Macintosh®

The MacApp® Interim Manual

Working Draft 4 (APDA)
June 14, 1988

APPLE COMPUTER, INC.

20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

6/14/88

Table of Contents

Preface

The MacApp® object-oriented application framework is a set of libraries you can use to speed your software development process and to create more robust applications for Macintosh® computers. This manual is a preliminary draft of what will ultimately be a two-volume suite (Volume 1 will be a tutorial; Volume 2 will be a reference). Most of the reference material is missing from this draft, as are the labs that will show you, step-by-step, how to build an application with MacApp. However, this draft is a complete (if not comprehensive) introduction to MacApp and object-oriented programming.

Prerequisites

To make any use of MacApp, you must have certain hardware and software; to make good use of this manual, you must have had some experience programming.

Hardware

You must have a Macintosh computer with a hard disk and at least one megabyte of memory to develop an application with MacApp. However, most applications will require you to have at least two megabytes of memory while you are developing them.

It is possible to create applications with MacApp that can run on any computer in the Apple® Macintosh line, including the Macintosh, the Macintosh Plus, the Macintosh XL, the Macintosh SE, and the Macintosh II, so long as the application doesn't exceed the memory limits of the computer and you are careful to avoid using features that are not supported across the product line. (MacApp itself is written to run on any Macintosh.)

Software

MacApp works under the MPW® development system. MPW is the Macintosh Programmer's Workshop. The final version of MacApp 2.0 will work with MPW 3.0, but this version works with MPW 2.0.2. MacApp comes with an interface for MPW Pascal; final versions of the software will also come with an interface for MPW C++.

What you should already know

If you are a competent Pascal or C programmer and you have played with a Macintosh enough to have a feeling for what makes a good application, then you know enough to use this manual. You don't have to know anything about object-oriented programming and you don't have to have written a single line of code for a Macintosh program.

However, if you have never programmed a Macintosh before, you can do some reading to make learning MacApp much easier. The books listed here appear in the order of how vital they are to your understanding:

- Before you begin with this manual, read Chapters 1, 2, and 3 in *Programmer's Introduction to the Macintosh* to get an overview of how programming the Macintosh is different from programming other computers. These chapters will also give you a good idea what it's like to program the Macintosh without an aid like MacApp.

  You should go on to read the rest of the book, especially Chapter 4 ("Memory Management"), Chapter 5 ("Display and Graphics Routines"), and Chapter 7 ("File Management") as they seem necessary. The information in Chapter 6 ("The User Interface Toolbox") is interesting and important, but it may seem confusing. Wait until you understand MacApp fairly well before you read Chapter 6.

- At your leisure, read the "User Interface" chapter of *Inside Macintosh,* Volume I and browse through Apple's *Human Interface Guidelines* to get an idea of the different elements you can include in your interface. These works were the guidelines by which MacApp was created; you should use them as your bible to create exciting and effective applications.

- After you have finished the Primer section of this book and before you start programming, read the "QuickDraw" chapter of *Inside Macintosh,* Volume I to learn how to create images on the Macintosh screen.

- When you are ready to create programs that will write files to disk, read the "File Manager" chapter of *Inside Macintosh,* Volume I.

- After you understand the basic concepts of MacApp, read the "Resource Manager" chapter of *Inside Macintosh,* Volume I.

- The more you know about MPW, the Macintosh Programmer's Workshop, the more efficiently you will be able to develop applications with MacApp. See the *MPW Reference Manual.*

See "Other Books You Should Have," later in this preface, for bibliographic information on these works. These books contain information about Macintosh ROM routines. Remember that because *you* use MacApp, you will not need to use many of these routines.

What this manual contains

The primer introduces the concepts behind MacApp, starting with an overview of MacApp itself. It continues with an introduction to object-oriented languages in general, and Pascal in particular. Next, it describes the physical, hierarchical, and conceptual organization of MacApp and shows the flow of control, as governed by MacApp. Finally, it tells you how to install MacApp.

The cookbook contains a number of segments, each of which answers a question in the form of How do I _______? For instance, one segment might answer the question How do I create windows? Another one might resolve the question How do I track the mouse?

The appendixes contain information for readers who may be familiar with MacApp 1.1. Appendix A describes the differences between the old and new versions of MacApp. Appendix B describes the method of converting applications based on MacApp 1.1 to applications based on MacApp 2.0.

How to use this manual

The more you already know about object-oriented programming, programming the Macintosh, or using MacApp, the more of this manual you can skip.

The only readers who may find it useful to read this whole manual are those who already know how to program the Macintosh but are not yet familiar with MacApp or object-oriented programming. Others may want to customize their reading as follows:

If you have never programmed a Macintosh you should carefully consider the suggestions in the section "What You Should Already Know."

If you are familiar with object-oriented programming you can skip chapters 2 and 3.

If you are an impatient learner you should still read Chapter 1. Then, if you are adventurous, skip to Chapter 6, "How to Install and Use MacApp." If you feel confused, go back and read the rest of the information in the primer.

If you are well-acquainted with MacApp 1.1 you may choose to read only Appendixes A and B, "Changes Since MacApp 1.1" and "How to Convert Your MacApp 1.1 Application." Then you will probably have to rely heavily on the cookbook for the first applications you write with MacApp 2.0.

Notation

This manual uses `courier` type to represent code fragments. Note that bold type is sometimes also used to highlight important concepts, especially at the beginning of paragraphs.

Like all technical manuals, this manual uses some words in special, conventionalized ways. These words and other conventions are explained in detail in the text, but here is a list for quick reference:

- **Routine** means any procedure or function.
- **Method** means any routine belonging to a class of objects.
- ***Your*** indicates an identifier or part of an identifier that you are expected to replace, as in **T*Your*Application**. If your application were named Corporate, you would replace the name as TCorporateApplication.
- **Generic references** to instances of a particular class are given the name of the class minus the "T" — for example, in this manual a generic instance of the class TWindow is just called a window.
- **Pascal reserved words** and predefined type names are written in all capital letters.
- **ROM routines** are routines in the Macintosh ROM and system software. They are documented in *Inside Macintosh.*

Conventions in the MacApp code

- All variable and constant identifiers begin with a lowercase letter.
- Global variable identifiers begin with a lowercase *g* (for example, gTarget).
- Command constants begin with a lowercase *c* (for example, cUndo).
- Most other constants begin with a lowercase *k* (for example, kWantHScrollBar).
- Fields of object classes begin with a lowercase *f* (for example, fDocument).
- All routine names, both object-oriented and otherwise, begin with an uppercase letter (for example, Enable).
- Names of classes begin with an uppercase *T* (for example, TApplication).
- Compile-time variables begin with a lowercase *q* (for example, qDebug).

MacApp and Object Pascal do not add any special checking to enforce these conventions. Uppercase and lowercase characters are equivalent in Pascal identifiers because the Pascal compiler coverts all characters to uppercase. When you use the MPW Assembler, all identifiers are also converted to uppercase, unless you use the Assembler directive to make case significant. You should not use that directive when using the Assembler with MacApp.

Other books you should have

Here is a list of books you should have within reach while programming with MacApp.

- *Human Interface Guidelines* (Addison-Wesley, 1987). This book describes how Macintosh applications should appear to their users. A tremendous amount of research and experience have gone into this book, and you may wish to consult it often to avoid inventing new solutions to problems that have already been solved.
- *Inside Macintosh.* There are five volumes of *Inside Macintosh* and you need all but the third one. Volumes I and II (Addison-Wesley, 1985)

describe the different managers in the ROM. Volume IV (Addison-Wesley, 1986) discusses special capabilities of the so-called 128K ROM introduced in the Macintosh 512K Enhanced and the Macintosh Plus computers. Volume V (Addison-Wesley, 1988) discusses the Macintosh SE and the Macintosh II (the color section may be of special interest).

One of the main advantages of using MacApp is that you don't need to know in detail much of the information in *Inside Macintosh.* However, you should pay attention to information on the User Interface, QuickDraw, and the File Manager. You may also want to browse through chapters concerning the Window Manager, the Resource Manager, the Menu Manager, and the Control Manager as well as any information concerning sound, color, and MultiFinder.

- *MPW Assembler Reference* (Apple Computer, 1986). If you intend to include Assembler language routines in your applications, you will need this manual.
- *MPW Pascal Reference* (Apple Computer, 1986). This manual documents Apple's version of the Object Pascal language.
- *MPW Reference and MPW Command Reference* (Apple Computer, 1986). These manuals describe the development system you need to use with MacApp. The more familiar you are with MPW, the more efficiently you'll be able to work.
- *Programmer's Introduction to the Macintosh* (Apple Computer, 1987). This book explains the important concepts you need to know in order to program the Macintosh.

Addison-Wesley books are available at commercial bookstores. Books and manuals published by Apple are available through APDA™, the Apple Programmer's and Developer's Association. In addition, technical notes and other materials of interest to Macintosh application developers are available from APDA.

Chapter 1
Why MacApp?

MacApp® is a tool that helps programmers create better applications in less time. It helps programmers work faster by allowing them to program in a style well suited to Macintosh® applications. It also facilitates the implementation of the standard Macintosh interface, including everything from scrollable, resizable windows to multipage printing and undoable commands. It makes programs better because it provides code carefully written by expert Macintosh programmers; as long as you exercise similar care, MacApp applications are virtually guaranteed to be compatible across the whole Macintosh line and to work with all other Apple® products, from A/UX® to MultiFinder™.

What is MacApp?

In technical terms, MacApp is an object-oriented application framework. This phrase—"object-oriented application framework"— contains a lot of information and deserves further explanation. MacApp is *object-oriented* because it uses a style of programming that is organized around **objects** instead of procedures and functions. Chapter 2 explains object-oriented programming and the definition of objects in detail. In MacApp, objects represent Macintosh entities such as windows and dialog boxes as well as real-world entities such as chess pieces, rectangles, and automobiles.

MacApp is an *application framework* because it provides a general structure that you can use to build almost any application. With MacApp and less than one page of your own code, you can have a complete Macintosh application that creates windows, handles mouse clicks, prints files, and supports almost every standard feature Macintosh applications are likely to have. Of course, this generic program will have no specific functionality; that much you will have to do yourself. But with MacApp, when you wish to use the standard features, you will find MacApp does most of the work for you. If you wish to stray a little from these standards, you will have to do a little extra work. In general, the more you want to stray, the more work you will have to do.

In more specific terms, MacApp is a large library of code written in Object Pascal and MPW Assembler Language. The library includes global procedures, functions, and methods that provide the standard Macintosh features, and optional units that will provide printing, dialog boxes, and text editing. MacApp also includes logic that will supervise when your own customized code is to be called.

Finally, you should note that MacApp is a framework for *applications* only. MacApp is not the appropriate tool for building other sorts of programs. It cannot be used to create device drivers or desk accessories.

Programming without MacApp

Programming a Macintosh, whether or not you use MacApp, is different from programming most other computers. While programs for other computers constrain the user to one of a few actions, Macintosh applications put the user in charge. Macintosh applications must be ready to handle anything at any time: without warning, the user can move the mouse, type a number, click in the menu bar, insert a disk, enter a command from the keyboard, or use the network.

Because Macintosh applications give users so much flexibility, they require a different structure than traditional programs. Macintosh application programs are built around a loop, called the **main event loop**, which continually looks out for any event that requires a response. When an event comes along, it temporarily passes control to some other piece of the program responsible for handling that particular type of event. The program handles the event and returns control to the main event loop. The process continues until the user quits the application.

While the main event loop itself does not require very much code, the code required to handle all the events is extensive. For instance, each time the mouse button is clicked, the application must determine whether the action requires a response and, if so, what sort of response it merits. A click in the menu bar may cause a menu to appear and will require that the mouse be tracked; if the mouse button is released over a menu item, the application will have to execute the appropriate command. A click in a window might activate the window. A click in a scroll bar might cause the view to change. And a click in a title bar might cause the window to move.

Moreover, applications require a fair amount of code to provide error checking and standard features such as printing, saving, and posing queries through dialog boxes. Much of this work is provided by code in the toolbox of the Macintosh ROM, but not as much as you might think. The toolbox knows nothing of windows with scroll bars, undoable commands, or clusters of radio buttons. It takes a lot of code just to provide these standard features.

Even someone who has no programming experience can tell that the features don't come automatically. Most Macintosh users can tell you, in between swear words, which programs have some crucial command that is not undoable. You can probably think of a program that has windows without scroll bars or that doesn't allow multiple windows to be open at once. If these features were free and easy, every program would have them.

That is the idea behind MacApp. MacApp makes it so easy to include standard features that programmers find it less work to provide the features than not.

Programming with MacApp

If you have programmed much at all, you have probably learned that you can save a lot of time by stealing lines of code from other programs and changing a few parameters here and there. Macintosh application programs have a great amount in common with each other, so borrowing code is particularly effective. However, plagiarizing code is a difficult, dangerous endeavor, even if it is your own code you are stealing. It is easy to overlook a missing parameter or a line of code that was meaningful only in some other program. MacApp puts the very best of Macintosh programming at your disposal without these pitfalls.

The manner in which MacApp allows you to borrow and customize code is clever and subtle, relying heavily on the use of object-oriented programming techniques. The next two chapters are devoted to this subject. For now, imagine that MacApp was not written in an object-oriented language, but rather in the procedure-oriented style more familiar to most programmers. In this hypothetical (and false) model, MacApp is a complete template application. It has a main event loop that waits for events, and a number of routines that handle them. You just fill in the blanks.

The division of labor

In general, you can assume that MacApp does as much as it can without being able to read your mind. Thus, the portion of MacApp responsible for windows will know how to respond if the user clicks in any of the window's controls, because virtually every application handles these clicks in the same way. If users want to change the size of the window, they always click the size and zoom boxes; if they want to move the window, they always drag the title bar. However, you could not expect MacApp to know what the content of any particular window might be. It is MacApp's job to handle events involving the controls of the window, since these are predictable; it is your job to handle events involving the content of the window, since only you know what this might be.

Commonsense logic of this sort will help you figure out what MacApp does for you and what you must do for yourself. Sometimes MacApp knows *when* something must be done, but not *how*. When this is the case, MacApp will call a routine with a specific name which it expects you to have written. Much of this manual will be devoted to teaching you which routines you have to write and how they should be named. Here are some examples of how the division of responsibility works in MacApp.

- **Managing windows.** Windows can have a variety of standard components, pictured below.

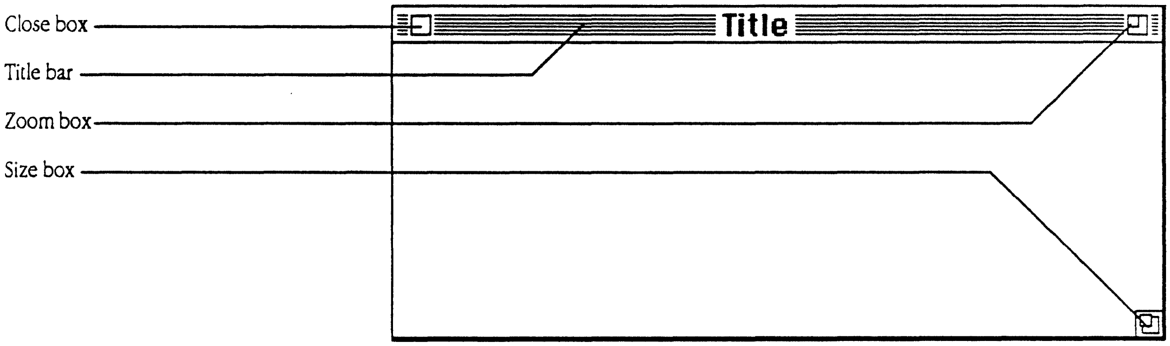


Figure 1-1
Components of a window

MacApp provides some of these components at your discretion. MacApp's code draws the window and responds if the user manipulates the size box, title bar, or zoom box. Your code is responsible for drawing the contents of the window and responding if the user manipulates the contents. In addition, if you want any of these components to work in an unconventional manner, you will have to add some code of your own.

- **Managing menus.** MacApp does most of the work involved in managing most menus. You must inform MacApp of the items that are to be listed in the menus, how they might be chosen from the keyboard, and their state (enabled, checked, and so forth). MacApp will automatically display the menus and highlight the commands in the usual way.

- **Executing menu commands.** Since MacApp cannot anticipate what commands your application is likely to have, you must write the code for each command yourself. If a command is undoable, you will have to write at least three pieces of code: (1) how to execute the command; (2) how to undo it; and (3) how to redo it. MacApp knows when to call which piece of code.

- **Handling errors.** MacApp provides a framework for error handling, as well as a number of routines for detecting and handling errors that fit into that framework. You supplement MacApp's error handling in a variety of ways: by calling MacApp's error handling and detection routines, by providing more specific error messages for those routines, and by providing your own routines.

- **Printing.** MacApp provides standard multipage printing capabilities which you can enable by linking in an extra module and writing three lines of code. If you wish to add any extra functionality, like drawing borders around the text or placing headings on every page, you must add them yourself, though MacApp has provisions to make this work relatively easy.

- **Editing text.** MacApp provides a text editor which you can access by linking in an extra module and adding another three lines of code. Again, you can enhance this editor by adding your own custom code.

- **Filing documents.** MacApp provides a framework for saving files and checking to make sure sufficient disk space is available. Since the format in which you save the data is not predictable, you must provide the code to read and write data. MacApp knows when to call this code.

This division of responsibility is one of the most difficult concepts in learning how to use MacApp. Even once you learn who does what, you may lose faith that MacApp will do its part, since you are not providing the code yourself.

This manual will make a special effort to make your responsibilities clear; you can trust MacApp to do the rest.

The benefits of MacApp

If you develop applications with MacApp, *you* will benefit because MacApp provides you with a superior development system and the *users* of your application will benefit because the application will work the way they expect. MacApp is written according to Apple's own specifications. As a result, your application is likely to work on any computer in the Macintosh line. It can work on the Macintosh XL as well as the Macintosh II, and it will work with MultiFinder and A/UX, provided you maintain MacApp's standards of compatibility. Moreover, because MacApp is supported by Apple, your application is likely to continue working with any new Macintosh systems.

Besides adding more functionality to your application and making it easier to maintain, MacApp makes the development cycle more productive. Since MacApp provides the basic framework, programmers can focus on the more interesting portions of an application. As a result, you can speed up development or get more done in the same amount of time. In addition, MacApp allows you to develop the application in an extremely modular fashion. You can recompile after adding each small method and get a testable, working version every time. You will always have a working application to show your boss or your clients. And if there is a problem, MacApp provides you with a high-level object-oriented debugger.

Perhaps the greatest benefit MacApp affords you is the pleasure your users will get from running your new application. Because your application was written with MacApp, the user interface—the window controls, menus, and so on—will be likely to work just as the user expects, and he or she will spend less time learning the application, and more time enjoying it. Because your application was written with MacApp, it's more likely that the user will find it works even under MultiFinder and with future revisions of the Macintosh Operating System. These are the criteria by which the user decides whether an application is truly professional and worth the money.

In short, an application written with MacApp is likely to be a better application.

Chapter 2

An Introduction to Object-Oriented Programming

MacApp is a powerful programmer's tool. A great deal of this power stems directly from the fact that MacApp is written in an object-oriented programming language, and that this tool can easily be enhanced by you—the application programmer. However, to take advantage of MacApp, you must be working in an object-oriented language, such as Object Pascal or C++. This chapter is not meant to be, and couldn't be, a complete guide to object-oriented programming. However, if you have a solid programming background, you should be able to learn the essentials of Object Pascal by reading this chapter—even if you have no experience with object-oriented programming. Chapter 3 then fills in the details by describing the exact syntax of Object Pascal.

The big picture

Before getting into the specifics of object-oriented programming, this section describes the theory behind this new way of programming, to show exactly how and why it differs from conventional algorithmic programming in languages such as Pascal or C.

In more conventional programming environments, the organization of a program is centered around the procedures and functions of that program. Typically, a program solves a problem by dividing it into a list of simpler tasks, each divided into simpler tasks still. This gives programs a treelike structure organized around their routines.

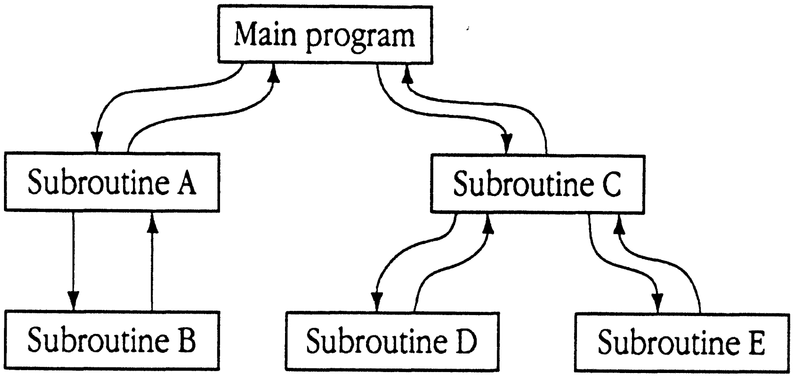


Figure 2-1
Treelike structure based on routines

This conventional programming paradigm allows large programming problems to be solved by dividing programs into routines. Therefore, problems that can be broken up into tasks that execute in a fairly predictable, linear manner are easily solved. Unfortunately, on a computer like the Macintosh, the user interface presents a programming task that is typically not predictable or linear.

Object-oriented programming allows programmers to solve complex, nonlinear problems in small, easily-managed functional units. This process requires forgetting the old habit of breaking programs into linear algorithmic solutions. Instead the idea is to break problems up into functional units called objects.

Theoretically, an object is a data structure in memory that provides some functionality in addition to some data storage. In this way, objects are equivalent to little computers in memory, storing data, doing calculations, and sometimes communicating with other objects. Dividing the responsibilities wisely among different objects can lead to well-organized programs that can solve monumental tasks.

Although objects are most fairly thought of as a method for organizing programs, it might be easier at first to think of them as records (in Pascal) or structs (in C) with some new features. In fact, the way objects are declared and the way they use data fields are strikingly similar to record and struct declaration and data field use.

Record types and record variables

If you are familiar with the use of records, then you already have a beginning understanding of objects since objects are analagous to records with extended functionality. When you decide what types of records you want to use in a program, you create **record type definitions,** which name the record types and declare the data fields belonging to each record type. These definitions don't actually create any records, however. In the same way that a cookie cutter defines the shape of a cookie but not its contents, the record type definition defines the shape of a record variable but contains no data itself.

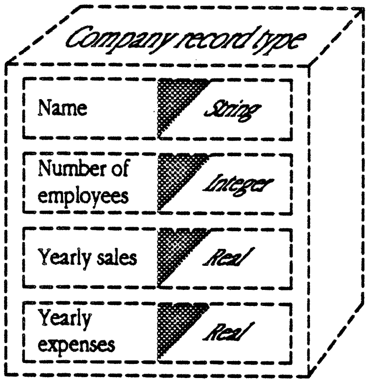


Figure 2-2
A record type

Once a company record type like the one in Figure 2-2 has been defined, actual records of that type can be created when the program is executing. To create actual **instantiations** of a record type, you would typically declare variables of that type. These variables will represent actual space in memory where data can be stored.

It is important to remember that the record type definition has no space allocated in memory. Only after a record variable of that type is declared is a space in memory reserved for storing values into the data fields. The distinction between the type definition, and the actual record in memory, is a very important one.

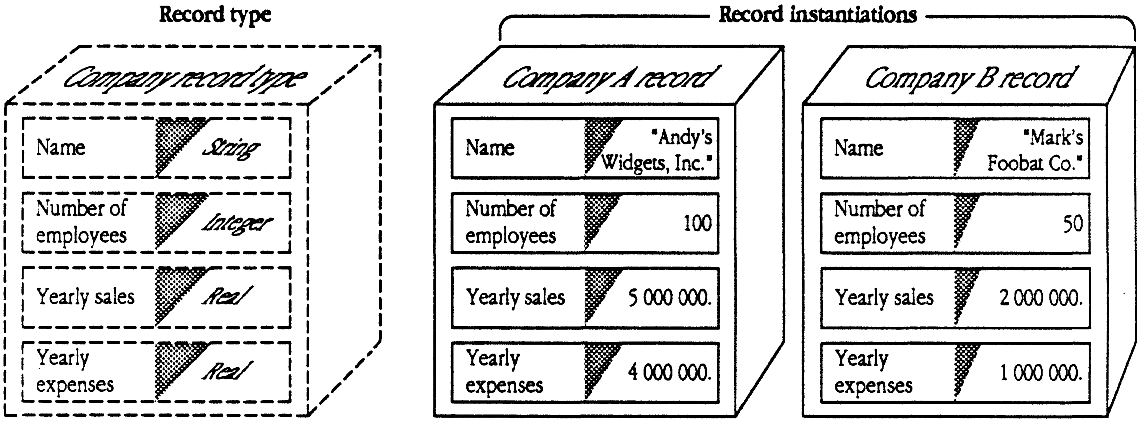


Figure 2-3
A record type and record instantiations

Object classes and object instances

Objects, then, are simply records with some added features. An object is made up of two parts: the object's data and the object's routines that operate on that data. Like a Pascal record or a C struct, the data that belongs to an object is stored in **fields**. The routines that manipulate the data in those fields are called **methods**.

As with the Pascal records, objects must have type definitions to define their "shape"—that is, the number and nature of their fields and methods. To help distinguish record types from object types, object types are called **object**

classes, or simply **classes.** Keep in mind that type and class mean approximately the same thing.

The actual objects that exist in memory while a program is running are called **object instances,** or simply **instances.** Object instances are like small pseudocomputers. Each object instance has its own memory for data (the fields), and its own functionality (the methods that work on those fields).

As a simple example of an object class, let's look a class with data fields but no methods. In Figure 2-4, notice that object classes with no methods are conceptually equivalent to record types.

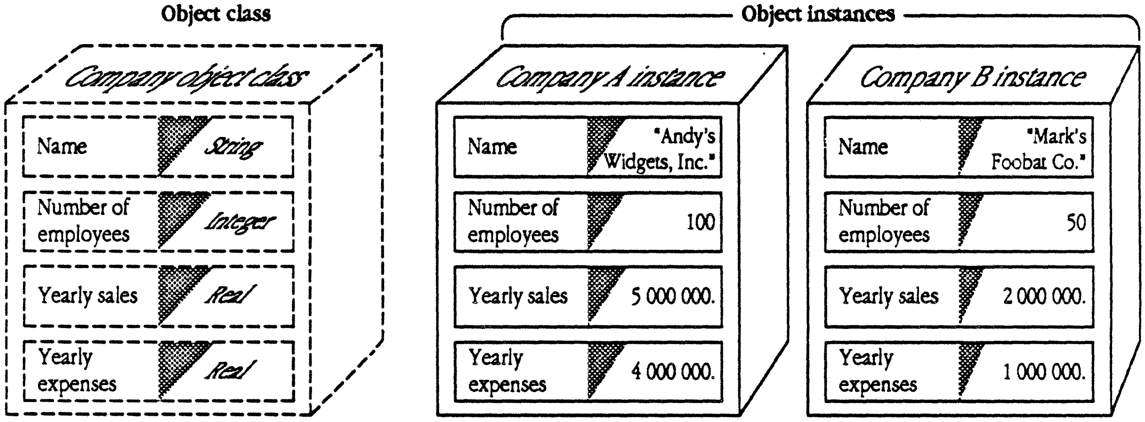


Figure 2-4
An object class and an object instance with no methods

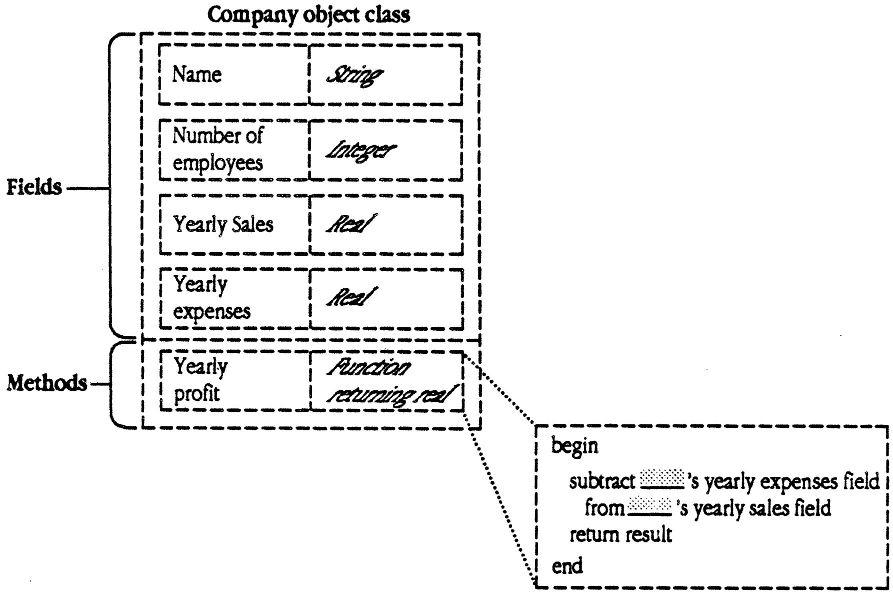
You can only use this company object class to create instances that are very limited pseudocomputers, because any instance of the company object class has storage ability but no functionality. To give functionality to instances of the company class, you must add a method to the company object class definition.

