parameter. Whatever is specified inside the *DoIt* method gets done to the element.

Action Objects

An action object resembles the cloaking device approach to traversal we looked at earlier in this chapter. The action object has a *DoIt* method with one *TObject* parameter. *DoIt* conceals some action—or even a sequence of actions—to be performed on the element passed to it. Here's class *TTraversalAction*:

```
type
  TTraversalAction = object(TObject)

      procedure DoIt(elem: TObject);
    end;  { Class TTraversalAction }
```

As you can see, *TTraversalAction* is extremely simple, although its descendants can become more complex.

DoIt

For *TTraversalAction*, *DoIt* is a stub. To use the class, we have to subclass it and fill in what *DoIt* does. Let's look at an example.

A traversal action example

Suppose that a *TList* contains *TInteger* elements, each with an *fInteger* instance variable. Suppose further that we'd like to traverse the list, summing those fields. Here's a traversal action subclass that will do it:

```
type
  TSumTraversal = object(TTraversalAction)

      fSum: Integer;                { Initially set to zero--DoIt adds to it }

      procedure TSumTraversal.Init;
        { Initializes fSum to zero }
      procedure TSumTraversal.DoIt(element: TObject);
      override;
        { Does the action to element--adds its contents to fSum }
      function TSumTraversal.SumOf: Integer;
        { Returns fSum after a traversal ends }
    end;  { Class TSumTraversal }
```

The *fSum* field stores the cumulative sum as the elements are traversed. We have to initialize *fSum* to *0*, which is all that *Init* does. With these declarations and calls:

```
var
  sum: Integer;
  s: TSumTraversal;
  list: TList;
⋮
New(s);
s.Init;
list.DoToEach(s);          { Calls TSumTraversal.DoIt for each element }
sum := s.SumOf;            { Returns the sum of the elements }
```

we can retrieve the sum at the end of the traversal by sending the action object a SumOf message.

We've seen how *DoToEach* works, so let's look now at what *s.DoIt* does:

```
procedure TSumTraversal.DoIt(elem: TObject);
  override;
begin
  self.fSum := self.fSum + TInteger(element).Get;
end;  { TSumTraversal.DoIt }
```

This code converts the element to the underlying *TInteger* type, sends the element a Get message to retrieve the element's integer value, and adds that value to the previous value of *fSum*.

DoIt could do just about anything to the element. For instance, it could simply send *element.Display* to have the current element display itself. It could count elements, sum them as we have it do here, or perform any other sort of action on them.

DoIt parameters

Although the *DoIt* method has only one parameter, note that, in effect, it uses the *fSum* field (which is global to *DoIt*) as an "output parameter." By storing data in *fSum* as a side effect and providing a *SumOf* method to retrieve it, *DoIt*'s object operates somewhat as a function does.

If you need to pass parameters in to a *DoIt* method, just declare some instance variables in your *TTraversalAction* subclass and an *Init* method to set them. Then *DoIt* can use the instance variables as if they had been passed to it as actual parameters.

This makes *DoIt* quite flexible and useful in structures such as lists and trees. We'll look at more examples when we take up binary search trees in Chapter 25.

Summary

In this chapter, we discussed several solutions to a second problem associated with encapsulated lists: the problem of traversing a list, sending arbitrary messages to the objects in it. After all this list work, you should have a good understanding of polymorphism—of both its benefits and its occasional awkwardness.

Projects

- Implement *DoToEach* for *TLinkedList*. Provide several *TTraversalAction* subclasses for commonly useful list traversal methods, such as *Display*.
- Implement *EachItem* for *TLinkedList*.
- What should *TSumTraversal.DoIt* do if the list contains elements that aren't *TIntegers*? The list might contain *TIntegers* and *TReals*, and we might want to sum only the integers. How can we skip the reals? Hint: Remember the *Member* function. Alter *DoIt* to skip any non-integer elements. (We developed this technique for *TMail* in Chapter 22).
- Develop a collection of commonly useful *TTraversalAction* subclasses that can count items, sum values of fields, and more. A subclass might get an element reference, convert it to a generic Macintosh handle, and call *GetHandleSize*. The subclass could be used to generate a list of object sizes, sum the sizes of all objects in the list, lock the handles in the heap with *HLock* (not the same thing as the object locking mechanism we developed in Chapter 19), and do anything else you can do to either handles or objects.
- Look back at the fourth project in Chapter 16, in which we outlined a tool to help PicoApp programmers estimate the sizes an application would need for its memory reserves. Use the results of this chapter's previous projects' *TTraversalAction* subclass methods to help implement that size estimating scheme. As part of your effort, develop a list class based on *TList* with methods for getting handles to all CODE resources at a particular moment, putting them into a list, and summing them. You might need to refer to the Resource Manager chapter in *Inside Macintosh,* I-103.

24

COMPONENTS FROM COMPONENTS

Having completed a highly general, flexible, and reusable software component in *TList*, we'll focus in this chapter on extensibility. We'll investigate ways to create other useful components cheaply and easily by building from *TList* and similar components. We'll look first at a number of possible *TList* subclasses, having in mind a library that contains a wide range of reusable classes that are extensible, versatile, and handy. Rather than one list class, we'd like to have many different classes—some array-based, some pointer-based. Some would be small and fast, sacrificing power and versatility for space and speed. Others, like *TList*, would be the Swiss army knives of classes. Some would be highly general collection classes, and others would be optimized for listing specific kinds of data, such as integers, characters, or strings.

After our consideration of subclasses, we'll look at alternatives. Subclassing isn't the only way to create new classes from old. We'll see

- A hierarchy of list classes
- Approaches to building a stack from a list
- A general collection class

In Chapter 25, we'll explore a selection of other possible components.

Lists from Lists

TList is a strong basic list. It has facilities for all the standard things you might want to do with a list. Have we written the last word on lists, then? Not by a long shot. Let's explore just a few of the possibilities.

Several directions present themselves. One is to extend *TList* in subclasses with added features. Another is to customize *TList* in subclasses that change the focus or function of the list. Let's look at a few examples.

Extending *TList*

We can extend *TList* in subclasses that add new methods. Here's an example.

Shuffling the list

HyperCard's buttons and fields appear to be in different layers on the screen. If you move a button in front of another button, the front button obscures the rear button. Using HyperCard's Bring Closer and Send Farther menu commands, a user can alter the order of his or her buttons—shuffle them around on the screen.

We can use *TList* to implement a similar capability. Buttons stored earlier in the list would be "ahead of" buttons stored later in the list.

TList doesn't have specific methods for moving list elements forward or backward in the list. It's easy enough to do, of course, using a combination of *TList*'s present methods. We could implement a Bring Closer command, for instance, with *Move*, *Extract*, and *Insert*:

```
procedure TList2.BringCloser(elem: Integer);
  var
    obj: TLockable;
begin
  self.Move(elem);
  obj := self.Extract;          { Mark ends up at elem - 1 }
  self.Insert(obj);             { Element is inserted before the mark }
end;  { TList2.BringCloser }
```

A *TList* subclass might add these new methods:

- *BringCloser*—move element one slot toward the head of the list
- *SendFarther*—move element one slot toward the tail of the list
- *BringFront*—move element to the head of the list
- *SendRear*—move element to the tail of the list

Of course, the subclass would also inherit all of *TList*'s methods.

Customizing *TList*

A good class library might also contain a number of specialized list classes and classes derived in various ways from *TList*. A specialized class doesn't simply do more, the way an extended class does; a specialized class does the same thing differently.

Many variations on the list structure are possible: sets, tables, stacks, queues, deques, rings, bags. Let's look at a couple.

Set classes

If we subclass *TList* and override its addition methods to suppress duplicate elements, the result is a set class. Elements appear in the set once, without regard for order. We can ask a set whether item *X* is a "member" of the set, and we can combine sets according to various principles: intersection, union, difference. We'd want to add methods implementing set operations such as *MemberOf*, *Intersection*, *Union*, and so on.

To suppress duplicate elements in the set when we call the *Add* method, we call *Find* before calling *inherited Add*. Here's a sample *Add* method for a set of integer objects:

```
procedure TSet.Add(elem: TObject);
  override;
  var
    s: TMySearchKey;
begin
  New(s);
  s.Init(TInteger(elem).Get);
  if not self.Find(s) then
    inherited Add(elem);
end;  { TSet.Add }
```

This code assumes a *TSearchKey* subclass whose *Init* method sets up a search key object to use with *Find*. We initialize the search key object by typecasting *elem* to a *TInteger* and calling its *Get* method:

```
TInteger(elem).Get
```

and then passing the results to *Init*:

```
s.Init(TInteger(elem).Get);
```

Then we call *Find*. If any object already in the list has the same value as the search key object, we skip the last line of code. Otherwise, we call *inherited Add*—the version of *Add* belonging to *TSet*'s immediate ancestor, which is either *TList* or a subclass of *TList*.

This approach takes advantage of the code already developed for *Add* and uses other *TList* methods to get the job done. (But read on. Later in this chapter we'll consider some drawbacks to subclassing *TList* this way.)

Hash tables

Arrays are usually used to implement hash tables for extremely fast lookups. *TList*, which rests on a dynamic array, can be the basis for a hash table object.

At initialization, we'd specify an appropriate size for the table. We'd also supply a hashing function appropriate for our data. The function might take the form of a procedural parameter or of something like a *TSearchKey* object. You can find a detailed account of hashing in a good computer science text.

Subclassing vs. Importing

Subclassing is a powerful way to derive one class of objects from another. You can take advantage of inheritance, override methods that need to work differently, add new instance variables and methods, and even use the *inherited* mechanism to call methods that were overridden. But sometimes subclassing is not the best way to develop a new class from an old one, as we'll see.

The problem is that, although you might be able to derive one abstraction from another by subclassing (say, a stack from a list) as Figure 24-1 suggests, the resulting class might be less than satisfactory. You're stuck with method names from the old abstraction, and you can't change the parameter lists of inherited methods. And you might be stuck with useless or even dangerous methods from the ancestor.

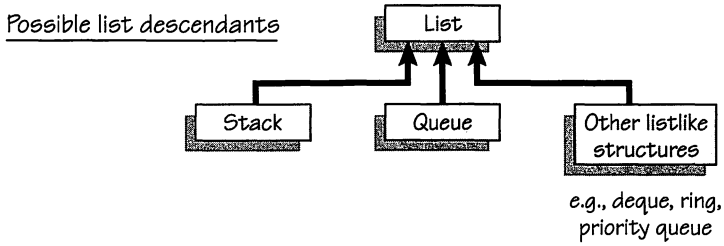


Figure 24-1.
Subclassing other abstractions from a list class. Almost any list class might serve as the base class of this hierarchy.

The solution in such a situation is to build the new class not as a subclass but as a higher-level layer on top of the old class. To do this, you "import" the old class's unit and call upon the methods of the old class to implement the methods of the new. In the rest of this chapter, we'll explore such layered development by basing *TStack* and other data-structure–based classes on our existing *TList* classes.

We'll also take a look at how to handle non-object data types, such as Pascal integers, chars, reals, strings, and so forth, in our object-oriented data structures.

And we'll look at a general "collection class." Based on *TList* (or possibly on other kinds of classes), the collection class will put a more general, less listlike interface on top of *TList*'s functionality.

Lists and Stacks

Some of the common data structures of computer science have a strong family resemblance to one another. Lists and stacks look and act a lot alike (as, for that matter, do queues, deques, rings, and other structures, although we won't look at those in this chapter).

Abstractly, a list is a sequence of items. You can add to it or remove from it anywhere. A stack is a list that allows adding and removing at only one end, called the "top." You can't add and remove at the other end or anywhere in between. A queue is a list that allows adding at one end, the rear, and removing at the other end, the front. You can't add at the front, you can't remove at the rear, and you can't do either in the middle.

Note the phrase "is a" in the descriptions. A stack *is-a* list. A queue *is-a* list. That phrase is the heart of the idea of inheritance—a subclass *is-a* whatever its immediate ancestor is (and more, or different, as well).

Subclassing a list class would appear to be the best way to create a stack class. For the stack, you'd override the addition and removal methods of the list to make them work only at the end of the stack defined to be the top. And for the queue, you'd override those same methods to make the addition methods work at the rear and the removal methods at the front. In some OOP languages, that's exactly what you'd do. But in Object Pascal, as we'll see, Pascal's constraints can make subclassing a poor way to go.

Subclassing a Stack from a List

Let's see what happens when we do subclass a list class to derive a stack class.

Recall that items are added and removed only at the top of a stack. A stack can be empty, and, abstractly at least, it can grow without limit. (Practically, the amount of RAM will limit the stack's size, of course.) The operations on a stack are traditionally called *Push* ("add an item at the top"), *Pop* ("remove an item from the top and throw it away"), and *Top* or *TopOf* ("return the value of the item at the top without removing it"). In implementations, *Pop* often becomes a function that combines the features of *Pop* and *TopOf* from the abstract concept of a stack. Other operations are *IsEmpty* and *IsFull*. We might also want some handy methods such as *Copy*, or *Clone* ("make a duplicate of the stack"), *Clear* ("return the same stack in an empty condition for reuse"), *DepthOf* ("return the number of items in the stack"), *IsEqual* ("indicate whether this stack is exactly equal to that stack, in size and contents"), and probably a method for traversing the stack (see Chapter 23) so that we can examine its full contents or send all its elements some common message without disturbing the stack itself. This might call for a *Peek* method, or for an *EachItem* method that creates a *Sequence* object, or for a *DoToEach* method. One last aspect of the stack abstraction is the exceptions or errors that can occur:

overflow—running out of memory for stack growth
underflow—trying to pop or retrieve the top of an empty stack

Here's a sketch of what we'd like a stack class to look like:

```
type
  TStack = object(SomeListType)

      fContents: SomeType;

      procedure IStack;          { Set a new stack to empty }
      procedure Push(item: AType); { Add an item to top }
      function Pop: AType;       { Remove an item from top }
      function TopOf: AType;     { Examine data at top without disturbing stack }
      function IsEmpty: Boolean; { True if stack is empty }
      function IsFull: Boolean;  { True if stack is full }
      function DepthOf: Integer; { Number of items currently in stack }
      procedure Clone: TObject;  { Make a duplicate copy of the stack }
      override;
      procedure Clear;           { Delete stack's data but not the stack itself }
      function IsEqual(toStack: TStack): Boolean; { True if self has same size }
                                                  { and contents as toStack }
      procedure EachItem;        { Make a sequence object for traversing stack }
      function StackError: SomeCodeType; { Report errors from other methods }
      function TypeOf: Str30; { Return the string 'TSTACK' }
    end; { Class TStack }
```

For convenience, we'll let both *Pop* and *TopOf* return a reference to the top item. The difference between the two is that *Pop* removes the item and *TopOf* doesn't.

Which List to Subclass?

Now we'd usually run into a small embarrassment of riches. If we had a real class library, we'd probably have many lists from which to choose the stack's immediate ancestor. Let's use *TList*.

Remember, of course, that we inherit a number of *TList* characteristics: all the instance variables and any methods that we don't override. And some of the methods we want in a stack class aren't available from *TList*: in particular, *Push*, *Pop*, *TopOf*, and *DepthOf*. Some of these methods have analogs in *TList*: *Push* is like *AddHead*, *Pop* is like *ExtractHead*, *TopOf* resembles *Peek* (if *Peek* were preceded by *Head*), and *DepthOf* functions as *LengthOf* does.

Problems

Unfortunately, the resulting stack class is not very pretty, and it could even be dangerous.

Let's look at the difficulties:

- Many of the names are "wrong," and we can't change them. For example, *fTheLength* would be better if it were *fTheDepth*. Much more important, *AddHead* and *ExtractHead* might do what we'd want *Push* and *Pop* to do, but a

client is going to feel uncomfortable "adding" to and "extracting" from a stack rather than "pushing" and "popping." The list method names simply aren't appropriate for the traditional stack abstraction familiar to programmers everywhere. It's only tradition, and we can call the operations anything we want, but at a cost. And Pascal, of course, won't let us rename an overridden method or change its parameter list in any way. (Some languages do let us overload names and change parameter profiles.)

- Having unusable methods around, such as *Append* and *Move*, is potentially unsafe. As inherited, *Append* adds items where we wouldn't want them to be added in a stack. Yes, we could disarm *Append* by overriding it and either leaving the overridden version as a stub method or generating an error if it's ever used. Thus disarmed, *Append* would be inherited as your appendix is from your ancestors: Sometime in the distant past, it must have had its uses, but now it just leads to appendicitis.

  The problem with *Move* is just as serious. If a client can call *Move* to move the mark in a stack, he or she can violate the integrity of the stack by, for instance, appending in the middle.

These dangers are reason enough for us to consider alternatives to subclassing as a way of deriving a stack class from a list class.

Building in Layers and Importing

One great achievement of software engineering is the notion of building complex software as a series of layers.

Layered Software Design

The general idea is to relate to the machine level with a layer of primitive instructions and then relate to that layer with a higher layer that calls the primitives to get things done, and so on, as Figure 24-2 on the next page suggests.

The Mac's QuickDraw graphics library, for example, was written in tightly coded, highly efficient 68000 assembly language. That code is inextricably tied to the Mac (or, at least, to 68000 machines)—it's machine dependent. Programs that call QuickDraw routines, though, are dependent on QuickDraw rather than on the underlying hardware. These programs can be written in high-level languages like Pascal instead of assembly language. Together, QuickDraw and the programs built on top of it constitute two layers. If we wanted to, we could write a version of QuickDraw—with the same routine names, the same parameters, and the same behavior—for some other machine. We could write this version of QuickDraw in, say, assembly language for the IBM PS/2 machines. (Let's commit a little heresy for the sake of an example.) Then the same program that used QuickDraw calls on the Mac could (assuming that it wasn't dependent on other Mac-specific features) be compiled on a PS/2.

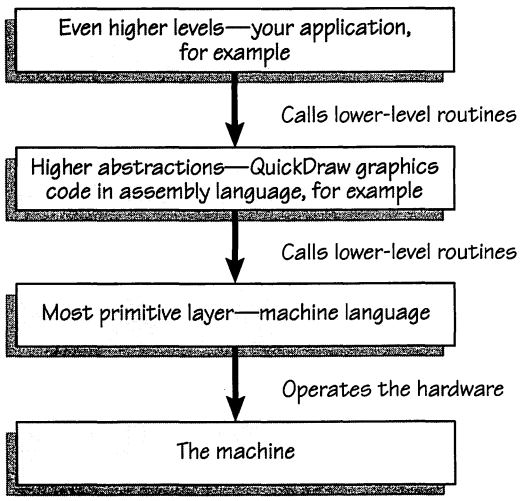


Figure 24-2.
Layered software—a program built in layers from machine language routines through higher-level routines.

Although the QuickDraw layer is dependent on the Macintosh hardware, it is independent of the higher-level program. As long as QuickDraw's interface doesn't change, programs that use it will be fine, even if the underlying implementation of QuickDraw changes drastically.

Layered software design can overlay great complexity with simplicity, make development easier and faster, and improve the portability of software. MacApp is a good example of this principle. Writing Macintosh programs using the Toolbox is complex and hard to learn, so it takes a lot of time to develop a program that incorporates the full Macintosh user interface. By providing a layer of software that handles all the usually most difficult parts of programming the Mac, MacApp simplifies and speeds the process of developing fully functional Mac programs. As Figure 24-3 suggests, all you have to do is add another layer on top of MacApp to make your program work as a spreadsheet, or a word processor, or a drawing program, or whatever. (Of course, using MacApp puts you on another learning curve, which is part of what this book is about.) The same principles are true for the THINK Class Library and our own PicoApp.

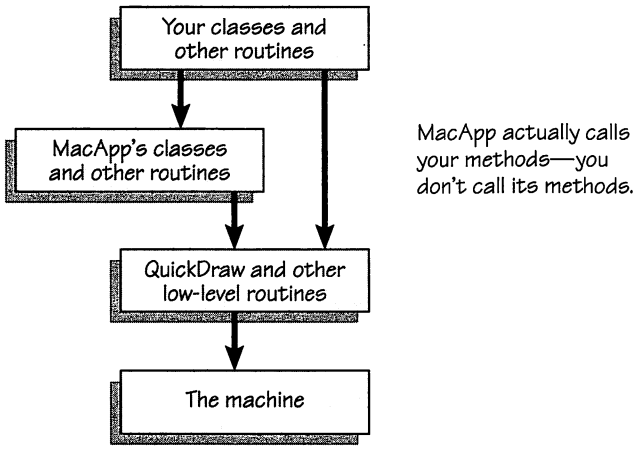


Figure 24-3.
Layering with MacApp or any other application framework. Just as the ROM provides lots of built-in features, so does MacApp, but your code can (and must) call QuickDraw and other low-level routines directly.