The way fields and methods relate to classes and instances is the single most important concept in object-oriented programming. Here is an overview of this relationship:

- Fields are defined to belong to an entire class of objects.
- The content of a field belongs to an individual object instance. When you create an object instance, space in memory is reserved for that instance to store values for each of the fields of its class. An instance can store values in its own fields that are different from the values stored in other instances' fields, as in Figure 2-4.
- Methods are defined to apply to an entire object class.
- When you call a method, you must specify which instance the method should operate on. When the method executes, it uses the data from the fields belonging to that specified instance.

Let's create an example method for the company object class. Since an instance of the company object class represents an individual company, you might find it useful to have a company instance be able to calculate its yearly profit by subtracting its yearly expenses from its yearly sales.

This yearly profit method definition, which must belong to the entire company object class, might look like Figure 2-5.

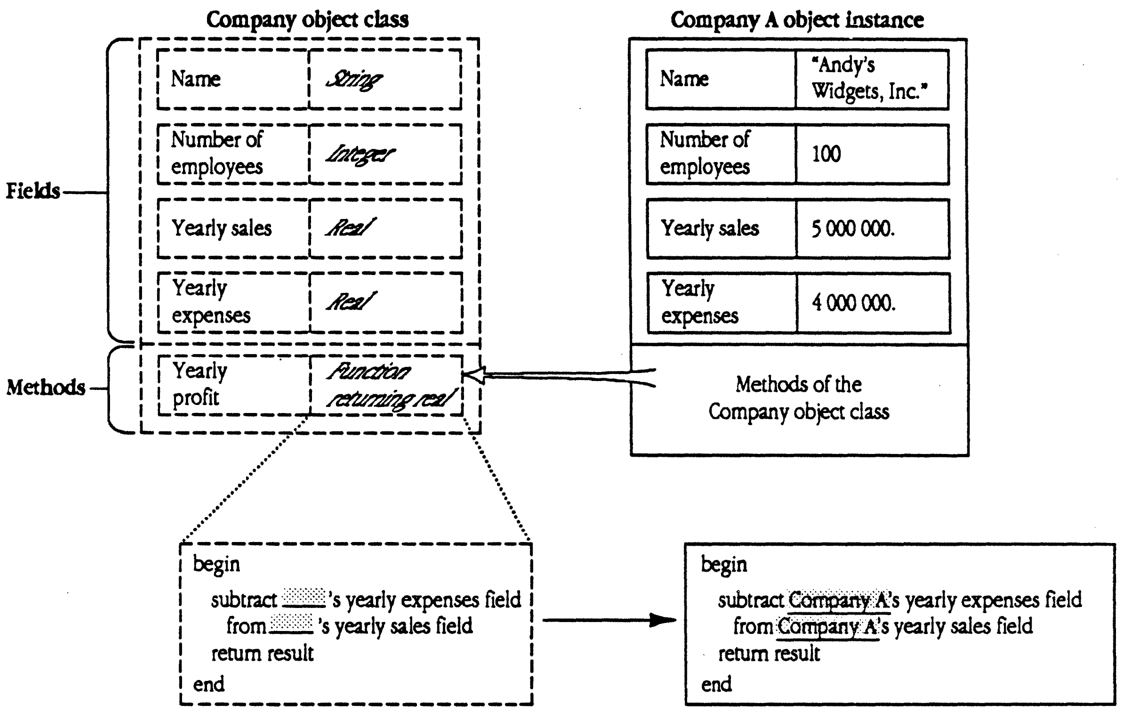


```
begin
  subtract ▨'s yearly expenses field
    from ▨'s yearly sales field
  return result
end
```

Figure 2-5
An object class with a method definition

Because this method belongs to the entire class of company objects, you cannot simply call it as you would a procedure or function. If you did, how would it know which instance's yearly sales or yearly expenses field to use? Remember, an object *class* cannot store values in fields—only object *instances* can, and there can be many object instances of the same class. Therefore, you must call a method to operate on a particular instance. To do this, you must **send a message** to that instance, telling it to execute the method on itself.

When you send a message to a particular instance, the method that actually executes is conceptually a specialized version of the method defined for the entire class—a specialized version that operates on that particular instance's data fields. For example, in Figure 2-6 you can see the differences between the yearly profit method as it was *defined* for the the class, and as it actually *executes* when you send the yearly profit message to the CompanyA object instance.



```
begin
  subtract ░░'s yearly expenses field
    from ░░'s yearly sales field
  return result
end
```

```
begin
  subtract Company A's yearly expenses field
    from Company A's yearly sales field
  return result
end
```

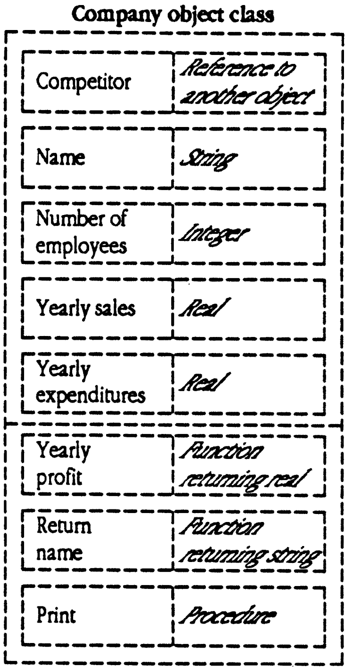
Figure 2-6
A method as it is defined and as it executes

To summarize then, an object class definition defines the generic shape of one class of objects—the number and size of the data fields, and the definition of the methods. An object instance is an actual instantiation of an object class—it has its own data in its fields, and you can send it a message telling it to call any of the class's methods on itself.

Flow of control in object-oriented programs

An object-oriented program is one based on objects instead of routines. When an object-oriented program executes, many object instances are created. They exist as space in memory where they store their data. Instead of following the typical treelike path of one procedure calling another procedure and returning, an object-oriented program follows a slightly different path. In the environments you'll be using with MacApp, there is still a conventional main procedure to start things rolling, but then the object instances take over. One instance's methods will send messages to other instances, as the flow of control bounces around from one instance to another. To be sure, each method eventually ends and returns to the method that sent the message calling it, as in conventional programming—the difference, especially at first, is largely in the way you think about the running program.

To give a clear picture of how control flows through an object-oriented program, let's first expand the object class defined previously.



Company object class

Competitor	*Reference to another object*
Name	*String*
Number of employees	*Integer*
Yearly sales	*Real*
Yearly expenditures	*Real*
Yearly profit	*Function returning real*
Return name	*Function returning string*
Print	*Procedure*

Figure 2-7
An expanded business object class

Notice that this object class has a new field—the competitor field. This field will be used by one object instance to refer to the object instance that represents its competitor. In other words, this field is like a pointer that allows one object instance to reference another object instance.

For example, in your program you can instantiate this expanded company object class by creating two object instances, and making them each other's competitors.

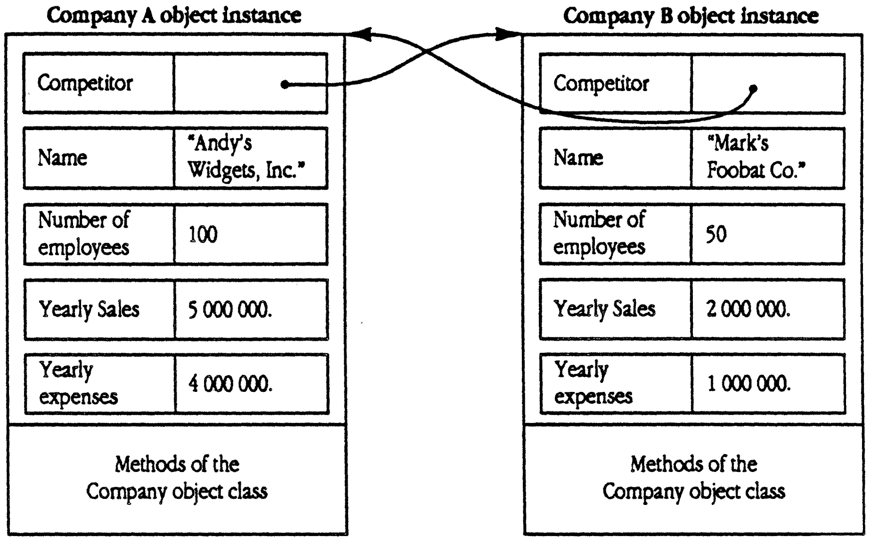
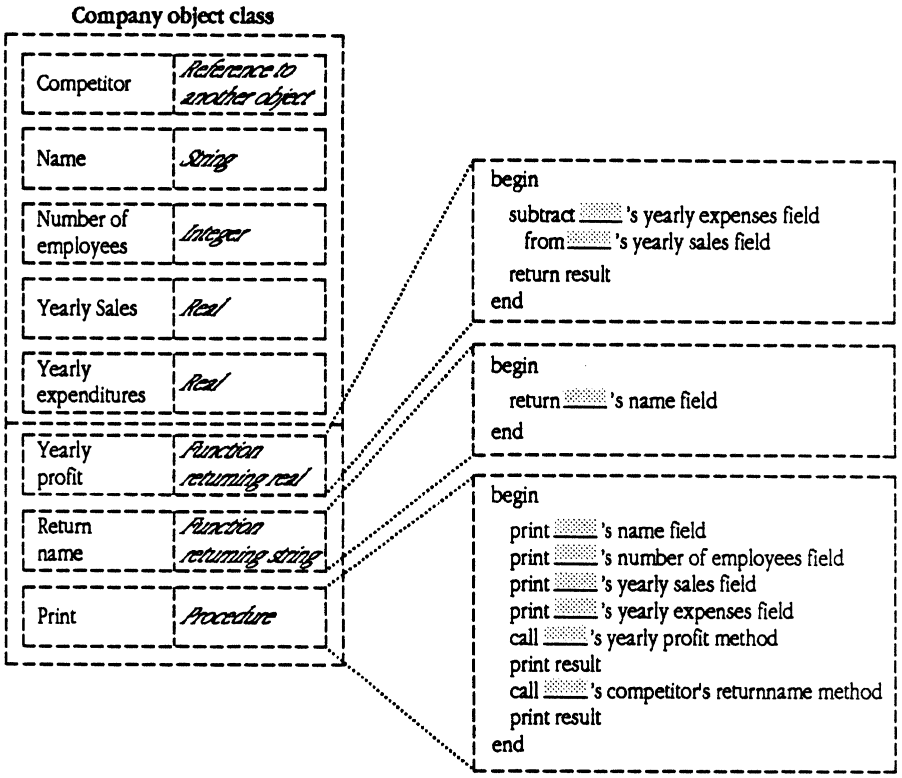


Figure 2-8
Object instances referring to each other

You'll soon see why it is important for one instance to be able to refer to another.

Going back to the new class definition, you'll see that there are also two new methods—a printing **procedure method,** and a name-returning **function method.** The purpose of the printing method is to print all the information about a particular instance. The name-returning function method will just return the contents of the name field to whatever piece of code called it. Figure 2-9 shows the generic method definition for all three company class methods.



```
begin
  subtract ░░░ 's yearly expenses field
    from ░░░ 's yearly sales field
  return result
end
```

```
begin
  return ░░░ 's name field
end
```

```
begin
  print ░░░ 's name field
  print ░░░ 's number of employees field
  print ░░░ 's yearly sales field
  print ░░░ 's yearly expenses field
  call ░░░ 's yearly profit method
  print result
  call ░░░ 's competitor's returnname method
  print result
end
```

Figure 2-9
The completely defined company object class

You've seen the yearly profit method in Figure 2-5. The name-returning method is just as simple. It merely returns the contents of the name field as its result.

The printing method is also fairly straightforward. Let's examine it in detail, but first let's introduce some terminology.

When you call a method, the following steps occur:

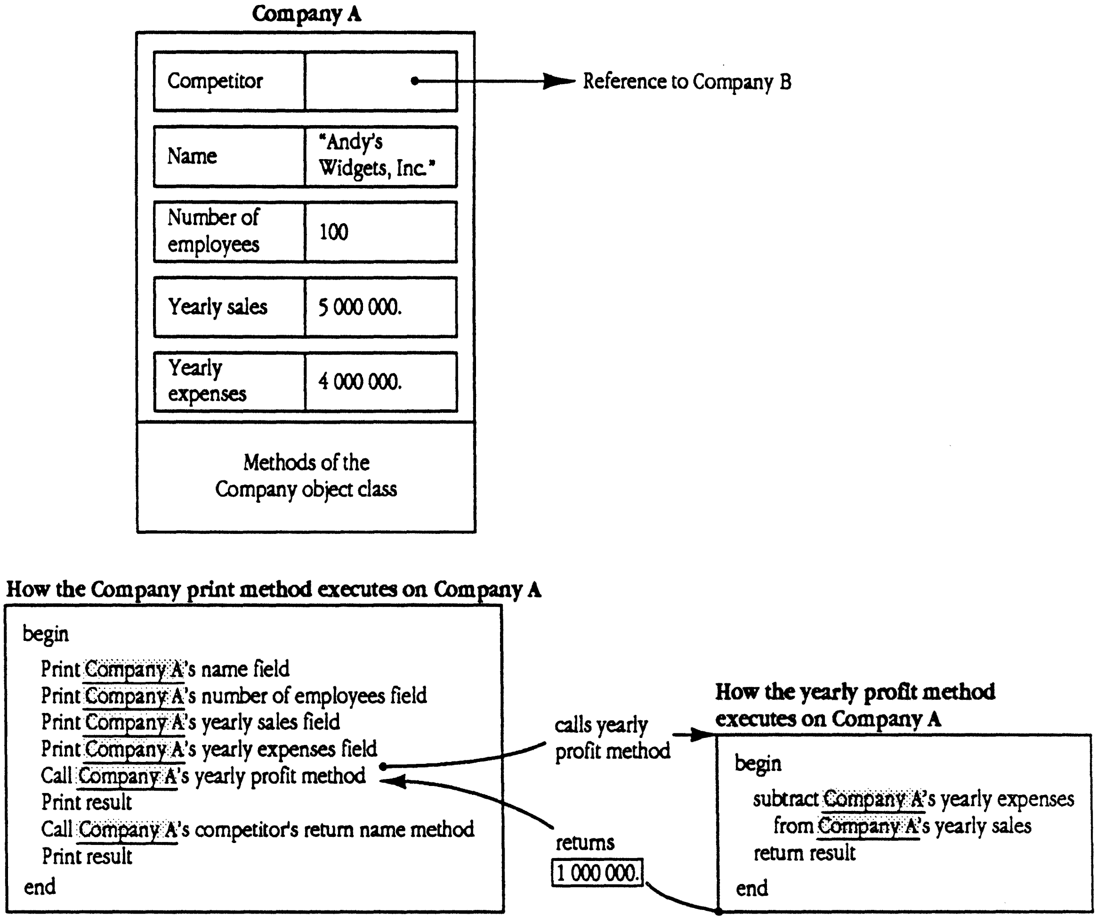
- You must send a message to a particular instance, telling that instance to execute one of its methods.
- That instance then calls the requested method, which was defined for the entire object class, but which will execute specifically on this instance's data.

The instance that you send the message to, which then calls the requested method, is referred to as the **calling instance.**

The first four lines of the printing method print out the values *of the calling instance's* fields. The next two lines call the yearly profit method *of the calling instance*, and then print the result of that method call. The next two lines call the return name method of the instance referred to by the competitor field and then print the value returned by that method.

Now you might be able to see why it is important to have one object instance be able to refer to another object instance: The last piece of information this method prints is actually information about a different instance—the *name* of its *competitor*. An object instance normally does not have access to other instance's name fields, but only to its own name field. References between object instances allows the methods of one instance to call methods of (or send messages to) another instance.

When the printing method of Company A is called, the first four fields of Company A will be printed. Then the printing method of Company A will call the yearly profit method of itself, as shown in Figure 2-10.


Company A
Reference to Company B
Methods of the Company object class
calls yearly profit method
returns
1 000 000.

Competitor	
Name	"Andy's Widgets, Inc."
Number of employees	100
Yearly sales	5 000 000.
Yearly expenses	4 000 000.

How the Company print method executes on Company A

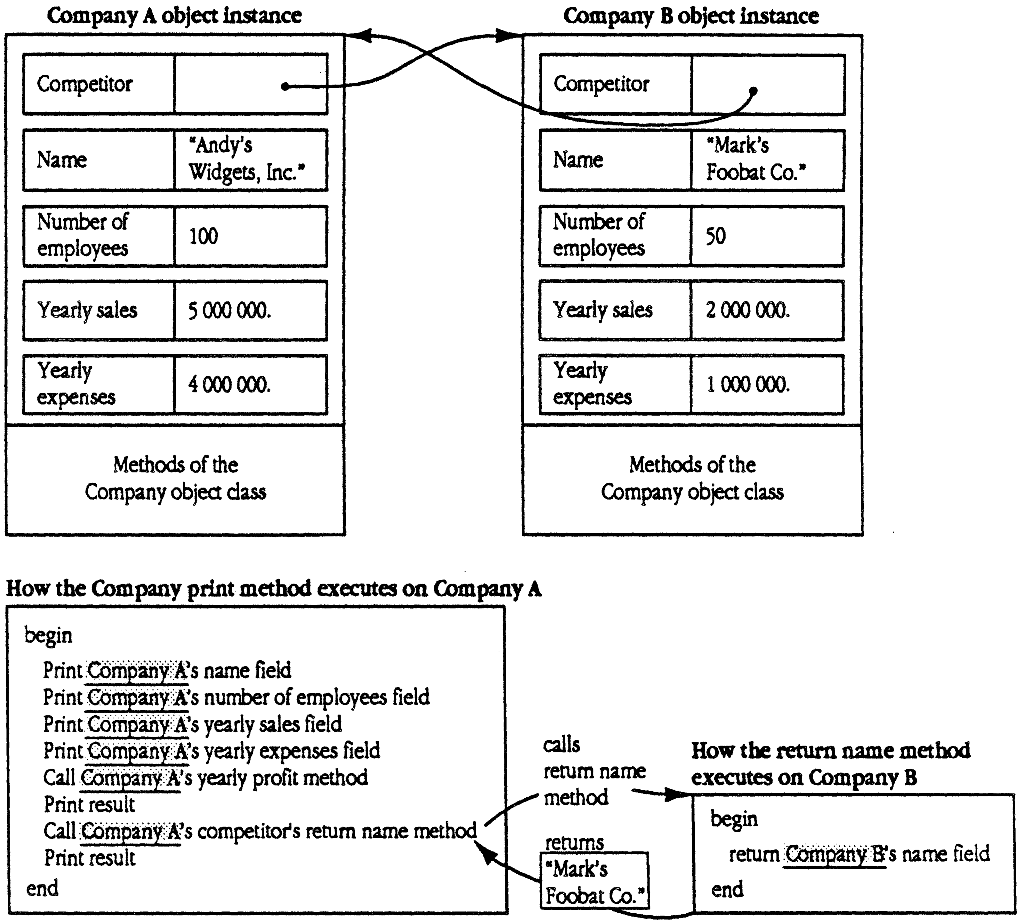
```
begin
  Print Company A's name field
  Print Company A's number of employees field
  Print Company A's yearly sales field
  Print Company A's yearly expenses field
  Call Company A's yearly profit method
  Print result
  Call Company A's competitor's return name method
  Print result
end
```

How the yearly profit method executes on Company A

```
begin
  subtract Company A's yearly expenses
    from Company A's yearly sales
  return result
end
```

Figure 2-10
An object method sending a message to the calling instance

After Company A's print method prints the yearly profit, it calls the return name method of whichever instance is referred to by Company A's competitor field, which in this case happens to be Company B.



Company A object instance
Company B object instance
Methods of the Company object class
Methods of the Company object class
How the Company print method executes on Company A
calls return name method
returns
"Mark's Foobat Co."
How the return name method executes on Company B

Company A object instance	
Competitor	
Name	"Andy's Widgets, Inc."
Number of employees	100
Yearly sales	5 000 000.
Yearly expenses	4 000 000.

Company B object instance	
Competitor	
Name	"Mark's Foobat Co."
Number of employees	50
Yearly sales	2 000 000.
Yearly expenses	1 000 000.

```
begin
  Print Company A's name field
  Print Company A's number of employees field
  Print Company A's yearly sales field
  Print Company A's yearly expenses field
  Call Company A's yearly profit method
  Print result
  Call Company A's competitor's return name method
  Print result
end
```

```
begin
  return Company B's name field
end
```

Figure 2-11
An object method sending a message to a different object instance

Company B's return name method returns the contents of Company B's name field (which is "Mark's Foobat Co.") to Company A's printing method, and then A's printing method prints that result. Company A's printing method is then finished, and it ends and returns to the piece of code that called it. Each of

Company A's fields has been printed, A's profit has been printed, and the name of A's competitor has also been printed.

Notice that Company A knows Company B exists because Company A has a reference to Company B in the competitor field. If Company A's competitor field pointed to some other company object instance (for example, a Company C), then A's printing method would have called the name-returning method of company C.

In other words, the last two lines of Company A's printing method call the name-returning method of whichever object instance is pointed to by Company A's competitor field. In this example, it happens to be Company B.

To finish the big picture of object-oriented programming, let's look at the flow of control in a complete, if arbitrarily simple, program written in an English-like programming language for simplicity.

In the object-oriented environments that you'll be working with there's a main procedure just like the ones found in standard Pascal or C. The purpose of this main procedure is very simple. First, it must allocate space in memory for any **global object instances**—instances that are created by the main program. Then the main program must initialize those instances, typically by calling their initialization method if they have one. Finally, the main procedure must call the methods of those instances to get things rolling.

Usually, the main procedure will only create one global object instance, call its initialization procedure, and then call its first method. The methods of this instance then do the rest of the work. Other objects may be created, and lots of methods will be executed, but the main procedure does nothing else itself.

In the current example, though, the main procedure will create two object instances and call one method of each. Let's expand the company object class so that it includes an initialization method. You'll see a lot of information on initialization methods a little later, so for now just assume there's a working initialization method.

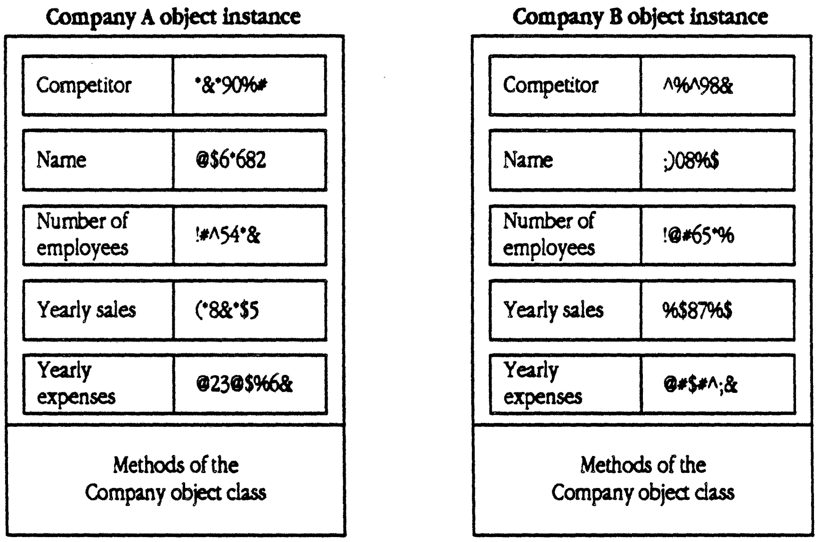
The main procedure will follow a structure like this:

```
main begin
    create an instance of the company class called Company A
    create an instance of the company class called Company B

    call the initialization method of Company A
    call the initialization method of Company B

    call the printing method of Company A
    call the printing method of Company B
main end
```

After the first two lines of this main procedure have been executed, the object instances will exist in memory, as shown in Figure 2-12.



Company A object instance

Competitor	*&*90%#
Name	@$6*682
Number of employees	!#^54*&
Yearly sales	(*8&*$5
Yearly expenses	@23@$%6&

Methods of the Company object class

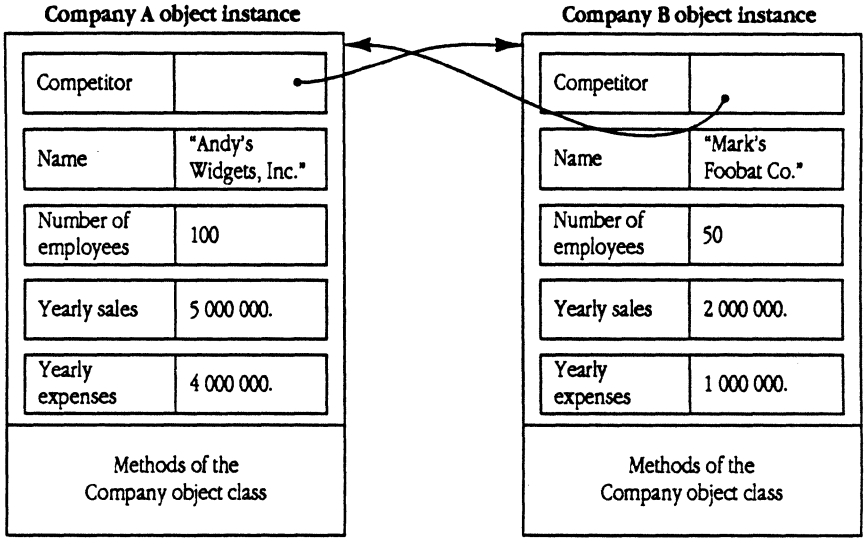
Company B object instance

Competitor	^%^98&
Name	;)08%$
Number of employees	!@#65*%
Yearly sales	%$87%$
Yearly expenses	@#$#^;&

Methods of the Company object class

Figure 2-12
Instances exist in memory

After the initialization methods are called, the instances' fields should all have values. This is true even of the competitor field that links the two instances. Exactly how this initialization method works will be examined in later chapters.



Company A object instance

Competitor	
Name	"Andy's Widgets, Inc."
Number of employees	100
Yearly sales	5 000 000.
Yearly expenses	4 000 000.

Methods of the Company object class

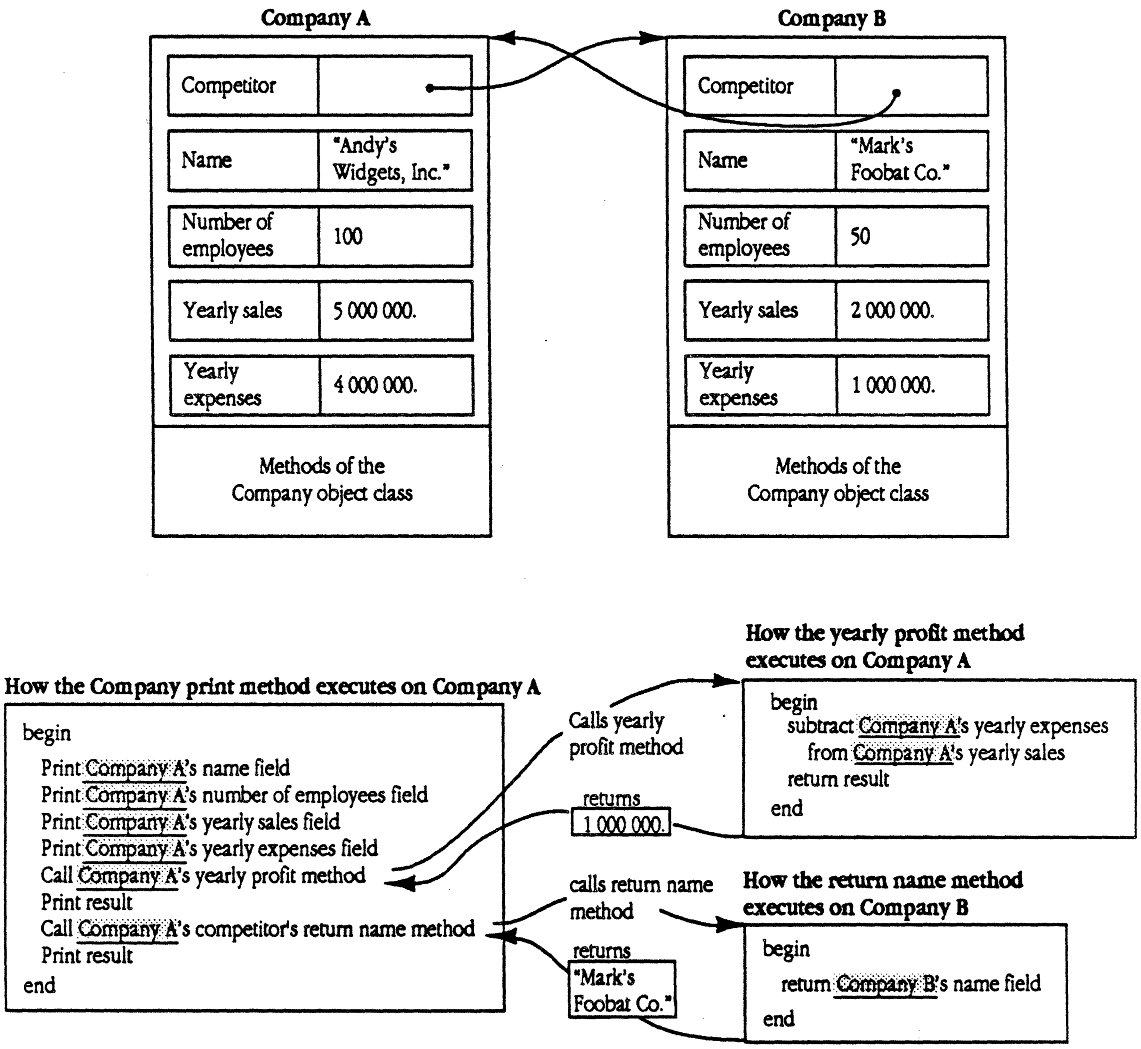
Company B object instance

Competitor	
Name	"Mark's Foobat Co."
Number of employees	50
Yearly sales	2 000 000.
Yearly expenses	1 000 000.

Methods of the Company object class

Figure 2-13
Instances have been initialized

The next step is for the main procedure to call the printing method of Company A. This leads to the flow of control shown in Figure 2-14.



How the Company print method executes on Company A

```
begin
   Print Company A's name field
   Print Company A's number of employees field
   Print Company A's yearly sales field
   Print Company A's yearly expenses field
   Call Company A's yearly profit method
   Print result
   Call Company A's competitor's return name method
   Print result
end
```

How the yearly profit method executes on Company A

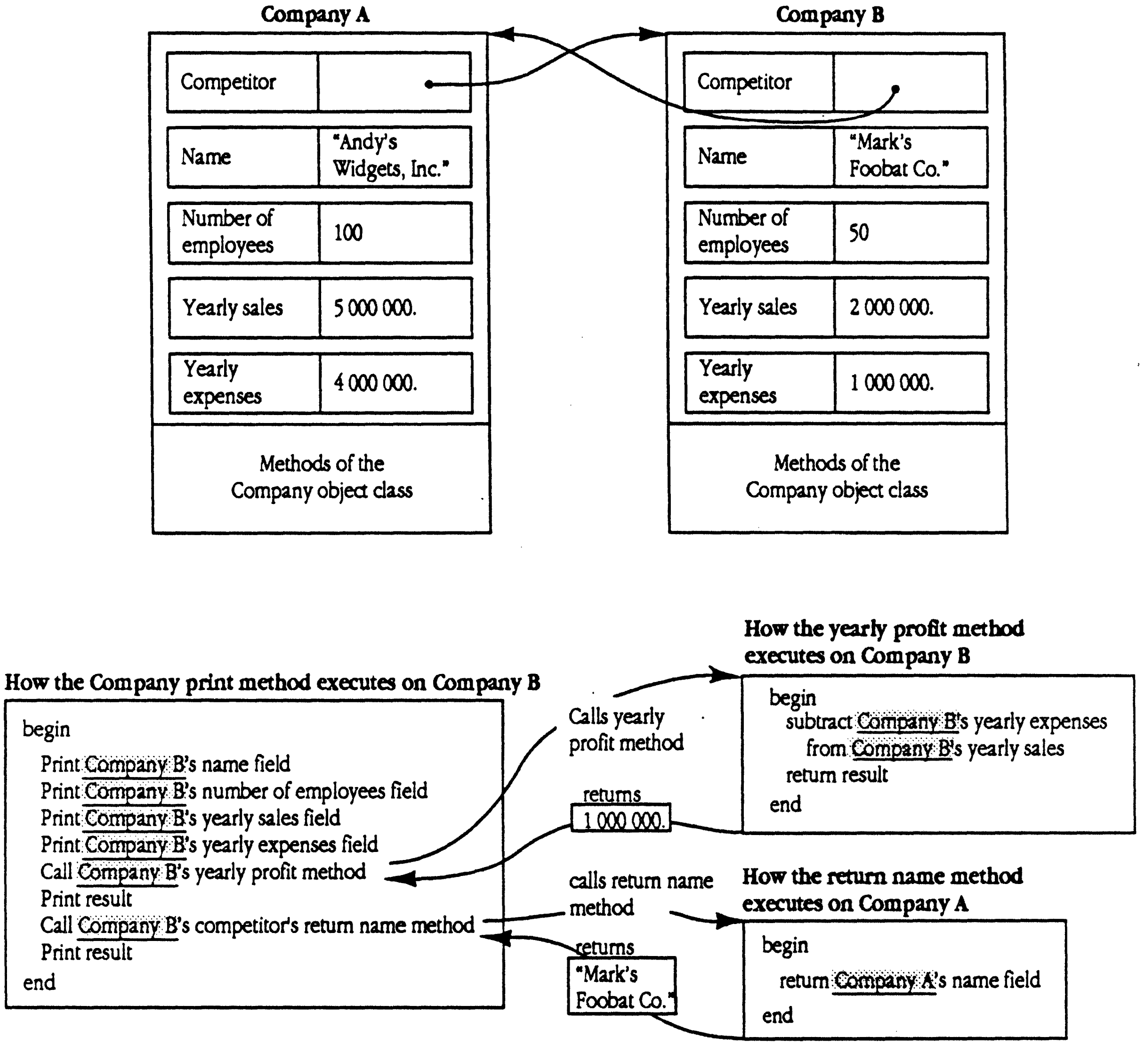
```
begin
  subtract Company A's yearly expenses
     from Company A's yearly sales
  return result
end
```

How the return name method executes on Company B

```
begin
  return Company B's name field
end
```

Figure 2-14
Company A's printing method is called

Finally, the main procedure calls Company B's printing method. This flow of control finishes the program:



How the Company print method executes on Company B

```
begin
  Print Company B's name field
  Print Company B's number of employees field
  Print Company B's yearly sales field
  Print Company B's yearly expenses field
  Call Company B's yearly profit method
  Print result
  Call Company B's competitor's return name method
  Print result
end
```

How the yearly profit method executes on Company B

```
begin
  subtract Company B's yearly expenses
    from Company B's yearly sales
  return result
end
```

How the return name method executes on Company A

```
begin
  return Company A's name field
end
```

Figure 2-15
Company B's printing method is called, and the program is finished

The output from this program (given a sophisticated printing routine) should look something like this:

```
The name of this company is: Andy's Widgets, Inc.
The number of employees is:  100
The yearly income is:        $ 5,000,000.00
The yearly expenses are:     $ 4,000,000.00
The yearly profit is:        $ 1,000,000.00
The competitor is:            Mark's Foobat Co.

The name of this company is: Mark's Foobat Co.
The number of employees is:  50
The yearly income is:        $ 2,000,000.00
The yearly expenses are:     $ 1,000,000.00
The yearly profit is:        $ 1,000,000.00
The competitor is:            Andy's Widgets, Inc.
```

In summary, you've seen the difference between records and objects—that objects have methods as well as data fields.

You've also seen that object classes, like record types, define the generic shape of object instances by defining both the number and size of the data fields as well as the generic structure of the object's methods.

Finally, you've seen three ways in which an object method can be called:

- The main procedure can call a method of a global object instance.
- A method of an object instance can call other methods of that instance.
- A method of an object instance can call methods of other instances, as long as the first instance has an object reference field that refers to the second instance.

Object classes

Object-oriented programs are organized at two levels. The first, which you've already seen, is the organization that exists during runtime between object instances. One instance creates a reference to another instance through object reference fields, and can subsequently make calls to the other instance's methods. This is the flow of control of an object program. The second level of object organization involves object class definitions. Unlike record types, each of which is wholly independent of every other type, object classes can be organized into hierarchies, where classes inherit characteristics from other classes. You can put a great deal of structure in your program by creating well-organized hierarchies of object classes.

The hierarchical structure of object classes is the basis of an object-oriented program's organization. Object classes are organized into hierarchies where each object class can be the **descendant** of another object class, called its **ancestor** class. Each object class can be an ancestor class to any number of descendant classes. Object classes are therefore organized into tree structures.

❖ *Note:* In some object-oriented programming environments an object class can be the descendant of more than one other object class. Although this can be a powerful tool, it can also lead to overly complex class organizations. In the MacApp environment, you will be dealing with classes that descend from only one other class, and all class organizations will be tree hierarchies.

A descendant object class contains not only its own fields and methods, but also those of its ancestor class. Of course, that ancestor class contains the fields and methods of its ancestor class as well. This concept, whereby an object class gains extra fields and methods from its ancestor class, is called **inheritance**.

Inheritance

Let's look at an example of an object class hierarchy by creating an employee object class. An employee object class might have fields for the employee's name and title, as well as a method for printing the information in these fields.

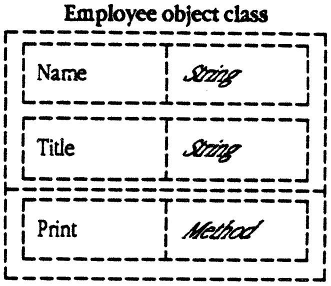


Figure 2-16
An employee object class

This object class, because of its generality, might not include fields and methods for all the important information about each kind of employee that a company might have. There may be many different kinds of employees, and each type of employee could have its own special object class. For example, there could be both contractors and paid-weekly employees. You could make both of these more specific employee classes a descendant class of the more generic employee class. As in this example, descendant classes are **customizations** of their ancestor class.

A contractor employee class might require two extra fields—a contract number field and a contract dollar amount field, for instance.

A paid-weekly employee class might require one extra method—a method for calculating a weekly paycheck.

Of course, both of these new employee classes will inherit the data fields and method of the ancestor employee class.

Once these two classes are added, the hierarchy starts to look like a tree.

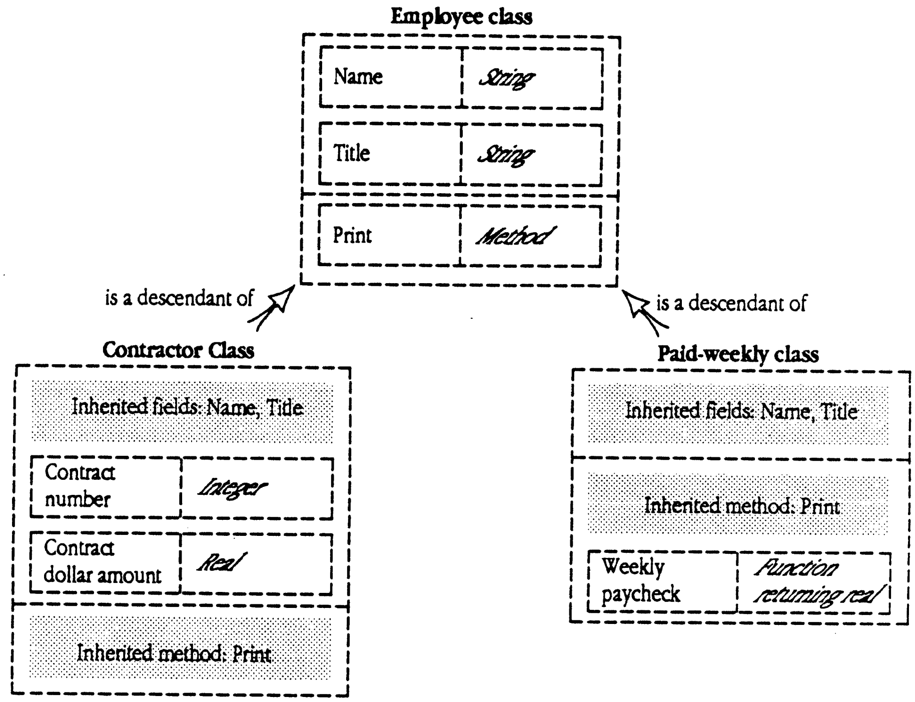


Figure 2-17
A simple hierarchy of object classes

In the body of the program, you will be able to declare object instances belonging to any of these three object classes. An instance of the employee class will have two data fields and one method. An instance of the contractor class will have four data fields and one method. To be sure, two of these fields and one of these methods are inherited from the employee class. Nevertheless, object *instances* of the contractor class are not able to distinguish between fields and methods that are inherited and those that are not. It's almost as if you declared three separate object classes, but remembered to include the fields and methods of the employee class in its two descendant classes. Object-oriented programming *helps by remembering to include them for you.*

Of course, inheritance of fields and methods applies through more than one level of ancestry. For example, you can now declare descendant classes of the paid-weekly employee class. These classes will inherit all the fields and methods of the paid-weekly class, including those that were already inherited from the employee class.

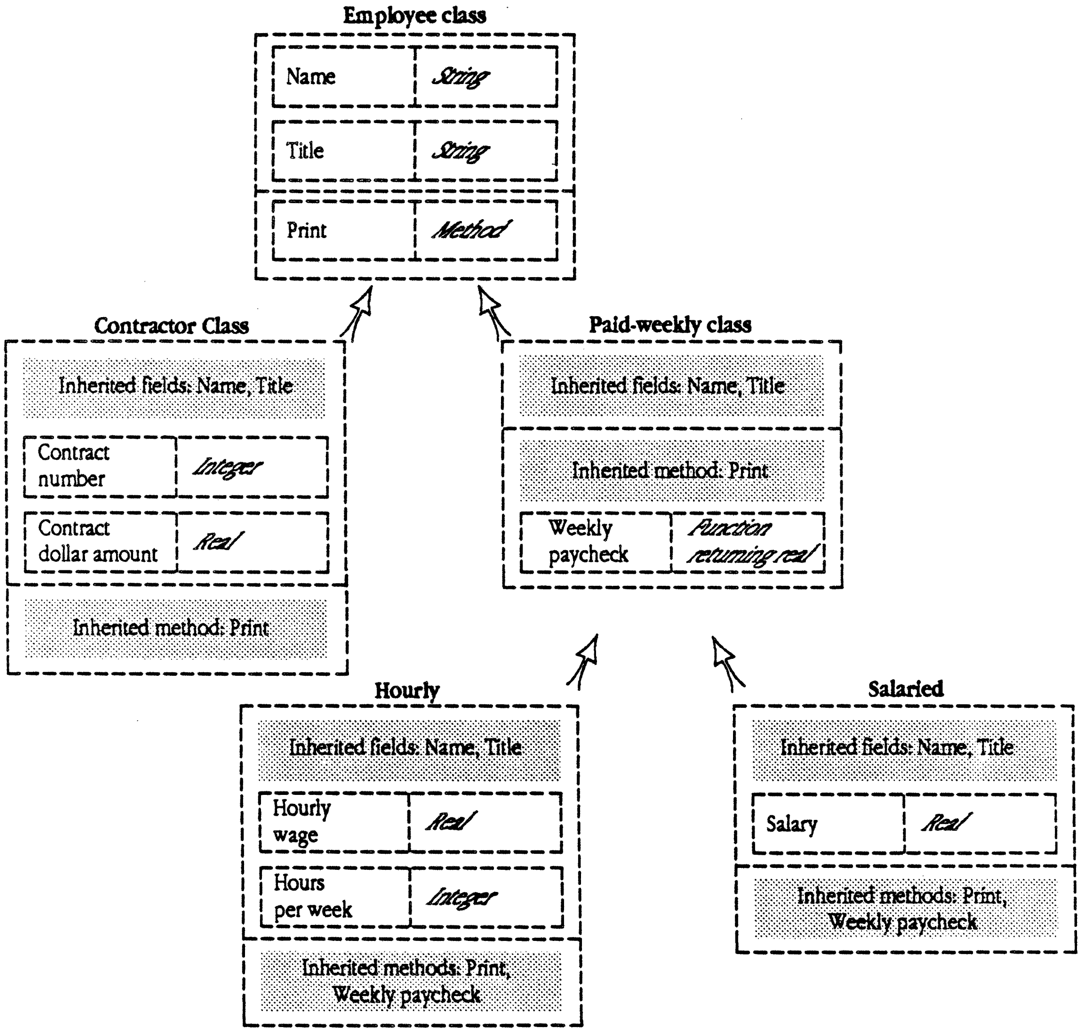


Figure 2-18
Multiple levels of inheritance

Once again, while you are using object instances, you cannot in any way distinguish between those fields and methods that are inherited and those that are not.

In the above example, the employee class and the paid-weekly class are ancestors of the hourly and salaried classes. The paid-weekly class is the **immediate ancestor** of the hourly and salaried classes. The hourly class is a descendant of the employee and paid-weekly classes, and is the **immediate descendant** of the paid-weekly class.

Method definitions

An important part of the class definition is the method definition. Remember that when a method is defined it belongs to an entire class of objects. Therefore the definition must be in a general form. However, only methods belonging to actual object instances are called. When a method of some instance is called, it operates on the data of that particular instance.

The method definition defines the *generic* method that applies to an entire class of objects. Therefore the method definition cannot refer to the fields or methods of any particular instance. Instead, the method definition must refer to data fields and methods by their generic class name. When you call a method, you must specify which instance the method is to operate on. This instance is referred to as the **calling instance.** As the method executes, references to the generic field and method names in the method definition will be automatically replaced by references to the specific fields and methods belonging to the calling instance.

As an example, let's define the generic form of the print method for the employee class. This method is supposed to print the values of the instance's fields. Figure 2-19 shows the generic method definition:

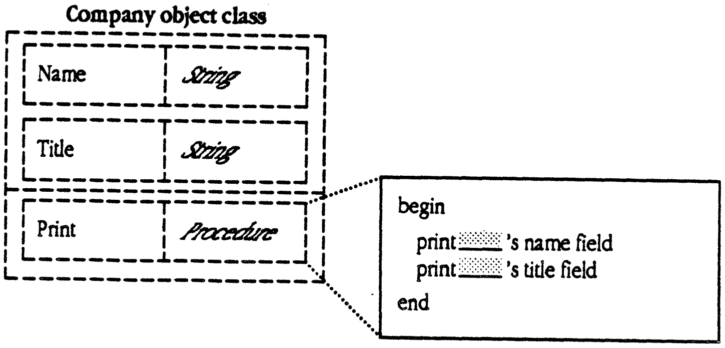


Figure 2-19
The generic method definition of the print method of the employee object class

Since this is a **method definition**, it applies generically to an entire class of objects. Whenever this method is called, however, it is called to execute on the specific data of some particular instance—the calling instance.

Notice that this method definition references both the name field and the title field of the employee object class. Of course, neither of these fields has any value until an employee object instance is created. Once an employee instance is created, you can call its print method, and the print method for the employee class will be executed specifically on the data of that instance. Part of the power of object-oriented programming is that you don't have to create a specialized version of each method for every employee instance that you create. The "specialized" version will be executed for you automatically, based on your generic method definition.

Override methods

The next section, "Object Instances," deals further with the distinction between methods as they are defined and methods as they are executed. Before that, however, there is another issue concerning method definitions and the object class hierarchy.

A class inherits the methods of its ancestor class. Usually this is a good thing; that's why you make one class a descendant of another class. Remember, though, that a descendant class is a *customization* of an ancestor class. Therefore it is sometimes desirable to have a descendant class that inherits the methods of its ancestor, but somehow customizes those methods. Object-oriented programming allows that flexibility.

There are two ways a descendant class can change an inherited method. It can either have its own completely rewritten version of the method, or it can add a little to the inherited method. Either way, the new method is called an **override** method. The ancestor's method definition is **overridden,** and this process is called **overriding** a method.

Sometimes the override method needs to be so different that it cannot even use the inherited method. In fact, it is quite common for a method to do absolutely nothing, requiring that every class that inherits that method completely override it.

The reasoning behind is very simple. Consider the paid-weekly class from the earlier example. This class has a weekly paycheck method. However, it has no fields that could possibly help in calculating the size of a weekly paycheck. But, the descendant classes of the paid-weekly employee class (hourly employees and salaried employees) do have fields that allow calculation of a weekly paycheck.

The paid-weekly employee class was never meant to be instantiated: the program will not have any instances of this class. It was created solely to help organize the class hierarchy, and therefore it is called an **abstract object class.** Abstract object classes are quite common in object-oriented programming—they provide a common framework of fields and methods for their descendants. Frequently, you will have to override the methods of abstract object classes.

Notice that ancestor classes are not synonymous with abstract classes: the employee class, at the very top of the class hierarchy, will be instantiated later. Of course, the compiler will allow you to instantiate any class. It's your decision which classes are to be instantiated and which are to be abstract.

In the case of the weekly paycheck method, you simply leave the definition of the inherited version blank, and write the override methods from scratch.

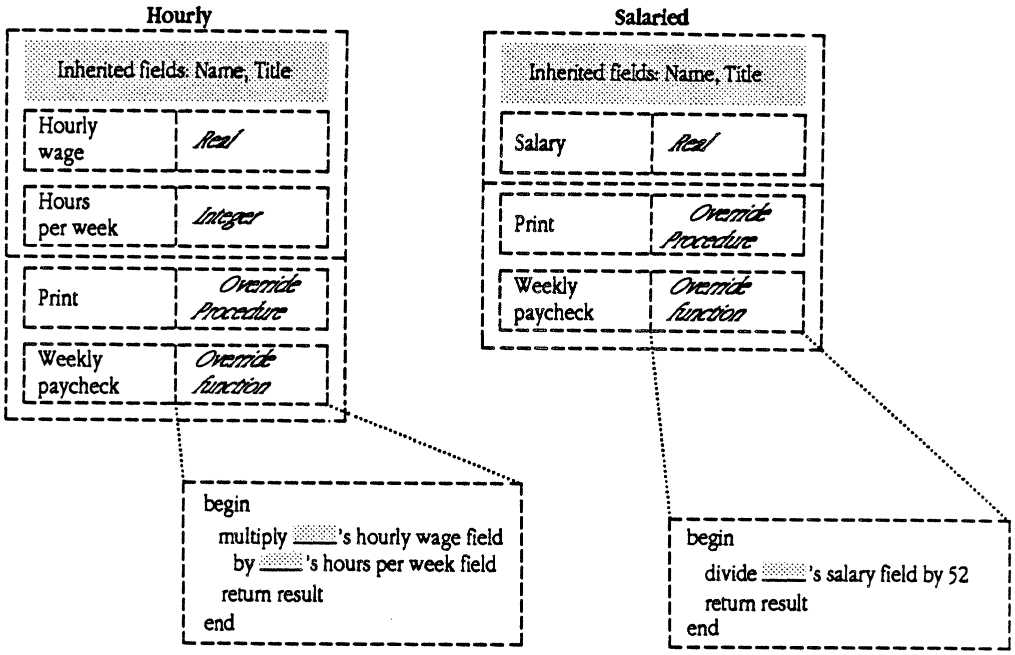


Figure 2-20
Override methods that do not call the inherited method

Sometimes the override method does not need to completely replace the inherited method, but rather only augment it. For example, take the print method defined in Figure 2-19. This method prints only the values of the name and title fields. This is fine for instances of the employee class; however, this method is inherited by both the contractor and the paid-weekly classes. In the case of the paid-weekly class, the inherited method will be fine because the paid-weekly class doesn't have any new fields. However, the contractor class has two fields that the employee class does not: the contract number and the contract dollar amount. The contract employee class, then, needs a print method that augments the print method of the employee class.

This override method ought to:

- call the inherited method, which will print the name and title fields
- print out the contract number and contract dollar amount fields itself

This is exactly what you can define the override method to do.

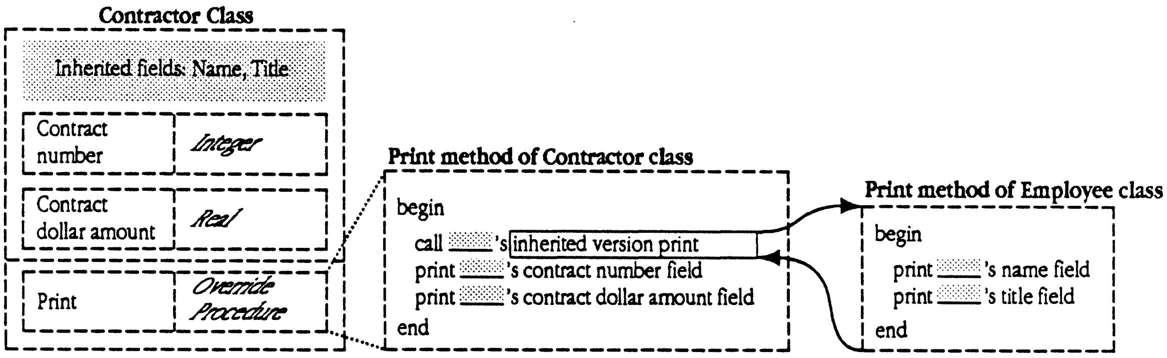

Contractor Class
Inherited fields: Name, Title
Contract number
Integer
Contract dollar amount
Real
Print
Override Procedure
Print method of Contractor class
begin
call ▨'s inherited version print
print ▨'s contract number field
print ▨'s contract dollar amount field
end
Print method of Employee class
begin
print ▨'s name field
print ▨'s title field
end

Figure 2-21
An override method that calls the inherited method

Now that the class hierarchy is defined, it is possible to create object instances and examine how inherited and overriden methods work during runtime.

Object instances

To create an object class, you declare the methods and fields of that class and define the methods. When you create an instance of that class, the compiler will allocate space in memory for the fields of that instance and set up a mechanism for the instance to call its methods when it receives messages.

An instance method

To begin, let's create an object instance of each employee class defined except for the paid-weekly class, which is an abstract class used only for hierarchical organization.