Imports

In Pascal, we "import" a type, a constant, a procedure or function, or a variable from another unit by declaring that unit's name in a *uses* clause. Then any declaration in the interface part of the imported unit is available to the importing unit. We can freely use all the imported constants, types, and so on in the importer's code.

This isn't some new object-oriented concept—importing by means of the *uses* clause is perfectly normal for any version of Pascal that supports units. But, of course, among the program elements we can import are object classes and their methods.

Thus, we could define *TStack* as a new class in its own unit, name *TList*'s unit in the *uses* clause, and use objects of the imported list type to implement *TStack*'s methods—which is exactly what we'll do.

Layering a stack on top of a list

For the sake of demonstration, let's import *TList*'s unit and build *TStack* on top of it. Of course, this version of *TStack* is going to be fully formed and not abstract at all. That's partly a consequence of building on top of a complete list type.

The interface part of *TStack*'s unit, *UStack*, looks like this:

```
unit UStack;

interface

  uses
    ObjIntf, UErrorHandler, UHandleArray, UList;
      { This is the import, which consists of several units }

  type
    TStack = object(TLockable)  { Note its heritage }

        fContents: TList;          { Here's the stack's list }

        procedure IStack(wantSlots: Integer; eh: TErrorHandler);
        procedure SetLocking(doLock: Boolean);
        procedure Push(item: TObject);
        function Pop: TObject;
        function TopOf: TObject;
        function IsEmpty: Boolean;
        function IsFull: Boolean;
        function DepthOf: Integer;
        function Clone: TObject;
        override;
        procedure Clear;
        function IsEqual(toStack: TStack): Boolean;
        function EachItem: TListSequence;
        function StackError: Integer;
        procedure Free;
        override;
      end;  { Class TStack }
```

Building *TStack*'s methods out of *TList*'s

The stack itself, the data structure part, *is-a TList* object, just as we'd like. A *TStack* object, in turn, is a "manager object" for a *TList* structure, calling *TList*'s methods to accomplish stack operations such as *Push* and *Pop*. Figure 24-4 illustrates the stack-built-upon-list structure.

Layering *TStack* on top of *TList* doesn't pose the difficulties that subclassing *TList* did. Layering permits us to use the correct method names—*Push*, *Pop*, and so on, and the consumer can use dangerous methods such as *Append* and *Move* only by breaking encapsulation and accessing them through the *TList* instance variable *fContents*.

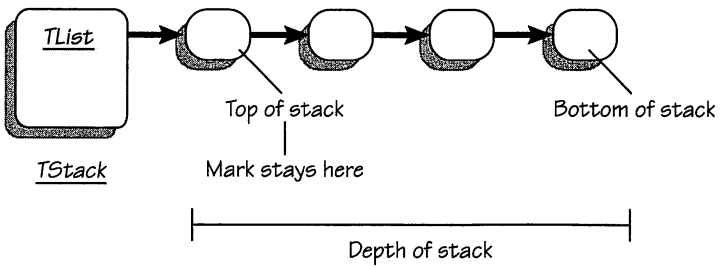


Figure 24-4.
Structure of a list-based stack. Parts of TList *lend themselves to the creation of* TStack.

Let's look at several *TStack* methods.

TStack.IStack To initialize a stack, we create and initialize its underlying empty *TList*.

```
procedure TStack.IStack(wantSlots: Integer; eh: TErrorHandler);
begin
  self.fContents := NewList(wantSlots, eh);
end;  { TStack.IStack }
```

TStack.Push To push an item onto the top of the stack, we position the mark at the first item in the underlying list. Then we call *list.Insert* to add the item to the list right before the marked item. *TList* combines those operations in *AddHead* (which we planned with an eye toward its future usefulness in *TStack*).

```
procedure TStack.Push(item: TObject);
begin
  self.fContents.AddHead(item);  { fContents is a TList }
  { Might call TStack.StackError here }
end;  { TStack.Push }
```

TStack.Pop To pop an item from the stack, we first position the mark at the top of the stack (the front of the underlying list). Then we extract the head item. *TList* combines these steps in *ExtractHead*.

```
function TStack.Pop: TObject;
begin
  Pop := self.fContents.ExtractHead;
  { Again, we might want some error-checking code }
end;  { TStack.Pop }
```

TStack.TopOf To retrieve the value of the top item on the stack (without altering the stack), *TopOf* positions the mark at the first item in the underlying list and then calls *list.Peek* to get the item's value. *TopOf* returns a reference to the top item, which will be a *TObject*, so we have to typecast the returned value. (This is true for *Pop*, too).

```
procedure TStack.TopOf: TObject;
begin
  self.fContents.Head;
  TopOf := self.fContents.Peek;
end;  { TStack.TopOf }
```

TStack.StackError To check for errors after each call to a stack method, we'll provide a *StackError* method, which checks with the stack's error-handler object and returns the latest error code.

```
function TStack.StackError: Integer;
begin
  if self.fErrorHandler <> nil then
    StackError := self.fErrorHandler.Error
  else
    StackError := kNoErrHandler;
end;  { TStack.StackError }
```

***TStack.Free* (overrides *TObject.Free*)** We have to override *Free* to dispose of the underlying list before freeing the stack object. We might also want two versions of *Free*, as in *THandleArray.Free* and *THandleArray.FreeAll*.

```
procedure TStack.Free;
  override;
begin
  self.fContents.Free;
  inherited Free;
end;  { TStack.Free }
```

Let's leave the remaining methods, and the error checking, to you.

Efficiency Considerations in the Layered Approach

You might have been wondering how efficient the layering approach to software design is. Let's take a look.

Clearly, the approach takes some overhead. Each time a consumer calls a stack method, that method in turn calls the underlying list method. Method calls exact a greater time penalty than ordinary procedure calls because method calls might involve looking up the correct method to call in the object class's method lookup table (and possibly in that class's superclass's table). The main alternative to chains of method calls is to design a complete stack class from scratch (as you did in the projects earlier in Part 3), inevitably duplicating much of the code and effort you've put into building the list class. Sometimes, of course, when speed is critical, that's what you need to do. But often the ease of developing the new class from the old outweighs the overhead. It takes only minutes to write a fully functional, highly general, and reusable stack component based on *TList*—and the component is already tested. Be aware, though, that method-call overhead goes with the territory in layered software design.

Second, we should ask whether *TList* is the best choice of ancestor for stacks. The "mark" mechanism does impose some overhead, incurred when we have to move the mark for stack operations. Subclassing a simpler traditional linked list would allow us to eliminate any move method overhead. But the solution is already in some *TList* methods. Anticipating layering, we bothered to include a method in *TList* called *AddHead*, for instance. Our method for *TList* can set the mark to the first item in the list without having to traverse the whole list to find the head and then add there. Because we anticipated such needs for the list class and related classes, we gave *TList* facilities from the start to maintain a reference to the head and update it in all the appropriate methods. That foresight and extra effort now pays off in a better stack.

A Few Improvements

Let's briefly reconsider specialized list-derived classes such as sets and hash tables. Would they be better implemented by importing *TList* rather than subclassing it?

If we subclassed *TList*, the set class, for example, would be stuck with some list methods it didn't need. Sets are unordered, so methods such as *Insert*, *Append*, *AddHead*, and *ExtractHead* don't fit the set abstraction very well. Better to have general *Add* and *Remove* methods through which all adding and removing is done without regard to location. And better not to have the troublesome list methods.

Similar reasoning applies to the other classes. Each represents a concept different in important ways from the list abstraction. The list is a good implementation vehicle for them, but it would be cleaner to import *TList* and layer a new interface over its functionality than to subclass it.

In Retrospect

And, let's reconsider, by the same token, our design of classes such as *TList*. Recall that *TList* is a subclass of *THandleArray*. As a consequence, *TList* inherits many array-based features—and has to override some methods to make them more list-like. We need to be aware of some dangers in using certain array methods with a list.

Although arrays are often used to implement lists, that doesn't mean that an array and a list represent the same abstract concept. They don't, at all. The fit is close enough to make implementing *TList* from *THandleArray* easy (by subclassing or importing). But the bones of *TList*'s array ancestor show through, sometimes too clearly.

Looking back, we can see that we probably should have built *TList* atop an imported *THandleArray*. The result would have been a cleaner list interface and less danger from inherited methods.

Handling Non-Object Data Types

We now have list and stack classes that can hold an object of any class descended from *TObject*, that is, almost any object at all—objects only. But some of the most common kinds of data to add, push, or enqueue aren't objects at all. What can we do with Pascal's standard data types—chars, integers, reals, strings, arrays, records, sets, enumerations, even files? Can we push a real or a string or a set onto a stack if we need to?

Of course, we could write standard data structures for these data types and encapsulate them inside an object. A stack of integers would thus use a stack object with *Push*, *Pop*, and other methods to manage an ordinary linked list–based stack (with record-type nodes and handles for links). But there's also a way to use the structure components we've just developed.

An Object Approach

We've already seen one approach. If we need to stack integers, say, we just convert each integer into an integer object first. We looked at an integer object type earlier. It looked like this:

```
type
  TInteger = object(TLockable)

      fInteger: Integer;              { Contains an integer }

      procedure Put (i: Integer);     { Adds i, which equals fInteger }
      function Get: Integer;          { Returns fInteger }
      procedure Display;              { Shows value }
      override;
      ⋮
    end;  { Class TInteger }
```

The idea is to wrap an ordinary integer up in an object, which manages its integer contents. The object can, at the least, store an integer value and report the value. (The object might also contain methods for adding one *TInteger* to another, subtracting one from another, and other common operations on integers.) We could do the same thing for other non-object data types by defining *TChar*, *TString*, *TReal*, and so on.

The main drawback to using such non-object objects with our stacks is that, in order to add an ordinary integer to a stack, we have to perform three steps. We have to create the integer object, give it a value, and then push it. That's a headache. Is there a better way?

Specialized List and Stack Classes

A cleaner approach, one that puts less burden on the consumer using the list or stack, is to design specialized list or stack classes for those types.

We might appear to be losing the ground we've gained by being able to write a polymorphic stack. The Pascal problem we were trying to overcome was the burdensome requirement, in standard Pascal, that we write a separate stack for every different data type.

A couple of facts work in favor of specialized classes, even so. First, we need to write such specialized classes for only a handful of basic data types, at least initially. We'll want list and stack classes for integers, chars, reals, and strings. We can't really write them for arrays in general, records in general, sets in general, and so on. Only when there's a specific array or record type can we write a list or stack class for it.

Second, even then, when we need a stack for records of type *employee*, say, writing the stack in relation to our existing stack class shouldn't be too hard. A lot of the work will have already been done for us in the original classes.

Finally, of course, many data types that used to be handled with records will now be handled with objects, anyway—employees, for example—objects that can be used directly in the data structures we've already developed.

An integer stack class

As an example, let's design a *TIntegerStack* class based on *TStack*.

What do we mean by "based on"? Subclassed from, or imported from? If we write *TIntegerStack* as a subclass of *TStack*, we'll run into the same kind of lumpy, unpleasing results we managed to avoid by developing *TStack* from *TList* by importing the list class. Here's why. When a consumer is using *TIntegerStack*, we'd like him or her to be able to say

```
aStack.Push(3);
```

But *TStack*'s *Push* method, which *TIntegerStack* would inherit, takes a *TObject* parameter, not a raw integer. To get the effect we want, we'd need two *Push* methods, one that takes integers and another (inherited) that takes objects. In Pascal, of course, we can't have two methods in the same class (or in the same hierarchy) with the same name— even if their parameter lists are different. The new integer-type *Push* would have to be called something like *PushInt*. Of course, we'd also be stuck with the inherited *Push*, which we wouldn't need or want for pushing integers on an integer stack.

The better approach is to import *TStack* into *TIntegerStack*'s unit, just as we imported *TList* and used it as a substrate on which to develop *TStack*.

Here's what the resulting interface part of *TIntegerStack*'s unit, *UIntegerStack*, might look like:

```
unit UIntegerStack;

interface

  uses
    ObjIntf, UErrorHandler, UHandleArray, UList, UStack, UInteger;
      { The import }

  type
    TIntegerStack = object(TLockable)  { Note the ancestor }

        fContents: TStack;             { The actual stack }

        procedure IIntegerStack(wantSlots: Integer; eh: TErrorHandler);
        procedure Push(theInt : Integer);
        function Pop: Integer;
        function TopOf: Integer;
        :
      end;  { Class TIntegerStack }
```

TIntegerStack methods

Let's see how some of those methods would be implemented using *TStack* methods as their underpinning.

TIntegerStack.IIntegerStack We create a new *TStack* object (to hold *TObjects*) and initialize it to empty. It's stored in the *fContents* field of a *TIntegerStack*.

TIntegerStack.Push To push a raw integer onto the object-based stack, *fContents*, we have to create a *TInteger* object, store the raw integer in it with *TInteger.Put*, and then call *TStack.Push* to push the integer object. *TIntegerStack.Push* encapsulates all of that for us.

```
procedure TIntegerStack.Push(theInt: Integer);
  var
    anInt: TInteger;              { Intermediary object }
begin
  New(anInt);
  anInt.Put(theInt);              { Store the integer }
  self.fContents.Push(anInt);     { The real push onto the stack }
end;  { TIntegerStack.Push }
```

The remaining methods of *TIntegerStack* would be implemented similarly, using *TStack*'s methods to do the stack work and converting data into objects where necessary.

Other specialized classes—*TCharStack*, *TIntegerList*, *TStringList*, and so on—would be built as new layers over the existing general stack and list classes. These we'll leave as a rather easy project. You can probably write all the specialized classes in an hour or two.

A General Collection Class

Another list-based data structure, one that polymorphism makes possible, is the collection. Recall that a collection is an object that can hold a variety of other objects, of different types, polymorphically. For a real-world example, consider our miscellaneous parts bin from Chapter 6. The bin could hold several different types of auto parts. A collection is a sort of burlap bag into which we can toss various things as they come up.

We might have been tempted simply to name *TList* "*TCollection*" instead, a name that's certainly general enough. But *TList* presents a more listlike interface than we might like for a general collection class. A collection can be implemented with a list, but it needn't always be, and the operations we want to perform on a collection differ somewhat from those for a list.

The two main ways to implement collection classes are by means of arrays and by means of linked lists (although we could use other structures, such as trees). Our strategy will be to import *TList* and use it as the basis for a highly general collection class, just as we used *TList* as the basis of our stacks.

At the most general level, a collection doesn't maintain any particular order of elements or exclude duplicate elements. A general collection abstraction can be extended to include "ordered collections" (collections arranged in some special order) and "set" classes (collections that exclude duplicates).

Collections provide methods for adding and removing items, for finding an item based on a key or keys, for finding out how many items are in a collection, and for visiting the items in a collection one by one to perform some task, such as displaying the items.

The collection classes we've seen so far had limited capacities because they were based on simple variables or arrays. Now we'll work out a general collection class, *TCollection*, based on *TList*. The *TCollection* objects will be able to expand as more and more objects are added, and the *TList* methods will support all the necessary functions of a collection.

We'll go a step further. We'll design *TCollection* so that we can implement subclasses of it based, not simply on *TList*, but also on *TLinkedList*, a binary search tree, a hash table, or any other suitable structure. *TCollection* will be the abstract ancestor of a wide variety of collection classes. Yet *TCollection* will not be so abstract that we can't use it directly. We'll give it a default implementation based on *TList*. Then our other *TCollection* subclasses can simply replace the list implementation details with a tree or a hash table implementation or something else. You might want to look at classes *CCollection*, *CCluster*, and *CList* in the THINK Class Library for a different approach from the one we'll take here.

Problems and Approaches

If we were simply to subclass *TList* to create *TCollection*, the result would be a collection class with some inappropriate methods. In particular, the *Move*, *MoveToNode*, *Head*, *Tail*, *PositionOf*, and *Append* methods are probably too list oriented, as is *IList*. Once again, the bones of the underlying implementation show through too clearly. Collection classes need to be at a higher level of abstraction, less concerned with the mechanics of the underlying data structure itself. In fact, as we've conjectured, *TCollection* shouldn't even depend upon what kind of data structure is used to implement it.

By hiding the intimate details, we make it possible to implement the *TCollection* class in different ways for different purposes. To handle this capability, we'll use a technique that lets us plug various kinds of underlying data structures into a slot in the collection class.

A "Slot" Approach to Specifying Details

If the details of an abstract class are hard to specify, let's just put in a slot and let the abstract class's subclasses fill the slot.

For an example, consider this declaration for *TCollection*:

```
type
  TCollection = object(TLockable)

      fContents: TObject;              { Here's the slot }

      procedure ICollection;
      procedure Add(anItem: TObject);
      function Find(data: TSearchKey): TObject;
      function Remove(anItem: TSearchKey): TObject;
      procedure Free;
      override;
      function SizeOf: Integer;
      procedure DoToEach(action: TTraversalAction);
      :
    end;  { Class TCollection }
```

The *fContents* field should hold some underlying data structure, such as an array, a linked list, a tree, or a set. We don't want to pin it down to one of those structures at this level of abstraction, so we call it a *TObject*. The *ICollection* method's purpose is to create an instance of whatever the underlying structure is—the array, linked list, tree, or set—and install it in the *fContents* slot. That's exactly what *TIntegerStack.IIntegerStack* did except that in that method we installed a stack object in a *TStack* slot, and in this one we install a *TList* or an array object or whatever polymorphically in a *TObject* slot. Of course, the underlying structure other than *TList* is specified only in a subclass of *TCollection*. Suppose we write a subclass of *TCollection* called *TCollection1*. Here's its declaration:

```
type
  TCollection1 = object(TCollection)

      procedure ICollection;
      override;
      procedure Add(anItem: TObject);
      override;
      function Find(data: TSearchKey): TObject;
      override;
      function Remove(anItem: TSearchKey): TObject;
      override;
    end;  { Class TCollection1 }
```

TCollection1 inherits the *fContents* field and the *Free*, *SizeOf*, *DoToEach*, and perhaps other methods, but it must override *ICollection*, *Add*, *Find*, and *Remove* because those methods can't be fully specified until we know what data structure underlies the collection. In this case, suppose the collection were based on a binary search tree. Here are the overridden methods (without error checking):

TCollection1.ICollection We create a new binary search tree object and install it in the *fContents* slot.

```
procedure TCollection1.ICollection;
  override;
begin
  self.fContents := NewBinarySearchTree;
end;  { TCollection1.ICollection }
```

If the collection were, instead, based on a hash table or on a different kind of list, we'd instantiate an instance of that class and install that in *fContents*.

TCollection1.Add We access the underlying tree (by typecasting) and use its *Insert* method. We use a binary search tree to make an ordered collection, relying on the tree's natural ordering properties to make it easy.

```
procedure TCollection1.Add(anItem: TObject);
  override;
begin
  TBinarySearchTree(self.fContents).Insert(anItem);  { Typecast }
end;  { TCollection1.Add }
```

Other overrides *TCollection.Find* and *TCollection.DoToEach* are special situations. *Find* takes a search key object as its parameter and then loops through the underlying *TList*, comparing the key to the data in each list node. In a subclass, *Find* might traverse an underlying tree, comparing the key to the data in each tree node.

The approach we're using builds a whole new abstraction (the collection) on top of an underlying abstraction (the list, tree, hash table, or whatever). We might build other new classes on top of *TCollection*, by either subclassing or importing it. The combination of subclassing and importing methodologies makes classes almost infinitely extendable.

Using *TCollection*

We can use *TCollection* in several ways. We can subclass it to create different specialized kinds of collections, such as ordered collections or sets. We can import it into other units and use its types and methods to implement other, even more abstract, classes. And, we can use it as is because by default it's implemented with *TList*.

Among the interesting kinds of collections we might create by subclassing or importing *TCollection* are *TFileFolder*, *TMailBox*, *TEnvelope*, *TGroceryList*, and *TTable*.

Summary

In this chapter we focused on the extensibility of software components. We looked at two ways to build new components from old: subclassing, of course, but also importing and then layering.

Along the way, we sketched out several useful software components: a variety of lists; general stacks; specific stacks for Pascal's built-in data types, such as *Integer*, *Char*, and *String*; and a general collection class layered on top of an imported list.

This, of course, is only the beginning of what's possible. We'll look at a few more components in the next chapter.

Projects

- We've only sketched out some of the possibilities for extending and customizing *TList*. Write some of those classes for yourself, using either subclassing or importing.
- Go ahead and build the general stack class *TStack*.
- Build a general queue class *TQueue*.
- Add to your library by building a priority queue (a queue that gives priority service to certain elements), a deque (a queue that allows adding and removing at both ends), and a ring (a list that is circular—having a mark but no beginning or ending element).
- Flesh out *TCollection*, basing it on *TList*.