For example, you might want to create four employee instances:

- Employee A, a contractor (of class contractor)
- Employee B, an hourly employee (of class hourly)
- Employee C, a salaried employee (of class salaried)
- Employee D, an employee that doesn't fit any of the above categories (of the most generic class—employee)

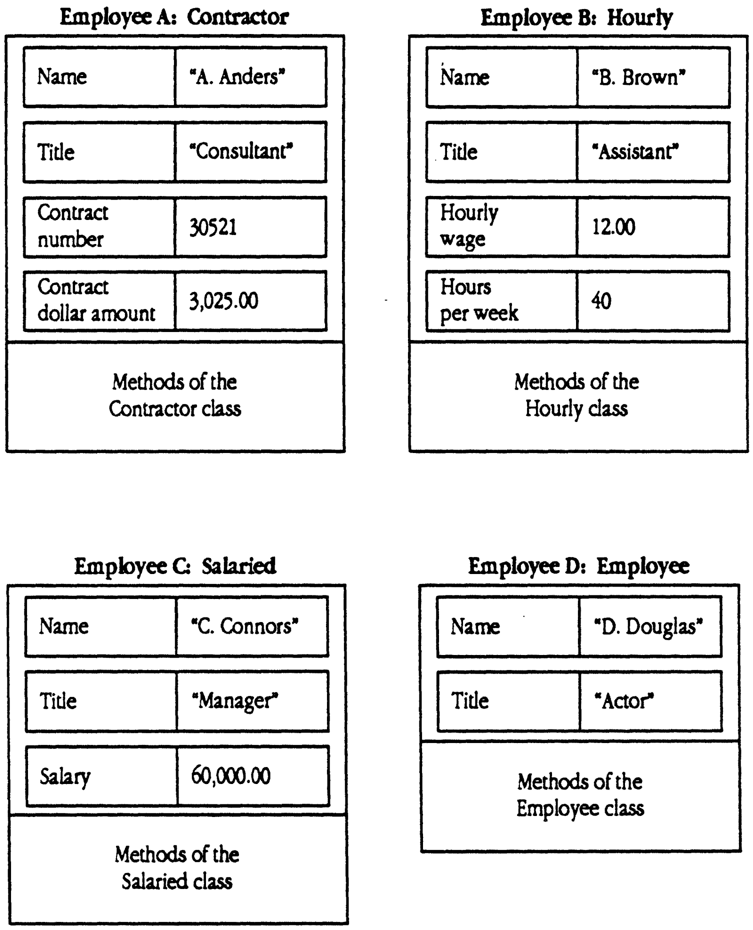


Figure 2-22
The four employee object instances

An object instance is said to be a **member** of its class and of all its ancestor classes. Therefore Employee A is a member of both the contractor class and the employee class. Employee B is a member of the hourly class, the paid-weekly class, and the employee class. Employee C is a member of the salaried class, the paid-weekly class, and the employee class. Employee D is a member of only the employee class.

Perhaps you might think that because the contractor class is a descendant of the employee class that Employee A is a descendant of Employee D. In fact, the Employee A instance has no relationship to Employee D, since these are object instances and only object classes have that sort of hierarchy. Object *instances* can only be related to each other through the use of **object reference fields.**

These instances can have three types of methods: methods specific to their class, methods inherited from an ancestor class, and inherited methods that are overridden.

Employee D, which is of the employee class, has a print method that is defined by the employee class. When the print method of Employee D instance is called, it's as if a version of the method that is specialized for Employee D is actually executed.

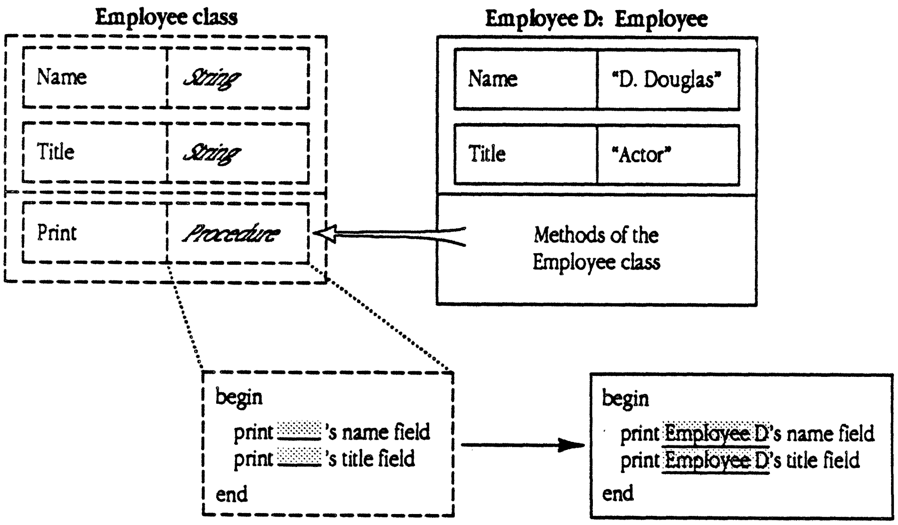


Figure 2-23
The print method definition and a specific instance

When the print method of Employee D is called (by the main program, for instance), it executes, referencing and printing the data fields of Employee D, and then returns control to the main program.

The case with methods that are inherited from an ancestor class is exactly the same. When an inherited method of an instance is called, the method executes specifically on that instance's data. For example, if you were to create an object instance (let's say Employee E) of the paid-weekly employee class and call the print method (which is inherited from the employee object class and not overridden), the method that would execute would look the same for Employee E as the print method above looks for Employee D.

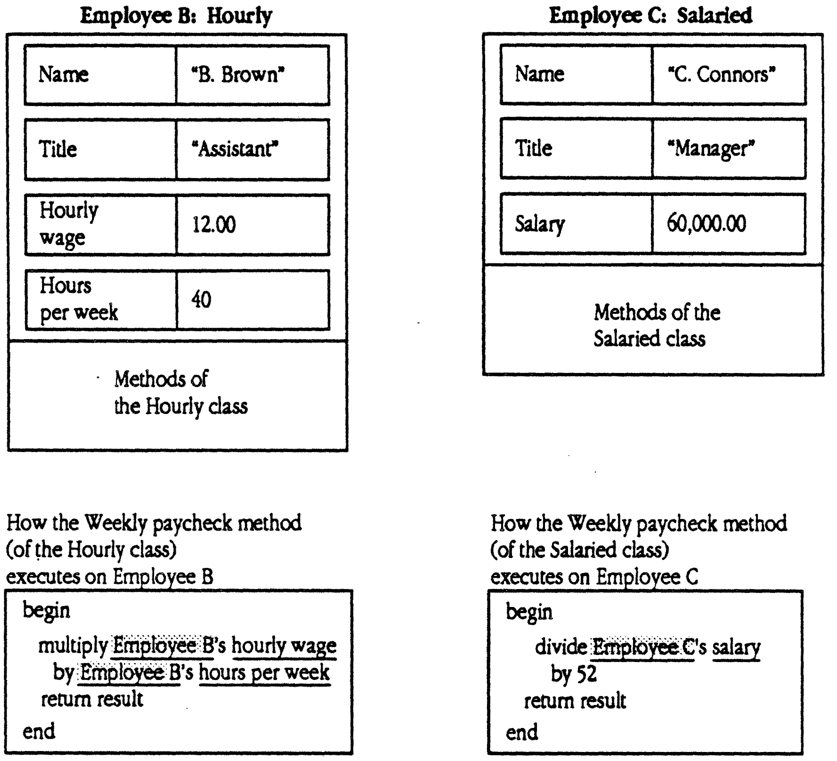
Override methods

Override methods are not much trickier. Remember that there are two kinds of override methods. The first kind are override methods that completely change the inherited method. These methods do not call their inherited method at all. The second kind are override methods that add something to the inherited method. These methods call their inherited method, usually as their first or their last action.

Override methods that don't call their inherited method

This category is the simpler of the two. If an override method completely overrides the inherited method, then it acts just like an entirely new method. For example, hourly employees and salaried employees both have a weekly paycheck method that completely overrides the weekly paycheck method of the paid-weekly employee class, as shown in Figure 2-20.

Figure 2-24 shows how a call to Employee B's weekly paycheck method is quite a different thing from a call to Employee C's weekly paycheck method. First of all, the methods actually execute on the data of the calling instance. Secondly, object-oriented programming makes sure that the correct version of the method is executed. For Employee B the correct version of the weekly paycheck method is the Hourly class version. For Employee C the correct version is the Salaried class version.



Employee B: Hourly

Name	"B. Brown"
Title	"Assistant"
Hourly wage	12.00
Hours per week	40

Methods of the Hourly class

Employee C: Salaried

Name	"C. Connors"
Title	"Manager"
Salary	60,000.00

Methods of the Salaried class

How the Weekly paycheck method (of the Hourly class) executes on Employee B

```
begin
  multiply Employee B's hourly wage
    by Employee B's hours per week
  return result
end
```

How the Weekly paycheck method (of the Salaried class) executes on Employee C

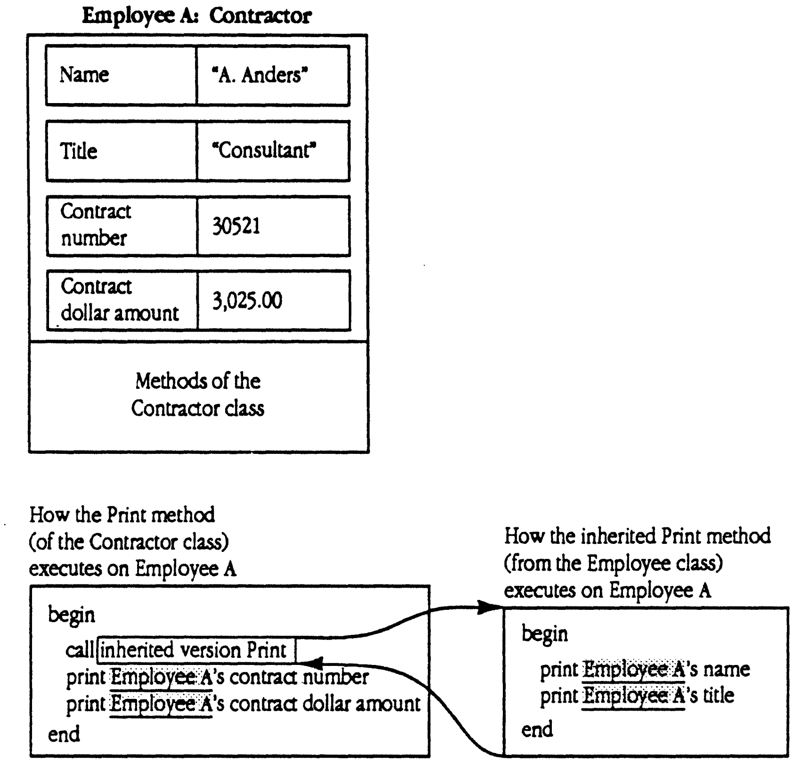
```
begin
    divide Employee C's salary
      by 52
  return result
end
```

Figure 2-24
The weekly paycheck method of the hourly and salaried employee classes

As before, when either of these methods is called, it executes linearly and then returns the real result to the calling line of code.

Override methods that do call their inherited method

Override methods that call their inherited method are only a little more complex. When such an override method is called, the method operates on the calling instance's data. Then, when the override method calls the inherited method, the inherited method also operates on the calling instance's data. For example, the print method of the contractor class overrides the print method of the employee class, but it also calls that method. When the print method of Employee A is called, the code in Figure 2-24 is what executes.


Employee A: Contractor
Name
"A. Anders"
Title
"Consultant"
Contract number
30521
Contract dollar amount
3,025.00
Methods of the Contractor class
How the Print method (of the Contractor class) executes on Employee A
How the inherited Print method (from the Employee class) executes on Employee A

```
begin
   call inherited version Print
   print Employee A's contract number
   print Employee A's contract dollar amount
end
```

```
begin
   print Employee A's name
   print Employee A's title
end
```

Figure 2-25
The specialized version of a method that calls an inherited method

Take notice of these three points:

- Employee A's print method is called; therefore the print method that executes operates specifically on Employee A's fields.
- Employee A's print method makes a call to the print method inherited from the ancestor object class—the employee class.
- This inherited print method also operates only on Employee A's fields.

When the print method of Employee A is called, it first calls the inherited print method, which prints the name and title fields of Employee A. Then the override print method prints the contract number and contract dollar amount fields of Employee A itself. Finally, the method returns control to its caller.

As another example, consider the print methods of the hourly and salaried employee classes. These are both override methods, overriding the print method of the paid-weekly employee class which is inherited from the employee class.

When the print methods of Employee B and Employee C are called, the methods that execute might look like those in Figure 2-26.

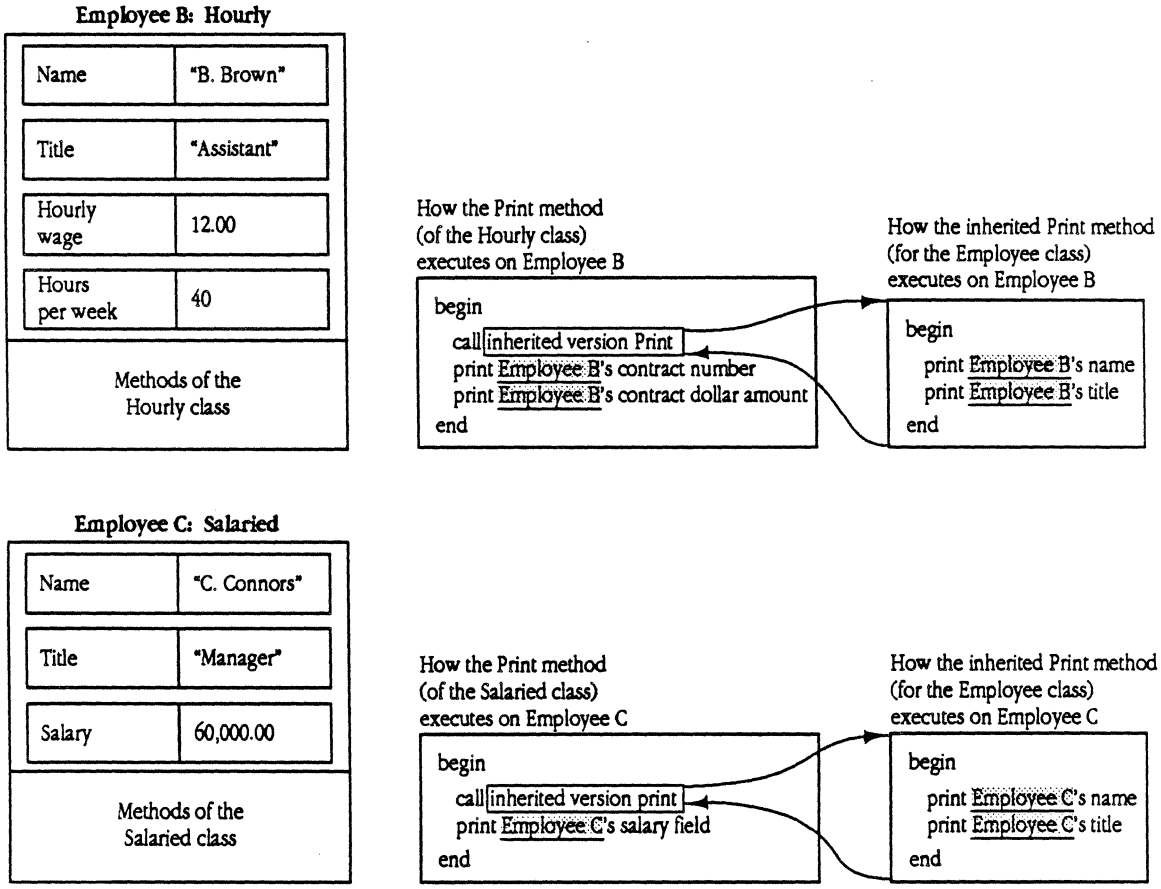

Employee B: Hourly
Name
"B. Brown"
Title
"Assistant"
Hourly wage
12.00
Hours per week
40
Methods of the Hourly class
How the Print method (of the Hourly class) executes on Employee B
begin
call inherited version Print
print Employee B's contract number
print Employee B's contract dollar amount
end
How the inherited Print method (for the Employee class) executes on Employee B
begin
print Employee B's name
print Employee B's title
end
Employee C: Salaried
Name
"C. Connors"
Title
"Manager"
Salary
60,000.00
Methods of the Salaried class
How the Print method (of the Salaried class) executes on Employee C
begin
call inherited version print
print Employee C's salary field
end
How the inherited Print method (for the Employee class) executes on Employee C
begin
print Employee C's name
print Employee C's title
end

Figure 2-26
More examples of override methods called by specific instances

Privacy between instances

The employee object instances discussed in this section do not have object reference fields like the company object instances did in the first section of this chapter. This means that no employee object instance knows of the existence of any other employee object instance. Because of this, no method of Employee A, for example, can make reference to any field or call any method of Employee B.

In the company object example, however, instances did have object reference fields. One instance, Company A, had a link to another instance, Company B, and vice versa. In that example, the methods of one instance could reference the fields or call the methods of another instance.

In Object Pascal, one instance can access any field or method of another instance, as long as the first instance has a reference to the second instance. Generally the data stored in an instance's fields is considered private, which means that an instance can examine and manipulate its own data (in its methods) but should not examine or manipulate the data in another object instance. This means that in method definitions you should only directly examine or manipulate the data fields of the calling instance.

If one instance needs some information from another instance, the first can call a method of the other instance asking it to return the value of the desired field. This is how the problem was handled in the company example. When Company A wanted to print its competitor's name, it called the return name method of Company B, which allowed Company B to access its own name field, and return the value as a function result. This way, Company A didn't look into Company B's name data field directly.

In C++, you are allowed to specify which fields and methods are accessible to other instances and which should be kept private.

Ramifications of object-oriented programming

Object-oriented programming has several advantages over conventional programming. In conventional programming, you typically must meddle with code all over your program to add functionality. In object-oriented programming, you simply create a new object class and instantiate it. The number of changes necessary to hook new objects into your existing program is quite small.

Most Macintosh programs are similar in many ways because they follow Apple's user interface guidelines for the Macintosh. And of course these guidelines are what make the Macintosh so easy to use. Since Macintosh applications have an unusual number of similarities, the engineers at Apple have used object-oriented programming techniques to put together an extremely powerful but completely generic program for the Macintosh. That program is MacApp. MacApp implements all the standard user interface features or creates an framework for you to implement them yourself. It is your job as a Macintosh programmer to create the objects not included in MacApp that do what you want your program to do.

You might be thinking that MacApp will coerce you into doing things a way you don't want to, but that's not true. Although MacApp does most things right all by itself, sometimes you'll want something better, or something done a different way. In such cases, you only need override the methods of MacApp you don't want. In general, the more standard your application's user interface is, the more of MacApp you can use unaltered; and the more unusual your application is, the more of MacApp you will have to override.

Object-oriented programming, then, allows you to organize your programs into objects. Object-oriented programming also allows for easy expansion of programs—often it's as simple as adding another object and changing a constant or two. This ease of expansion allows MacApp to implement a lot of your program for you, freeing you to create the specific objects that make your program unique. Finally, this generic application framework doesn't box you into anything, as object-oriented programming allows you to override any part of the framework easily, without even deleting a line of code.

Chapter 3
Object Pascal

Now that you've seen the wonders of object-oriented programming, you must be eager to learn the syntax of a real object-oriented language so you can get to work. This chapter specifies the syntax of Object Pascal, the language in which MacApp is written.

Let's start with the overview of how to organize your Object Pascal program into files.

Object Pascal file organization

The format of an Object Pascal program is similar to the format of a conventional Pascal program. The program must have a header, followed by the constant declarations, type definitions, and variable declarations. These are followed by procedure and function definitions, and finally, the main procedure.

Of course, there are new elements in an object-oriented program, namely class definitions and method definitions. The standard format still applies, however.

```
PROGRAM ProgramName;

CONST <constant declarations>

TYPE  <type and class declarations>

VAR   <global variables>

<procedure, function, and method definitions>

BEGIN
    <main program code>
END.
```

Since Macintosh programs are typically complex, this linear organization leads to unmanageably large files. The solution is to break the organization into two files, using Pascal **units**, which are discussed fully in the MPW Pascal Reference Manual. As a quick reminder, a Pascal unit is a separate file that has two parts: an **interface** and an **implementation.** The declarations and definitions in the interface part are available to any file that **uses** the unit. The implementation part contains implementation information private to the unit, for example, procedure and function definitions.

The convention for MacApp programs is to break them into two parts: a unit that defines the object classes and methods, and a main program that uses the unit.

The main file, which stores the global declarations and the main program, is organized like any Pascal program. It makes reference to the unit where the object-related code is stored, by means of the uses statement.

```
PROGRAM ProgramName;

USES ObjectUnitName;

CONST <constant declarations>

TYPE <type definitions>

VAR  <global variables>

<procedure, function, and method definitions>

BEGIN
    <main program code>
END.
```

The unit file defines the classes and methods that will be used by the program, and is typically organized like this:

```
UNIT ObjectUnitName;

INTERFACE

CONST <public constant declarations>

TYPE  <class definitions>


IMPLEMENTATION

CONST <private constant declarations>

VAR   <private variables global to the unit>

<method definitions>

END.
```

As this unit becomes more unwieldy itself, it is typically divided into two files. The implementation part is usually removed to another file, which is **included** in the first one. When this is done, the interface file has the form:

```
UNIT ObjectUnitName;

INTERFACE

CONST <public constant declarations>

TYPE  <class definitions>


IMPLEMENTATION

{$I ImplementationFileName}

END.
```

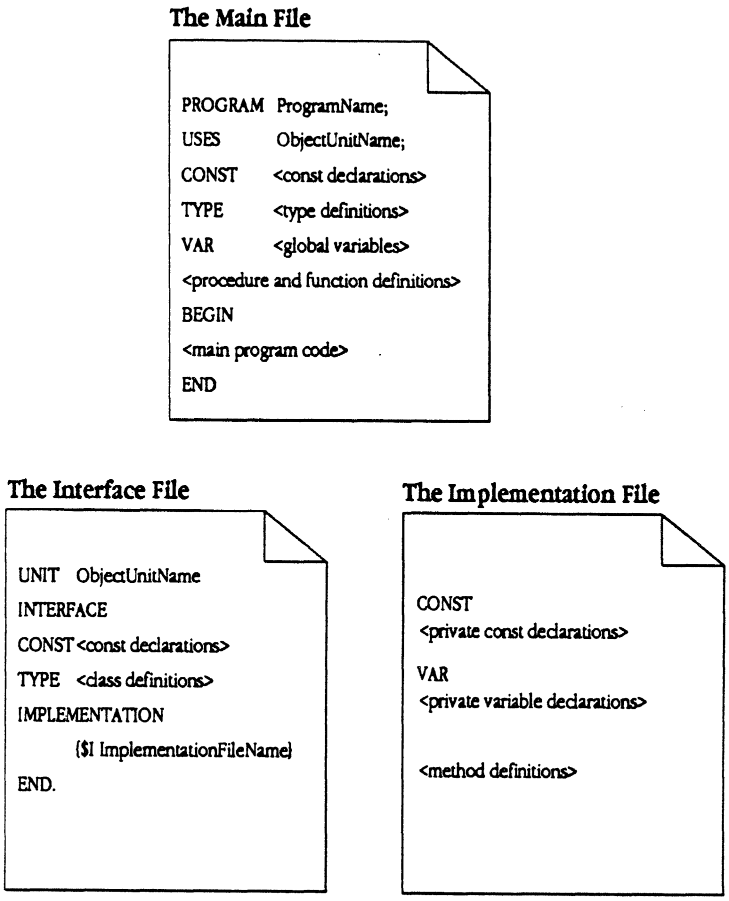
The implementation file, then, looks like this:

```
CONST <private constant declarations>

VAR   <private variables global to the unit>

<method definitions>
```

These three files—the main file, the unit interface file, and the unit implementation file—constitute the entire object program. You'll see this pattern of files repeatedly when working with MacApp.



The Main File

```
PROGRAM  ProgramName;
USES     ObjectUnitName;
CONST    <const declarations>
TYPE     <type definitions>
VAR      <global variables>
<procedure and function definitions>
BEGIN
<main program code>
END
```

The Interface File

```
UNIT  ObjectUnitName
INTERFACE
CONST<const declarations>
TYPE  <class definitions>
IMPLEMENTATION
        {$I ImplementationFileName}
END.
```

The Implementation File

```
CONST
<private const declarations>
VAR
<private variable declarations>

<method definitions>
```

Figure 3-1
Typical file organization of an Object Pascal program

As mentioned earlier, you are not limited to three files for your code. You may implement any number of units and any number of implmentation files for each unit. In this manner you can keep your files to a reasonable size—and remember, units can be compiled separately. Figure 3-2 shows a possible organization of a large Object Pascal program.

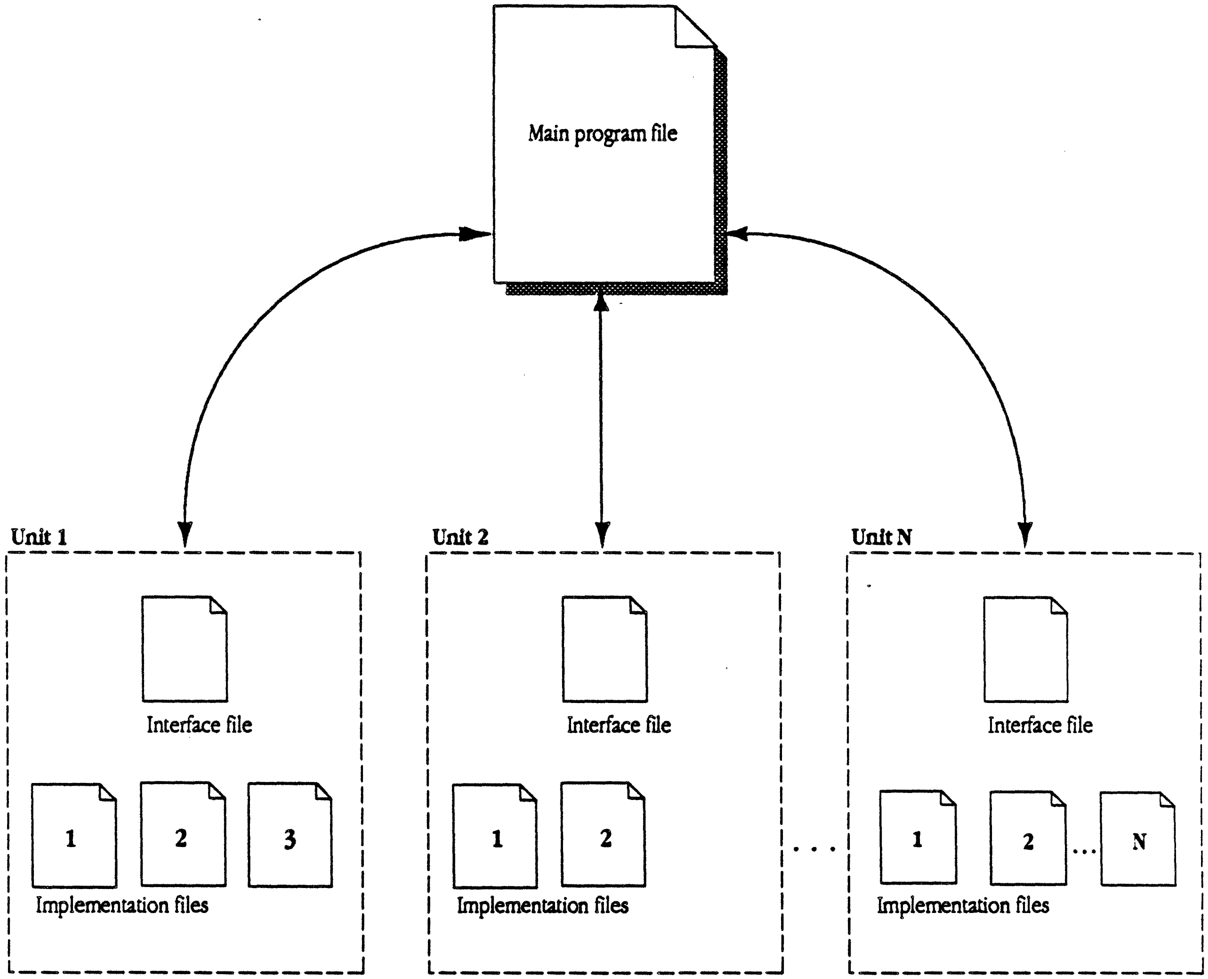


Figure 3-2
Possible file organization of a large Object Pascal program

Object class definitions

Let's start to fill in some of the blanks in those file specifications. In the interface part of the unit, there's a section called <class definitions>. Defining an object class in Object Pascal is similar to defining a record type. First, you need to name the object class, specify that it is an object class, and then declare the fields and methods. The specific syntax looks like this:

```
TYPE
        ObjectClassName = OBJECT
                {field declarations}
                {method declarations}
        END;
```

OBJECT is a reserved word that acts almost exactly like the reserved word RECORD. As you can see, the object class definition above has the same structure as a record type definition.

It is the convention in MacApp to begin object class names with a T. (This stands for Type—remember "object class" means "object type.") For example, if you wanted to implement the company object class from chapter 2, you might make the following declaration:

```
TCompany = OBJECT

        {TCompany's field declarations}
        {TCompany's method declarations}

END;
```

In this declaration, object class TCompany has no ancestor. By convention in MacApp, if an object class has no ancestor, then it is made a descendant of class TObject. TObject is a class with no fields and only a few "utility" methods that will come in handy later. For now, just remember that object classes with no immediate ancestor should use TObject as their immediate ancestor.

In Object Pascal, the way to declare the ancestor of an object class is simple. Since the TCompany class should descend from the TObject class, you can simply rewrite the TCompany definition like this:

```
TCompany = OBJECT(TObject)

        {TCompany's field declarations}
        {TCompany's method declarations}

END;
```

Notice the TObject class name appears in parentheses after the keyword OBJECT. This defines the class TCompany as an immediate descendant of the TObject class.

Field declarations

Now that the object class has a name and an ancestor, you need to declare the object class's fields. As object classes are named with an uppercase T by convention, object fields are named with a small f. There's actually a good reason for this, as you will see in the next section. Object fields are declared in precisely the same manner as record fields. For example,

```
TCompany = OBJECT(TObject)

        fMainCompetitor:  TCompany;
        fName:  string;
        fNumberOfEmployees:  integer;
        fYearlySales:  real;
        fYearlyExpenditures: real;

        {TCompany's method declarations}

END;
```

TCompany has five fields. The first field, fMainCompetitor, is a reference to another company object. You'll see more on object references later in this chapter, but for now, remember that an object reference is similar to a pointer to an object instance. An object reference field, like fMainCompetitor, allows one object instance to call the methods of another object instance. As you can see in the example above, object reference fields are declared to be of the class that the field refers to. In this case, fMainCompetitor is declared to be of class TCompany, meaning that fMainCompetitor will be used to refer to instances of class TCompany.

The other four fields are straightforward: they exactly mimic the declaration of record fields.

Method declarations

The class definition includes field declarations and method declarations. Method *declarations* should not be confused with method *definitions*. A **method declaration** simply states the name of the method, its parameter list, whether it is a procedure or function, and whether it is an override method. The **method definition** actually defines the method—contains the code that describes the method's behavior. Method declarations are a lot like forward declarations in Pascal. They describe the method's nature, but do not define the method's behavior.

The TCompany class has three methods. Let's look at the method declaration line for each of them.

```
FUNCTION TCompany.YearlyProfit:  real;

FUNCTION TCompany.ReturnName:  string;

PROCEDURE TCompany.Print;
```

The first two of these methods are function methods. Therefore, their method declaration line begins with the keyword FUNCTION, just as any Pascal function would. The third method is a procedure method, and, similarly, it is declared with the keyword PROCEDURE.

The next thing on the method declaration line is the name of the method. Since methods belong to an entire object class, the complete name of the method is the name of the class, followed by a period, followed by the name of the method. Always writing the complete method name saves confusion when two or more object classes have a method of the same name.

Finally, the function methods end the method declaration line with the specification of the type that is returned by the function. TCompany.YearlyProfit returns a real value and TCompany.ReturnName returns a string value.

Methods, like any procedures and functions, can take parameters. Later in this book you'll see many methods that take parameters. If a method takes parameters, the parameter list comes after the method name, and looks like any parameter definition list. For example, if the three methods above took parameters, their declaration lines might look like this:

```
FUNCTION TCompany.YearlyProfit(LessTaxes: real):  real;

FUNCTION TCompany.ReturnName(Capitalized: boolean):  string;

PROCEDURE TCompany.Print(LongForm: boolean; HowManyTimes: integer);
```

Reverting to the earlier method declaration lines, you can now put together a complete method definition:

```
TCompany = OBJECT(TObject)

        fMainCompetitor:  TCompany;
        fName:  string;
        fNumberOfEmployees:  integer;
        fYearlySales:  real;
        fYearlyExpenditures: real;

        FUNCTION TCompany.YearlyProfit:  real;
        FUNCTION TCompany.ReturnName:  string;
        PROCEDURE TCompany.Print;

END;
```

Override method declarations

The only aspect of defining object classes left to explore is that of declaring certain methods to be override methods of ancestors' methods. For instance, if you wanted to define the employee object class from Chapter 2, you might end up with a definition like this one:

```
TEmployee = OBJECT(TObject)

        fName:  string;
        fTitle:  string;

        PROCEDURE TEmployee.Print;

END;
```

Then you could define the contractor descendant class.

```
TContractor = OBJECT(TEmployee)

        fContractNumber:  integer;
        fContractDollarAmount:  real;

        PROCEDURE TContractor.Print;  OVERRIDE;

END;
```

You should notice three things about this class definition:

- The TContractor class has been declared to be a descendant of the TEmployee class.

- Only the fields specific to the TContractor class are listed. To be sure, the TContractor class has four fields: fName, fTitle, fContractNumber, and fContractDollarAmount. Since the first two are inherited, however, only the last two need be listed in this class definition.

- The Print method, which is an inherited method, is listed again. If you wanted the Print method of TContractor to be exactly the same as the Print method of TEmployee then you would not have to list TContractor.Print—it would be automatically inherited. However, since TContractor.Print overrides TEmployee.Print, it does need to be listed, and the keyword `OVERRIDE`, followed by a semicolon, is written at the end of the method declaration line.

Object Pascal makes one limitation on override methods: an override method must have the same interface as its inherited method. Therefore an override method must have the same parameter list as its inherited method. Since an override method is supposed to be "equivalent" to the inherited method—only customized for a descendant class—this rule makes sense. However, there are times when it will pose a bit of a problem, especially with initialization methods, which will be dealt with later in this chapter.

A class definition is hardly complete without defining the methods that belong to that class. However, it will be useful to examine Object Pascal instances before exploring the syntax of method definitions.

Object Instances

Since object instances in Object Pascal are similar to records in standard Pascal, let's first look at record instantiation.

Record handles and record instantiations

Standard Pascal allows records to be instantiated a number of different ways: through record variables, through record pointers, and through record handles.

Instantiation through record variables

Records, of course, can simply be instantiated as variables, for example:

```
ThisRecord: RecordType;
```

These lines each declare an actual record instantiation, named ThisRecord, to be created, which reserves space for ThisRecord in memory.

To reference the fields of this record, simply use a dot:

```
SomeVariable := ThisRecord.AField;
```

Instantiation through record pointers

Another frequent means of instantiating a record is through the use of a pointer, for example:

```
ThisRecord: ^RecordType;
```

When you use pointers, you must first call Pascal's NEW routine to allocate space in memory for the record. Then the fields of the record can be referenced by the special up-arrow notation:

```
NEW(ThisRecord);
SomeVariable := ThisRecord^.AField;
```

Instantiation through record handles

Finally, a third means of instantiating records, and one that is very popular with Macintosh programmers, is through the use of a handle. A handle, as veteran Macintosh programmers know, is a pointer to a pointer. Declaring handles looks like this:

```
ThisRecord: ^^RecordType;      { "a pointer to a pointer to a RecordType" }
```

Before the record actually exists, you must call the NewHandle routine. Then you can access the fields of the record through the use of the double indirection symbol:

```
NewHandle(ThisRecord);
ThisRecord^^.AField;
```

Using handles and double indirection may look like a lot of extra work, since it does approximately the same job as using pointers. However, handles allow more efficient memory management, and handles are frequently used in place of pointers in Macintosh programming. Since object classes are instantiated solely through the use of handles, the rest of the instantiation examples are shown solely using handles.

It is important to remember that declaring a record handle does not allocate space in memory for a record. Only after the NewHandle procedure is run is a space in memory created for storing values into the data fields.

Object references and object instances

Object classes are instantiated in the almost the same manner as record handles. If it were exactly the same, you would declare variables that would store the handle like this:

```
EmployeeA: ^^TContractor;              { But it's not really done this way. }
```

Since this handle variable will "refer" to the object instantiation, it is usually called an **object reference variable**, or just an object reference.

Since all object reference variables are handles, Object Pascal lets you take a shortcut. In fact, it forces you to take the shortcut. The shortcut is this: *all double indirection is done for you automatically*. You never see the double up-arrow when dealing with object reference variables. So, the above declaration becomes:

```
EmployeeA: TContractor;                { This is actually how it's done. }
```

EmployeeA is still an object reference variable. In other words, EmployeeA is a handle that points to an object of class TContractor. In order to create space in memory for an actual object instance, you must call the Pascal NEW routine:

```
NEW(EmployeeA);
```

Notice that this is not the same NEW routine as in standard Pascal. This Object Pascal NEW routine takes an object reference variable as a parameter, and creates an object instance for it to refer to through a handle.

Finally, to reference a field of the object instance, you use a syntax exactly like that for records:

```
SomeStringVariable := EmployeeA.fName;
```

Note that once again, the double indirection is done for you. Otherwise the above line would read:

```
SomeStringVariable := EmployeeA^^.fName;          { This is NOT how it's done. }
```

Even though this is what is actually happening, you never see this in an Object Pascal program—Object Pascal notices that EmployeeA is an object reference variable and automatically does double indirection for you. While this may seem confusing at first, it actually simplifies things greatly. Except for always remembering to call the NEW procedure, you can usually just think of object reference variables as plain variables instead of handles. This also makes code much easier to read.

Method call syntax

The secret of calling methods in Object Pascal can be easily learned by memorizing one sentence: *In Object Pascal making a method call looks just like referencing a field.* In other words, an object's methods are called the same way an object's fields are referenced—the use of a dot.

For example, if you wanted to call the Print method of the EmployeeA object instance, the call would look like this:

```
EmployeeA.Print;
```

This looks just like referencing a field of a record variable, but actually it is calling the method of an object instance. Here again you can see why it is so important to begin all field names with a lowercase f. You will find you get used to any residual ambiguity quite quickly, especially when parameters are involved. For instance, if the Print method required parameters, the call might look like this:

```
EmployeeA.Print(true, 7);
```

This line of code would call the Print method of the instance referred to by the EmployeeA reference variable with true and 7 as the input values for the two parameters.

If the method is a function method, as is the YearlyProfit method of the TCompany object class, then it is used syntactically just like standard functions:

```
VAR CompanyA:  TCompany;

...

NEW(CompanyA);
SomeIntegerVariable := CompanyA.YearlyProfit;
```

Now that you've seen object reference variables and object instances in Object Pascal, you're ready to see how methods are defined.

Method definitions

Once your object classes have been defined, you must write the code for each of the methods you have declared. The syntax for such method definitions is also similar to standard Pascal, with a few exceptions.

Remember that method definitions define a method for an entire class of objects. They describe the general behavior of a method. The actual methods that are called in a program belong to specific object instances, and each object instance of a class has its own specialized version of the generic method. Object Pascal takes care of creating these specialized versions for you, but when you define the generic method, you must give the directions directing what you want the specialized versions to do. You do this using Object Pascal's SELF keyword.

The SELF keyword

Defining object methods—that is, writing the code for an object method—is syntactically very simple. The opening line of the definition must exactly match the opening line of the method declaration (the one found in the object class definition). The opening line is then followed by the body of the method. In the body of the method, you will often need to refer to a field of the class that owns this method. You can refer to the class's fields using the `SELF` keyword. For example,

```
PROCEDURE TEmployee.Print;

BEGIN
    writeln("My name is ", SELF.fName);
    writeln("My title is ", SELF.fTitle);
END;
```

This method definition is fairly simple. When you call a specific instance's version of this Print method, it will print that instance's fName field, then it will print that instance's fTitle field. In other words, the keyword `SELF` is kind of a marker that means "the specific instance that this specialized method belongs to". The instance that `SELF` refers to is called the **calling instance.**

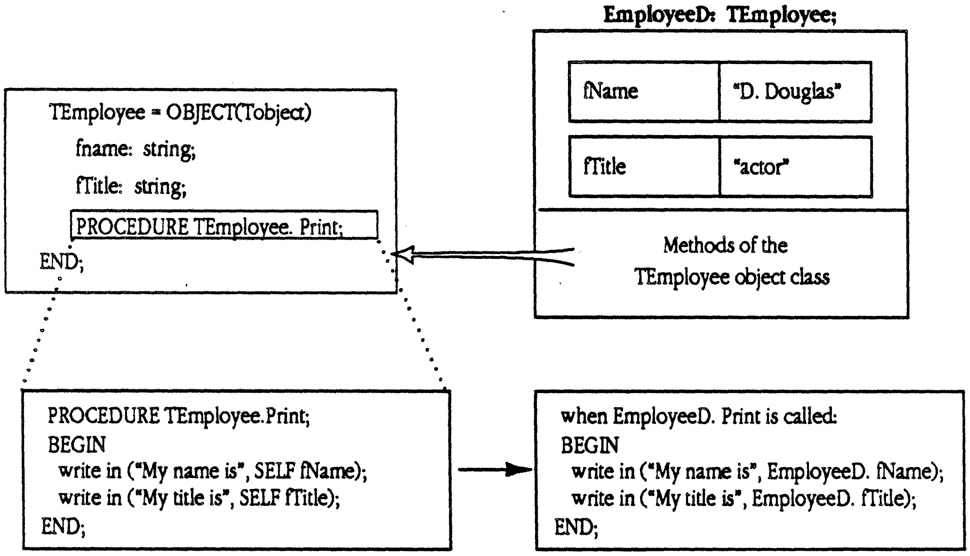
Let's take an example. Suppose you have created an employee reference variable:

```
VAR  EmployeeD:  TEmployee;
```

and then you have instantiated that variable, and initialized it:

```
NEW(EmployeeD);
EmployeeD.fName := "D. Douglas";
EmployeeD.fTitle := "actor";
```

Finally, you make a call to its Print method. The Print method that actually is called is not the generic one defined above, but a specialized copy of it, where every occurrence of the keyword "SELF " has been replaced by "EmployeeD".



```
TEmployee = OBJECT(Tobject)
   fname: string;
   fTitle: string;
   PROCEDURE TEmployee. Print;
END;
```

```
PROCEDURE TEmployee.Print;
BEGIN
  write in ("My name is", SELF fName);
  write in ("My title is", SELF fTitle);
END;
```

```
when EmployeeD. Print is called:
BEGIN
  write in ("My name is", EmployeeD. fName);
  write in ("My title is", EmployeeD. fTitle);
END;
```

Figure 3-3
A generic method definition and a specialized version of the method

When a call is made to the Print method of EmployeeD:

```
EmployeeD.Print;
```

EmployeeD is called the calling instance. The keyword SELF is always replaced in the specialized versions of methods by a reference to the calling instance.

So, you can see that the keyword SELF is the trick that allows you to define a method once, even though a specialized version is created for each different object instance you create.

One more thing—the use of the word SELF in Object Pascal is optional, though it often makes your code clearer. You can act as if the code part of every method is surrounded by

```
WITH SELF DO BEGIN
  .
  .
  .
END.
```

In other words, every variable or method name is checked to see if it belongs to SELF. For example, the above Print method could be redefined like this:

```
PROCEDURE TEmployee.Print;

BEGIN
    writeln("My name is ", fName);
    writeln("My title is ", fTitle);
END;
```

The keyword SELF, then, rarely *must* used. The only situations that require the use of SELF are when you need to pass a reference to the calling instance to another routine or need to assign a reference to the calling instance to a variable. However, always using SELF may help make your code clearer.

Methods that call other methods of the calling instance

You've seen how to refer to a field of the calling instance in the method definition using SELF. Making a call to another method of the calling instance works similarly. For instance, here is a possible definition of the Print method of the TSalariedEmployee class:

```
PROCEDURE TSalariedEmployee.Print;

BEGIN
    writeln("My name is ", SELF.fName);
    writeln("My title is ", SELF.fTitle);
    writeln("My salary is ", SELF.fSalary);
    writeln("My weekly paycheck is ", SELF.WeeklyPaycheck);
END.
```

As you can see, the fourth line of this definition looks strikingly similar to the first three lines. The difference is that the first three lines access fields of the calling instance, while the fourth line calls another method (a function method) of the calling instance. In general, function method calls are indistinguishable from field references, which explains the use of a lowercase f to begin field names. Procedure method calls also look the same, but of course they return no value and you can distinguish them in context.

Methods that call methods of other instances

As you just saw, calling methods of the same instance is as simple as referring to fields of that instance. Calling methods of other instances is different. (Remember, by convention in Object Pascal, one instance should not reference fields of other instances—it should only call their methods.) In order to call methods of another instance, the first instance must have access to the other instance. There are only three ways this is possible.

Calling a method of a global object instance

Imagine you have declared a global object reference variable:

```
VAR globalObject:  TSomeObjectClass;
```

Assuming that this global object reference variable has been instantiated and initialized somewhere in the main program, you can effectively call its methods in any method of any object. For instance,

```
PROCEDURE TSampleObject.SampleMethod;

BEGIN
    { Calls to other methods of the calling instance look like this.}
    SELF.OtherMethod;

    { Here's a call to a method of an object instance that is not the calling
instance.}
    globalObject.SomeMethod;
END.
```

Calling a method of a local object instance

The second possibility for calling another object instance's methods is having the object be a local variable to the method. Remember, methods are procedures and functions, and as such can have their own local variables—including object reference variables. Using this strategy, you could redefine the TSalariedEmployee.Print method again.

```
PROCEDURE TSampleObject.SampleMethod;

VAR localObject:  TSomeObjectClass;

BEGIN
    { Calls to other methods of the calling instance look like this.}
    SELF.OtherMethod;

    { Before calling any methods of the local object, you must instantiate it.}
    NEW(localObject);

    { Here's a call to a method of the local instance -- not the calling instance.}
    localObject.SomeMethod;
END.
```

Calling a method of an instance linked by an object reference field

The final possibility for calling other object instance's methods is the most common, and the one discussed in Chapter 2—having an object reference field linked to the other instance. For example, remember the company object class:

```
TCompany = OBJECT(TObject)

        fMainCompetitor:  TCompany;
        fName:  string;
        fNumberOfEmployees:  integer;
        fYearlySales:  real;
        fYearlyExpenses: real;

        FUNCTION TCompany.YearlyProfit:  real;
        FUNCTION TCompany.ReturnName:  string;
        PROCEDURE TCompany.Print;

END;
```

The Print method of class TCompany prints not only the fields of the calling instance, but also the name field of the calling instance's competitor. The Print method does this by calling the name returning method of the instance referred to by the fMainCompetitor field. The syntax for this method definition is this:

```
PROCEDURE TCompany.Print

BEGIN

    { First, print the values of the calling instances fields. }
    writeln("Our company's name is ", SELF.fName);
    writeln("The number of happy employee's here is:", SELF.fNumberOfEmployees);
    writeln("Our yearly sales are: ", SELF.fYearlySales);
    writeln("while our yearly expenses are: ", SELF.fYearlyExpenses);

    { Then, print the value returned by another method of the calling instance. }
    writeln("which means our yearly profit is: ", SELF.YearlyProfit);

    { Finally, print the value returned by a method of the instance }
    { that is referred to by the fMainCompetitor field.           }
    writeln("Our biggest competitor's name is", SELF.fMainCompetitor.ReturnName);

END.
```

Notice the last line of code in the above example. It makes the following method call:

```
SELF.fMainCompetitor.ReturnName
```

This syntax might seem a bit strange at first. What it means is actually quite simple—it's like referencing a field of a record field of a record. The SELF keyword refers to the calling instance. Adding the ".fMainCompetitor" makes it refer to the instance linked by that field—the calling instance's main competitor instance. Adding the ".ReturnName" specifies that a call to the ReturnName function method of the main competitor instance is to be made.

In order to see this in action, let's create some object instances of class TCompany.

```
VAR CompanyA, CompanyB:  TCompany;

BEGIN
    {instantiate the object reference variables }
    NEW(CompanyA);
    NEW(CompanyB);

    { initialize CompanyA's fields }
    CompanyA.fMainCompetitor := CompanyB;          { This is how the one link is
established. }
    CompanyA.fName := "Andy's Widgets, Inc."
    CompanyA.fNumberOfEmployees := 100;
    CompanyA.fYearlySales := 5000000.00;
    CompanyA.fYearlyExpenditures := 4000000.00;

    { initialize CompanyB's fields }
    CompanyB.fMainCompetitor := CompanyB;          { This is how the other link is
established. }
    CompanyB.fName := "Mark's Foobat Co."
    CompanyB.fNumberOfEmployees := 50;
    CompanyB.fYearlySales := 2000000.00;
    CompanyB.fYearlyExpenditures := 1000000.00;

    ...

END.
```

After the first two lines of code have been executed, unitialized instances have been created in memory.

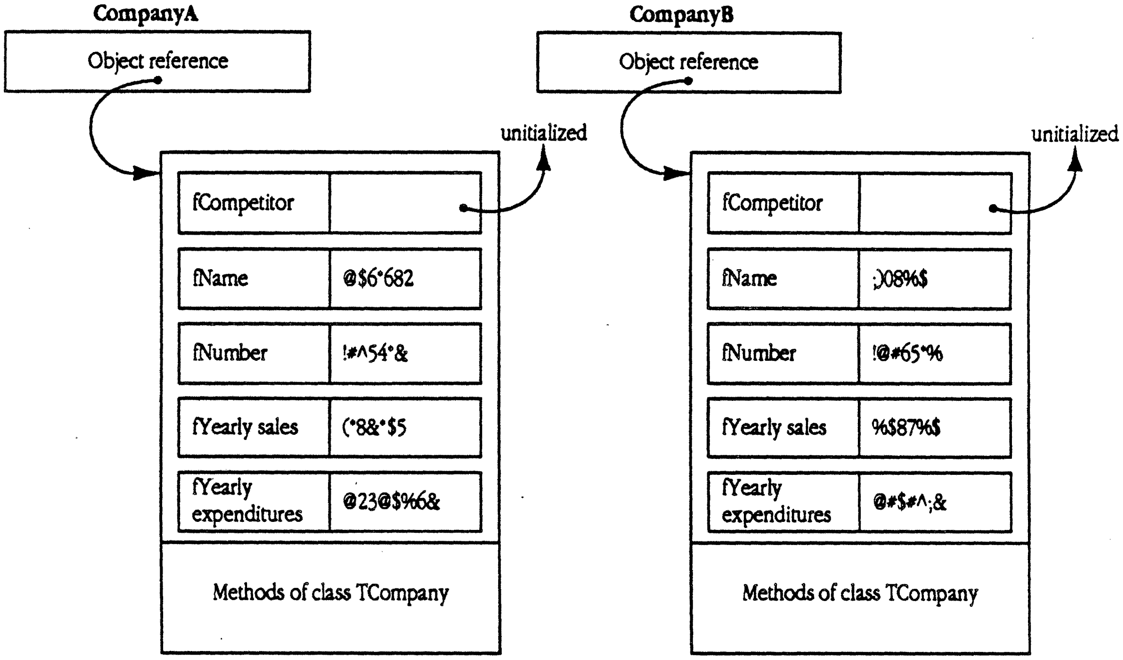


Figure 3-4
Unitialized instances in memory

The next ten lines of code initialize these instance. Most of this initialization is straightforward. The only unfamiliar initialization is the establishment of the references between the two instances. If you remember that object reference variables are actually handles, these two lines look like standard Pascal code. Take the first of these as an example:

```
CompanyA.fMainCompetitor := CompanyB;
```

This line specifies that the fMainCompetitor field should point to the instance that the CompanyB object reference variable points to.

Once all of the code above has executed, the following situation will exist in memory.

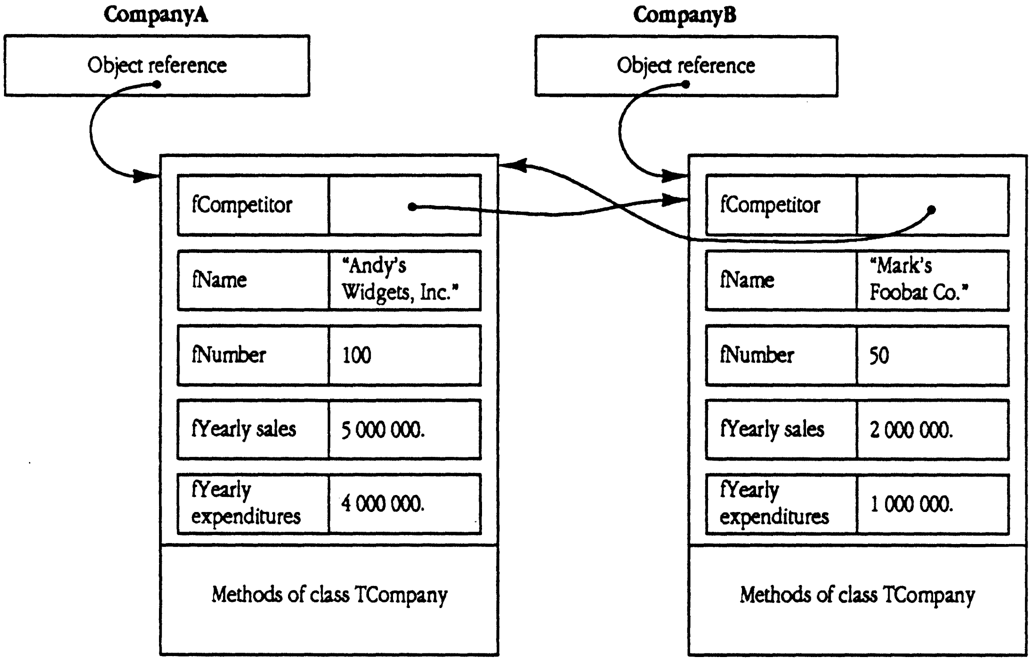


Figure 3-5
Object references and instances in memory

Now, if you call the Print method of CompanyA at the end of the main program above, like this:

```
CompanyA.Print;
```

the line that read

```
writeln("My competitor's name is", SELF.fMainCompetitor.ReturnName);
```

in the method definition will execute just as this line would during runtime:

```
writeln("My competitor's name is", CompanyA.fMainCompetitor.ReturnName);
```

When this line is executed, the ReturnName method of the object referenced by the fMainCompetitor field of CompanyA is called, and the result is printed, which is exactly what was wanted.

Override methods

An override method is defined just like any other method, except for the use of the keyword OVERRIDE, just as in the method declaration line.