25

MORE COMPONENTS

In this final chapter we'll look at a smorgasbord of other useful components. The selection is, of course, arbitrary. We could have included lots of other kinds of components, but these are useful and demonstrate several more OOP concepts we should touch on. We'll look, to varying degrees, at

- Trees and "treeability"
- A general node class
- A general binary search tree software component
- Array objects for open array parameters
- Streams in brief
- Filters

Trees and "Treeability"

So far we've looked at software components based on lists. Another common kind of structure in computer science is the tree, and the most commonly used tree is the binary tree.

Whereas a list strings nodes together linearly, a tree strings them together hierarchically. A doubly linked list node has a previous node and a next node (unless it's at one end of the list). A tree node has a "parent" and zero or more "children," each a node. The first node in the tree, which has no parent, is called the "root" node. In a tree, a node can be at the root, at the terminus of a branch (a *leaf*), or somewhere in between (an *interior node*). Usually, a tree node doesn't contain a pointer to its parent node—although we can build such a pointer in. A tree node does contain pointers to its children. Figure 25-1 on the next page illustrates the three kinds of tree node positions and the parent-child relationships among tree nodes.

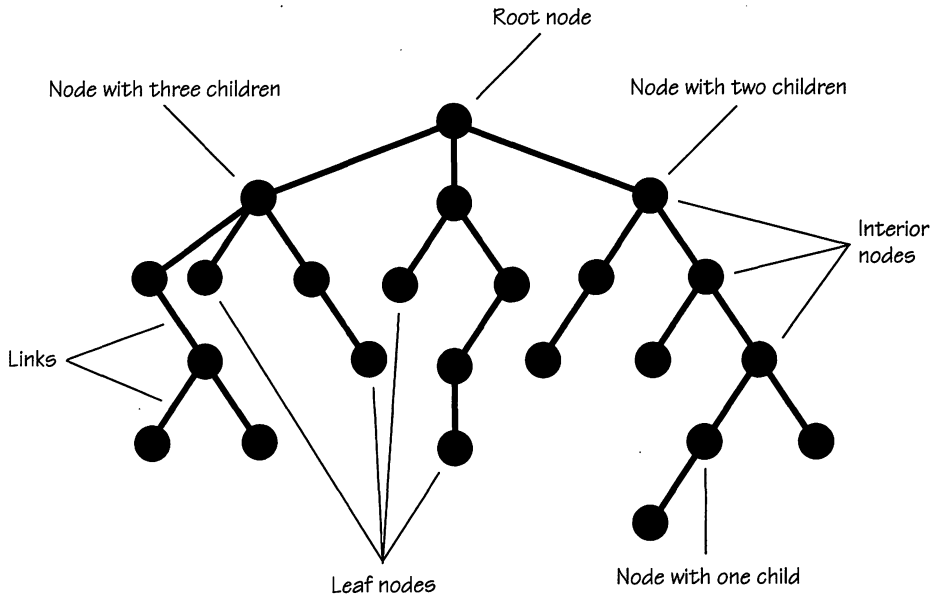


Figure 25-1.
Structure of a tree.

Trees can be *binary* (a maximum of two children per node), trinary, and so on. The nodes of a totally general tree such as the one shown in Figure 25-1 can have any number of children, even varying numbers of children, per node. Some kinds of trees are adjusted after each addition or removal to "balance" the tree. The adjustments keep the tree more efficient and easier to work with. One advantage of tree structures is that we can use them in a way that allows us to access a given node much more quickly than if the node were in a list. For more on the care and feeding of trees, see your local data structures text.

"Treeability"

By analogy with lists, we'll consider a property that might be called "treeability." An object is "treeable"—that is, it can be added to a tree—if it descends from a class called *TTreeable*. *TTreeable* greatly resembles *TDoublyLinkable*:

```
type
  TTreeable = object(TLockable)

      fRightChild, fLeftChild: TTreeable;

      procedure ITreeable;
      procedure SetRight(o: TTreeable);
      function RightOf: TTreeable;
```

```
    procedure SetLeft(o: TTreeable);
    function LeftOf: TTreeable;
  end;  { Class TTreeable }
```

But, having considered the *TTreeable* class, we're not going to implement it.

TNode, a General Node Class

A hypothetical example will illustrate why *TTreeable* is a bad approach. Suppose that we have some object *O* that we'd like to use in a list at one point and in a tree later. What should *O*'s class descend from? It appears to need both *TDoublyLinkable* and *TTreeable* properties. Of course, both *TDoublyLinkable* and *TTreeable* descend from *TLockable*, a contender to be their common ancestor class, but *TLockable* lacks any kind of linkability or treeability whatsoever.

We need a more general strategy. Recall the problems we had in Chapter 19 with using the pointer-based *TLinkedList* in PicoApp (or in MacApp or TCL). The most common kinds of data in an application framework all descend from the framework's view class. But in order to be both a node in *TLinkedList* and a view in PicoApp, an object had to descend from both *TDoublyLinkable* and *TPicoView*. Integrating the two separate class hierarchies proved to be difficult—not impossible, but with some negative side effects. In the end, we decided to put our emphasis on the array-based class *TList* instead, even though it, too, required a small adjustment to *PicoApp* to make its node locking work well.

We really face two problems: the problem of storing the same object in lists, trees, and other data structures at different times and the related problem of integrating data structures with application frameworks. *TList* is a good start toward a solution, but implementing trees and other nonlinear data structures inside an array turns out to be pretty complicated. Is there a better, more general solution, then?

Toward Real Generality

The real problem with software components such as *TLinkedList* and a similarly implemented tree stems from our using a piece of data as itself a node in the structure. In such a setup, the data object—a *TInteger*, for example—inherits node properties from its *TLinkable* and *TDoublyLinkable* ancestors (or their tree equivalents). That avenue seemed attractive back in Chapter 18, and we later saw how much more storage it saved than rival lists implemented with old-fashioned record-type nodes.

But despite the savings, if components built with data objects as nodes are going to throw up roadblocks when we try to use them as we'd like to in PicoApp and in relation to other kinds of structures, we might need to rethink the approach.

Let's look at two alternatives. The first falls back on the traditional linked-list, linked-tree structure that stores object-type data in a field of a record-type node. We considered this model briefly in Chapter 18. The second alternative does use objects

for nodes but separates "nodeness" from data objects. Each node object has an instance variable of type *TObject* in which it stores its data. Any descendant of *TObject* can be stored in such a node, and there's no conflict with PicoApp or other structures. We can put the same data object into a *TList*, an old *TLinkedList* (provided we install the data in a *TNode* and then add the *TNode*), a revised *TLinkedList* (built from the start with *TNodes*), a *TBinarySearchTree*, or any other structure that doesn't require data objects to know how to be nodes.

The Old Old-Fashioned Way

In a traditional linked list, a node declaration would look something like this:

```
type
  NodePtr = ^Node;
  NodeHandle = ^NodePtr;
  Node = record
      data: TObject;              { Space for data }
      next: NodeHandle;           { Links }
      prev: NodeHandle;
    end;  { Node Record }
```

This structure resembles the structure of a *TDoublyLinkable*: It has space for data (in an instance variable) and two links, one to the next node in the list and one to the previous node.

But the data is kept separate. We can store any object whatsoever in the *data* field. And the data item does not have to be a subclass of *Node*, of *TDoublyLinkable*, or of anything else—except *TObject*.

List methods can still work perfectly well with the data. To add a new item to the list, for instance, an addition method first creates a new, dynamically allocated block of storage in the heap for a *Node* record pointed to by a *NodeHandle*. Then the addition method stores its input data in the new *Node*'s *data* field. Finally, the addition method links the node into the list in the appropriate place.

That's not much different from the way we did it in *TLinkedList*. The addition methods simply skipped the intermediate step of creating a new node object. They linked the data object itself into the list because the data had inherited linkability and double linkability from its ancestors.

Of course, we noted that a traditional record-based list takes quite a bit more heap space than a *TDoublyLinkable*-based list. The record-based list needs to allocate two blocks in the heap for each list element: one block for the node and another for the data object stored in the node. Each block has 8 bytes of block header overhead. In the space-saving department, both *TLinkedList* and *TList* outperformed *TOldList*, as we called it.

The New Old-Fashioned Way

Now let's modify the traditional scheme a bit. Instead of allocating a record-based node for each list element, we'll use a node object. But, like the old-fashioned record-based node, our node object will have a separate field for storing the actual data. We've again separated nodeness from data objects. Here are some initial declarations:

```
type
  TNode = object(TObject)

      fData: TObject;              { Space for data }
      fNext: TNode;                { Links }
      fPrev: TNode;

      procedure INode (itsData: TObject);
      procedure SetNext (node: TNode);
      function NextOf: TNode;
      procedure SetPrevious (node: TNode);
      function PrevOf: TNode;
      procedure SetData (itsData: TObject);
      function DataOf: TObject;
      :
    end;  { Class TNode }
```

A *TNode* instance is an object that can hold a piece of object-type data descended from *TObject* and that can be linked with other *TNode*s. It has methods for setting and getting all its fields and for initializing itself.

Notice the listlike field and method names: *fNext*, *fPrev*, *SetNext*, and so on. This node model would be a suitable replacement for the nodes in a redesigned *TLinkedList*. But what about trees? We'll get to that topic shortly.

The important aspect of *TNode* is that the data and the node are not the same entity. We'll have no problem integrating lists or other structures made of *TNode*s with PicoApp or anything else.

A More General *TNode*

It's not essential that the *TNode* class used for lists be the same as the class used for trees. But let's at least consider a way to make one class do double duty.

Picture the structure of a doubly linked list against the structure of a binary tree. Figure 25-2 on the next page helps us to compare the two. Normally, we visualize a list extending from left to right across a page. The head is at the left end, and the tail is at the right. The nodes are doubly linked, so each node points to its predecessor (on the left) and its successor (on the right). Each node is also pointed to by both its predecessor and its successor. A binary tree, on the other hand, begins at the top

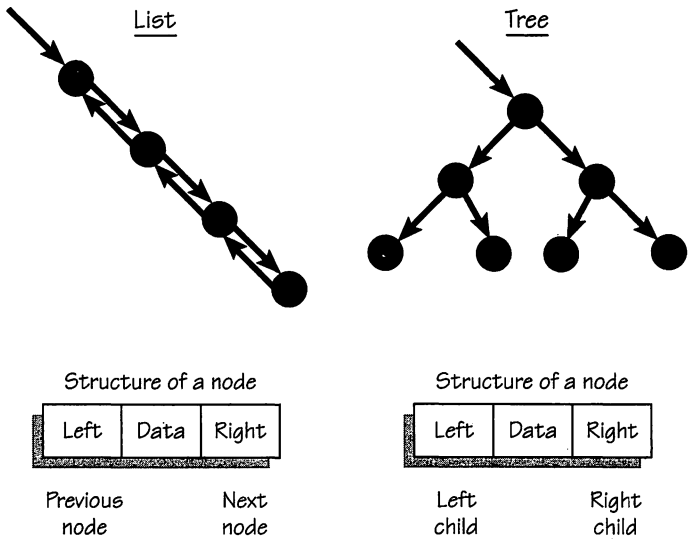


Figure 25-2.
TNode*s in a doubly linked list and a binary tree.*

with a single node, the root. Each node, including the root, has the ability to point to two child nodes. And each node is pointed to by its own parent node, to which, normally, it doesn't have a "back pointer."

The structures are dissimilar: One is linear, the other hierarchical. And in a doubly linked list, we can traverse both forward and backward, but in a tree, we can traverse only downward.

Yet the nodes are considerably alike: A list node has two links and a slot for data; a binary tree node also has two links and a slot for data. The only differences between the kinds of node are the naming conventions used for their links and the methods that operate on the nodes. A list node uses *fPrev* and *fNext*; a tree node uses *fLeft* and *fRight* to refer to its left and right children.

What's in a name? A lot—and yet not so much. Maybe we can work out a solution to the name problem.

We picture a tree node's children hanging from the node's left and right links, so we name the children for that spatial arrangement. It's not much of a conceptual leap to think of list nodes in a similar way: Instead of "previous" and "next," suppose we refer to a node's predecessor as the node on its left and its successor as the node on its right. That's natural enough when we visualize the list strung out from left to right on a page.

Let's give up the traditional list terminology and adopt more treelike terms instead, all in the name of generality and reusability. We just have to remember how to think about a list—it goes from left to right, with the head at the left and the tail at the right. Here's a new and more thorough node declaration based on this way of thinking:

```
type
  TNode = object(TObject)              { Note ancestry }

      fData: TObject;                  { Space for data object }
      fLeft, fRight: TNode;            { Previous and next or left child and right }
                                       {  child }
      fStructure: Integer;             { Structure type--specifies list or tree }
      fErrorHandler: TErrorHandler;    { Error-handler object }

      procedure INode (data: TObject; treeOrList: Integer; eh: TErrorHandler);
      procedure SetLeft (o: TNode);
      procedure SetRight (o: TNode);
      function LeftOf: TNode;
      function RightOf: TNode;
      function NumberOfChildren: Integer;  { Used only in trees}
      procedure SetData (data: TObject);
      function DataOf: TObject;
      procedure SetErrHandler (e: TErrorHandler);
      function ErrHandlerOf: TErrorHandler;
      function Clone: TObject;
      override;
      procedure Free;
      override;
      function TypeOf: Str30;
      override;
    end;  { Class TNode }
```

The final version, which you can see in the file UNode.p on the code disk in the folder Binary Tree, Ch 25, has more methods, but this is a good start.

TNode's Ancestry

We've made *TNode* a descendant of *TObject* rather than of *TLockable*. We don't plan to lock the nodes themselves. We'll lock the data the nodes contain because consumers might have outside references to the objects they store in a list or a tree, but no consumer is likely ever to have an outside reference to a node. Nodes are a mechanism entirely internal to a list or tree.

When a list or tree adds a new data item, it creates a new node to hold the item and locks the data—not the node.

Locking *TNode*'s Data

The *fData* instance variable in *TNode* is of type *TObject*. As long as locking is not in effect, the consumer can store any *TObject* descendant in that field, regardless of whether it also descends from *TLockable*.

By default, lists and trees are initialized with locking enabled. A list or a tree will automatically lock all data objects unless the consumer calls *SetLocking(false)* right after initializing the list or tree or any time the structure is empty. That's the way we did it in *TList*. *SetLocking(false)* works only if the structure is currently empty, and normally we'd call it only right after initialization.

For convenience, we'll also add *Lock*, *Unlock*, and *Locked* methods to *TNode*. These methods take care of locking and unlocking a node's data, calling upon the data object's own *TLockable* fields and methods.

The consumer doesn't have to make nodes, doesn't have to install his or her data in nodes, and isn't responsible for locking the data. The list or tree does all of that. But the consumer is responsible for adding only unlocked *TLockable* data as long as the structure is enabled for locking.

TNode's Methods

Almost all the node methods work for both listlike structures and treelike structures. The exception is *TNode.NumberOfChildren*, which indicates by its very name that it's meaningful only for trees. It returns the value *0* (no children under this node), *1*, or *2*. If a consumer did happen to call *NumberOfChildren* for a list, though, the method would correctly return *0* or *1*—a list node either has a successor node or it doesn't.

The rest of *TNode*'s methods are ambidextrous, working for both kinds of structure. That means we could build a new *TLinkedList* or a *TBinaryTree* using *TNode*.

In a list, we set a node's link to its predecessor by sending the node a SetLeft message, passing a reference to the predecessor node. To set a node's link to its successor, we send the node a SetRight message. To get a reference to the predecessor, we send LeftOf, and to get a reference to the successor, we send RightOf.

In a tree, *SetLeft* and *LeftOf* refer to a node's left child; *SetRight* and *RightOf* refer to its right child.

Both lists and trees can use the *Clone* and *Free* methods. To empower these methods for both list-type and tree-type structures, we've included in *TNode* an *fStructure* instance variable of type *Integer*. For a list-type instance, we set *fStructure* in all nodes to the constant *kList*, which equals *1*, and for a tree-type instance, we set *fStructure* in all nodes to *kTree*, which equals *2*.

The data structure object sets its nodes to the proper structure type in *INode*:

```
procedure TNode.INode (data: TObject; treeOrList: Integer; eh: TErrorHandler);
begin
  self.fStructure := treeOrList;        { Values are kList or kTree}
  self.fData := data;
  if self.fErrorHandler = nil then
    self.fErrorHandler := NewErrorHandler;
  { A new node doesn't point to anything else }
  self.fLeft := nil;
  self.fRight := nil;
end;  { TNode.INode }
```

Then other methods, such as *Clone* and *Free*, can use the value that represents the structure type to implement list or tree behavior.

Cloning Nodes

TNode.Clone does different things depending on whether the structure type is a list or a tree. Here's *TNode*'s *Clone* method—keep in mind that the overall list or tree object has a *Clone* method of its own, which calls this one to copy nodes:

```
function TNode.Clone: TObject;
  override;
  var
    temp: TNode;
begin
  if self.fErrorHandler <> nil then
    self.fErrorHandler.SetError(kNodeNoErr);
  temp := TNode(inherited Clone);  { Clone self (this node), and then... }
  if self.fData <> nil then
    temp.fData := self.fData.Clone;  {  clone self's data }

  case self.fStructure of
    kList:                              { Clone following nodes in the list }
      begin
        if self.fRight <> nil then      { Clone next node }
          temp.fRight := TNode(self.fRight.Clone);
        Clone := temp;  { Return a reference to first following node's clone }
      end;

    kTree:                  { Clone both left and right children and their }
                            {  children, recursively }
      begin
        if self.fLeft <> nil then       { Clone left child }
          temp.fLeft := TNode(self.fLeft.Clone);
```

```
        if self.fRight <> nil then  { Clone right child }
          temp.fRight := TNode(self.fRight.Clone);
        Clone := temp;
      end;
    otherwise
      begin
        if self.fErrorHandler <> nil then
          { Tree or list not specified }
          self.fErrorHandler.SetError(kAmbiguousStructure);
        Clone := nil;
      end;
  end;  { case }
end;  { TNode.Clone }
```

For a list (*fStructure* = *kList*), the *TNode.Clone* function recursively clones the next node (which recursively clones its next node, which...). When all the recursive calls have unwound, we have a copy of the list, with all its *fRight* (= *fNext*) links correctly set. That's what the nodes can do. It's up to the list object's own *Clone* method to hook up the *fLeft* (= *fPrev*) links, reset its *fMark* if it has one, and so on. The node's *Clone* method presented here doesn't do anything about *fLeft* links.

For a tree (*fStructure* = *kTree*), the *TNode.Clone* function recursively clones the root's left child (which recursively clones its left child, which...) and then recursively clones the root's right child. At each level of recursion, the node at that level will send a Clone message to its own left child and then, when that message returns, send a Clone message to its own right child. Here's the tree cloning code:

```
if self.fLeft <> nil then              { Clone left child }
  temp.fLeft := TNode(self.fLeft.Clone);
if self.fRight <> nil then             { Clone right child  }
  temp.fRight := TNode(self.fRight.Clone);
Clone := temp;
```

The first *if* statement sends the current node's (*self*'s) left child a Clone message (if the left child exists). The second *if* statement does the same for *self*'s right child. Once the left and right subtrees under the current node have been copied, the references to them are stored in the left and right child references of *temp*, the copy of *self*. Finally, *temp* is returned as the result of the *Clone* function for the current node. As each node's *Clone* function returns, it hands back a complete copy of all children, grandchildren, great-grandchildren, and so on, for that node.

Each node is responsible for cloning the subtree hanging from its own left child link, cloning the subtree hanging from its own right child link, and cloning itself, including references to its copies of the left and right subtrees. The process is like the node cloning for *TLinkedList* in Chapter 20. If you followed the recursion there, you'll be able to work it out here.

Freeing Nodes

Freeing the nodes of a locked list or tree poses some special problems. As we free the whole data structure, we have to

- Unlock each node's data object (*fData*)
- Possibly free the node's data object
- Free the node itself

That turns out to be a bit tricky as we move down the structure from node to node. In order to free each node, we need the list's or tree's *fSignature* value that was used to lock the node's data originally. But the *Free* method, as inherited from *TObject*, takes no parameters, and we can't simply add parameters to an override.

We'll write a new method, *TNode.PRIVATERelease*, instead of using *TObject*'s *Free*. We prefix the method name with *PRIVATE* because we don't want consumers to call the method.

PRIVATERelease takes two parameters: an *Integer* signature and a *Boolean* value, *killData*. The signature is used to unlock the node's data item. If *killData* is *true*, *PRIVATERelease* instructs the node to destroy its data item by sending the data a Free message. If *killData* is *false*, the data item is not destroyed. We make it possible to free the node object without freeing the node's data because the consumer might need the objects in the list or tree after their roles in the data structure are over.

PRIVATERelease unlocks the node, optionally frees the node's data, and sends recursive PRIVATERelease messages to the node's successor (in a list) or its children (in a tree). When those recursive calls return, *PRIVATERelease* calls

```
self.Free;
```

to free the node object. At that point, all of the node object's successors or children have already freed themselves.