For example, remember that class TSalariedEmployee is a descendant of class TPaidWeeklyEmployee, and therefore TSalariedEmployee inherits the WeeklyPaycheck method of TPaidWeeklyEmployee. Of course, TPaidWeeklyEmployee was an abstract object class, and its WeeklyPaycheck method was never meant to do anything:

```
PROCEDURE TPaidWeeklyEmployee.WeeklyPaycheck;

BEGIN

    { This method does nothing. It is meant to be overridden. }

END.
```

Defining the override version of this method looks exactly like defining any method, except for the keyword OVERRIDE.

```
PROCEDURE TSalariedEmployee.WeeklyPaycheck;  OVERRIDE;

VAR amountOfCheck: real;

BEGIN

    { Determine the amount of the paycheck, using the calling instance's fields. }
    amountOfCheck := SELF.fSalary / 52;

    { Return the result, just as you would for any Pascal function. }
    WeeklyPaycheck := amountOfCheck;
END.
```

Override methods that call the inherited method

Sometimes it is desirable for an override method to make use of the inherited method that it is overriding. Remember from Figure 2-21 that the Print method of the Contractor class overrode the Print method of the Employee class, but still made a call to that inherited method. In Object Pascal, the definition of the override method would look like this:

```
PROCEDURE TContractor.Print

BEGIN

    { First, call the inherited version of Print to print name and title. }
    INHERITED Print;

    { Then, print out the fields specific to this class. }
    writeln("My contract number is ", SELF.fContractNumber);
    writeln("My contract dollar amount is ", SELF.fContractDollarAmount);

END.
```

You can see that the syntax in Object Pascal for making calls to inherited methods is simple:

```
INHERITED Print;
```

If the method has any parameters, they are listed after the method name, as usual. For example,

```
INHERITED Print(parameter1, parameter2);
```

More about Object Pascal

By now, you've seen all of the syntax of Object Pascal. You know how to define object classes and their methods, and declare and instantiate object reference variables. Syntactically, you're ready to start creating Object Pascal programs. However, there are a few use and convention issues that you should be familiar with before diving into your first MacApp program.

Creating and freeing instances

Earlier in this chapter, the object class TObject was mentioned. The TObject class has no fields and only a few utility methods. These utility methods are helpful enough that it is desirable for all object classes to have them. This is accomplished simply: just make your highest ancestor objects descend from class TObject. For example, in the employee class hierarchy, TEmployee is the highest ancestor—it has no ancestor itself. In this case, all you need to do is make TEmployee a descendant of TObject. Since all other classes in that example descend from the TEmployee class, all classes therefore descend from TObject as well. As a result, all object classes inherit the methods of class TObject.

You already know how to allocate memory for an object instance—use the NEW routine on an object reference variable. To deallocate the memory of an object instance, you can use the **Free** method. The Free method is a method of TObject and therefore can be called by any object instance, if you've declared your hierarchy properly. You can override the Free method so that objects of a given type can perform cleanup tasks of their own when they are about to be deallocated.

Initialization methods

Immediately after instantiating an object reference variable, it is a good idea to initialize the object instance. Typically this is done with an initialization method—a method that takes as many parameters as the class has fields, and initializes each field to one of the parameters.

For example, the TEmployee object class might have the following initialization method.

```
PROCEDURE TEmployee.Initialize(name, title: string);

BEGIN

    { Initialize each field to the appropriate parameter. }
    SELF.fName := name;
    SELF.fTitle := title;

END;
```

If you have declared an object reference variable, for example EmployeeD, to be of class TEmployee, then after you instantiate it you can call it's initialization method:

```
EmployeeD.Initialize("D. Douglas", "actor");
```

Like the Print method example, you might think it would be nice to give each employee object class an override version of this method. The override version could call the inherited version to initialize the first two fields, and then initialize the rest of the fields itself.

For instance,

```
PROCEDURE TContractor.Initialize(name, title: string;
                                number: integer;
                                amount: real);  OVERRIDE;

BEGIN

    { Call the inherited method -- the method of TEmployee }
    INHERITED Initialize(name, title);

    { Initialize the remaining two parameters. }
    SELF.fContractNumber := number;
    SELF.fContractDollarAmount := amount;

END;
```

Unfortunately, Object Pascal requires that the method declaration line not be changed for override procedures. As you can see in the above example, there were two parameters added to the method declaration line. This is not legal and the above override procedure would not compile.

Since initialization methods almost always require varying numbers of parameters, the result of this restriction is that initialization methods are not override methods. Each class has its own initialization method. For example, the two classes above might have these two initialization methods:

```
PROCEDURE TEmployee.IEmployee(name, title: string);

BEGIN

    SELF.fName := name;
    SELF.fTitle := title;

END;


PROCEDURE TContractor.IContractor(name, title: string;
                                 number: integer;
                                 amount: real);

BEGIN

    SELF.fName := name;
    SELF.fTitle := title;
    SELF.fContractNumber := number;
    SELF.fContractDollarAmount := amount;

END;
```

As you can see, the convention for naming initialization methods is to begin them with an uppercase I, followed by the name of the class.

Even though you cannot have IContractor call an inherited initialization method (because there isn't one), you can do something almost as good. Since class TContractor inherits all of the methods of class TEmployee, it must also inherit IEmployee. Therefore, you can have IContractor call IEmployee directly—almost as if you were calling an inherited initialization method:

```
PROCEDURE TContractor.IContractor(name, title: string;
                                  number: integer;
                                  amount: real);

BEGIN

    SELF.IEmployee(name, title);
    SELF.fContractNumber := number;
    SELF.fContractDollarAmount := amount;

END;
```

In this manner, making separate initialization methods that are not override methods for each class does not cause you to have to duplicate any code.

Privacy between instances

As you've seen a few times, the method of one instance can call the method of another instance. Under the strict application of convention, this is the only way one instance should communicate with another. However, there are times when you will want a method of one instance to have access to information from another instance and there will be no method of the second instance that returns that data. In cases like this, it would be best to add a method to the second instance's class that did return that data, but this is not always possible.

It is legal in Object Pascal syntax for the method of one object to reference a field of another object. For example, the method definition for TCompany.Print could include something like this:

```
PROCEDURE TCompany.Print

BEGIN

    { Print the value of the fName field of the competitor instance. }
    writeln("My competitor's name is", SELF.fMainCompetitor.fName);

END.
```

Here, you can see that the method is accessing the fName field directly. This ability exists in Object Pascal, but use it sparingly!

Object reference variables

Object reference variables must be declared to be of a specific object class. For example,

```
VAR SomeEmployee:  TEmployee;
```

SomeEmployee is an object reference variable, and is a member of class TEmployee. Therefore, it must reference an instance of class TEmployee. However, in Object Pascal, SomeEmployee is also allowed to reference an instance of any class that is descended from TEmployee. For instance, imagine you have also declared a variable EmployeeA:

```
VAR EmployeeA:  TContractor;
```

In the main body of the program, after instantiating and initializing EmployeeA, you can make the following assignment:

```
SomeEmployee := EmployeeA;
```

Now, SomeEmployee is referencing an instance of class TContractor. Since TContractor descends from TEmployee, this is legal. SomeEmployee now has access to all of the fields methods of TContractor, as well as the fields and methods of TEmployee. For example, it is now legal to reference SomeEmployee.fContractNumber.

The real power of object-oriented programming comes from what happens when you access a method of SomeEmployee. Regardless of what class the instance that SomeEmployee references, the correct method will always be called.

For example, if you now called SomeEmployee.Print, the override Print method belonging to the TContractor class is the version that is called—not the original Print method belonging to the TEmployee class. This means that as late as runtime your object-oriented program is checking which instance is the calling instance, and making sure the right version of the method is called.

Summary of Object Pascal syntax

- Object classes are defined in the TYPE declaration part of a Pascal program. These definitions include the class name (beginning with an uppercase T by convention), the type name OBJECT, the ancestor class name in parentheses, and the field declarations and the method declarations (including overriden methods followed by the keyword OVERRIDE.

```
TObjectClassName = OBJECT(TObjectClassAncestor)
        {field declarations}
        {method declarations}
END;
```

- Object fields (beginning with a lowercase f by convention) are declared just as record fields.

```
fFieldName:  fieldType;
```

- Methods are declared as normal Pascal procedures and functions. Their names include the name of their object class. If it is an overridden method, the keyword "OVERRIDE;" ends the declaration line.

```
PROCEDURE TObjectClass.TMethodName;
FUNCTION TObjectClass.TMethodName: ResultType;

PROCEDURE TObjectClass.TMethodName(parameter list);
FUNCTION TObjectClass.TMethodName(parameter list): ResultType;

PROCEDURE TObjectClass.TMethodName; OVERRIDE;
FUNCTION TObjectClass.TMethodName: ResultType; OVERRIDE;

PROCEDURE TObjectClass.TMethodName(parameter list); OVERRIDE;
FUNCTION TObjectClass.TMethodName(parameter list): ResultType; OVERRIDE;
```

- Methods are generally considered "public", that is, they can be accessed by other objects' methods, while fields are generally considered "private", that is, other objects should not access them directly. This is just convention and is not enforced by Object Pascal.

- Object Pascal does all object-related double-indirection. Even when an object reference variable is declared, no indirection is expressly used.

```
ObjectReferenceVariableName : ObjectClass;
```

The function New is still necessary, however, as object reference variables are still handles.

```
NEW(ObjectReferenceVariableName);
```

- Fields and Methods are both accessed with a dot. No indirection is ever needed.

```
ObjectReferenceVariableName.fFieldName
ObjectReferenceVariableName.MethodName
```

- Methods are defined (actually coded) much like standard Pascal procedures and functions. The title line must match the method declaration line exactly (including parameters and OVERRIDE;). The keyword SELF is used to refer to the object instance that called this method. SELF is always optional.
- Inherited methods can be called in an overriden method by prefixing the line of the call with the keyword INHERITED.

```
INHERITED InheritedMethodCallName;
```

Chapter 4

Introduction to MacApp: Organization

By now, you should have a good idea of what an object-oriented programming is all about, and should be ready for an in-depth look at MacApp. MacApp is nothing more than an extensive library of code. As with any large system, it seems you have to know it all before any of it makes sense. This chapter and the next present an overview of MacApp. This one renders a still-life portrait of MacApp; the next one shows the engine in motion: how the user's actions are interpreted and handled while the application is running.

The still-life in this chapter offers three different perspectives on how MacApp is organized:

- the physical location of files on disks
- the hierarchy of inheritance among the objects
- the way MacApp's objects represent the concepts of the Macintosh user interface

Knowing the physical layout of the files will help you to include all the necessary code; it will also help you track down the location of any piece of code you want to investigate more thoroughly. Knowing the ancestry of a particular object tells you what fields and methods it has in common with other objects. And knowing how the different objects in MacApp provide metaphors for the concepts of the Macintosh world allows you to understand why MacApp is put together the way it is, and gives you the power to manipulate the Macintosh so you can create effective applications.

Source code: organization of the files

MacApp is written mostly in Object Pascal. Pascal code is organized into units. Technically, a **unit** is a piece of code that is compiled separately from all the rest; conceptually, all the code compiled into one unit should have something in common. For instance, all the code related to printing might be compiled into a single unit. Once you compile all the units separately, you then have to link them together—a single unit by itself is not executable.

MacApp is divided into 18 units. Many of the units must be included in every application; others are optional, for instance depending on whether you want to use the debugger or take advantage of the printing or text-editing features of MacApp. All of MacApp's units have names that begin with an uppercase U. Table 4-1 lists these units. The table gives a thumbnail description of each unit, mentions the circumstances under which it should be included, and indicates how much of your attention it requires. Many of these units will be of no concern to you, either because their use is optional or because their routines and data are used only by the code in the MacApp library.

Table 4-1
MacApp units

Unit	Responsible for	When to include	Needs attention?
UAssociation	associating one object with another	always	rarely
UBusyCursor	changing cursor shape when the application is busy	always	rarely
UDialog	dialog-style windows	optional	optional
UFailure	handling failures	always	sometimes
UGridView	displaying arrays of information	always when debugging; otherwise optional	optional
UInspector	implementing the inspector	only when debugging	never
UList	maintaining lists of object-oriented information	always	optional
UMacApp	the core of MacApp's functions	always	always
UMAUtil	global routines and data	always	always
UMemory	memory management	always	always
UMenuSetup	menu maintenance	always	always
UObject	defining the root ancestor of all other objects	always	when creating custom objects
UPatch	implementing patches to the Macintosh ROM	always	never

UPrinting	printing	when your application requires printing	when customizing
UTEView	text editing	optional	optional
UTrace	MacApp debugger	debugging	never
UWrite-LnWindow	debugger window	debugging	never
UViewCoords	32-bit coordinate support	always	rarely

Each unit consists of at least two files: one interface file and one or more implementation files. The interface files, which are always much shorter than the implementation files, contain only declarations and type definitions. The implementation files contain the code that implements these declarations. That means that if you want to take a quick look at what parameters a particular method call requires, you only need to look at the short interface file, where it is declared. Each interface file also lists all the implementation files that are associated with it, since they all need to be compiled together. The file names in MacApp adhere to a rigid convention, as laid out in the table below.

Table 4-2
Naming conventions for MacApp files

File Type	Naming Convention	Example
Assembly	U*filename*.a	UMacApp.a
Interface	U*filename*.p	UMacApp.p
Implementation	U*interface.implementation*.p	UMacApp.inc1.p *or* UMacApp.TApplication.p

In most cases, the implementation of a unit is short enough to be put into a single file. In that case, the *implementation*'s name is traditionally just "inc1," which stands for "include file #1." However, when the implementation is too long to be managed easily in a single file, it is broken up across a number of files; each is given a mnemonic name. The most notable example of a unit broken into a large number of files is UMacApp, which is broken up into about a dozen files, each named after the most important thing defined in the unit, usually an object class. So, the file UMacApp.TApplication.p is the implementation portion of the UMacApp unit that defines the class named TApplication.

Each unit contains code related to a particular aspect of MacApp. A unit may contain any proportion of Object Pascal and standard Pascal. Though MacApp is an object-oriented application framework, much of the MacApp library consists of routines written in standard Pascal to make your application run more efficiently.

In sum, the 60 or so files in the MacApp library belong to 18 units, organized according to function. The names of the files reflect the unit name and whether the file contains the interface or the implementation.

Ancestry: organization of the classes

MacApp defines about 45 different classes of objects, an overwhelming number to learn at once. Fortunately, you only have to understand about a dozen of these to get a good overview of the library. The other three dozen or so classes are less important to understand immediately, either because you will never use them directly (though they are used by other pieces of the MacApp library) or because their purpose is highly specialized. For instance, almost half of the classes are devoted to manipulating dialog boxes, and though you will probably use some of these classes quite frequently, you don't need to know about them all to get an overview of MacApp.

There are two ways of analyzing these classes: by inheritance and by functional design. Inheritance tells you which methods and fields different classes may share but it does not tell you how the objects were intended to work together. This section discusses inheritance only; the next section discusses functional design (which is also called architecture). Figure 4-1 is a picture of the ancestry tree for the different classes. The object class **TObject**, from which all the others descend, is all the way to the left. TObject has five immediate descendants: **TCommand**, **TEntry**, **TAssociation**, **TList** and **TEvtHandler**. The uppercase T at the beginning of each of these names indicates that it is an object *type* (which is another way of saying *class*), not an actual instantiation of an object. Remember, when you see a name beginning with T it is a cookie cutter, not the cookie itself. In any case, the arrows in Figure 4-1 point from the ancestors toward the descendants.

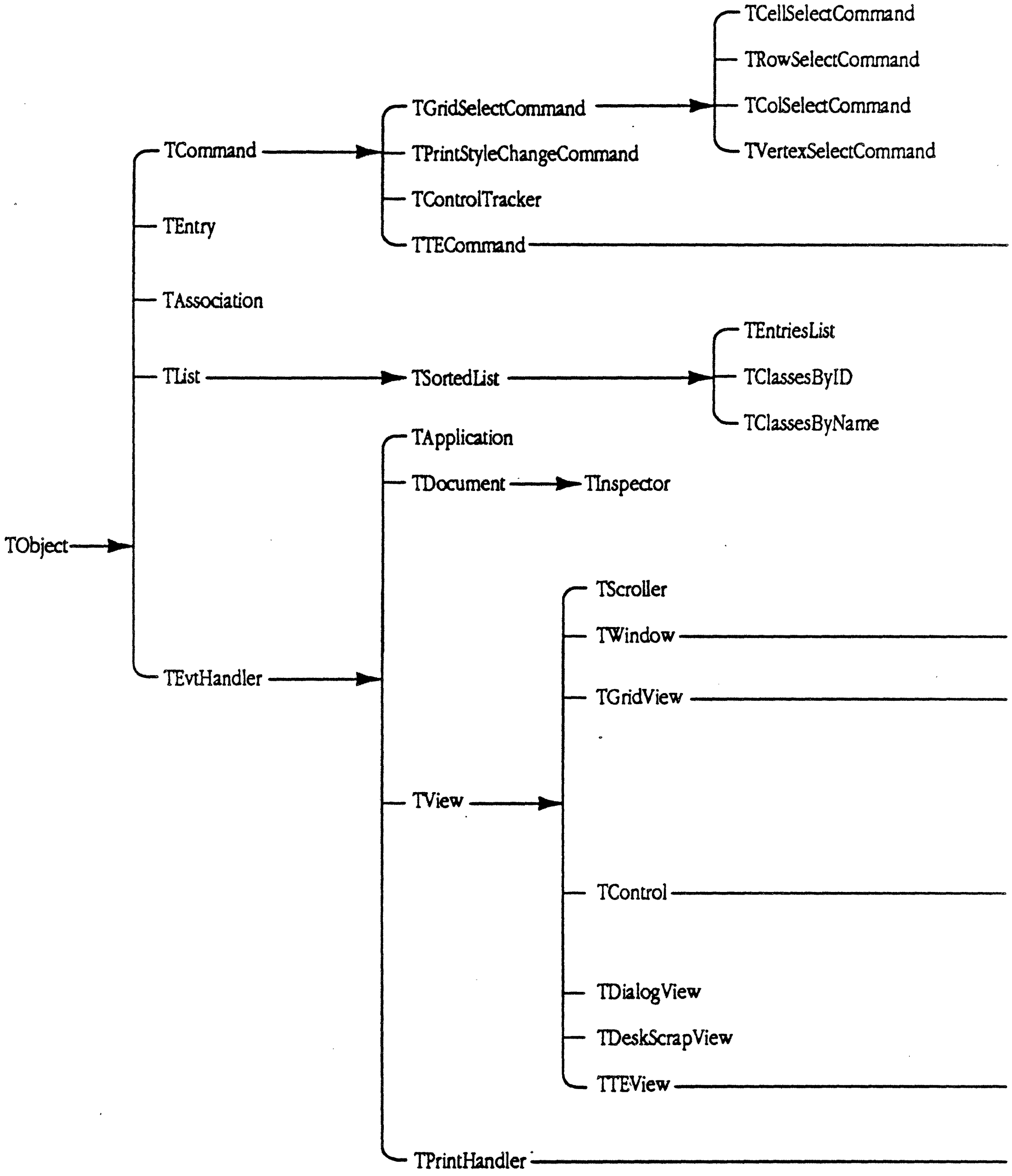
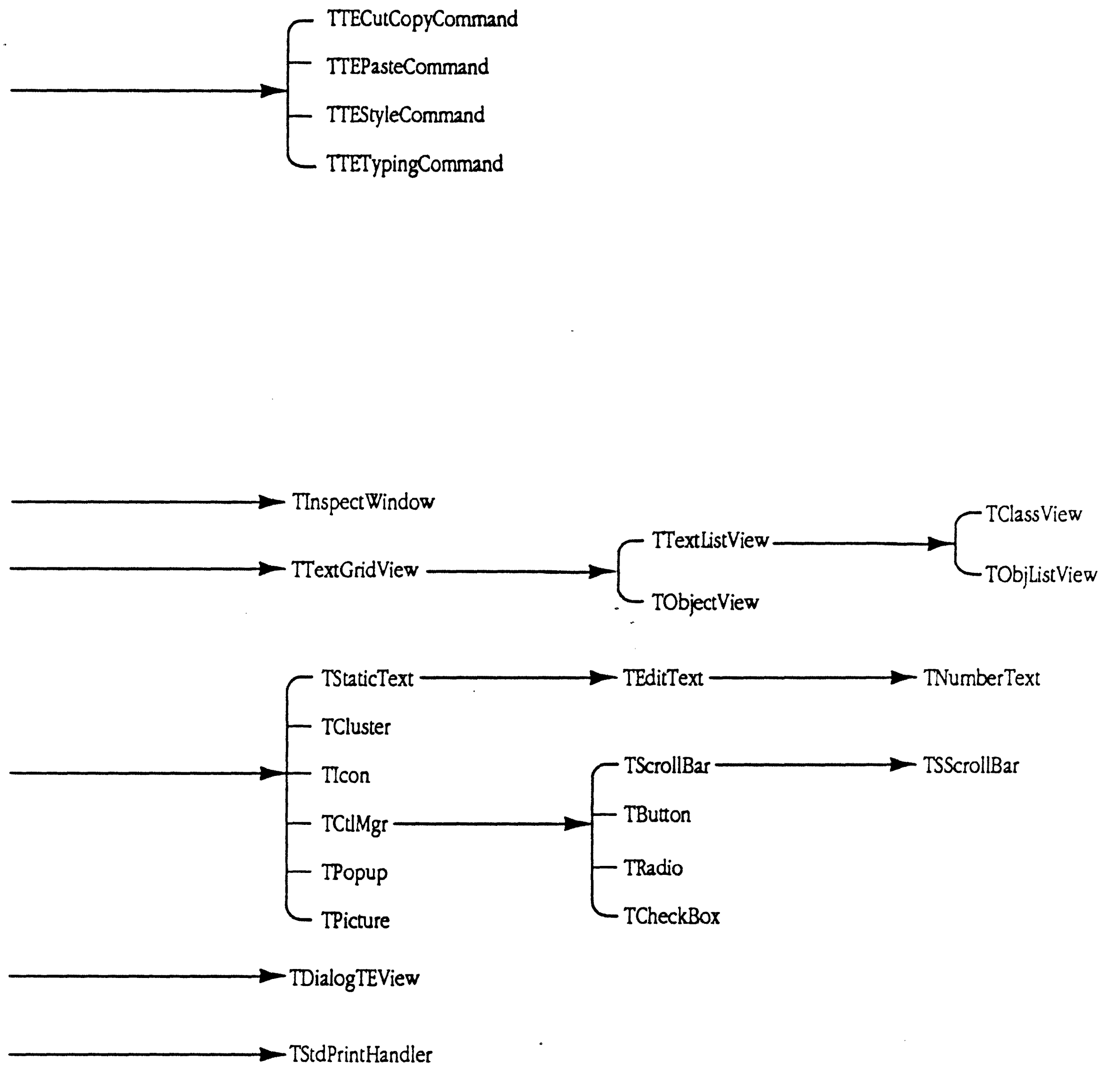


Figure 4-1
The ancestry tree for the classes n MacApp


TTECutCopyCommand
TTEPasteCommand
TTEStyleCommand
TTETypingCommand
TInspectWindow
TTextGridView
TTextListView
TObjectView
TClassView
TObjListView
TStaticText
TEditText
TNumberText
TCluster
TIcon
TScrollBar
TSScrollBar
TCtlMgr
TButton
TPopup
TRadio
TPicture
TCheckBox
TDialogTEView
TStdPrintHandler

As you look at this chart and read these sections, remember that descendants are specialized versions of their ancestors. So when you see that TWindow is a descendant of TView, translate that into the words "TWindow is a specialized version of TView."

Each of these objects is described more completely in the next section on architecture. Here, each object is presented only in its sketchiest outline, so you can see how it fits into the hierarchy. Most of the objects described in this section are objects that you will use repeatedly. You can "use" objects in two ways: by instantiating them or by creating new descendants of them. Classes that are never instantiated are called **abstract classes.** Abstract classes are used only to create descendants who will then share all the methods and fields inherited from their parent.

Higher-level classes

There are four commonly used classes at the top of the ancestry tree: TObject, TList, TCommand, and TEvtHandler.

- **TObject.** TObject is an abstract class, the ultimate ancestor of every other class. The main purpose of TObject is to bestow upon all its descendants a very few abilities all objects should have in common, namely making clones of itself and deallocating any memory it might have required so you can delete it. You will never create instances of TObject itself. Rather, both you and MacApp will use it to create more useful descendant classes and then will make instantiations of *those* classes.

TObject has three important descendants: TList, TCommand, and TEvtHandler. Of these three, TEvtHandler is by far the most important, both in terms of how often you will use it and in terms of understanding how MacApp works. Well over two-thirds of the classes defined in MacApp are descendants of TEvtHandler, and they are conceptually important classes, the ones that represent windows and documents and the application itself.

- **TList.** (Ancestry: TObject → TList) An instance of the TList class implements a simple list of objects. It is used throughout the the MacApp library to store such lists; you can have each instance in the list perform some operation on itself. For example, in a chess program, you could keep a list of all the instances that represent chess pieces. Then, if you wanted to implement a command that would display the location of every chess piece, you could simply go down the list and execute the Report method for every instance in the list.

- **TCommand.** (Ancestry: TObject → TCommand) The TCommand class is one of the most important to understand, and also one of the most difficult to explain. While it is easy to picture a window or a scroll bar, it is hard to visualize a command. But if you remember that an object is just some data and the methods that manipulate that data, the concept is not difficult at all. An instance of the command class has fields that keep track of the effects of a command and methods to perform the command, to track the mouse, and perhaps to undo and redo the command.

- **TEvtHandler.** (Ancestry: TObject → TEvtHandler) TEvtHandler is an abstract class, the ancestor of the many object instances that have to deal with events. The word *event* has a special meaning in the Macintosh world, including almost anything that might occasion a response from an application, ranging from user actions like pressing a key on the keyboard to ROM-generated requests to refresh a region on the screen.

Much of the code that you write yourself will be written to implement event-handling classes. The most important descendants of TEvtHandler are TApplication, TDocument, and TView. The great majority of TEvtHandlers are descendants of TView.

- **TApplication.** (Ancestry: TObject → TEvtHandler → TApplication) The TApplication class implements the main event loop and directs events to the objects responsible for handling them. The application class itself handles some events, if they affect the application as a whole. To get an idea of the sorts of things the application class is responsible for, imagine an application with no windows open. Any commands that can still be executed (including Open, New, About, or Quit) are handled by an instance of the application object. There is one, and only one, instance of a TApplication class in each application.

- **TDocument.** (Ancestry: TObject → TEvtHandler → TDocument) The TDocument class represents a collection of data (which is usually stored in a file). For example, a word processor uses documents consisting of formatting commands and text; a drawing program might use bitmapped documents; and other programs might use several different types of documents at once. Documents are generally responsible for commands that directly affect the data, such as the Save command.

- **TView.** (Ancestry: TObject → TEvtHandler → TView) The TView class is responsible for everything your application displays on the screen, except the menus. Views are used to render a document's data. For example, a spreadsheet application may have two views of the same data—tabular and graphic. Views are generally responsible for handling commands that a user might enter by clicking and dragging in the content region of a window.

Descendants of TView

Since the Macintosh provides a highly visual environment, the TView class includes a highly diverse group of classes among its descendants.

- **TWindow.** (Ancestry: TObject → TEvtHandler → TView → TWindow) The TWindow class represents Macintosh windows. Since it is a descendant of TEvtHandler, it inherits many fields and methods for intercepting and handling events; and since it is a descendant of TView, it inherits fields and methods for drawing on the Macintosh screen. TWindow overrides some of these, and has extra attributes that allow it to store and display a title and the other parts of a window.

- **TScroller.** (Ancestry: TObject → TEvtHandler → TView → TScroller) The TScroller class calculates coordinate translations to create the illusion of scrolling through a document.

- **TDialogView.** (Ancestry: TObject → TEvtHandler → TView → TDialogView) The TDialogView class duplicates some of the Dialog Manager's functions, creating the illusion (with the help of some other classes) that a MacApp window is a Dialog Manager dialog.

- **TTEView.** (Ancestry: TObject → TEvtHandler → TView → TTEView) The letters TE stand for TextEdit, the set of Macintosh ROM utilities that make providing a rudimentary text editor relatively easy. TTEView is a class that displays and manipulates text.