> **CAUTION:** *You'll seldom see or use the message self.Free. Notice that we sent this message from within a* PRIVATERelease *method, not from an overridden* Free *method. In this context, it's safe—the node's inherited* Free *method is called. But if you called* self.Free *from inside the node's overridden* Free *method (if it had one), you'd be recursively calling the same method again, and again, and again, with no base case to stop the recursion.*

Here's the code for *PRIVATERelease*:

```
procedure TNode.PRIVATERelease (signature: Integer; killData: Boolean);
begin
  if self.fErrorHandler <> nil then
    self.fErrorHandler.SetError(kNodeNoErr);
```

```
    { Unlock this node and then release its successors or children }
    { When all successors or children are released, free this node }
    if TLockable(self.fData).Unlock(signature) then
      begin
        case self.fStructure of
          kList:
            if self.fRight <> nil then   { Free next node }
              self.fRight.PRIVATERelease(signature, killData);
        kTree:
          begin
            if self.fLeft <> nil then    { Free left subtree }
              self.fLeft.PRIVATERelease(signature, killData);
            if self.fRight <> nil then   { Free right subtree }
              self.fRight.PRIVATERelease(signature, killData);
          end;
        otherwise
          if self.fErrorHandler <> nil then
            self.fErrorHandler.SetError(kAmbiguousStructure);
          end;  { case }
        if killData then                 { Destroy data if indicated }
          self.fData.Free;
        self.Free;                       { Nodes inherit Free }
      end  { We're calling Free from PRIVATERelease }
    else if self.fErrorHandler <> nil then
      self.fErrorHandler.FatalError(kUnauthorizedFree);
  end;  { TNode.PRIVATERelease }
```

TNode Error Handling

Like *TDoublyLinkable*, *TNode* is not descended from *TErrorHandler* but merely contains a field for holding a *TErrorHandler* object (the "composite object" approach we looked at in Chapter 8). *INode* might instantiate an error-handler object and install it in *fErrorHandler*; or the list or tree might install a reference to its own error-handler object in each node by means of the *eh* parameter so that all nodes could use the same error handler as the list or tree itself. Treating *TErrorHandler* this way also reduces space overhead a little because we need only one 4-byte field for the handler instead of inheriting two integer fields and a Boolean field—a small savings, but a savings nonetheless.

TNodes vs. Record-Type Nodes

Now that we've seen both a traditional record-type node and the new *TNode*, let's figure out which is better.

The traditional node, because it's a record, inherits nothing. The space cost per node is 12 bytes—4 each for the data and the two links—plus an 8-byte block header,

plus a separate block for the data object. Let's say that the data object is a *TInteger*, which has 2 bytes of data and 2 bytes for its class link, plus its own 8-byte block header. To store the *TInteger* instance in a traditional node, we'd need 32 bytes.

TNode inherits no instance variables from *TObject*, but it does require 2 bytes for a class link, 4 bytes for data, 8 bytes total for left and right links, 4 bytes for an error handler (which is lacking in the traditional node but could be added), 2 bytes for the *fStructure* field, and 8 bytes for the node's block header—for a total of 28 bytes. When we add in 12 bytes for a *TInteger* object, the total is 40 bytes.

So a *TNode* costs 8 bytes more per node (4 if we discount the error handler). That might be enough in some cases to make us prefer the old-fashioned record-based node. The old-style node can also be made to be ambidextrous in exactly the same way as the object-type node, so it will work for both lists and trees.

But there's one more consideration on *TNode*'s side. A record can't be extended without directly revising source code. *TNode*, as a class, can be extended by means of subclassing. Suppose we need a node type for trinary trees, say. A *TTrinaryNode* class can be derived easily, and it will still work in lists as well as trees.

The bottom line? A good class library will have both record-type nodes and object-type nodes, and various lists and trees built from both kinds. If space is at a premium, the old-style node is cheaper. If extensibility is more important than space, *TNode* is preferable.

Our example binary search tree class will be constructed with *TNode*s.

Real Trees

Just as *TDoublyLinkable*s provide the raw material for lists but don't actually build the lists, *TNode*s provide the raw material for both lists and trees. But we still have to build the structures.

To implement some particular kind of tree object, we'd need to write a tree object class for it: for instance, a binary search tree class. This class would have methods that call the *TNode* methods to implement the structure and characteristics of a binary search tree—which is only one of many different kinds of tree. The tree class would be the tree equivalent of *TLinkedList* or some other linked list component. We'll look at a binary search tree class a little later in this chapter.

Beyond Binary

With its *fLeft* and *fRight* fields and corresponding methods, *TNode* is optimized for binary trees, the most commonly used kind. To implement trees of higher "degree" (more children per node), you can subclass *TNode* to add more fields and methods. One naming scheme is to retain Left and Right and add new children in between as A, B, C, and so on. Methods would include *SetA*, *BOf*, and the like.

To implement a completely general tree, with any number of children at each node, you might create a special node class and, instead of individual child fields, install a list of node objects in each node. Then the node could install its children in the list.

A Binary Search Tree Software Component

A binary search tree is often used to implement the abstraction of a search table—in compilers, databases, file managers, and so on—because it allows fast access. A binary search tree also has useful properties for sorting data. Each node contains some data and references to two children. A data field of the node is used as a key field to establish an ordering relationship for the nodes in the tree. Suppose we have a binary search tree in ascending order. Let's look at the data value of a node: The node will have a data value greater than the values of any of its left children (or their children), and less than the values of any of its right children. Figure 25-3 illustrates binary search trees with nodes in ascending and descending order. The relationship between a node and its children is defined in a way that prohibits duplicate values.

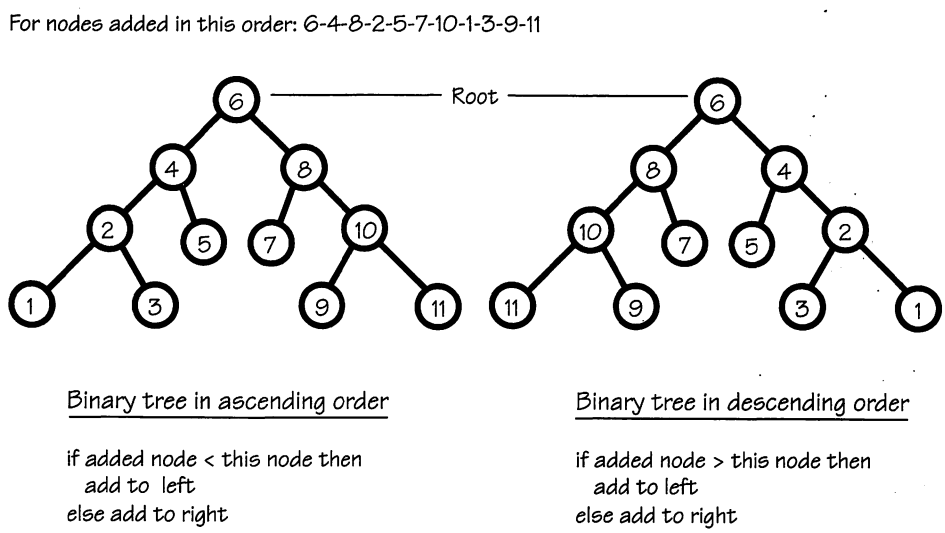


Figure 25-3.
Ascending and descending order in binary search trees.

When we add a node to a search tree, our addition method has to search for the right place to insert the node according to our tree's ordering principle. Let's assume ascending order—we might be ordering strings alphabetically, say, with one string per node. First, the new node's data is compared to the root node's. If it's equal, the duplicate is suppressed—it isn't added. If the new node is greater than the root, the comparison is moved to the root's right child. Each comparison moves to the left or the right child until there is no left or right child to move to. Then the new node is added as the child of the appropriate node.

Removing a node is trickier. Removing a leaf node (one with no children) is no problem. But removing an interior node (or the root) calls for reconstructing the tree after the target node has been clipped out. If the node to be removed has only one child, that child simply moves up to replace it. But if the node has two children (which could, in turn, have children of their own), it must be replaced with its proper successor. This is neither a data structures book nor an algorithms book, so we'll avoid the messy details. Consult any standard data structures text, and examine the code in the *TBinarySearchTree.Delete* method on the code disk.

Binary Search Tree Objects

A binary search tree object resembles a list object, such as a *TList*, to some extent. The tree object is a manager object that contains an instance variable of type *TNode* to point to the root node. The root node, in turn, has *fLeft* and *fRight* instance variables to point to its two children. And so on. We build the tree by inserting new nodes as either left or right children according to the algorithm for inserting nodes in a binary search tree.

Here's a simplified binary search tree class. (The real one on the code disk has more instance variables and methods.)

```
type
  TraversalOrders = (preorder, inorder, postorder);

type
  TBinarySearchTree = object(TLockable)
      fRoot: TNode;                { First node of the tree }
      fAscendingOrder: Boolean;    { Used to make tree ascending or descending }
      :
      procedure IBinarySearchTree (ascending: Boolean);
      function TypeMatches (o: TObject): Boolean;
      procedure Insert (o: TObject; s: TTreeSearchKey);
      function Delete (s: TTreeSearchKey): TObject;
      function IsPresent (s: TTreeSearchKey): Boolean;
      function Find (s: TTreeSearchKey): TObject;
      function PreorderTraversal (node: TNode; a: TTraversalAction;
        level: Integer);
      function InorderTraversal (node: TNode; a: TTraversalAction);
      function PostorderTraversal (node: TNode; a: TTraversalAction);
      procedure DisplayTree (order: TraversalOrders);
      function Clone: TObject;
      override;
      procedure Free;
      override;
      function RootOf: TNode;
```

```
    function AscendingOrder: Boolean;
    procedure Clear(ascending: Boolean);
    function TypeOf: Str30;
    override;
  end; { Class TBinarySearchTree }
```

The tree class provides methods to insert and delete nodes, to determine whether a value is present in the tree, and to find and return a given node's data. For *Delete*, *IsPresent*, and *Find*, we pass in a *TTreeSearchKey* object, which provides appropriate instance variables, an *Init* method, and *Equals* and *LessThan* methods that can be used for data comparisons. The idea is very much the same as in lists.

The *TBinarySearchTree* class also provides a display method and methods to traverse the tree, visiting each node in the tree in some order and accomplishing some task at each node. We've already been through the difficulties of doing some arbitrary action at each node of a list. It's the same here, and the solution parallels one we came up with in Chapter 23 for *TList*: using a traversal action object. In this case, we pass an object of a *TTraversalAction* subclass to one of the traversal methods. For the display method, we specify the order we want for the display: preorder, inorder, or postorder. This means that overriding *TObject*'s *Display* method won't work because *TObject.Display* has no parameters. Instead, we create the method *DisplayTree*, which takes one "order" parameter. If the parameter is *inorder*, the traversal will return nodes "inorder." Figure 25-4 illustrates a partial inorder tree traversal that would call the *Display* method. *DisplayTree* shows the tree's nodes in the specified order by traversing in that order and calling each node's *Display* method.

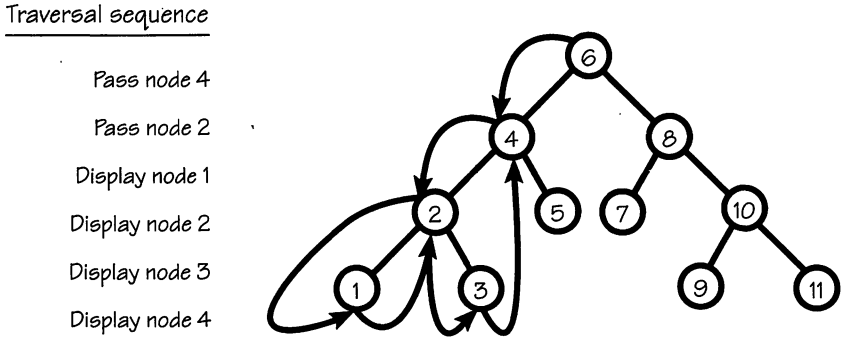


Figure 25-4.
A partial inorder tree traversal. Visits go first to a node's left child, then to the node, and then to the node's right child.

The *AscendingOrder* method gets the value of *fAscendingOrder*. The search tree class can be used to build either an ascending order tree or a descending order tree. The consumer isn't allowed to change the order once it's established unless he or she

clears the tree. *Clear* frees the existing nodes and reinitializes the tree object to empty. We can respecify whether nodes are to be stored in ascending or descending order by passing *true* or *false* to *Clear*.

The binary search tree class also overrides *TObject.Clone* and *TObject.Free* to take care of the whole tree. *Clone* must clone not only the tree object but also all of its nodes, just as in *TLinkedList.Clone*. *Free* has to free the tree nodes and then call *inherited Free* to free the tree object.

Traversing a Tree

TBinarySearchTree comes with several methods for traversing the tree to perform actions on its nodes. All of the traversal methods use recursion—for example, to traverse the tree inorder, the traversal method would

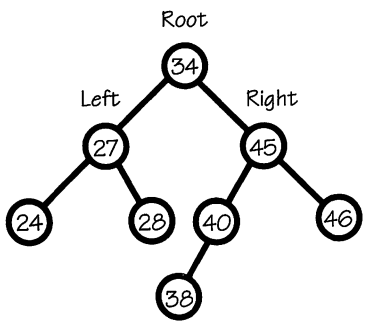
1. Call itself recursively to traverse the left subtree (the current node's left children, grandchildren, and so on)
2. Perform an action on the current node
3. Call itself recursively to traverse the right subtree

TBinarySearchTree has three traversal methods. *PreorderTraversal* visits each node in the tree before visiting the node's children. *InorderTraversal* visits each node in the tree between visits to the node's left and right children. *PostorderTraversal* visits each node in the tree after visiting both its left and right children. Here's *InorderTraversal*:

```
procedure TBinarySearchTree.InorderTraversal (node: TNode; a: TTraversalAction);
begin
  { Recursive base case--last node has no left or right child }
  if node <> nil then
    begin
      { Traverse the left subtree }
      self.InorderTraversal(node.LeftOf, a);
      { Visit the node and perform action.DoIt }
      a.DoIt(node);
      { Traverse the right subtree }
      self.InorderTraversal(node.RightOf, a);
    end;
end;  { TBinarySearchTree.InorderTraversal }
```

The *DisplayTree* method is based on the traversal methods. It takes one parameter, of the enumerated type *TraversalOrders*. Depending on the parameter passed, *DisplayTree* calls *PreorderTraversal*, *InorderTraversal*, or *PostorderTraversal*, passing to the method a *TDisplayTraversal* object that sends each node's data object a Display message.

PreorderTraversal is special in that the consumer can use it to display a staggered diagram of the tree with depth. Figure 25-5 shows two views of the same tree: a conventional view, with the root at the top and its children hanging down, and the indented view produced by the method call *DisplayTree(preorder)*. *PreorderTraversal* uses the method *PRIVATEDisplayLevel* to indent displayed data values. We can turn the indenting off by setting *PreorderTraversal*'s *level* parameter to 0. We can also subclass the tree and override the *PRIVATEDisplayLevel* method to change the amount of indenting, make the display more graphic, and so on. See the definition of *PreorderTraversal* in file UBinarySearchTree.p on the code disk.



Format from *DisplayTree(preorder)*:

```
34      Root node's value
  27    Root's left child
    24  Root's left child's left child
    28  Root's left child's right child
  45    Root's right child
    40  Root's right child's left child
      38  Root's right child's left child's left child
    46  Root's right child's right child
```

Figure 25-5.
A conventional tree view and the indented notation form accomplished by DisplayTree(preorder).

Other Tree Features

TBinarySearchTree also implements optional locking, optional freeing of the contained data when the tree is freed, and optional type screening. These are implemented very much as they were in *TList*, so we won't go into the details. See the file UBinarySearchTree.p in the folder Binary Tree, Ch 25 on the code disk.

Example Program: Tree Demo

The folder Binary Tree, Ch 25 on the code disk contains the source code for classes *TNode* and *TBinarySearchTree*. The folder also contains code for a number of *TTreeSearchKey* and *TTraversalAction* classes. The sample program, Tree Demo, simply puts the binary search tree class through its paces, adding nodes, deleting nodes, displaying, traversing, clearing, and so forth. Figure 25-6 shows the compilation order for the files in that program.

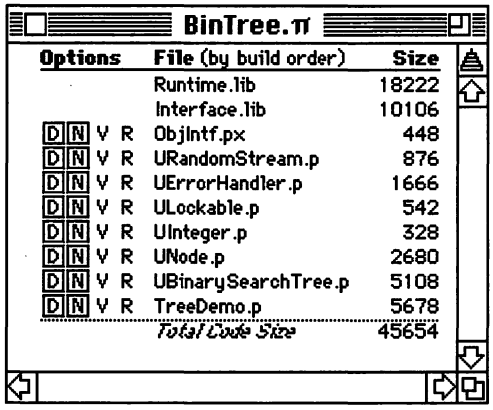


Figure 25-6.
Compilation order for binary search tree demo program.

About the files

You should be familiar by now with Runtime.lib, Interface.lib, and ObjIntf.px (my modified version of the ObjIntf.p file that comes with the THINK Pascal package).

URandomStream.p contains the random number–generator class we used in Chapter 6 to generate random auto parts. This file is on the disk in the folder Auto Parts, Ch 6. We use it here to generate random integers to store in the tree.

UErrorHandler.p is the error-handling mechanism we've been using all along. We covered a basic version in Chapter 8, and a fuller version is embedded in the structure of PicoApp. This file is on the code disk in the folder Common Units.

UNode.p contains class *TNode*, which lets objects be nodes.

UBinarySearchTree.p provides class *TBinarySearchTree*, which, like a list manager object, manages an underlying binary search tree composed of *TNode* objects. You can add to the tree, delete from it, traverse it to accomplish various tasks, and so on. You can also subclass it to build fast lookup tables.

TreeDemo.p is a simple, non-PicoApp test program designed to put the tree through its paces.

To compile the Tree Demo program, you'll need to create a THINK Pascal project that duplicates the list of files by the build order shown in Figure 25-6.

Open Array Parameters

While we're on the general subject of data structures, let's deal with a perennial shortcoming of standard Pascal: the problem of array of *Char* parameters in procedures and functions that require array bounds. The following solution for providing "open" array parameters should make writing generic procedures easier. (Actually, we'll look at two solutions, both of which rely on polymorphism.)

Suppose we have a procedure that needs to loop through an array of *Char* for some purpose—perhaps to make a copy, or perhaps to search for a key. In Pascal, formal parameters need to be specific array types, declared complete with array bounds. This makes writing a general *Search* routine difficult because we might want to call it for *Char* arrays of a variety of lengths. Standard Pascal requires us to write a separate *Search* procedure for each array length, but the *Search* function procedure heading might look like this in Pascal's descendant, Modula-2:

```
PROCEDURE Search(s: ARRAY OF CHAR; key: ARRAY OF CHAR): INTEGER;
```

An array of *Char* of any size could be passed for either parameter. Such parameters are called open array parameters. They specify the type of the array but not its length. For instance, arrays of the following sizes could be passed to the same function in Modula-2:

```
TYPE StringType = ARRAY[1..256] OF CHAR;
     KeyType = ARRAY[1..15] OF CHAR;
```

Inside function *Search*, how can the function tell how long its *s* and *key* actual parameters are? How can it loop through the *s* array looking for *key*? The loop would look something like this in standard Pascal:

```
for i := 1 to ??? do                    { What's the upper bound? }
  if s[i] = ...
```

How can the function determine the upper bound of the *for* loop? In Modula-2, a built-in function, HIGH, can tell the upper bound of any array such as *s* or *key*, so the loop inside *Search* would look like this in Modula-2:

```
FOR i := 1 TO HIGH(s) DO          { Using the HIGH function }
  IF s[i] = ... THEN
    :
  END;
END;
```

Could we arrange to do something remotely similar in Object Pascal? Let's find out.

An Array Object Class

In Object Pascal, the only way to come close to Modula-2's open array parameters is with the same kind of object-type parameter we've been using for other purposes, such as searches on keys and traversal actions. A parameter might be of type *TArray*, for instance. Then, with polymorphism and runtime binding, any descendant of *TArray* could be passed in the parameter.

That has important consequences for arrays, though. We still can't pass ordinary Pascal arrays of different lengths. We'll have to pass them as array objects.

What's an array object? It's a Pascal array encapsulated inside a manager object that manages access to the array. The array object contains both the array and methods to put data into the array, take data out of the array, and find out how big the array is. Of course, using such an encapsulated array will feel and look different from using an ordinary array.