- **TGridView.** (Ancestry: TObject → TEvtHandler → TView → TGridView) TGridView is also a descendant of TView

Summary of the ancestry tree

TObject is the ancestor of all objects. It has three important descendants: TList, which manages lists of objects; TCommand, which provides a framework for doing and undoing commands; and TEvtHandler, whose descendants populate most of the rest of MacApp. TEvtHandler, in turn, has three important descendants: TApplication, TDocument, and TView. TView and its more specialized descendants make up a plurality of the classes in MacApp.

Architecture: organization of the concepts

Now that you have the pieces of MacApp in front of you, it's time to put the puzzle together. Every application has three major sorts of operations: storing data, displaying information, and executing commands. How does MacApp accomplish these operations?

Storing data

Almost every application needs a metaphorical scratch pad where it can jot down information and perform calculations. In the MacApp world, this scratch pad is called the **document**. The English word *document* generally refers to a piece of paper with text on it. But in MacApp, a document is can store any sort of data. For instance, in a graphics program it could store shapes; in a word processor it could store text and formatting information; in a spreadsheet application it could store numeric information; or in a music program it could store data concerning pitch, timbre, and rhythm.

A program could also have a document that stored both text and graphics; or a single program could have two (or more) different types of documents. Also, many Macintosh programs allow more than one document to be open at a time, and often allow documents of different types to be open at the same time.

How does MacApp implement documents? MacApp provides a class called TDocument, which is an immediate descendant of TEvtHandler. In addition to the inherited **event-handler** fields, it has fields that record the title, creator, type, and date of the document. MacApp's design requires each document instance to take initiative for getting itself displayed. Thus, it has methods that create the views through which it will be displayed and fields that keep track of all the views associated with the document. Since MacApp cannot anticipate how you want to display the data or what sorts of windows and views you might want, these methods are empty in the MacApp library. You must override them to provide your own custom definitions.

Exactly how each sort of document stores information is up to you. It could be in memory or on disk. Applications often start by reading a document off the disk, keeping as much of it in memory as RAM allows, and then writing it back to the disk at the user's request. When you design your application, you must choose how you want to represent your data and where you want to store it. Since most applications require documents to be read from and written to the disk, the TDocument class also has fields for storing information about the file associated with a particular document and methods for determining the necessary disk space, and reading and writing them. Again, since MacApp cannot determine what sort of information you will be writing to the disk, many of these methods require you to override them and provide the actual code yourself.

In addition to reading and writing the disk files and creating and maintaining the views and windows associated with them, a document's methods are responsible for handling any requests from the user that affect the document. Every event-handler class has a method called **DoMenuCommand.** The document's DoMenuCommand method will call the appropriate methods in response to the user's choice of commands such as Save, Save As, and Revert to Saved. By overriding this method, you can allow your documents to handle other commands you might care to implement.

You never instantiate the TDocument class directly. Instead you will create one descendant of TDocument for each different sort of document you will have. You will customize each descendant class with fields for whatever data structures you need to keep the data in memory while the application is running and implement methods to maintain those fields. You might also add fields that reference related documents and views; and you might override and customize all the empty methods responsible for reading, writing, and so on. Then, whenever the user opens an old or creates a new document, the MacApp code you have customized will dynamically create new instances of these classes.

Displaying information

The Macintosh has a rich visual interface, which means the task of creating images on the screen is particularly important. Everything your application displays on the screen (except the menu bar) is handled by an instance of the class TView or one of its descendants. Programmers who have written software for the Macintosh *without* MacApp know a system of displaying images via grafPorts, bit maps, and bit images. Fortunately, MacApp overlays a simpler architecture on top of this old, complex one. Eventually you will have to know quite a bit about QuickDraw, the set of ROM routines that actually put images on the screen. But this introduction assumes no such knowledge. (If you do know about QuickDraw, please put that knowledge aside for the next few pages.)

TView is an abstract class. You seldom, if ever, instantiate it directly. Everything that appears on the Macintosh screen except for the menus is the responsibility of a descendant of TView. TView (Ancestry: TObject → TEvtHandler → TView) is an event handler, so it of course inherits fields and methods that make handling user events relatively easy. It represents a 32-bit visual coordinate system.

Hierarchy of views

Before discussing the concept of a hierarchy of **views**, it is important to clarify the terminology. As discussed in Chapter 2, *classes* have an ancestral hierarchy, but *instances* do not. Since the views being discussed here are instances, not classes, what sense is there in the phrase "hierarchy of views?" Each view has **object reference fields** that record information about its relationship to other views. All of these relationships together form a hierarchy of views. In MacApp's terminology, each view instance may have one superview, and any number of subviews. Subviews may themselves have other subviews, resulting in a whole tree of views, as depicted abstractly in Figure 4-2. (See Figures 4-4 and 4-5 for less abstract depictions.)

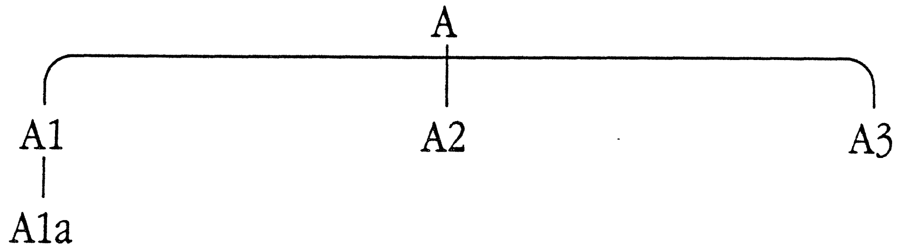
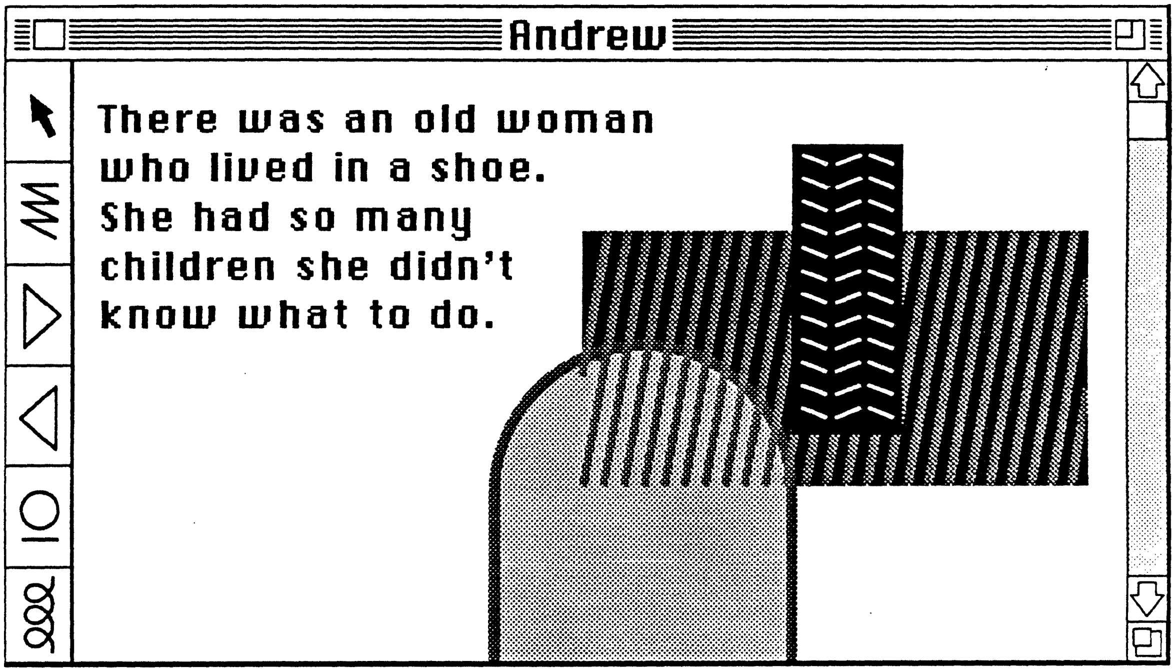


Figure 4-2
Tree of views

For now, don't try to picture how this tree of views would look on a Macintosh screen. It is enough if you realize that there can be a hierarchy of views. For instance, in the figure above, view A has no superviews, but it has three subviews, A1, A2, and A3. Note that the terms subview and superview are relative and not absolute. Just as you can be the child of one person and the parent of another, so one view can be a subview in one case and a superview in another. For example, A1 is the subview of A, but it is the superview of A1a. Each view has its own coordinate system which you define when it is created.

The TView class has fields that keep track of the superview and the subviews, the location and the size of the view, and a pointer to the object responsible for printing it. It has methods that translate coordinates from one view to another and that maintain the lists of subviews and references to superviews, and that can perform actions on all the views, or one view that meets particular criteria. It also has methods for opening, closing, and activating views. (**Activating** takes place when the view's window is brought to the front.) It has other methods for managing the location and size of the view, for drawing and updating the contents of the view, for handling mouse clicks, and for exporting and importing data to and from the Clipboard. Though the TView class is itself never instantiated, these fields and methods are passed down to all its descendants.

In the MacApp world, everything on the screen except the menu bar appears in a window. Thus, the natural place to begin a discussion about the architecture of views is with the window. A window is at the top of the tree of view instances; in other words, window instances are the only view instances in MacApp that have no superview. The contents of windows can be very simple, displaying only a static image in a fixed-size view with no scroll bars, or they can be very, very complex. Here is a blown-up diagram of a window with fairly complex contents.



There was an old woman who lived in a shoe. She had so many children she didn't know what to do.

Figure 4-3
A fairly complex window with one scroll bar and a palette

There are instances of five different classes represented in this particular window: a window, a scroll bar, a scroller, a content view, and a palette. Here is a blown-up version:

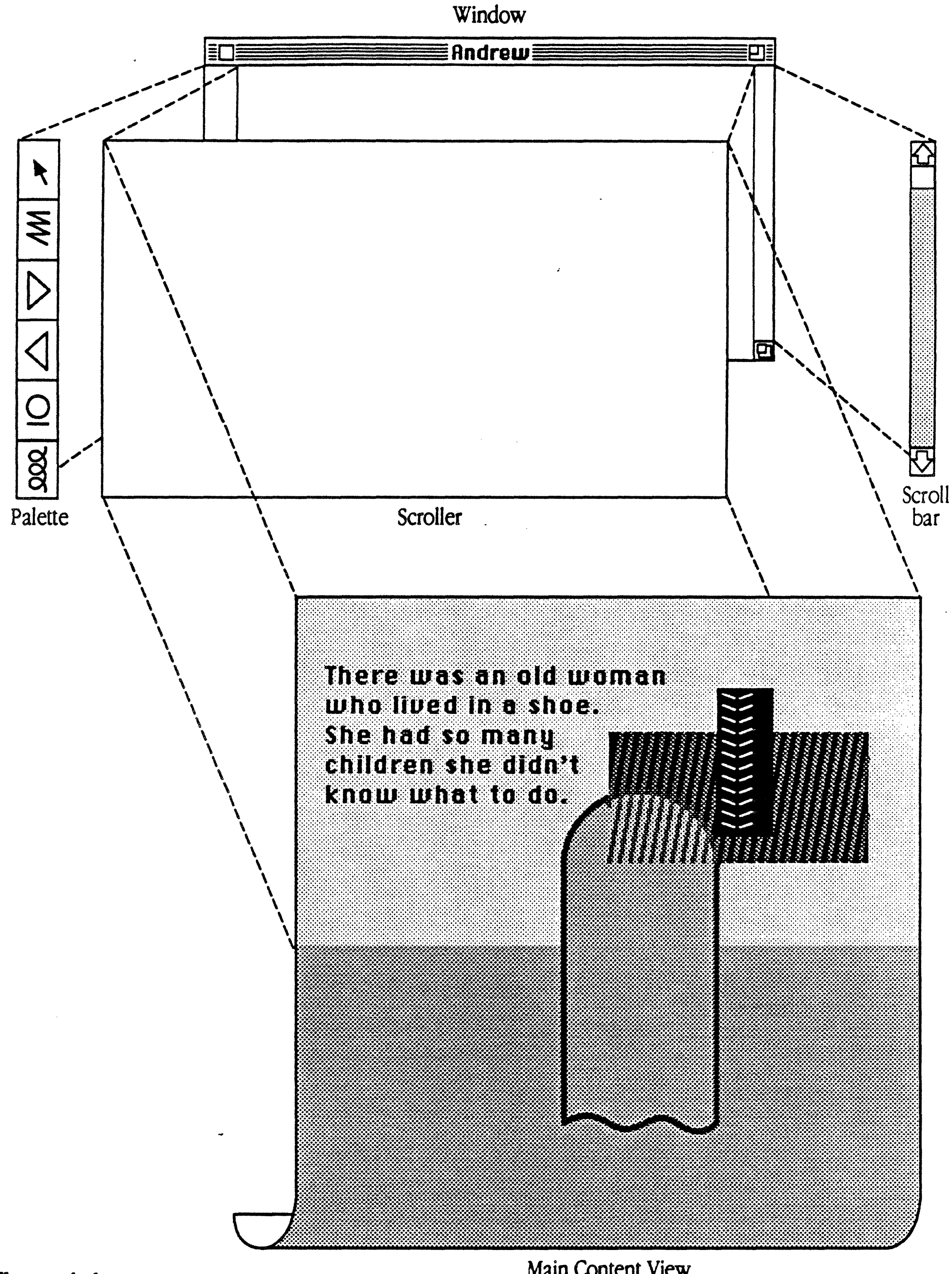

Window
Andrew
Palette
Scroller
Scroll bar
There was an old woman who lived in a shoe. She had so many children she didn't know what to do.
Main Content View

Figure 4-4
Blown-up version of window in Figure 4-3

The view tree for this window looks like this:

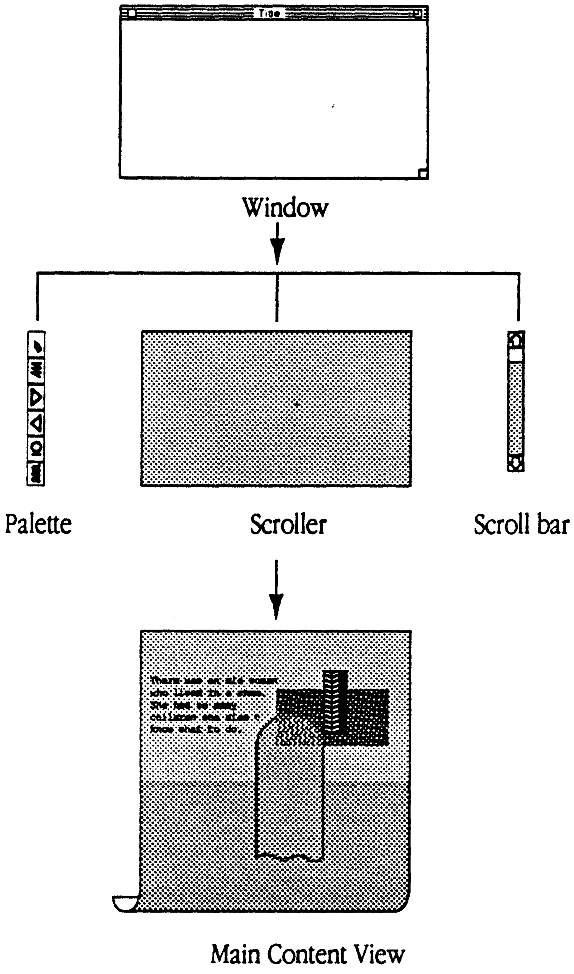


Figure 4-5 View tree for window in Figure 4-3

Each of these parts merits a separate discussion, beginning at the top of the tree.

Windows

At the top of the tree is an instance of TWindow. TWindow implements standard Macintosh window behavior, and you will only need to create descendants of it when you want to change its behavior. The TWindow *class* is an immediate descendant of TView, customized to give it the special attributes of a window. It inherits all the fields and methods of TView. The superview of every instance of TWindow is NIL, since windows are at the top of the tree of view instances. Additionally, it has fields that record the window's name and the attributes of the window (Can it be closed or resized? How big or small can the user make it? What should happen when the user closes the window?).

TWindow has special methods for opening, closing, activating, and resizing; it also has methods for handling commands a window is likely to receive via the mouse or the menu. If the user clicks the close box, the zoom box, or the size box, TWindow's methods can handle the command. Because the appearance and behavior of windows (but not their contents) are pretty much standardized, MacApp's own code can handle almost anything a window has to do. You don't need to do much with windows, except tell MacApp initially what sort of attributes you want them to have. Note that MacApp does not know how to manipulate the contents of a window, just the window frame and its controls.

* **Note**: If you are familiar with *Inside Macintosh*, you should note that TWindow is MacApp's representation of a Window Manager window. In fact, a Window Manager window is created for every instance of TWindow, and the visual representation of the window depends on the type of Window Manager window you've defined in your resource file.

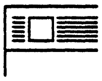
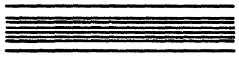
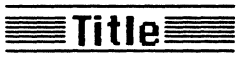
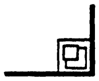
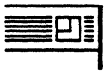
Name of element	Appearance on screen	How user manipulates control
Close box		Clicks to close the window.
Title bar		Drags to move the window.
Title		Not a control; displays name of window.
Size box		Drags to adjust size of window.
Zoom box		Clicks to make window as large as possible; clicks again to return window to its original size.

Figure 4-6
Parts of the window

The window instance in this example has three subviews: a scroller, a scroll bar, and a palette.

Scrollers

The **scroller** is the most difficult of the three subviews to understand, perhaps because it is has no visible correlate on the screen. Once again, it is helpful to remember that *objects* in the object-oriented programming sense are not necessarily tangible, they are merely data and the methods that manipulate that data for some particular purpose. The purpose of the scroller class, called TScroller, is to calculate coordinate translations to create the illusion of scrolling through a view, either horizontally or vertically or both. The user typically accomplishes scrolling by manipulating scroll bars. In that case, the scroll bars and the scroller work in concert. But because MacApp provides a scroller separate from the scroll bars, you may implement any other scrolling mechanism that you care to.

Generally, you can create instances of TScroller directly, without having to override any of its methods, so for the most part you can treat TScroller as a black box. Just be aware that anything scrollable is probably implemented as a subview of an instance of the TScroller class. Your main job is initializing the scroller with the correct references to scroll bars (if you want to use scroll bars) and making a scroller the superview for anything you want scrolled. TScroller has a number of fields that record offsets, scaling factors, and the increments by which you want the subviews to be scrolled. It has methods that actually perform the translation and coordinate with the scroll bars to keep all objects concerned synchronized.

Main content

The main content of the window is a separate view, which is often a subview of a scroller, as it is in this example. The content of any window is hard to predict, so MacApp's library of code does not contain specific classes for every possible main content view. Chances are overwhelming that you will have to create your own descendant of TView that knows how to display whatever it is you need to display. You will probably want to add fields that keep track of what is being displayed, and methods that will do the drawing, probably by calling the Draw methods belonging to the instance of whatever is to be displayed.

Like all other views, the main content view is an event handler. The main content view is usually responsible for a great many of the commands a user is likely to issue. For instance, in a graphics program, if the user is supposed to draw a box by holding down the the mouse button and dragging, it is the view instance that will be responsible for intercepting the mouse click and calling the appropriate methods and routines in response. Often the content view will have no subviews, but it could.

Palettes

In addition to the window, scroller, scroll bar, and main content views, the particular window in the example above also has a **palette**. Since every palette is different, the MacApp library has no special class for creating palettes, so you would probably create a palette class as a direct descendant of TView or perhaps TGridView. (Creating a palette is really not much different from creating the main content view.) Palettes are generally not scrollable (though they could be), so usually the palette will have the window instance as its superview. (If it were scrollable, it would have a scroller instance as its superview.) The most important methods you need to customize in a palette view are the ones responsible for responding to the user's click inside the palette, especially the methods that highlight and record the selection.

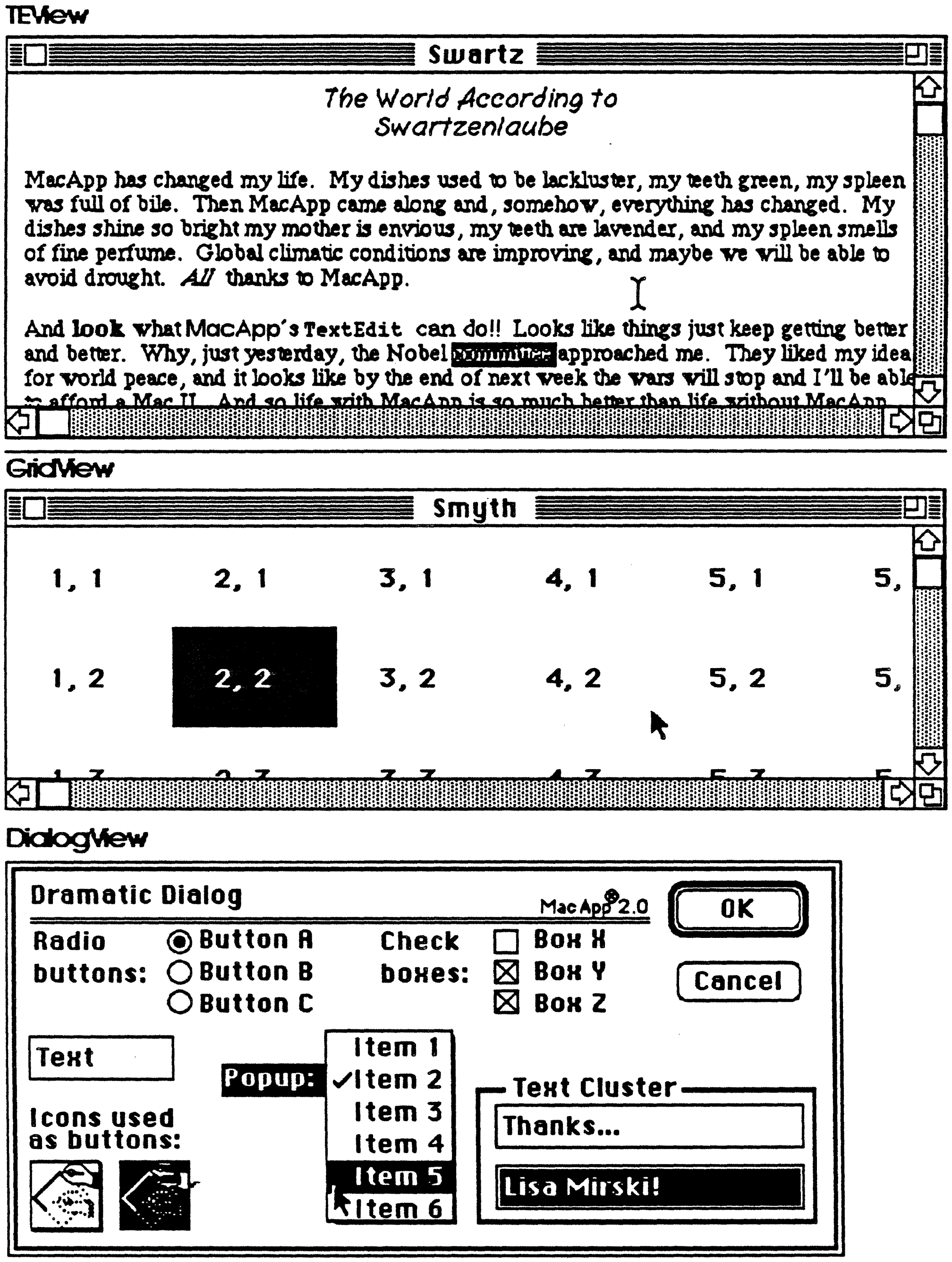
Scroll bars

Another object in the example above is the **scroll bar**. This particular window has only a vertical scroll bar. The scroll bar is an instantiation of the TSScrollBar class (Ancestry: TObject → TEvtHandler → TView → TControl → TCtlMgr → TScrollBar → TSScrollBar). TSScrollBar is an indirect descendant of TControl, which in turn is a descendant of TView. Because scroll bars are highly conventionalized, you do not need to be too concerned with the inner workings of this class. It has fields that point to the scroller instance(s) it is associated with and that record the limits of the scrolling, and methods that respond to the user's mouse clicks and maintain the appearance of the scroll bars. Your scroller instance has methods to create instances of TSScrollBar.

* **Note:** MacApp has two classes with similar names, both of which are responsible for displaying and manipulating scroll bars: TScrollBar and TSScrollBar. The latter—the one with two S's—is a descendant of the former which has been customized to work with scroller instances. The extra *S* stands for "scroller". The former class, TScrollBar, is a simple control which you can use as an analog dial in dialogs.

Other specialized views

Though no one could anticipate every sort of view a programmer might want to offer, MacApp does provide several specialized descendants of TView that you can use to display text, columnar data, or dialog boxes. Text can be displayed and manipulated with an instance of the class TEView, which is part of the optional UTEView building block. Columnar data, such as you might see in a spreadsheet program, can be manipulated and displayed with an instance of the class TGridView. In fact, members of this class are used to display information as part of the UInspector unit for the debugger. And sophisticated dialog boxes, including static text, editable text, check boxes, radio buttons, and so on can be manipulated and displayed with a wide variety of classes found mostly in the UDialog unit. These specialized views are pictured in Figure 4-7. (Appendix A, "Changes Since MacApp 1.1," lists all the view classes provided in MacApp 2.0.)



TEView

Swartz

The World According to Swartzenlaube

MacApp has changed my life. My dishes used to be lackluster, my teeth green, my spleen was full of bile. Then MacApp came along and, somehow, everything has changed. My dishes shine so bright my mother is envious, my teeth are lavender, and my spleen smells of fine perfume. Global climatic conditions are improving, and maybe we will be able to avoid drought. *All* thanks to MacApp.

And **look** what MacApp's `TextEdit` can do!! Looks like things just keep getting better and better. Why, just yesterday, the Nobel committee approached me. They liked my idea for world peace, and it looks like by the end of next week the wars will stop and I'll be able to afford a Mac II. And so life with MacApp is so much better than life without MacApp.

GridView

Smyth

1, 1	2, 1	3, 1	4, 1	5, 1	5,
1, 2	2, 2	3, 2	4, 2	5, 2	5,

DialogView

Dramatic Dialog

Radio buttons: ● Button A ○ Button B ○ Button C

Check boxes: □ Box X ☒ Box Y ☒ Box Z

Text

Popup: Item 1 ✓Item 2 Item 3 Item 4 Item 5 Item 6

Text Cluster

Thanks...

Lisa Mirski!

Icons used as buttons:

Figure 4-7
TEView, TGridView, and TDialogView

Executing commands

In addition to storing and displaying data, applications must allow users to manipulate the data. In the Macintosh world, any input from the user, even typing simple text, is considered to be a command. However, because MacApp reserves the word "command" for menu commands, the term **user event** is used to describe all sorts of commands. Users have two tools for creating user events: the mouse and the keyboard. With the mouse, the user can choose commands from menus or click the controls of a window or in its content. In the content of a window, users can click an object to select it, drag an object to move it, or click in a palette to choose a tool. With the keyboard, users can type text or can enter Command-key combinations. Command-key combinations are equivalent to menu items; for instance, there is no difference in most applications between choosing Save from the File menu or typing Command-S from the keyboard.

When the user chooses a menu command using either the keyboard or the mouse, MacApp receives a report of the event from the Macintosh ROM. Methods in the MacApp library begin a search for the correct instance to handle each sort of event. The algorithm for this search is the topic of the next chapter. But there are rules of thumb for figuring out what instances are responsible for handling which commands. First of all, only event-handler instances respond to user events. The main players in the game are these instances: the application, the document(s), the window(s), and the specialized view(s). MacApp gives each of these objects a shot at resolving any incoming command, in a specific order. There is generally only one candidate for each class. For example, only one view, window, and document are active at any one time, and there can only be a single instance of the TApplication class.