But like an ordinary array, a particular array object class will be able to handle only an array with a fixed element type and a fixed size (although we'll look at an alternative shortly). Here's a possible *T20CharArray* class, for instance:

```
type
  T20CharArray = object(TObject)

      fArray: array[1..20] of Char;
      fHigh: Integer;

      procedure I20CharArray;
      procedure Put (c: Char; at: Integer);
      function Get (from: Integer): Char;
      function High: Integer;
    end;  { Class T20CharArray }
```

This array object class manages an array of fixed size (*20*) and type (*Char*). The initialization method sets the value of *fHigh* to *20*. The *High* method simply returns that value—the array's upper bound, which can be used in loops. It shouldn't be difficult to generalize *T20CharArray* to a whole hierarchy of *Char* array classes. At the top would be class *TCharArray*, whose *Put*, *Get*, and *High* methods would be complete and functional but which would have a stub for its initialization method. All *Char* array objects with specific lengths would descend from *TCharArray*.

Here's function *T20CharArray.High*:

```
function T20CharArray.High: Integer;
begin
  High := self.fHigh;                { Simply reads the fHigh instance variable }
end;  { T20CharArray.High }
```

Of course, in Pascal we can have an array of any other element type—integer, real, enumerated, record, pointer, object reference, set—even of other array object elements. And an array can be indexed by other types besides integer—for example,

```
for c := 'a' to 'z' do ...
```

This array object scheme is not completely general, so we do need a superclass for each type we might want to put into an array: one for integers, one for characters, and so on. (And we might want classes in which, say, *fHigh* is of type *Char*, or even type *TFruit*.)

Here's some code that creates, initializes, and uses a 20-element array object:

```
var
  anArray: T20CharArray;
⋮
New(anArray);
anArray.I20CharArray;              { Set array object's fHigh field to 20 }
⋮
for i := 1 to anArray.High do      { Use High method }
  anArray.Put(Chr(i + 32), i);     { Use Put method }
⋮
for i := 1 to anArray.High do
  Write(anArray.Get(i), ', ');     { Use Get method }
WriteLn;
⋮
```

The code creates an array object, sets its upper bound information to *20* with *I20CharArray*, uses *High* to extract the upper bound for use in the *for* loops, puts ASCII characters into the array with *Put*, and writes out the values obtained from the array with *Get*.

Writing a Procedure with Open Array Parameters

Recall that the whole purpose of designing array objects was to generalize passing array parameters so that we can write procedures and functions (or methods) general enough to handle, say, any one-dimensional array of *Char*. Our example will be a procedure to display one-dimensional arrays of *Char* of any size. This procedure helps us out with the open array problem by saving us from having to write as many different display procedures for *Char* arrays as there are lengths of *Char* arrays. Here's the display procedure (which could be a method for some class):

```
procedure DisplayArray(a: TCharArray);  { Polymorphic parameter }
  const
    rowLength = 10;
  var
    i: Integer;
begin
  for i := 1 to a.High do
    begin
      if (i mod rowLength = 0) then
        WriteLn;
      Write(a.Get(i), ', ');
    end;  { for }
  WriteLn;
end;  { DisplayArray }
```

By using a parameter of type *TCharArray*, the procedure can accept actual parameters of any array descendant of *TCharArray*—hence, any *Char* array of any size. Inside the procedure, we ask the array object for its size by sending it the High message.

We could just as easily pass a parameter of type *T80CharArray* or *T256CharArray*.

What About Multidimensional Arrays?

Arrays with two, three, or more dimensions can be handled by extending the mechanism.

A two-dimensional array, for instance, would have two upper bound fields, *fHigh* and *fHigh2*. It would also need two functions on the order of *High*, one to extract the first dimension's upper bound from *fHigh* and the other to extract the second dimension's upper bound from *fHigh2*. Alternatively, one method could serve for both, by taking a parameter indicating which dimension was needed and returning the upper bound value for the correct dimension:

```
function High(whichDim: Integer): Integer;
```

A Dynamic Array Class

I hinted earlier at an array object approach that doesn't have to specify an upper dimension. Thus, instead of creating a class for *T33CharArray*, say, we could create a *TCharArray* class and specify its maximum size at runtime by means of a parameter. How?

We developed a *TDynamicArray* class in Chapter 21, although we didn't spend much time on it. A class like *TDynamicArray* manages an underlying array implemented as a dynamic data structure based on a handle. At runtime, we allocate a handle of the desired size and treat it as a handle to an array of *TObjects* (or possibly to something else).

When you try to add an item to a *TDynamicArray* instance that is already full, the array accommodates the new data by growing.

LengthOf reports the index value of the highest slot that currently has non-*nil* data in it—the upper bound of the current logical array. *HighOf* is analogous to the *High* function we've developed in classes like *TCharArray*. It returns the number of allocated slots, hence the upper bound of the current physical array. Depending on our needs, we could use either *LengthOf* or *HighOf* in procedures and methods to get the effect of Modula-2's HIGH function. In most cases, *LengthOf* would be the more efficient.

Put and *Get* are used to put data into the array and take it out. For each, we specify which slot by index value. If we're putting non-object data into the array, we have to typecast. For example, to put a Macintosh handle into slot 3, we'd call

```
if anArray.Put(TObject(aHandle), 3) then ...
```

and to take the handle out, we'd do this:

```
h := Handle(anArray.Get(3));          { Uses a typecast }
```

DoToEach lets us traverse the array, doing some action to each object (or non-object) in it. The approach is one we covered in Chapter 23. You can see the code for *TDynamicArray* in the file UHandleArray.p in the folder List, Part 3 on the disk.

Can We Use *TDynamicArray* as an Open Array Parameter?

Sure. We can use *TDynamicArray* to provide open array parameters. To use it for open arrays, we make a parameter of type *TDynamicArray*. Inside the procedure, we call the array's *HighOf* or *LengthOf* method to get the upper bound.

What about *Char* arrays for the open array parameter type instead of *TObject* arrays? There are two basic approaches. One is to rewrite *TDynamicArray* to hold *Char*s instead of *TObject*s. That might not be a bad idea (and it's one proposed as a project in Chapter 21). The other approach is to derive a *TDynamicCharArray* class from *TDynamicArray* by importing. *TDynamicCharArray* would use a *TDynamicArray* as the underlying data structure, calling its methods to implement the *Put* and *Get* methods to store and retrieve *Char*s as *TChar* objects. We could create similar classes for other useful array types, as we did in Chapter 24 for stacks.

Streams

One topic we haven't covered so far is input/output. As it turns out, even files aren't protected from our desire to abstract and to see the whole world as objects.

Files as Streams

Figure 25-7 shows an abstract stream sequence. The term "stream" refers to a sequence of data items. Thus a file *is-a* stream. You can look at a file—or any other stream—on two levels. At the lower level, a file is a sequence of raw bytes. At a higher level, it's a sequence of data items: integers or characters or personnel records. If you look at the Macintosh File Manager, of course, what you'll see are routines for reading and writing bytes. If you want to create your own file at some higher level, you can either use the Pascal file types, such as *File of Char* or *File of PersonnelRecord*, or you can use more primitive File Manager calls, such as *FSRead* and *FSWrite*. For instance, you might call something like

```
err := FSRead(refNum, SizeOf(Integer), data);
```

to implement a procedure that reads an integer.

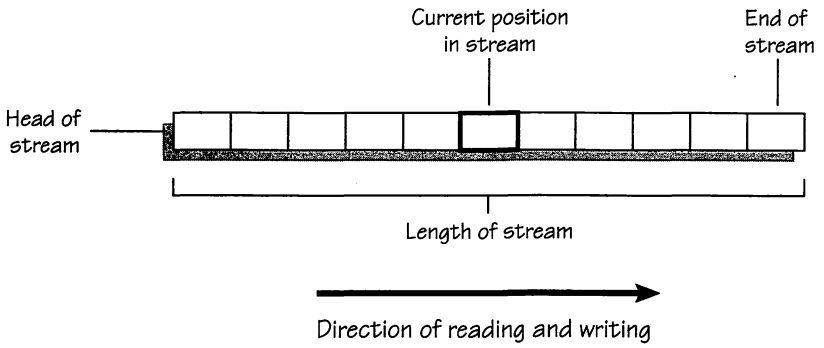


Figure 25-7.
An abstract stream structure. A stream can be read only, write only, or read-write, and a stream can be empty.

It's possible, certainly, to write object classes that represent files. TCL does just that with its *CFile* and *CDataFile* classes.

But our goal is even more abstract and general. We want an abstract stream class, plus several more specific stream subclasses, to represent streams of data at a higher level than the file. We'd like streams of *Integer*s, streams of *Char*s, streams of *PersonnelRecord*s, streams of objects, and even mixed streams. We'd like to put an item into the stream or get an item from the stream without thinking about the messy details.

Non-file streams

An extra motivation for that goal is that there are other data structures besides files that fit the stream abstraction. First, as Wilson (1990) points out, the stream abstraction fits nicely with *data buffers*—pointers or handles to some data in memory that you want to "read" or "write" just as if you were doing so with a file. Wilson presents a stream subclass for "handled" data, calling it *THandleStream* (something like our *THandleArray*). You can get the next data item from such a stream, put a new data item into the stream, and so forth, just as you can with a file. At the abstract level, there's no real difference between a block of handled data and a disk file.

As a second non-file stream example, consider the design for the *TRandom* class, which we've already characterized as a "stream" of random numbers. At any point, we can get the next item from the stream. Of course, this stream is read only. Other streams might be write only or might allow both reading and writing, as direct-access files do.

The Goal

Our goal is to create a hierarchy of stream classes that embody everything from files to handled buffers to random number generators to whatever. As long as something

has streamlike structure and behavior, we'd like it to be a subclass of an abstract *TStream* class. Any source or sink of data items can qualify.

Much of this work has been done already. For three excellent treatments of stream classes, see Wilson, et al. (1990), who present a set of stream classes for MacApp; the example programs that come with Turbo Pascal 5.5 for MS-DOS; and the stream classes in Turbo Vision, the application framework that comes with Turbo Pascal 6.0 (Keller 1990).

Those sources present complete stream designs, so we'll simply refer to Wilson here.

Filter Objects

One other useful operation we can perform on a stream is to "filter" it. We can set up a mechanism that lets us filter out all but a certain kind of item when writing to or reading from the stream. We might, for instance, filter a random stream so that it produces only odd random integers, throwing even random integers away.

A filter object tests a piece of data passed to it to see whether the data meets some criterion. The filter object lets "good" items pass and filters out "bad" ones. The filter's *Good* method returns *true* if the data passes the test; *false*, otherwise.

Class *TFilter*

Here's what *TFilter* looks like. Notice how similar the filter object is to search key and traversal objects.

```
type
  TFilter = object(TObject)

      { Add data variables here }

      { Add an initialization method here }
      function Good: Boolean;     { Override this method }

    end;  { Class TFilter }
```

To use the *TFilter* class, we subclass it, building in an instance variable(s) of the type(s) we want to test. We also write an initialization method that sets the values of the variable(s). And we override *Good* to do the test.

When we instantiate our *TFilter* subclass,

- We install the *TFilter* object in a target stream object, using the stream's installation method, probably called *SetFilter*
- The stream calls the filter's initialization method to load data into the filter's instance variables
- The stream calls the filter's *Good* method to test the data

Example: filtering a stream

For example, here's a *TFilter* subclass that filters a stream delivering integers so that the stream delivers only odd integers. The subclass can filter any stream of integers.

```
type
  TOddFilter = object(TFilter)

      fInt: Integer;

      procedure Init (i: Integer);
      function Good: Boolean;
      override;
    end;  { Class TOddFilter }
```

Now, let's install an instance of *TOddFilter* in a *TRandomStream* stream (two methods of which deliver integers) so that the stream will deliver only odd random integers:

```
var
  anOddFilter: TOddFilter;
  aRandStream: TRandomStream;
⋮
aRandStream.SetFilter(anOddFilter);
⋮
num := aRandStream.Get;          { Delivers an odd random }
⋮
```

In the stream

Inside *TRandomStream*, the *Get* method tests to see whether its filter object is *nil*. If it is, the stream is unfiltered and *Get* simply gets the next item from the stream. If the stream is filtered, *Get* repeatedly gets an item and initializes the filter with the item until the item passes the *Good* test:

```
{ Inside TRandomStream.Get... }

if self.fFilter <> nil then          { Test whether stream is filtered }
  repeat
    rand := Random;                  { This does the stream's job }
    self.fFilter.Init(rand);         { This sets up the filtering }
  until self.fFilter.Good;           { This does the filtering }
else
  rand := Random;                    { If stream is unfiltered }
Get := rand;
```

Any stream's *Get* and *Put* method (or whatever we call them) should be set up along these lines so that the stream can use filter objects.

Inside a *TFilter* instance

All *TOddFilter.Init* does is to assign its parameter to the *fInt* instance variable.

TOddFilter.Good looks like this:

```
function TOddFilter.Good: Integer;
  override;
begin
  Good := Odd(fInt);
end;  { TOddFilter.Good }
```

Filters and Streams Together

The stream's accommodations to filtering are

- Providing an instance variable of type *TFilter* to hold a filter object
- Providing a method, *SetFilter*, to set that instance variable
- Providing the mechanism to use the filter, inside the stream's *Get* and *Put* methods

Of course, the kind of filter object we install in a stream must be compatible with the stream's data type. But we can install a filter for any kind of data the stream gets or puts. For example, given a stream of personnel objects, we could install a personnel filter that would pick out only the personnel of executive rank, or all personnel with salaries in a certain range. We could also use complicated filter tests with multiple Boolean conditions, just as in search key objects.

Other Kinds of Filter Objects

To complete our consideration of filters, let's look at two rather different ideas for filter classes.

The filter design we've done installs a filter object in a stream. Then, when our code calls the stream object, the stream calls its filter's methods to test its data. If the data passes muster, the stream delivers the data normally. The filtering action is hidden inside the stream. It's still the stream that we call, and our calls to it are the same regardless of whether it's filtered; the stream still generates the same kind of data.

But here's another problem we can address with filters. Suppose our stream delivers characters, one at a time. And suppose that we'd like to filter the stream so that it delivers words or tokens rather than individual characters. Clearly, the design we've done won't do that job.

As long as we install the filter in the stream and then call the stream, we can't get the stream to deliver words or tokens. A character stream delivers a character.

But suppose that we reverse relationships and install a stream in a filter. We'd create a filter object that knows how to call a stream to get characters, say. Then we'd

create a character stream and install it in the filter object and make our calls to the filter, not to the stream. Every time we called

```
aToken := aFilter.Get;
```

the filter would call

```
ch := aStream.Get;
```

enough times to get a complete token in a string. Then the filter would pass back the token. Figure 25-8 shows both the first approach we looked at and this new filter approach.

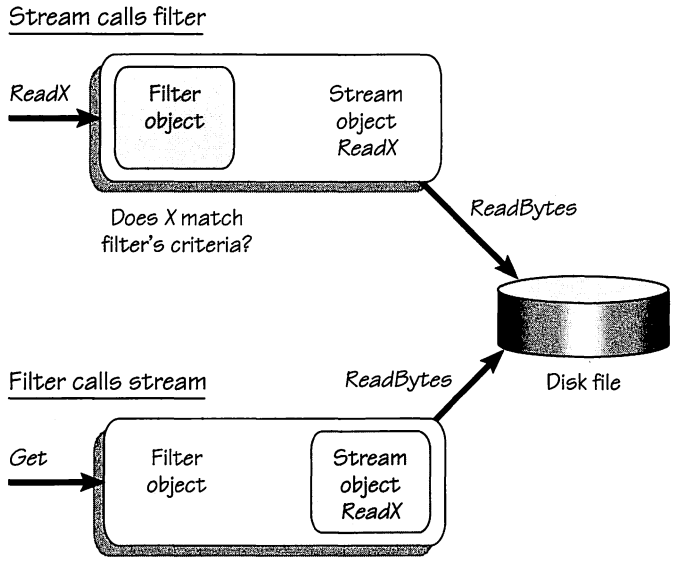


Figure 25-8.
Two filter schemes.

Would this design make a better general filter? In other words, would it also do the tasks for filters we looked at earlier, such as filtering a stream to get only odd random integers? It would. We'd use a filter designed to call a stream for integers and weed out the even integers. Then we'd install a stream in the filter so that the filter could call the stream.

But let's not throw out the original design. There's no reason at all not to have both kinds of filter objects in our class library. Then we'd have a variety of approaches suitable for all sorts of circumstances—and that's exactly what we want for our class library.

One other kind of filter we've briefly noted is the one postulated in Chapter 14 for working with an Undo mechanism. When a user makes an undoable change to a view, the data involved in the change is saved in some alternate data structure rather

than in the normal one for the view. This preserves the view's previous data so that it can be redrawn if the user chooses Undo. The purpose of a filter object in this case is to maintain the alternate data and draw it to the view instead of the data that's in the view's normal data structure. By filtering, we draw the data without changing the real data structure. We can leave the implementation of such a filter for those who choose to implement Undo in PicoApp.

Summary

In this chapter, we added several more items to the class inventory, at least in rough or theoretical form: tree classes; a new node class that can be used in lists, trees, and other data structures; array classes for open array parameters; various kinds of stream classes; and several kinds of filter classes.

Of course, these are only a few illustrations drawn from a vast range of possibilities. Anything we can conceive of as an entity can be represented as an object, and, as we've seen, whole applications can be built from objects.

Our ultimate goal is to have at our disposal a full shelf of easy-to-use, easy-to-modify software components, built with objects, that we can plug into our object-oriented application framework to produce software more quickly, cheaply, reliably, and maintainably than before. And, feeding that shelf of components, we can hope to see one day soon a whole industry supplying our needs with high-quality, well-tested, reusable components. Here's to that aim.

Projects

- Flesh out a collection of character array classes.
- Build a simple string class, on the order of *TInteger*.
- Build a dynamic string class. It might use some of the same techniques that *TDynamicArray does*, basing the string on a resizable handle. (See the projects for Chapter 21.)
- Build a collection class—an ordered collection—based on the binary search tree that we developed in this chapter. Note that the tree uses linked objects descended from the class *TNode*. Even so, the tree should fit easily into the *TCollection* framework developed in Chapter 24.
- Make the binary search tree an AVL tree, or height-balanced tree, for better efficiency.
- Build a general set class using hashing. You might have the user provide a hash object that defines a hashing function suitable to his or her data. Your set code would call the hash object's *Hash* method.
- Build an ordered collection class based on your set.
- Build a general hash table with collision resolution. You might use a dynamic array of lists, putting duplicates into the list for the particular array element.

- Build more kinds of trees: B-trees, B+-trees, a general tree (a tree whose nodes can have any number of children).
- Read Wilson (1990), and implement classes *TStream*, *TFileStream*, *THandleStream*, and *TRandomStream* for PicoApp. (And take a look at Wilson's *TCountingStream*.)
- Implement some form of object persistence, or "storability." You might try adding the appropriate methods to *TObject*.
- Implement a few potentially useful *TFilter* subclasses. You might develop one that calls a character stream and builds tokens. Or another that delivers words. Or one that delivers ASCII codes for the characters it reads. Use your imagination.
- Enjoy OOP.

Appendix A

PORTING TO OTHER PASCALS

The code examples throughout the book were written and compiled using Symantec's THINK Pascal 3.0, so you'll usually need to make a few changes if you want to compile the code under Apple's Macintosh Programmer's Workshop (MPW) using either MPW Pascal or TML Pascal. And if you have version 2.0 of THINK, you might need to change a few things.

The Pascals Available

The choices for doing object-oriented programming in Pascal on the Macintosh are, at this writing, limited to Apple's MPW Pascal, TML Pascal (no versions later than version 3.1—discontinued in 1990), and Symantec's THINK (formerly Lightspeed) Pascal. MPW Pascal has been object oriented all along. Both TML and THINK became object oriented with their second releases.

Both MPW Pascal and TML Pascal run under Apple's Macintosh Programmer's Workshop. If either is your Pascal of choice, you'll need to be familiar with your compiler's option switches, compiler directives, and segmentation techniques. In addition, if you're going to develop real Macintosh programs, you'll need to be able to work with other tools, such as Apple's Resource Editor, ResEdit, and one of the resource compilers available: RMaker or Rez. You'll also need to use the linker once you've compiled your code into object modules.

THINK Pascal uses its own development environment, which—although less flexible and extensible than MPW—features fast turnaround and easy project management with transparent linking, easy segmentation, and a standard Mac user interface. (MPW programmers work mostly with a command-line interface embedded in a Mac interface. That command line will be familiar to Unix users.) In THINK Pascal, you'll still need to use ResEdit and a resource compiler such as RMaker or a standalone version of Rez called SARez, which comes with THINK, but you won't have to deal with a separate linking process.

On the whole, THINK is the better environment for beginners, and MPW for full-scale commercial software development. A great many developers use both—THINK for the rapid prototyping its quick turnaround promotes, and MPW for the finished product. Porting code between the two environments is not exactly simple, but it is nevertheless straightforward. The main areas of incompatibility involve different segmentation techniques and different compiler directives. All three Pascals support an identical Object Pascal syntax and semantics.

At this writing, both MPW Pascal and TML Pascal will compile Apple's MacApp application framework—a collection of units containing Object Pascal classes. THINK supports MacApp as of version 3, although you have to make a few adjustments to the MacApp source files.

Porting to THINK Pascal 2.0

Version 2 was the earliest version of THINK Pascal to support object-oriented programming and is the earliest version of THINK Pascal you can port this book's programs to. Areas that might require changes include

- Compiler directives and segmentation
- Set constants
- L-value function calls

Compiler Directives That Might Need Changing

Some of the code in the book uses the compiler directive *$S*, a segmentation directive like MPW's. This directive was added to THINK Pascal with version 3.

Where to Get Them

MPW, MPW Pascal, and MacApp are available—each as a separate product—from the Apple Programmers and Developers Association (APDA):

Apple Computer, Incorporated
20525 Mariani Avenue
Cupertino, CA 95014-6299.

APDA membership is (currently) $20 per year, and anyone can join. You get access to a catalog of Apple and third-party Mac and Apple IIGS products and documentation and eligibility to sign up for one of several developer support plans (at extra cost).

THINK Pascal is available through APDA, Apple dealers, software retailers, and mail-order houses.

Versions of THINK Pascal earlier than version 3 allowed you to segment a project only file by file, by means of the segmentation view in the project window. Version 3 also provides the *$S* directive, which, like MPW's *$S*, lets you put individual procedures and method bodies in the same file into different segments. See the documentation for THINK Pascal 3.

We used *$S* mainly with PicoApp. To eliminate it, you might have to do some serious revision of the code. For instance, we used *$S* to put an *UnloadSegments* method into the main segment because you can't call *UnloadSeg* on a segment from the segment itself. You might have to put the whole file containing the *UnloadSegments* method into the main segment.

Set Constants That Might Need Changing

Somewhere in the book's code, you might find uses of the set constants that are allowed as of version 3. For example, you can declare a constant of a set type:

```
const
  letters = ['a'..'z', 'A'..'Z'];
```

That wasn't allowed in versions of THINK Pascal earlier than version 3. In the unlikely event that you find a set constant, change it to a variable and initialize it properly.

L-Value Function Calls That Might Need Changing

As of version 3 of THINK Pascal, you can use function and method calls in "l-value" contexts:

- On the left side of assignment statements

```
HeadOf^.next := nil;
```

- As arguments to *with* statements

```
with MyFunction do
```

- As method calls

```
itsDoc := TWindow(GetWRefCon(FrontWindow)).DocumentOf;
```

See the documentation for THINK Pascal 3. I've tried to use this feature only sparingly in the book's code, but you will certainly encounter it, particularly in the PicoApp code.

To replace function and method l-values, use auxiliary variables to get the l-value items on the right-hand side of the assignments:

```
head := HeadOf;
head^.next := nil;
x := MyFunction;
with x do
  ⋮
```

```
theRefCon := GetWRefCon(FrontWindow);
winObj := TWindow(theRefCon);
itsDoc := winObj.DocumentOf;
```

Other New Features of THINK Pascal 3.0 Not Used

I didn't use any of the other new features of THINK Pascal 3 (except the formerly undocumented *univ* qualifier for parameters): other compiler directives, new packed types, new Toolbox interfaces, segments greater than 64KB, indexing of string constants, qualified *sizeof* function calls, typecasting to a different size (except as allowed with polymorphic objects), new predefined functions.

Porting to MPW Pascal

As of version 3, THINK Pascal has become quite compatible with Apple's MPW Pascal, but, as always, there are some differences between the two Pascals. The main areas of difference to watch out for are

- Compiler directives
- Initialization of the Macintosh Toolbox managers
- Standard units and *uses* clauses
- I/O

and

- Comments
- Packing of files, records, and arrays
- Some predefined functions

You'll definitely have to consider the items in the first group. You might have to consider the items in the second sometime, but these areas won't be a problem for you in porting the book's code to MPW.

Compiler Directives

MPW Pascal and THINK Pascal don't agree entirely when it comes to compiler directives. THINK now supports the same *$S* segmentation directive as MPW. Other directives either aren't supported in both environments or have different meanings in MPW and THINK. The *$I* directive, which means "include" in MPW and "initialize" in THINK, is a good example.

THINK's documentation recommends that you consider all compiler directives to be nonportable and either remove or change them before you port your THINK code.

The main compiler directive we've used that might cause problems is the *$I* directive, which we'll talk about. In general, we've used very few compiler directives.

Toolbox Initialization

By default, THINK Pascal initializes the Macintosh Toolbox managers for us by making these calls:

```
InitGraf(@thePort);
InitFonts;
InitWindows;
InitMenus;
TEInit;
InitDialogs(nil);
SetApplLimit(current value of A7 – Run Options stack size);
MaxApplZone;
for i := 1 to 10 do
  MoreMasters;
```

You can make the calls yourself if you precede them with the *{$I-}* compiler directive that turns off THINK's automatic initialization. To make our code more portable, we've done just that with PicoApp (in our PicoApplications Crapgame and PicoSketch). In the main program file of any PicoApplication, a *{$I-}* directive and a call to the *InitTheMac* procedure appear. To port the code to MPW Pascal, just be sure to remove the *$I-* directive.

Our non-PicoApp code doesn't use *$I-*, so you'll have to add specific initialization code yourself. You can imitate the code in the *InitTheMac* procedure, found in file UPARoutines.p on the code disk.

Uses Clauses and Standard Units

Most MPW Pascal programs begin with this *uses* clause:

```
uses
  MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
```

THINK predefines most of the declarations in those units, so the book's *uses* clauses omit those units. You'll need to set up an MPW *uses* clause. For MPW compatibility, THINK does provide a set of unit interfaces for *MemTypes*, *QuickDraw*, *OSIntf*, *ToolIntf*, and *PackIntf*, but we've prepared the book's projects in THINK style rather than MPW style, so you will have to make some changes:

- Add these files to your MPW project: MemTypes.p, QuickDraw.p, OSIntf.p, ToolIntf.p, and PackIntf.p.
- Where the book's THINK projects include the THINK libraries Runtime.lib and Interface.lib (which MPW lacks), the files you've just added to your MPW project provide the same imports, so drop Runtime.lib and Interface.lib.
- Add the following units to the book's *uses* clauses: *MemTypes*, *QuickDraw*, *OSIntf*, *ToolIntf*, and *PackIntf*. Use these in addition to your own units already listed in your *uses* clauses.

Input/Output

None of the book's code relies on any odd features of I/O, with one exception: Some of the book's example programs use THINK's Text and Drawing windows. Look for calls to *ShowText*, *ShowDrawing*, *SetTextRect*, *SetDrawingRect*, *GetTextRect*, and *GetDrawingRect*. You'll need to provide replacements for these handy built-in windows. But setting up a window for the code to draw into is not terribly difficult. Most of the examples don't require scrolling or other fancy features.

Comments

THINK and MPW recognize both { } and (* *) as comment delimiters. MPW treats the two delimiters differently to allow nested comments. THINK doesn't distinguish between the two delimiters, so nested comments result in an error. THINK requires that a comment within a single set of comment delimiters not run over multiple lines. MPW allows multi-line comments, so anticipate no problem with the book code's THINK-style comments.

Packing Files, Records, and Arrays

The MPW and THINK compilers use different packing methods, which could create difficulties in porting. We've used no packing in the book's code at all except for a packed array of *Char* in the fourth project in Chapter 21, and that should be portable.

Predefined Functions

We've avoided nonstandard and nonportable functions and procedures in the book's code. You should have no trouble on this score.

Porting to Other Pascals

The four top Macintosh Pascal compilers are THINK, MPW, TML (discontinued in 1990), and Turbo. Any advice on porting from THINK to MPW also applies to porting from THINK to TML.

Turbo Pascal for the Macintosh doesn't support objects at this writing, so you can't port the book's code to Macintosh Turbo. If Borland ever does introduce objects into its Macintosh Turbo, it's quite possible that its Mac version of Object Pascal will resemble its MS-DOS version rather than Apple's Object Pascal. Turbo Pascal 5.5/6.0 for MS-DOS borrows many features from C++ and is in many ways unlike Apple's specification for Object Pascal. A good deal of our code might not be portable to a future Borland Macintosh Object Pascal.

Similarly, the current Pascal '9x project at Apple might, in a year or two, lead to a new Pascal++. This successor to today's Object Pascal will retain a Pascal syntax but incorporate many C++ features. Most likely, porting from Object Pascal to Pascal++ will be a big job.

Other Languages

Our ideas, if not our code, might be adaptable, not only to other Object Pascals, but also to other languages. Modula-2, sometimes called the "Son of Pascal," is similar to standard Pascal in many ways, and at least one object-oriented version of Modula-2 has been developed. Called p1 Modula-2, developed by Mindware (formerly Gesellschaft für Informatik G.m.b.H.) in Germany, and distributed by the MacApp Developers Association, this Modula-2 implements objects compatible with MPW Pascal. You can also program with p1 Modula-2 and MacApp. Porting our code to p1 should resemble porting to MPW Pascal, with some changes from Pascal to Modula-2 syntax: keywords in all caps, explicit END delimiters for IF statements, WHILE loops, and so on.

Symantec's THINK C implements objects in a way that seems to be modeled more on Object Pascal than on standard C++. THINK C supports both MacApp and its own C version of the THINK Class Library (TCL). Even so, translating our code to THINK C might be a pretty complex operation.

There are, of course, numerous other object-oriented languages: LISP, Objective-C, Smalltalk, Prograph, C++, and more. Ideas and components similar to ours should be implementable in all of those languages. Porting might not be feasible.

Pascals for Other Computers

Naturally, many parts of the book's code are Macintosh specific, relying on the Mac environment, its Toolbox calls, and its way of doing business with windows, menus, icons, and the like. Those parts of the code, of course, are not portable to other computers. Many of the book's ideas, however, could well be implemented on other computers: Turbo Pascal or Microsoft QuickPascal under MS-DOS could be used to create a DOS-style imitation of PicoApp. (Turbo already has an application framework, called Turbo Vision.) And much of the *TList* and other non-PicoApp code could be translated into one of those MS-DOS Object Pascal variants.

So far, the two main Object Pascals for non-Mac programming are Borland's Turbo Pascal 5.5/6.0 and Microsoft's QuickPascal. QuickPascal's syntax is much more like Apple Object Pascal's than it is like Turbo Pascal's C++-like syntax. Except for the differences in environments, much of our code should be acceptable syntactically under QuickPascal.

At this point, no other non-Mac Pascal is object oriented.

Appendix B

BOOKS AND ARTICLES ABOUT OOP

Object-oriented programming is a big subject, and a lot has been written about it already. The books and articles in this bibliography helped make my transition to OOP and to the idea of an application framework a little easier. They might help you, too.

I was influenced early on, pre–Object Pascal and practically pre-objects, by Grady Booch's *Software Components with Ada,* and later, of course, by his *Object-Oriented Design: With Applications.*

I found Brad Cox's *Object-Oriented Programming: An Evolutionary Approach,* about Objective-C, to be an especially good treatment of the idea of "consumers" of "software ICs." Bertrand Meyer's *Object-Oriented Software Construction,* about Eiffel, is another standout, with an excellent discussion of the relative merits of inheritance and other techniques for achieving similar effects.

Some object-language vendors have had the wit to engage the services of talented writers, and the manuals for their object-oriented language products are especially noteworthy additions to the literature on OOP. Philip Borenstein and Jeff Matson's *THINK Pascal: Object-Oriented Programming Manual* was especially useful. Jeff Duntemann's *Object-Oriented Programming Guide* for Borland International's Turbo Pascal 5.5/6.0 (for MS-DOS) accompanies a Pascal closer to C++ than to the Apple Object Pascal model, but Duntemann's elucidation of principles is a great service to the OOP neophyte. Robert Keller wrote the very fine manual for Turbo Vision, Borland's MS-DOS application framework for Turbo Pascal 6.0.

Not surprisingly, the literature on application frameworks is small. Before I came across Keller's manual for Borland, Apple furnished me with the Apple Programmers

and Developers Association's beta draft of their *Introduction to MacApp 2.0 and Object-Oriented Programming.* I went from that brief overview of Object Pascal, OOP, and MacApp to the best book yet on MacApp, *Programming with MacApp* by David A. Wilson, Larry S. Rosenstein, and Dan Shafer. But my first real introduction to the whole subject was Kurt Schmucker's *Object-Oriented Programming for the Macintosh,* which covers MacApp and includes a chapter on Object Pascal. Both the Keller manual and the Wilson, Rosenstein, and Shafer book provide particularly helpful explanations of streams.

A great many magazines run OOP articles. Among the most fruitful to read regularly are *BYTE, Dr. Dobb's Journal, Computer Language, Programmer's Journal, PC Techniques, Journal of Object-Oriented Programming,* and *Journal of Pascal, Ada, and Modula-2.* Apple-specific magazines for programmers include Apple's own *Develop*; *MacTutor,* the most widely read Mac programming journal; and *Frameworks,* the excellent journal of the MacApp Developers' Association. *MacTech Journal* (formerly *MacTech Quarterly*) stopped publication in March 1991 but published a number of good OOP articles before it did, notably those by Howard Katz.

Many of the titles in the list below treat other OOP languages or OOP in general. The entries for titles that treat Object Pascal specifically are marked with asterisks. To move quickly to all the entries that deal with a major OOP language or version of a language, see the entry under that language or version (Actor, C++, MPW Pascal, Objective-C, QuickPascal, Smalltalk, THINK Pascal, Turbo Pascal).

Abbott, R. J. 1983. "Program Design by Informal English Descriptions." *CACM,* November, 882. The application design method featured in Chapter 10 grew out of Abbott's article with an assist from Grady Booch and William Lorenson.

Actor. See Duff, Franz.

Apple Computer, Incorporated. 1984–. *Inside Macintosh,* Volumes I–V. Reading, Mass.: Addison-Wesley. The standard reference for Macintosh developers.

————. 1989. *Introduction to MacApp 2.0 and Object-Oriented Programming.* Beta Draft, APDA# M0300LL/A. Cupertino, Calif.: Apple Computer. A brief overview of Object Pascal, OOP, and MacApp.

————. 1990. *ResEdit Reference.* Reading, Mass.: Addison-Wesley. The official documentation for Apple's Resource Editor application, version 2.0b2. Consider it a "beta" document—it documents a beta product. Useful for all Mac programming, including PicoApp programming.

Ayers, Kenneth E. 1989. "An Object-Oriented Logic Simulator." *Dr. Dobb's Journal,* December, 72. An OOP example in Smalltalk.

Bailey, Stephen C. 1989. "Designing with Objects." *Computer Language,* January, 34.

Berry, John. 1988. *The Waite Group's C++ Programming.* Indianapolis: Howard W. Sams. An introduction to C++ for C programmers.

Bianchi, Curt. 1989. "Memory Management with MacApp." *Dr. Dobb's Macintosh Journal,* Fall, 23. An extremely helpful article on MacApp memory management that influenced the development of PicoApp. Echoes MacApp documentation on memory management (or vice versa).

Booch, Grady. 1987. *Software Components with Ada: Structures, Tools, and Subsystems.* Menlo Park, Calif.: Benjamin/Cummings. My introduction to the idea of objects, before Object Pascal. Technically, Ada doesn't support objects, but you can achieve a considerably object-oriented style with Ada's packages and generics.

*———. 1991. *Object-Oriented Design: With Applications.* Redwood City, Calif.: Benjamin/Cummings. A solid software engineering approach to application design. Real developers should read it. Has one fair-sized MacApp-based example. in Object Pascal.

*Borenstein, Philip, and Jeff Mattson. 1990. *THINK Pascal: Object-Oriented Programming Manual.* Cupertino, Calif.: Symantec Corporation. The THINK guide to using the THINK Class Library, with some discussion of object-oriented programming.

Bulman, David M. 1989a. "An Object-Based Development Model." *Computer Language,* August, 49. Bulman's ideas about OOD are considerably more rigorous than those I presented in Chapter 10. Especially good reading for more serious developers.

———. 1989b. "Objects Don't Replace Design." *Computer Language,* August, 151. Bulman says that the object should replace the function as the principal unit of modularity in software design. In Chapter 8, I argue for the object hierarchy instead.

BYTE Staff. 1989. "Object-Oriented Resources." *BYTE,* March, 270. Languages, databases, and related products.

C++. See Berry, Dlugosz, Gibbons, Goodwin, King, Koenig, Langowski, Miller, Mullin, Stevens, Treister, West, K. Williams.

Chernicoff, Stephen. 1985a. *Macintosh Revealed: Volume 1, Unlocking the Toolbox.* Hasbrouck Heights, N.J.: Hayden. The first of four volumes. Volume 1 presents the key Macintosh programming concepts: memory, ports, QuickDraw, resources, code segments, fonts. The series is an excellent introduction to programming the Mac for Pascal programmers.

———. 1985b. *Macintosh Revealed: Volume 2, Programming with the Toolbox.* Hasbrouck Heights, N.J.: Hayden. The second of four volumes. Volume 2 focuses on programming the Macintosh user interface: events, windows, menus, TextEdit, controls, dialog boxes, files.

———. 1989. *Macintosh Revealed: Volume 3, Mastering the Toolbox.* Hasbrouck Heights, N.J.: Hayden. The third of four volumes. Volume 3 shows how to customize QuickDraw, windows, controls, and menus; how to write device drivers; how to implement printing; how to generate and play sounds; and how to write desk accessories. Unlike Volumes 1 and 2, Volume 3 covers advanced topics.

———. 1990. *Macintosh Revealed: Volume 4, Expanding the Toolbox.* Hasbrouck Heights, N.J.: Hayden. The fourth of four volumes. Volume 4 covers features of the Toolbox added with the Mac SE and the Mac II: MultiFinder, color pixels and ports, color windows, styled TextEdit.

Constantine, Larry L. 1990. "Objects, Functions, and Program Extensibility." *Computer Language,* January, 34. Constantine's thesis is that OOP doesn't necessarily replace structured programming, and his article shows how to mix the two.

Cox, Brad. 1986a. *Object Oriented Programming: An Evolutionary Approach.* Reading, Mass.: Addison-Wesley. Required reading for a thorough OOP background. Cox's focus is pretty theoretical, and the book is about Objective-C, not Object Pascal, but Cox's ideas are the most advanced on software components, managing OOP libraries, documenting classes, and training object-oriented software engineers.

Cox, Brad, and Bill Hunt. 1986b. "Objects, Icons, and Software-ICs." *BYTE,* August, 161. Technical, the article is a concise explanation of the software-IC concept.

*Curtis, David B. 1988. "Menus as Objects in TML Pascal." *MacTutor,* June, 85. This is the one TML-specific article I've seen—on a very inventive and useful example. TML Systems has dropped its Pascal product.

Dawson, Joseph. 1989. "A Family of Models." *BYTE,* September, 277. Object-oriented databases vs. relational databases.

Dlugosz, John M. 1989a. "Better C++ Code Generation." *Programmer's Journal,* September/October, 56. Improving the code generation of current C++ compilers (on IBM PCs).

———. 1989b. "Object-oriented Programming for C." *Computer Language,* August, 103. Three OOP C hybrids: C_Talk, Complete C, and C+O.

———. 1989c. "The Secret of Reference Variables." *Computer Language,* August, 83. About reference variables in C++.

Dodani, Mahesh, Charles E. Hughes, and J. Michael Moshell. 1989. "Separation of Powers." *BYTE,* March, 255. Objects and user interfaces in Smalltalk. This article is the origin of Chapter 17's discussion of the model-view-controller (MVC) approach to software architecture.

Duff, Charles B. 1986. "Designing an Efficient Language." *BYTE,* August, 211. The design philosophy of the creator of the Actor language.

*Duntemann, Jeff. 1988. *Turbo Pascal: Object-Oriented Programming Guide.* Scotts Valley, Calif.: Borland International. Despite major differences between Turbo and Apple brands of Object Pascal, an excellent introduction to using objects. It gave me quite a boost.

*——— 1989a. "Admitting Objects to Pascal." *Dr. Dobb's Journal,* September, 128. A comparison of Turbo Pascal's and Microsoft QuickPascal's objects.

*——— 1989b. "Dodging Steamships." *Dr. Dobb's Journal,* July, 128. An early look at Turbo Pascal 5.5 with objects.

Dye, Rob. 1989. "Visual Object-Oriented Programming." *Dr. Dobb's Macintosh Journal,* Fall, 30. Writing OOP graphically with LabView.

*Ezell, Ben. 1990. *Object-Oriented Programming in Turbo Pascal 5.5.* Reading, Mass.: Addison-Wesley.

Fernhout, Paul D. 1989. "Simulating Interacting Intelligent Objects in C." *AI Expert,* January, 38. Doing OOP in plain old ANSI C. See also Jeffery.

*Fleener, Kevin. 1989. "Object-oriented Pascals: The Next Generation." *Computer Language,* August, 91. Turbo Pascal and QuickPascal for MS-DOS.

*Floyd, Michael. 1989. "Turbo Pascal with Objects." *Dr. Dobb's Journal,* July, 56. Turbo Pascal 5.5 reaches the marketplace.

Franz, Marty. 1989. "Writing Filters in an Object-Oriented Language." *Dr. Dobb's Journal,* December, 28. An example in Actor.

Gabriel, Richard P. 1989. "Using the Common LISP Object System." *Computer Language,* August, 73. Objects with Common LISP. See also Guillon, Lange, Shalit.

Gibbons, Bill. 1989. "MPW Extensions to C++." *APDALog,* Fall, 10. Apple's C++.

Glassey, C. Roger, and Sadashiv Adiga. 1989. "Conceptual Design of a Software Object Library for Simulation of Semiconductor Manufacturing Systems." *Journal of Object-Oriented Programming,* November/December, 39. Building a library of reusable objects for research in real-time resource allocation.

*Goldsmith, David. 1987. "A MacApp Text Editor." *MacTutor,* February, 43.

Goodwin, Mark. 1989. *User Interfaces in C++ and Object-Oriented Programming.* Portland, Oreg.: Management Information Source Inc. Very little OOP discussion, but does develop a full windowing system for IBM PCs using C++. Excellent information on PC video services.

Greenberg, Ross M. 1989. "Faces of Unix." *PC Magazine,* September 12, 143. Graphical user interfaces for Unix.

Guillon, Didier. 1988. "LISP Objects: Ford vs. Chevy." *MacTutor,* April, 92. Objects with Common LISP. See also Gabriel, Lange, Shalit.

Guthery, Scott. 1989. "Are the Emperor's New Clothes Object Oriented?" *Dr. Dobb's Journal,* December, 80. Extremely provocative essay about OOP limitations—in particular, the difficulties of using OOP in multi-programmer teams. Read!

Hayes, Frank, and Nick Baran. 1989. "A Guide to GUIs." *BYTE,* July, 250. Overview of 12 of the top graphical user interfaces, including the Mac's and Microsoft Windows.

Jeffery, David. 1989. "Object-Oriented Programming in ANSI C." *Computer Language,* February, 77. See also Fernhout.

Jensen, Kathleen, and Niklaus Wirth. 1984. *Pascal Report.* 3d ed. New York: Springer-Verlag.

Kaehler, Ted, and Dave Patterson. 1986. "A Small Taste of Smalltalk." *BYTE,* August, 145.

*Katz, Howard. 1987. "How to Think in MacApp." *MacTutor,* August, 59. Mostly an introduction to Object Pascal.

*———. 1989a. "Inheritance and Subclassing." *MacTech Quarterly,* Autumn, 22. A brief introduction to Object Pascal concepts.

*———. 1989b. "Object Oriented Programming for the Masses." *MacTech Quarterly,* Spring, 28. An example game program in Object Pascal.

———. 1989c. "OOP Notes: Something Special." *APDALog,* Fall, 13. A description of "MicroTV,™" a MacApp success story.

———. 1990. "The Think C Library." *MacTech Quarterly,* Winter, 102. The THINK Pascal–like object extensions to THINK C and the library of MacApp-style classes (TCL for C).

*Keller, Robert. 1990. *Turbo Vision Guide.* Scotts Valley, Calif.: Borland International. The best explanation so far of the preeminent OOP application framework for MS-DOS. Comes with Turbo Pascal 6.0. From my perspective, it contains good alternative ideas on application architecture and lots of implementation ideas that could be adapted for the Mac.

*Kienle, Steven. 1989. "Network Graphs in Object Pascal." *Dr. Dobb's Journal,* December, 17. An example in MPW Pascal.

King, Todd. 1989. "Force-Based Simulations." *Dr. Dobb's Journal,* September, 40. OOP and simulation in C++.

Knaster, Scott. 1986. *How to Write Macintosh Software.* Hasbrouck Heights, N.J.: Hayden. The best source of information on Mac debugging and on what's in the Mac's memory.

Koenig, Andrew. 1989. "How Virtual Functions Work." *Journal of Object-Oriented Programming,* January/February, 73. Virtual functions in C++—they were also adopted in Turbo Pascal 5.5.

Ladd, Scott R. 1990. "The OOP Elite." *Computer Language,* January, 109. COBOL and FORTRAN and OOP.

———. 1989a. "Chatting About Smalltalk." *Computer Language,* December, 109. Smalltalk as the best language to use in learning OOP.

———. 1989b. "Performance Issues." *Computer Language,* August, 125. What's the truth about OOP performance?

———. 1989c. "Taking the Plunge or Testing the Waters." *Computer Language,* June, 91. The future of OOP.

———. 1989d. "Super Paradigm." *Computer Language,* February, 103. OOP and structured programming.

*Lane, Alex. 1990. *Object-Oriented Turbo Pascal.* Redwood City, Calif.: M&T Publishing. For MS-DOS.

Lange, Jean Pascal J. 1989. "OOP in LISP." *MacTutor,* December, 41. See Gabriel, Guillon, Shalit.

Langowski, Jörg. 1989a. "Apple's C++." *MacTutor,* November, 60.

———. 1989b. "C++ Overview." *MacTutor,* September, 42.

———. 1988. "Object Forth Project." *MacTutor,* August, 66. Objects in Forth. See also Pountain, Snively 1987.

Leonard, Randy. 1989. "OOP: The Future for Macintosh Development." *MacTech Quarterly,* Spring, 22. Expect OOP to be the way things are done on the Mac.

Lorenson, William. 1986. "Object-Oriented Design." *CRD Software Engineering Guidelines,* General Electric Co. Contributed to Chapter 10's OOD discussion.

*McMath, Chuck, and Carl Nelson. 1989. "A Tale of Two Quadratic Plotters." *MacTutor,* August, 56. Shows how to redo (in MacApp) a quadratic equation plotting program originally written in Pascal—first by straightforward translation and then by a complete OOP redesign in MacApp. See Nelson and McMath for Part II of this two-part series.

Meyer, Bertrand. 1988. *Object-Oriented Software Construction.* New York: Prentice Hall. With the Eiffel language. An excellent treatment of many OOP topics and one of the influential books in the industry. Eiffel is an interesting language designed for serious software engineering.

———. 1989. "Writing Correct Software with Eiffel." *Dr. Dobb's Journal,* December, 48. Eiffel's assertion feature, based on the concept of proving program correctness.

Miller, William M. 1989. "Multiple Inheritance in C++." *Computer Language,* August, 63.

Mouhanna, Joseph, and Michael Vose. 1989. "The QuickPascal in QuickPascal." *Dr. Dobb's Journal,* December, 64. Microsoft used QuickPascal to write its own user interface. (Object Pascal for MS-DOS.)

MPW Pascal. See Kienle, West.

Mullin, Mark. 1989. *Object Oriented Program Design: With Examples in C++.* Reading, Mass.: Addison-Wesley. Design of one large business application using C++. So focused on the application that the OOP discussion gets lost.

*Muys-Vasovic, Jean-Denis. 1989. "Back to the Future: OOP and MacApp." *MacTutor,* September, 102.

*Nelson, Carl, and Chuck McMath. 1989. "A Tale of Two Quadratic Plotters, Part II." *MacTutor,* September, 18. This article completes a two-part series on MacApp. See also McMath and Nelson.

Objective-C. See Cox, Pries, Webster.

Object Pascal. See Apple, Booch, Duntemann, Katz, Keller, Kienle, Mouhanna, Schmucker, Shammas, Tesler, West, Wilson.

Pascoe, Geoffrey A. 1986. "Elements of Object-Oriented Programming." *BYTE,* August, 139. The elements that make a language object-oriented.

Petzold, Charles, Luisa Simone, and Tami D. Peterson. 1989. "GUIs for DOS and OS/2." *PC Magazine,* September 12, 111. Microsoft Windows, DeskMate, GEM, NewWave, Presentation Manager, and other graphical user interfaces.

Plauger, P. J. 1989. "Encapsulate It." *Computer Language,* December, 17. Encapsulation in structured programming.

———. 1990. "Inherit It." *Computer Language,* January, 17. Inheritance in structured programming.

Pountain, Dick. 1986. "Object-Oriented Forth." *BYTE,* August, 227. Objects in the Forth language. See also Langowski 1988, Snively 1987.

Pressman, Roger S. 1987. *Software Engineering: A Practitioner's Approach,* 2d ed. New York: McGraw-Hill. A standard text on software engineering, with a chapter on object-oriented design.

Pries, Jerri. 1989. "An Exercise in Object-Oriented Programming." *Journal of Object-Oriented Programming,* January/February, 77. Designing data structures in Objective-C. I found this article helpful as I wrote the data structure chapters of Part 3. Pries helped me get a grip on what OOP can do.

QuickPascal. See Duntemann, Ezell, Fleener, Floyd, Lane, Shammas, Udell.

Rasmus, Daniel. 1989. "Understanding the Objects in Nexpert Object." *MacTech Quarterly,* Autumn, 102. About an OOP expert system shell.

Rose, Philip F. H. 1989. "The Macintosh Finder: Pure GUI." *PC Magazine,* September 12, 133. The Mac's graphical user interface from a DOS perspective.

Rovira, Charles-A. 1989. "Persistent Objects." *Dr. Dobb's Macintosh Journal,* Fall, 41. Rovira's persistent objects are basically files consisting of collections of bytes and read as streams of data. See Wilson 1990.

———. 1990. "Rovira Diagrams." *Computer Language,* January, 59. A diagramming method for planning OOP user interface routines (dialog boxes).

Sando, Steve. 1989. "From C to Objects." *Programmer's Journal,* September/October, 44. Using classes to bridge the gap between C and OOP.

*Schmucker, Kurt J. 1986a. *Object-Oriented Programming for the Macintosh.* Hasbrouck Heights, N.J.: Hayden. Includes a chapter on Object Pascal. And see the second edition, 1989.

*———. 1986b. "MacApp: An Application Framework." *BYTE,* August, 189. An introduction to Apple's MacApp, which uses objects to provide a built-in, ready-to-extend, generic Macintosh application.

*———. 1986c. "Object-Oriented Languages for the Macintosh." *BYTE,* August, 177. A survey that includes Object Pascal.

Seymour, Jim. 1989. "The GUI: An Interface You Won't Outgrow." *PC Magazine,* September 12, 97. A review of 15 graphical user interfaces for the PC, the Mac, the Amiga, Unix, and other platforms.

Shalit, Andrew. 1986. "Graphics Objects in Scheme." *MacTutor,* November, 69. Objects with MacScheme, a Macintosh LISP. See also Gabriel, Guillon, Lange.

*Shammas, Namir. 1990. *Object-Oriented Programming with QuickPascal.* New York: Wiley. Microsoft QuickPascal is very Apple-like in its Object Pascal features, very Turbo-like in the rest of its Pascal implementation (for MS-DOS).

Smalltalk. See Ayers, Kaehler, Ladd.

*Smith, David E. 1989. "Introduction to MacApp." *MacTutor,* August, 10. A good overview. MacApp is terribly complex and difficult to learn to use. But it's so powerful that once mastered, it can cut development time tremendously.

Snively, Paul. 1987. "ForthTalk Brings Objects." *MacTutor,* February, 27. Objects in Forth. See also Langowski 1988, Pountain.

———. 1989. "NeXT Evolution." *MacTutor,* July, 25. Overview of programming objects on the NeXT computer.

Stevens, Al. 1990. *Teach Yourself C++.* Portland, Oreg.: MIS Press. An introduction to C++ for C programmers.

———. 1989. "C++: Of Books, Compilers, and a Window Object." *Dr. Dobb's Journal,* September, 121. Pop-up windows and menus in C++.

Swaine, Michael. 1989. "Is Multiple Inheritance Necessary?" *Dr. Dobb's Journal,* March, 107. One of the more thoughtful pieces I encountered in the midst of all the OOP cheerleading.

*Symantec. 1988a. *THINK Pascal: User Manual.* Cupertino, Calif.: Symantec Corporation. The manual for THINK Pascal 3.0.

*———. 1988b. *THINK Pascal: Object-Oriented Programming Manual.* See Borenstein and Mattson.

*———. 1988c. *THINK Pascal: Resource Utilities Manual.* Cupertino, Calif.: Symantec Corporation. THINK's manual for resource editing, including good information on using ResEdit.

Tesler, Larry. 1985. "Object Pascal Report." First published in *Structured Language World* 9 (February 14), Apple Technical Report No. 1. Amends Jensen & Wirth's *Pascal Report* to add object-oriented features to Pascal. Object Pascal and Tesler's report were endorsed by Niklaus Wirth, the designer of Pascal, who consulted on the project.

———. 1986. "Programming Experiences." *BYTE,* August, 195. Experiences of programmers using OOP.

THINK Pascal. See Borenstein and Mattson, Symantec.

**THINK Pascal: Object-Oriented Programming Manual.* See Borenstein and Mattson.

Thomas, Dave. 1989. "What's in an Object?" *BYTE,* March, 231. Extending applications.

Thompson, Tom. 1989. "The Next Step." *BYTE,* March, 265. Software development with objects on the NeXT computer.

Treister, Adam. 1989. "Pseudo Objects." *MacTutor,* August, 38. Doing OOP in standard C as a preparation for the coming of C++.

Turbo Pascal. See Duntemann, Ezell, Fleener, Floyd, Keller, Lane, Shammas, Udell.

**Turbo Pascal: Object-Oriented Programming Guide.* See Duntemann.

Turbo Vision Guide. See Keller.

*Udell, John. 1989. "Clash of the Object-Oriented Pascals." *BYTE,* July, 104. On the introduction of Borland's Turbo Pascal 5.5 and Microsoft's QuickPascal 1.0.

Vernon, Vaughn. 1989. "The Forest for the Trees." *Programmer's Journal,* September/October, 39. On designing class libraries. A very good article that helped me with Chapter 8's discussion of hierarchies.

Waters, Bryan. 1989a. "Object C and the Macintosh Control Panel." *Dr. Dobb's Macintosh Journal,* Fall, 50. Using THINK C to write CDEVs.

———. 1989b. "Writing Macintosh Device Drivers." *Dr. Dobb's Macintosh Journal,* Fall, 36. THINK C objects used to write a device driver template.

Webster, Bruce. 1989. *The NeXT Book.* Reading, Mass.: Addison-Wesley. Covers the NextStep operating system and Objective-C. Especially interesting in its view of object-oriented user interfaces.

Wegner, Peter. 1989. "Learning the Language." *BYTE,* March, 245. OOP in the 1990s.

West, Joel. 1987a. *Programming with Macintosh Programmer's Workshop.* New York: Bantam Books. Excellent guide to using Apple's MPW, including MPW Pascal. Chapter on Object Pascal and MacApp. Covers version 2.0 of MPW.

———. 1989b. "Mastering MPW: OOP(s) and C++." *APDALog,* Fall, 8.

Williams, Gregg. 1984. "Software Frameworks." *BYTE,* December, 124. The general concept, of which MacApp is a leading example.

Williams, Kent. 1989. "Tables Within Tables." *Computer Language,* August, 36. Brent's hashing algorithm for tables as a C++ reusable class.

*Wilson, David A., Larry S. Rosenstein, and Dan Shafer. 1990. *Programming with MacApp.* Reading, Mass.: Addison-Wesley. Part of Addison-Wesley's new *Macintosh Inside Out* series. Good but brief Object Pascal coverage, good coverage of MacApp, and an excellent Chapter 22 on streams that influenced me a great deal.

Wirth, Niklaus. 1975. *Algorithms Plus Data Structures Equals Programs.* New York: Prentice-Hall.

Yourdon, Edward. 1989. "The Year of the Object." *Computer Language,* August, 119. Selling the OOP concept to managers. Last but not least.

Appendix C

GLOSSARY

In addition to defining terms, this glossary includes a few useful organizations and publications of interest to OOP programmers on the Mac.

abstract class A *class* from which you will normally not make any *instances*. Such a class appears high in the object *hierarchy,* primarily to pass down facilities and capabilities to its *descendant* classes. *TObject* is the best-known example.

abstract data type (ADT) A data type plus the permitted operations on that type. Type *Integer*, for example, allows operations such as addition, subtraction, multiplication, and division. You can declare an ADT in standard Pascal code, but an object class is a better vehicle. Any standard Pascal code or any object class whose essence is the declaration of a single type and that provides all necessary operations on that type can be considered the implementation of an abstract data type.

ancestor A *class* higher in the object *hierarchy* than a given class. In Object Pascal, class *X* can have one and only one *immediate ancestor,* but that ancestor's ancestors are *X*'s ancestors, too. A class inherits all the *instance variables* and *methods* of all its ancestors (except those methods it overrides).

APDA The Apple Programmers and Developers Association, now maintained by Apple Computer. APDA supplies tools, documentation, and information about products for Apple computers, including *APDALog,* a quarterly technical journal and catalog. Anyone can join (currently for $20 annually) by contacting APDA at

Apple Computer, Incorporated
20525 Mariani Avenue, Mail Stop 33G
Cupertino, CA 95014-6299
Phone: 1-800-282-2732 in the U.S.
AppleLink: APDA

application framework A *class library* that provides basic application functionality through a group of interrelated classes. An application framework usually provides menu and window creation and handling, event handling, memory management, and the like. To build upon the framework and create a working application, you subclass the classes provided by the application framework and add classes of your own. The best-known Macintosh application frameworks are *MacApp* and the THINK Class Library (*TCL*). This book's PicoApp is an application framework.

base class The single *class* that serves as the base in a polymorphic data structure. The most usual way to visualize the base class is paradoxically as the topmost class whose *instances* can be added to the data structure. Given an array of *TObject*, for example, you can put *TObject* instances or instances of any descendant class of *TObject* into the array. *TObject* is thus the base class.

binding The compiler's making a connection between a procedure, function, or *method* call and the code the call invokes. The connection takes the form of the code's address. Binding can occur at compile time, the normal case for ordinary procedures and functions, or at *runtime,* for methods of polymorphic *objects.*

browser A facility that lets the programmer see the structure of a program's class hierarchies. Most good OOP compilers include some kind of browser that lets the programmer move around within the *hierarchy*, examining and editing classes.

C++ A variant of the C language that includes object-oriented constructs and facilities. Apple has endorsed C++ as one of the major OOP languages for application development. *Turbo Pascal* for MS-DOS is derived as much from C++ as it is from *Object Pascal.* The Pascal++ under development at Apple will be closer to C++ than to Object Pascal.

class A kind of template for making *objects.* A class defines the *instance variables* and *methods* of the objects that will be created, or *instantiated,* from it. We can consider the class of cats, for instance, of which Tippy, Striper, and Sydney are specific *instances* or members. The class *TCat* could define instance variables for color, weight, and personality, and methods for eating, sleeping, and playing with yarn. In Object Pascal, classes are implemented as special object types, and they look much like Pascal record types.

class hierarchy. See *hierarchy.*

class library A collection of *object classes* that can be used together in programs. *MacApp* and *TCL* are examples. Also see *library.*

class link A means of locating information about a class and its methods. *Classes* are simply templates that the compiler uses when it constructs actual *object instances* in the heap, dynamically, at *runtime.* The heap block for an object contains its *instance variables,* but it can't contain all the *method* code. So the compiler puts in a class link. The program uses the class link at runtime to find the right code for a method. Also see *method dispatch table, polymorphism, runtime binding.*

data field. See *instance variable.*

descendant A *subclass* of a higher *class.* Class *B* is a descendant of class *A* if *B* has been *subclassed* from *A*—in other words, if *B* has declared *A* in its *heritage spot.* Descendants inherit *instance variables* and *methods* from their *ancestors* and can have their own descendants as well.

encapsulation A kind of information hiding. *Objects* are, or should be, encapsulated. Their *instance variables* should not be accessed directly. Instead, you should access them only by means of the object's *methods.*

framework. See *application framework.*

Frameworks. See *MADA.*

heritage spot In a class declaration, the position after the keyword *object* in which you specify the class's immediate ancestor in parentheses. The declaration of an ancestor in the class declaration's heritage spot informs the compiler of the relationship between the classes.

hierarchy The *ancestor-descendant* relationships among a group of related *classes.* Often called the *object hierarchy* or the *class hierarchy.* The form of the hierarchy is a tree of which the nodes are classes and the root node is an ancestor of all the others. In Object Pascal, each node in the hierarchy can have one and only one *immediate ancestor* node but can have many or no *descendant* nodes.

immediate ancestor The *class* declared in the *heritage spot* of a class declaration, the class from which the current class is directly descended. A class can have many *ancestors* above it in the *hierarchy,* but it can have only one immediate ancestor.

inheritance A valuable way to derive one object type from another with minimal code duplication. A *descendant class* automatically contains the *instance variables* and *methods* of its *ancestor class.* The descendant thus doesn't have to (and can't) name those items again in its own declaration—it simply adds new instance variables and methods it needs or overrides its ancestors' methods in order to distinguish itself from its ancestors. Inheritance, along with *runtime binding,* also makes *polymorphism* possible. Also see *subclassing.*

Inside Macintosh The authoritative Macintosh programming documentation, written by Apple Computer and published by Addison-Wesley. At this writing, *Inside Macintosh* consists of five volumes that document all the Macintosh *Toolbox* routines and data structures used in programming windows, menus, dialog boxes, and other standard Macintosh user interface items.

instance An actual *object,* as opposed to the *class* from which the object is *instantiated.* Emerson and Buji, very real cats, are instances of the class *TCat,* for example. Each instance is implemented as a block in the heap, with storage for its own *instance variables* and a *class link* to its *method* code and to its *immediate ancestor* class. Classes are represented as types in Object Pascal, and instances are represented as variables.

instance variable A data field, similar to a field in a Pascal record, that stores a piece of data representing some aspect of an object's state. Field declarations take the form of variable declarations, as in records, and each object *instance* has its own copies of the fields declared by its *class*: hence, "instance" variables.

instantiate To create an actual *object* from its class template. To instantiate an object, you declare a variable of the class type and then call Pascal's *New* procedure, passing the variable as the parameter. You then usually need to initialize the object *instance* by sending it an initialization *message*.

library In its broadest sense, a compiled group of procedures, functions, data types, and perhaps *classes* that can be called upon from an application by means of an interface file. Also see *class library*.

MacApp The best-known and most powerful of the object-oriented *application frameworks* for the Macintosh. MacApp is a *class library* of *classes* that can manage Mac windows, menus, events, and so forth. To learn more about MacApp, see the books and articles listed in Appendix B or contact Apple Computer through *APDA*.

MacTech A quarterly technical magazine for Macintosh programmers that stopped publication in March 1990. *MacTech* back issues contain information at all levels, on all topics, but much is geared to skilled programmers and object-oriented programmers specifically.

MacTutor A monthly technical magazine for Macintosh programmers. A subscription is currently $30 in the U.S. To order, contact

MacTutor
1250 N. Lakeview, #0
Anaheim, CA 92807
Phone: (714) 777-1255

MADA The MacApp Developers Association, which promotes MacApp programming and provides various products and services to its members, including *Frameworks,* an excellent bimonthly technical journal. Anyone can join (currently for $75 annually in the U.S.) by contacting

MacApp Developers Association
P.O. Box 23
Everett, WA 98206
Phone: (206) 252-6946
Fax: (206) 259-2154
AppleLink: MADA

message A call to a method of the same name. To cause an *object* to do something useful, you "send it a message." Messages have the general form

```
objectName.MessageName(parameters);
```

which resembles a cross between record-access dot notation and a procedure or function call. "Sending a message" and "calling a method" are more or less synonymous. Also see *method*.

method A procedure or function declared as part of an object class declaration. Methods are the means by which a class's *instances* operate on themselves and other *objects*. Syntactically, methods behave much as ordinary procedures and functions do. To invoke a method from outside an object, send the object a *message* with the same name as the method, including any parameters the method requires. To invoke a method from inside another method of the same object, send a message to *self*. You can perform almost any operations with methods that you can with ordinary procedures and functions, including recursion. (You can't pass method names as actual procedural parameters to a procedure.) Also see *message*.

method dispatch table A table that contains method addresses for various *classes*. Because of *polymorphism*, the compiler must wait until runtime to bind method calls to actual method code (see *binding, polymorphism, runtime binding*). At runtime, the compiler extracts a *class link* from an *object instance*, traces the link to find the method dispatch table, locates the correct method address in the table, and jumps to that address to execute the *method*. Such tables can be implemented in a variety of ways, and they are subject to change between versions of a compiler.

MPW Macintosh Programmer's Workshop. A product of Apple Computer that provides a complete programming environment with both a Macintosh user interface and a Unix-like command line. You can use a variety of programming languages in MPW, including MPW Pascal, which is object oriented. MPW includes powerful scripting and toolmaking facilities.

object An *instance* of some *class* of similar instances. Perhaps best viewed as a software counterpart of some real-world object. Viewed as little capsules containing data (in *instance variables*) and operations on the data (*methods*), objects can be created and used dynamically in a variety of powerful and useful ways. Also see *class, encapsulation, polymorphism*.

object class. See *class*.

object hierarchy. See *hierarchy*.

Object Pascal Standard Pascal plus a few object-oriented extensions.

object reference An object variable, a variable of an *object type*, or *class*. An object reference is implemented as a Macintosh handle, which uses double indirection to point to the object's block in the heap. In effect, an object reference is also the object's name.

object type. See *class*.

Objective-C A variant of the C language that provides object-oriented extensions to C. Objective-C is an alternative to C++, at least on some computers. Objective-C is of interest primarily for its use on the NeXT computer.

overriding Including an inherited method's declaration in a *subclass*, following it with the *override* keyword, and writing a new implementation of the method. When

you write a subclass of some existing *class,* the new class automatically inherits its ancestors' *instance variables* and *methods.* If a particular method doesn't suit your needs, you can override it. The ancestor class still has its version of the method, but the new *descendant* now has a version of its own. In the code of the overridden method, you can revise what the method did, replace what the method did, "stub out" the method so that it does nothing, or use the keyword *inherited* to call the ancestor's version of the method.

polymorphism The ability to send the same *message* to *instances* of different *classes* and have each respond in its own way, by invoking its own corresponding *method.* You can send a Display message to all the *objects* stored in a linked list, say, even if they are of different types. Polymorphism also means that an instance of subclass *B* can be assigned to a variable of an ancestor class *A.* And when you send a message to the *A*-type object, the *B*-type object "contained" within the *A*-type variable will respond by invoking its *B*-type method. Whole data structures can be polymorphic—an array of *TObject,* for example, can contain at the same time instances of any classes that are *descendants* of *TObject*—all of one type or of varying types. *Runtime binding* is the essential mechanism that makes polymorphism possible. Also see *runtime binding.*

QuickPascal Microsoft's version of *Object Pascal* for MS-DOS computers. Unlike its chief competitor, Borland's *Turbo Pascal,* QuickPascal follows the Apple standard for Object Pascal fairly closely. Also see *Turbo Pascal.*

reference. See *object reference.*

runtime binding Having the compiler make the connection between a procedure, function, or method call and the operation's code at runtime rather than at compile time. Because object variables can be polymorphic, the compiler can't bind their method calls until runtime. The compiler has no way of knowing that, at runtime, a variable of type *A,* for example, will actually contain an *object* of the subclass *B.* So, at runtime, the compiler generates code to check the actual object *instance,* follow the object's *class link* to look up the right method address in a *method dispatch table,* and use that address to jump to the correct method code—all on the fly. Also see *binding, method dispatch table, polymorphism.*

self A pseudovariable by which a particular object *instance* can refer to itself. The object can use *self* when it needs to send itself a *message* or access its own *instance variables,* or it might pass a reference to itself to some other *object.* When a *method* is called, the compiler implicitly passes *self* along with the method's parameters, giving the method the ability to use *self.*

sending messages. See *message.*

Smalltalk An early OOP language developed by Xerox at its Palo Alto Research Center (PARC) in the 1970s. Smalltalk is a "pure" OOP language: totally object-oriented, not hybrid as *Object Pascal* or *C++* is. In Smalltalk, everything, even an integer, is an *object.* Apple's interest in Smalltalk led to the development of *Object Pascal.*

subclass A new *class* that is a *descendant* of an existing class. Subclasses can add to what they inherit from their *ancestors* and override the methods of their *inheritance.* They can extend or customize what their ancestors did.

subclassing The process of deriving a new *class* from an old one. Also see *inheritance, overriding, subclass.*

superclass An ancestor class. Each class can have one and only one *immediate ancestor,* hence only one immediate superclass. Any class on up the *hierarchy* is also a superclass. Also see *ancestor, descendant, hierarchy, subclass.*

TCL The THINK Class Library. TCL is an *application framework,* or a collection of object *classes* that together provide a rather complete Macintosh user interface, with menus, windows, event handling, and so forth, onto which you can graft your own application features, using OOP techniques, to produce a complete program. TCL is a newer and therefore somewhat less robust cousin of Apple's *MacApp* (but, in my view, a cousin with a cleaner design). MacApp is suitable for industrial-strength application development, and TCL is presently more suitable for smaller applications. Our own PicoApp is a cousin to both TCL and MacApp but suitable only for learning about application frameworks. Also see *application framework, class library, MacApp.*

THINK Pascal A Pascal compiler and programming environment produced by Symantec Corporation. THINK Pascal has the same *Object Pascal* features as MPW Pascal, a good class *browser,* and a wonderful integrated, source-level debugger.

TML Pascal A product of TML Systems, one of the first companies to produce a good Pascal compiler for the Mac as an alternative to Apple's MPW and Lisa Pascals. With its second version, TML Pascal became object oriented and was changed to run under the *MPW* environment. Most of what we say about MPW Pascal is also applicable to versions 2 and later of the now-discontinued TML.

TObject The usual name for the topmost class of an object *class hierarchy. TObject* is simple, providing the ability for any one of its *descendants* to clone itself or free itself. Also see *base class.*

Toolbox A wealth of routines and data structures for managing windows, menus, dialog boxes, and memory, editing text, and performing a variety of other tasks, built into the Macintosh ROM. (Technically, the ROM contains both the Toolbox traps and the operating system traps, but we'll lump them here.) Toolbox code is the most primitive code on the Mac, and your programs, including the *methods* of your *objects,* will frequently call Toolbox routines to do their work. Toolbox routines are documented in the (currently five) volumes of Apple's *Inside Macintosh.*

Turbo Pascal Although Turbo Pascal for the Macintosh is not (at this writing) object oriented, its cousin for MS-DOS is, as of version 5.5. Turbo Pascal implements *objects* quite differently than Apple's *Object Pascal* standard does, borrowing heavily from *C++* as well as from Object Pascal. With version 6, Turbo Pascal also has its own *application framework,* called Turbo Vision. Also see *QuickPascal.*

INDEX

References to figures and illustrations are in italics.

C

D

E

M

N

O

P

Q

R

S

T

U

V

W

Chuck Sphar has been teaching himself programming for 10 years, with small assists from the University of Southern California and New Mexico State University. He bought one of the first Macs and learned to program it in Modula-2 and Pascal despite the limitations of 128K of RAM and one tiny floppy drive. Chuck has been a miner, a surveyor, a GI, a legal clerk, and an assistant manager of a car wash. He was a college English teacher for 12 years. His first programming job was for the NASA contractor at White Sands that operates the TDRSS satellites that track the Space Shuttle. Chuck now works for Microsoft as a technical writer.

The manuscript for this book was prepared and submitted to Microsoft Press in electronic form. Text files were processed and formatted using Microsoft Word.

Interior text designer: Darcie Furlan
Principal word processors: Debbie Kem and Judith Bloch
Principal proofreader: Cynthia Riskin
Principal typographers: Lisa Iversen and Rodney Cook
Principal illustrators: Kim Eggleston, Connie Little, and Lisa Sandburg
Layout artist: Peggy Herman
Cover designer: Studio MD
Cover color separator: Color Control

Text composition by Microsoft Press in Garamond Light with display type in Futura Extra Bold, using the Magna composition system and the Linotronic 300 laser imagesetter.

Printed on recycled paper stock.

Other Titles From Microsoft Press

INSIDE SUPERCARD
The Complete Guide for Macintosh® Developers

Andrew Himes and Craig Ragland
Technical Assistance from Bill Appleton, Creator of SuperCard

"INSIDE SUPERCARD is the most accurate, informative, and complete book of its kind. I strongly recommend it to all SuperCard developers and advanced users."
Bill Appleton, Creater of SuperCard

If you currently use SuperCard to develop applications or stacks—or if you're starting to explore SuperCard's potential for the development of sound-enhanced, animated commercial applications for the color Macintosh—this book is a must-have reference. Comprehensive, up-to-date, packed with clear examples and scores of sample scripts, the information in INSIDE SUPERCARD is unavailable elsewhere. The heart of the book deals with the high-powered features of SuperCard version 1.5. Included is in-depth coverage of SuperEdit and Runtime Editor, Graphics and Animation Tools, Sound, SuperTalk ScriptTracer, Objects, Apple Desktop Interface, and Navigational Tool Design.
640 pages, softcover $22.95 Order Code INSU

THE PROGRAMMER'S APPLE® MAC® SOURCEBOOK
Reference Tables for Apple Macintosh® Hardware and System Software

Thom Hogan

At last! An information-packed reference book that integrates key technical data and information—previously published in scores of other sources into one convenient volume. With more than 400 pages of charts and tables, all thoroughly indexed, THE PROGRAMMER'S APPLE MAC SOURCEBOOK is your primary reference to Mac Plus, Mac SE, and Mac II hardware and software. It covers in detail data formats; interface toolbox managers, including Color QuickDraw; Window Manager, Font Manager, and Menu Manager; NuBus Slot Manager; operating system calls; HyperCard; Installer, System, and Finder; video driver information; ASCII codes; hexadecimal and binary conversions; and more. Included with each table are data sources, cross-references to related information in the book, and notes to help you use the information.
496 pages, softcover $22.95 Order Code PRAPMA

THE BIG BOOK OF AMAZING MAC® FACTS

Lon Poole

"Clearly written and superbly organized, THE BIG BOOK OF AMAZING MAC FACTS is an invaluable resource of inspiration for all Mac users." Macworld

Take advantage of shortcuts, practical tips, and new ideas to make work on your Mac faster, easier, and more fun! This is a superb collection of hundreds of useful tidbits from the author's *Macworld* magazine column "Quick Tips" to help you get the most out of your Macintosh—whether it's an older model or a new Classic, LC, or IIs. Tips cover System 7 software and applications of all types—word processing, spreadsheets, graphics, communications, and desktop publishing. See what your Mac can do for you with THE BIG BOOK OF AMAZING MAC FACTS.
528 pages, softcover $24.95 Order Code BIBOAM

Microsoft Press books are available wherever quality computer books are sold.
*Or call **1-800-MSPRESS** for ordering information or placing credit card orders.**
*Please refer to **BBK** when placing your order.*

* In Canada, contact Macmillan of Canada, Attn: Microsoft Press Dept., 164 Commander Blvd., Agincourt, Ontario, Canada M1S 3C7, or call (416) 293-8141.
In the U.K., contact Microsoft Press, 27 Wrights Lane, London W8 5TZ.

IMPORTANT—READ CAREFULLY BEFORE USING. By breaking the seal on the disk pack included with this book, you indicate your acceptance of the following Microsoft License Agreement.

Microsoft License Agreement

This is a legal agreement between you and Microsoft Corporation. By using the enclosed disk(s), you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the disk(s) and the accompanying items (including the book and other materials included herewith) to Microsoft Press for a full refund.

MICROSOFT SOFTWARE LICENSE

1. GRANT OF LICENSE. Microsoft grants you the right to use one copy of the enclosed Microsoft software program (the "SOFTWARE") on a single terminal connected to a single computer (i.e., with a single CPU). You may not network the SOFTWARE or otherwise use it on more than one computer or computer terminal at the same time.

2. COPYRIGHT. The SOFTWARE is owned by Microsoft or its authors and is protected by United States copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE like any other copyrighted material (e.g., a book or musical recording) except that you may either (a) make copies of the SOFTWARE solely for backup or archival purposes, or (b) transfer the SOFTWARE to a single hard disk provided you keep the original solely for backup or archival purposes. You may not copy the written materials accompanying the software.

3. OTHER RESTRICTIONS. You may not rent or lease the SOFTWARE, but you may transfer the SOFTWARE and accompanying written materials on a permanent basis provided you retain no copies and the recipient agrees to the terms of this Agreement. You may not modify, reverse engineer, decompile, or disassemble the SOFTWARE.

4. DUAL MEDIA SOFTWARE. If the SOFTWARE is acquired on both 3½" and 5¼" disks, then you may use only the disks appropriate for your single-user computer. You may not use the other disks on another computer or loan, rent, lease, or transfer them to another user except as part of the permanent transfer (as provided above) of all SOFTWARE and written materials.

5. SAMPLE CODE. If the SOFTWARE includes sample code, then Microsoft grants you a royalty-free right to reproduce and distribute the sample code from the SOFTWARE in object code form provided that you: (a) distribute the sample code only in conjunction with and as a part of your software product; (b) do not use Microsoft's and/or its author's names, logos, or trademarks to market your software product; (c) include the copyright notice that appears on the SOFTWARE on your product label and as a part of the sign-on message for your software product; and (d) agree to indemnify, hold harmless, and defend Microsoft and/or its authors from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your software product.

DISCLAIMER OF WARRANTY

THE SOFTWARE (INCLUDING INSTRUCTIONS FOR ITS USE) IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. MICROSOFT AND ITS AUTHORS DISCLAIM ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH REGARD TO THE SOFTWARE AND THE ACCOMPANYING WRITTEN MATERIALS. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS WHICH VARY FROM STATE/COUNTRY TO STATE/COUNTRY. FURTHER, MICROSOFT AND ITS AUTHORS DO NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIALS CONCERNING THE SOFTWARE IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY YOU. IF THE SOFTWARE OR WRITTEN MATERIALS ARE DEFECTIVE, YOU, AND NOT THE AUTHOR, MICROSOFT OR ITS DEALERS, DISTRIBUTORS, AGENTS OR EMPLOYEES, ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION.

NEITHER MICROSOFT, ITS AUTHORS, NOR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THIS SOFTWARE SHALL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH SOFTWARE EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATIONS OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

U.S. GOVERNMENT RESTRICTED RIGHTS

The SOFTWARE and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restriction as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software -- Restricted Rights 48 CFR 52.227-19, as applicable. Contractor/manufacturer is Microsoft Corporation/One Microsoft Way/Redmond, WA 98052-6399.

This Agreement is governed by the laws of the State of Washington.

Should you have any questions concerning this Agreement, or if you wish to contact Microsoft Press for any reason, please write: Microsoft Press/One Microsoft Way/Redmond, WA 98052-6399.

By breaking the seal on this disk pack, you accept the terms of the Microsoft License Agreement included in this book.

If you do not agree to these terms, please return the book and all the disk packs, unopened, along with the rest of the package contents, immediately to Microsoft Press.

Microsoft®

OBJECT-ORIENTED PROGRAMMING POWER

for THINK Pascal™ Programmers

If you program the Apple® Macintosh® in Pascal and you're intrigued by object-oriented programming (OOP) and application framework programming, this book is for you. It delivers what you need to know—in your native language. Learn OOP with Object Pascal, an elegant extension of Pascal. Discover the secrets of object-oriented application design and, in the process, develop PicoApp, the book's own small MacApp-like application framework. Follow the author's groundbreaking exploration of the relationship between class hierarchies and polymorphism, and learn how to exploit the implications of that relationship in reliable, reusable software components. Your knowledge of Pascal is all the preparation you need. The book features:

- **A Thorough Introduction to OOP.** Learn OOP from the ground up, revisiting the central concepts at higher and higher levels and in increasingly rich relationship to one another.
- **Clear Instruction on Application Frameworks.** Learn the fundamentals of using application frameworks such as the THINK Class Library (TCL) from Symantec and MacApp® from Apple. Look under the hood of the book's own tiny framework, PicoApp. Two PicoApplications teach framework techniques.
- **Techniques for Writing Fast, Reliable, Reusable Software Components.** Explore the author's original object innovations as he develops a robust, reusable list data structure that can hold any data type—or multiple data types at the same time. See *search key objects* that simplify searching lists and trees, *traversal action objects* that make it easy to perform the same action on every item in a list or tree, and *array objects* that overcome Pascal's limitations on character-array parameters.
- **Dozens of Projects and Examples.** Throughout, the author tests your progress with recommendations for projects of your own that build on the book's examples. The accompanying code disk contains full versions of all the examples. They are written in THINK Pascal but are usable with both MPW® and TML Pascal.

System requirements

Apple Macintosh with 1 MB RAM and a hard disk; THINK Pascal 2.0, MPW Pascal 2.0, TML Pascal 2.0, or later versions

Package contains

One 800 KB 3½-inch code disk

U.S.A.	**$39.95**
U.K.	£35.95 (VAT included)
Canada	$54.95

[Recommended]

Microsoft PRESS®

The Authorized Editions

ISBN 1-55615-348-1