Each class is responsible for handling user events related to its own area. The application object is responsible for user events that could be executed even when no window is open, including Quit, About, New, and Show Clipboard. The document is responsible for reading and writing data to disk. The window is responsible for clicks in the size box, title bar, and zoom box. Scroll bar instances handle clicks in the scroll bars. The view is responsible for manipulating the images it shows; usually it has methods that actually take input from the mouse to draw a new object or manipulate an old one, and then report the object's existence to the document so it can be recorded.

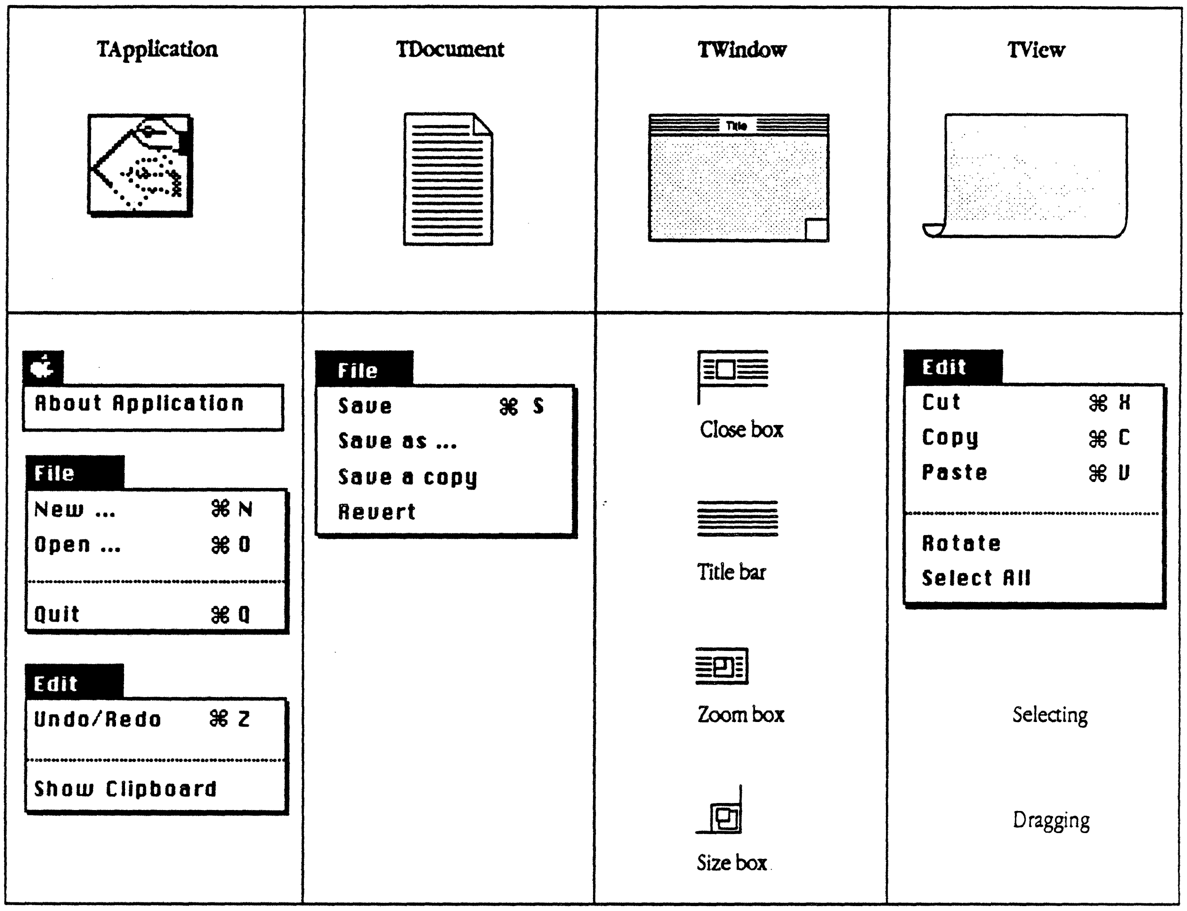


Figure 4-8
The main event handlers and their responsibilities for commands

The most important methods for handling the user's commands are DoMenuCommand, DoMouseCommand, and DoKeyCommand, collectively called the Do*Whatever*Commands. Because these methods are declared in TEvtHandler, they are shared by all event handlers.

- DoMenuCommand handles all menu commands, whether they come from a mouse click or from a Command-key combination.

- DoMouseCommand handles mouse clicks that occur outside the menus.
- DoKeyCommand handles all keystrokes not combined with the Command-key.

These three methods generally do *not* do the work of the command itself—they either call other methods of the same instance or methods of some other instance. The particular manner in which the Do*Whatever*Command does its work depends on whether the user chooses a simple command or a complex one. The terms *simple* and *complex* have a specific meaning in this case. A command is complex if it is undoable or if it requires mouse tracking. Many Macintosh commands fall into this category. For one thing, most commands change the document in one way or another, and thus they should be undoable. Many others require the mouse to be tracked. If a command is not undoable and does not require the mouse to be tracked, it is a simple command.

Table 4-3
Examples of simple and complex commands

Command	Reason
Simple commands	
Save (menu command)	Cannot be undone and does not require the mouse to be tracked.
Select All (menu command)	Cannot be undone and does not require the mouse to be tracked.
Complex commands	
Delete or Clear	Must be undoable since it changes the document.
Select item (with mouse)	Requires mouse to be tracked.
Move item (with mouse)	Requires mouse to be tracked and must be undoable.

The Do*Whatever*Command method usually executes simple commands by calling other methods. For example, the Save command is handled by a document instance which calls the document's own Save method. The

Do*Whatever*Command executes complex commands by creating an instance of some command class, methods of which will then be called by MacApp. You will need to design a command class for every complex command you wish to implement. Command classes should always be created as descendants of TCommand.

The TCommand class is an abstract class. It has fields that record important information about the command, including a reference number of the command, the relevant view instance, the object instance that is the target of the command, and information about the status of the command, such as whether the command is undoable, requires tracking, and whether it caused a change. It has methods that track the mouse and that do and undo the command.

In sum, simple commands are executed without instantiating new objects but complex commands—undoable commands or commands that require mouse tracking—require that a member of a command class be instantiated.

Summary

Now you should understand how MacApp is organized: how the files and units are related; the significance of the ancestry tree for the classes; and, most importantly, the conceptual design behind the MacApp library. The next chapter moves from this static view of organization to a dynamic picture of how a MacApp application works while it is running, focusing on how it handles events.

Chapter 5

Introduction to MacApp: Flow of Control

People learning MacApp may often ask themselves, "just who is in charge here, my code, or MacApp's?" At any moment, the user might click the mouse button; some part of the MacApp library would handle the click by calling a piece of your code. Your code would probably then relinquish control back to MacApp, which in turn might call a ROM routine to draw something on the screen. Then the user might click something else, starting the process over again. Thus, control over the computer is constantly traded among the user, your code, MacApp, and the ROM routines. This perpetual shuffling is called **flow of control** and is the subject of this chapter.

Any application might have to handle a great number of events while it is running; fortunately, they fall into a very few categories. Macintosh applications spend most of their time idling in an outer loop, called the **main event loop,** waiting for something to happen. When an event comes along that needs to be handled, the main event loop sorts it into one of these categories, opening a path down which the event will flow until it is taken care of. If you find it hard to picture this, imagine one of those coin banks that sorts your change into piles of pennies, nickels, dimes, and quarters. Every time a user creates an event, it is as if he or she has dropped a coin into the top. MacApp, like the coin bank, evaluates the event and sends it down the correct chute.

In MacApp there are eight chutes or categories, each representing a different sort of event. Each event falls into one of these eight categories. After the event starts down its path, MacApp must decide which instance should handle it. The process of assigning events to instances is one of the more subtle aspects of flow of control in MacApp, and it is the purpose of this chapter to help you understand how those decisions are made. Some of these assignments will be intuitive to you, even from the beginning; others will not concern you, because some events can be handled without ever having to call your customized code; but still others use wily algorithms to do the assigning, algorithms you must understand to create the proper code to handle them. Chances are, once you understand these, you will understand MacApp.

This chapter begins by explaining the main event loop and the mechanism by which events are assigned to instances, and at the same time introduces the diagramming method this manual uses to show flow of control. Then it will discuss which of the eight categories ought to concern you, and how you can get a feel for which events will be assigned to which instances.

Flowcharts and overview

In the beginning, you may be curious about how MacApp works, and there is nothing to stop you from opening up MacApp's code and reading along. Reading the raw code can be frustrating—it is hard to tell what's a method, what's a global routine, and what comes from the ROM. Moreover, you can get lost in the detail of the code. The flowcharts allow you to see an overview, so if you do decide to read the source code, at least you'll have a road map.

Once the program is running, your application will spend all its time in the methods of your various instances, except for a few lines of startup code and some global routines here and there which you or MacApp will call. A simple mouse click in one of your application's windows could generate messages among a handful of instances and several dozen of their methods. Rather than presenting you with every possible path along which the data could flow for a particular event, the flowcharts in this manual present the flow for some specific event, which will be explicitly defined. Each flowchart will be small enough to fit on one page, which means that many parts of it will be abstract and not detailed; those parts will be defined in further flowcharts.

The first flowchart, in Figure 5-1, is an overview of an entire MacApp application, from the time the user double-clicks the icon to the time he or she quits. It is not very detailed or very interesting, and doesn't look much different from a traditional procedure-based program. (Also, it is a good example of a limited chart. There are two other paths an application could take when it starts up, and this one shows only the simplest path—it assumes the user opens the application from the Finder™ without requesting that any documents be opened or printed.)

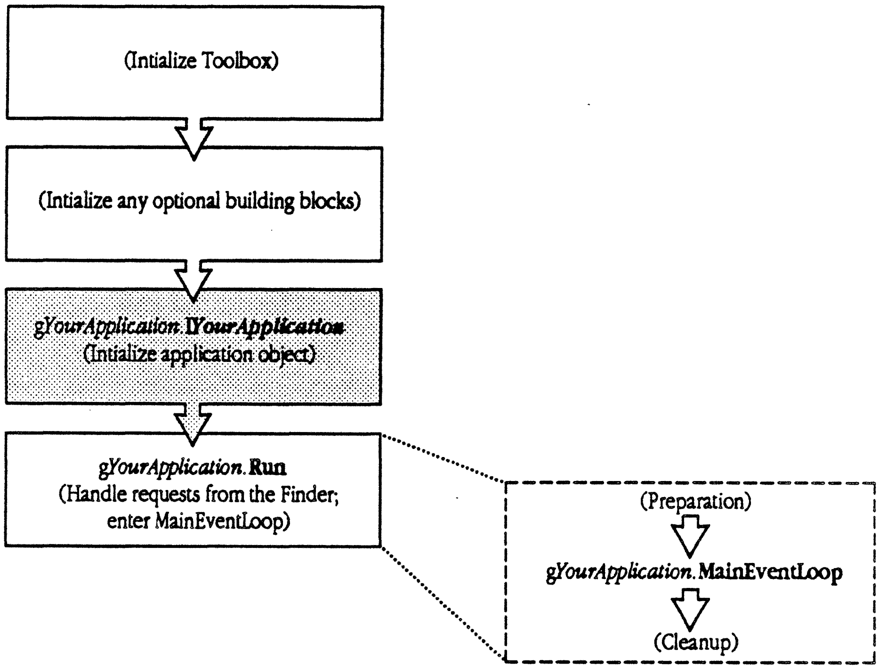

(Intialize Toolbox)
(Intialize any optional building blocks)
gYourApplication.IYourApplication
(Intialize application object)
gYourApplication.Run
(Handle requests from the Finder;
enter MainEventLoop)
(Preparation)
gYourApplication.MainEventLoop
(Cleanup)

Figure 5-1
Biggest overview

First, here are some conventions. As in a traditional flowchart, each box can be considered a single operation. Arrows indicate the direction of flow. All the flowcharts will start at the top, but almost none of the rest of the charts will be this simple. In this chart, the flow is entirely linear, starting at the top and dropping off the bottom, which is the end of the application. In all charts, each box will contain one of five items: the name of an object's method, the name of a global routine, the name of a ROM call, a Pascal statement, or a general description. If there is an asterisk in a box, it means that you can find more information about the indicated routine in the box in some other flowchart. If the box is shaded, it means that the MacApp version of that routine does almost nothing or does not exist, and you should write it or override it if you want it to work. In the first flowchart, the shading in the method I*Your*Application indicate that you should create or override this method. These conventions are summarized in Figure 5-2.

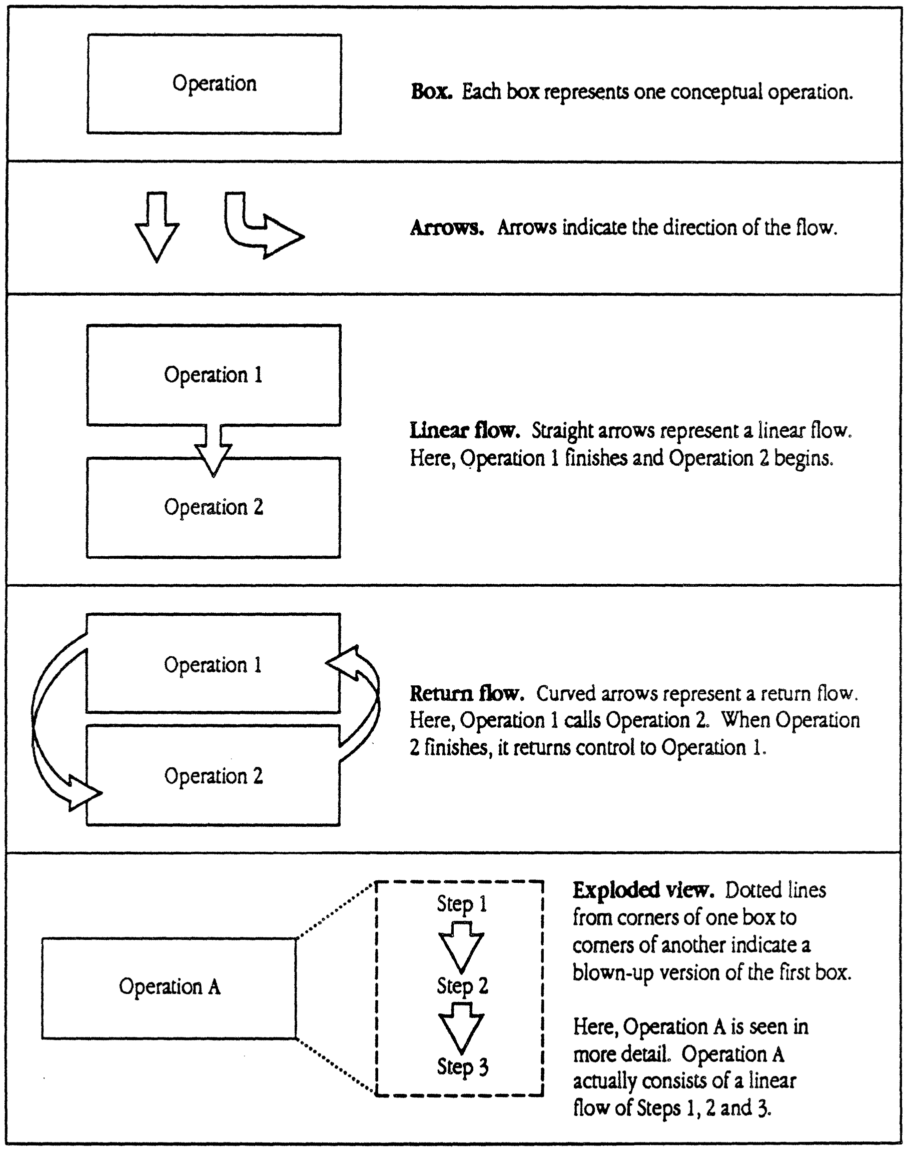
You may have noticed that most of the boxes in Figure 5-1 had asterisks in them. Some of these boxes are self-explanatory, even if you don't know the contents of the code. The routine Cleanup at the end of the program must take care of last-minute details. Other routines, though they require further explanation, are not very important for helping you to grasp the basic idea of MacApp.

One of the methods, MainEventLoop, is the loop discussed previously. Your program will spend much its time in this loop. Much of the rest of this section will be spent investigating this method of the TApplication class; even so, only the first few layers of its complexity will be described.

Figure 5-3 is the first nonlinear flowchart. The action both starts and ends at the method MainEventLoop. The first thing this flowchart tells you is that MainEventLoop calls PollEvent, which returns control to MainEventLoop after it's finished. MainEventLoop repeatedly calls PollEvent until the user chooses Quit.

But the flowchart also tells you that if an event is ready to be handled (or, more accurately, if a non-null event has occurred), PollEvent calls HandleEvent. Furthermore, it tells you what happens inside HandleEvent: the record of the event, which comes from the ROM, is stored in a special configuration. Afterward, HandleEvent calls DispatchEvent and PostHandleEvent.

Graphic Conventions



Box. Each box represents one conceptual operation.

Arrows. Arrows indicate the direction of the flow.

Linear flow. Straight arrows represent a linear flow. Here, Operation 1 finishes and Operation 2 begins.

Return flow. Curved arrows represent a return flow. Here, Operation 1 calls Operation 2. When Operation 2 finishes, it returns control to Operation 1.

Exploded view. Dotted lines from corners of one box to corners of another indicate a blown-up version of the first box.

Here, Operation A is seen in more detail. Operation A actually consists of a linear flow of Steps 1, 2 and 3.

Figure 5-2
Conventions in flowcharts

Typographic Coventions

aCommand.DoIt	**Method call.** The dot in the middle of the name indicates this is a method call. Here, the DoIt method of the aCommand instance is being called.
FailNil	**Global routine.** The lowercase name with no dot indicates this is a global routine. Here, the FailNil routine is being called.
NEWO	**Pascal reserved word.** The name in all uppercase letters indicates this is a Pascal reserved word.
(stores data)	**Comment.** The parentheses indicate a comment, as opposed to actual code. Here, an operation that stores data is represented.
GetNewWindow (ROM)	**ROM call.** The ROM in parentheses after the name of the routine indicates a ROM call.
▨ Text	**Highlighted grey.** A grey box indicates that you will need to override this method.
Text*	**Asterisk.** An asterisk after text indicates that the step in this box is expanded in another flow chart.

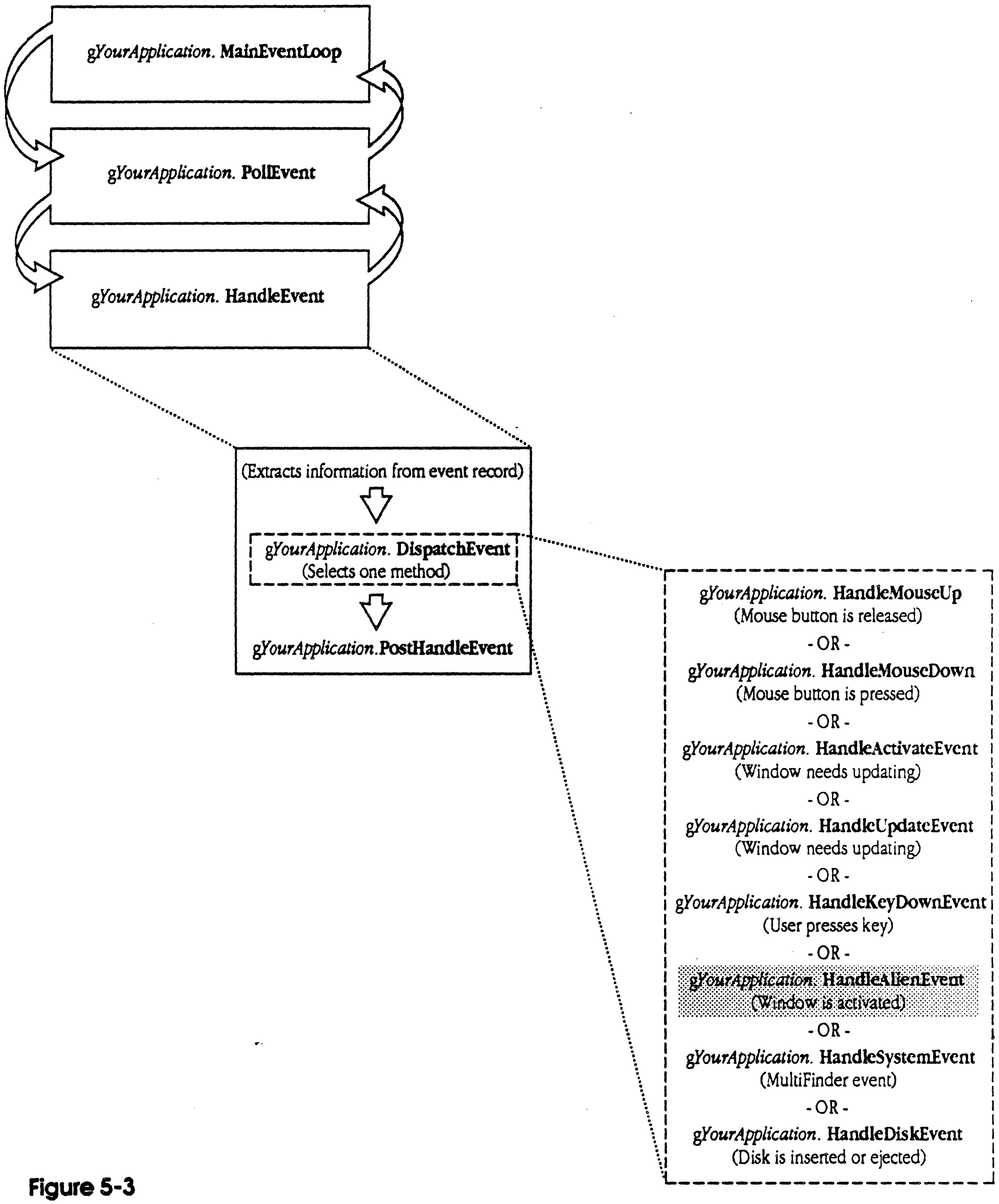

gYourApplication. MainEventLoop
gYourApplication. PollEvent
gYourApplication. HandleEvent
(Extracts information from event record)
gYourApplication. DispatchEvent
(Selects one method)
gYourApplication.PostHandleEvent
gYourApplication. HandleMouseUp
(Mouse button is released)
- OR -
gYourApplication. HandleMouseDown
(Mouse button is pressed)
- OR -
gYourApplication. HandleActivateEvent
(Window needs updating)
- OR -
gYourApplication. HandleUpdateEvent
(Window needs updating)
- OR -
gYourApplication. HandleKeyDownEvent
(User presses key)
- OR -
gYourApplication. HandleAlienEvent
(Window is activated)
- OR -
gYourApplication. HandleSystemEvent
(MultiFinder event)
- OR -
gYourApplication. HandleDiskEvent
(Disk is inserted or ejected)

Figure 5-3
Flow of control: MainEventLoop

Finally, the flowchart details how the DispatchEvent method works. The OR's between the boxes indicate that only one of these methods is called; the condition for choosing that particular method is indicated in parentheses in each box. In fact, as you might have guessed, this all comes in the form of a big CASE statement in the code:

```
CASE what OF
      mouseUp:
      commandToPerform := HandleMouseUp(theEventInfo);

      mouseDown:
      commandToPerform := HandleMouseDown(theEventInfo);

      activateEvt:
      commandToPerform :=
            HandleActivateEvent(theEventInfo);

      updateEvt:
      commandToPerform :=
            HandleUpdateEvent(theEventInfo);

      keyDown, autoKey:
      commandToPerform :=
            HandleKeyDownEvent(theEventInfo);

      diskEvt:
      commandToPerform := HandleDiskEvent(theEventInfo);

      app4Evt:
      commandToPerform :=
            HandleSystemEvent(theEventInfo);

      OTHERWISE
            commandToPerform :=
                  HandleAlienEvent(theEventInfo);

      END; {case}
```

Figure 5-4
The giant case statement in TApplication.DispatchEvent

The overall picture should now be apparent. MainEventLoop repeatedly calls PollEvent. Whenever there *is* an event, PollEvent will call HandleEvent, which will record it, dispatch it by calling one of nine methods, and then execute PostHandleEvent. When PostHandleEvent completes execution, HandleEvent also finishes and returns control to PollEvent, which in turn returns control to MainEventLoop. The process repeats until the user chooses Quit. Keep in mind that you can override any of the methods listed in this chart if you want to change the way MacApp handles a specific type of event. However, MacApp is capable of handling most events the user can generate; but there are other (**alien**) events that it does not anticipate, most notably network events, such as AppleShare® packets. You must override HandleAlienEvent if you wish your application to respond to these events.

Note that this flowchart, like all the others in this book, is incomplete in several ways. First, it does not tell you what happens in PollEvent if there is not an event waiting; that is shown in another flowchart. Second, it does not tell you everything that happens. For instance, it has left out all the error-checking code and the method call that actually checked for the event. This streamlines the flowchart, allowing you to focus on the main points.

Assigning events to instances

By now, you should have a mental picture depicting MacApp constantly on the lookout for events. If PollEvent sees an event, it will pass it on to DispatchEvent, which will send it down one of the eight possible paths. (If there is *no* event, PollEvent will open a ninth path so any work that should be done while the application is idling can have an opportunity to execute; there will be more about idling later in this chapter.) Now it is time to look at these eight paths in more detail.

First, how does MacApp know which sort of event has taken place? This is perhaps the easiest question of all. Whenever there is an event, it is intercepted by toolbox routines that report exactly what sort of event took place. All DispatchEvent has to do is look in a certain field of the event record. What it will find there is one of the following eight sorts of events:

MouseDown — The user presses the mouse button. What action MacApp takes depends on where the cursor was when the button was pressed and where the mouse was dragged while the button remained down.

MouseUp
The user releases the mouse button. Generally this event is ignored, except when the user drags the mouse.

Activate
A window is activated or deactivated. In general, an activate event occurs as an indirect result of a user action. For instance, a user may click on an inactive window, which in turn will cause a ROM routine to generate two activate events, one to deactivate the old window and another to activate the new one.

Update
A window needs refreshing. Usually an update event occurs as an indirect result of a user action. For instance, if a user had one window partially covered and then uncovered it, a ROM routine would generate an update event for the uncovered region to be redrawn.

KeyDown
The user presses a key or a key combination on the keyboard. KeyDown events include both plain keystrokes and Command-key sequences (a key pressed while the Command key is held down).

Disk
The user inserts or ejects a disk.

System
The user enters or leaves the application with MultiFinder.

Alien
Other events, such as network events or events which you define yourself, are called alien events. By default, MacApp ignores alien events.

Which of these events do you need to be concerned about? Remember, as a rule, MacApp will do as much as it can without being clairvoyant. That means that MacApp's code can take care of many of these events for you, as long as you are happy with the default operation MacApp gives you. In fact, that's a crucial idea behind MacApp. The application instance catches events as they come in and assigns them to the instances responsible for handling them. *You* don't have to be concerned with events themselves. Instead, all you need to know is which methods and classes you have to be implement or override. But in order to give you some intuition about which methods and classes are of concern, here is an overview of how MacApp deals with events of various sorts.

Three of the classes of events are dead ends that you can typically ignore: MouseUp events are generally discarded or handled automatically; Disk events require no action except error checking; and Alien events can be neglected, unless you are writing a networking or special-purpose program.

* **Note:** There are three typical cases when MouseUp events are not ignored: when the user drags the mouse; when the user double- or triple-clicks the mouse; and when the user releases the mouse button over a TControl instance. In each of these cases, MacApp's code intercepts the event and handles it appropriately.

Three more of the event classes require consistently predictable actions, which MacApp can take care of for you, so you can ignore them unless you want some special action: Activate, Update, and Switcher events require no special attention, except that update events will call the Draw methods of any views that need updating. Since MacApp cannot anticipate how you want views to be drawn, the default Draw methods are empty. You must override them if you want anything to be drawn. All these considerations aside, that leaves two sorts of events you need to pay special attention to: KeyDown and MouseDown events. These events can be further subdivided as shown below.

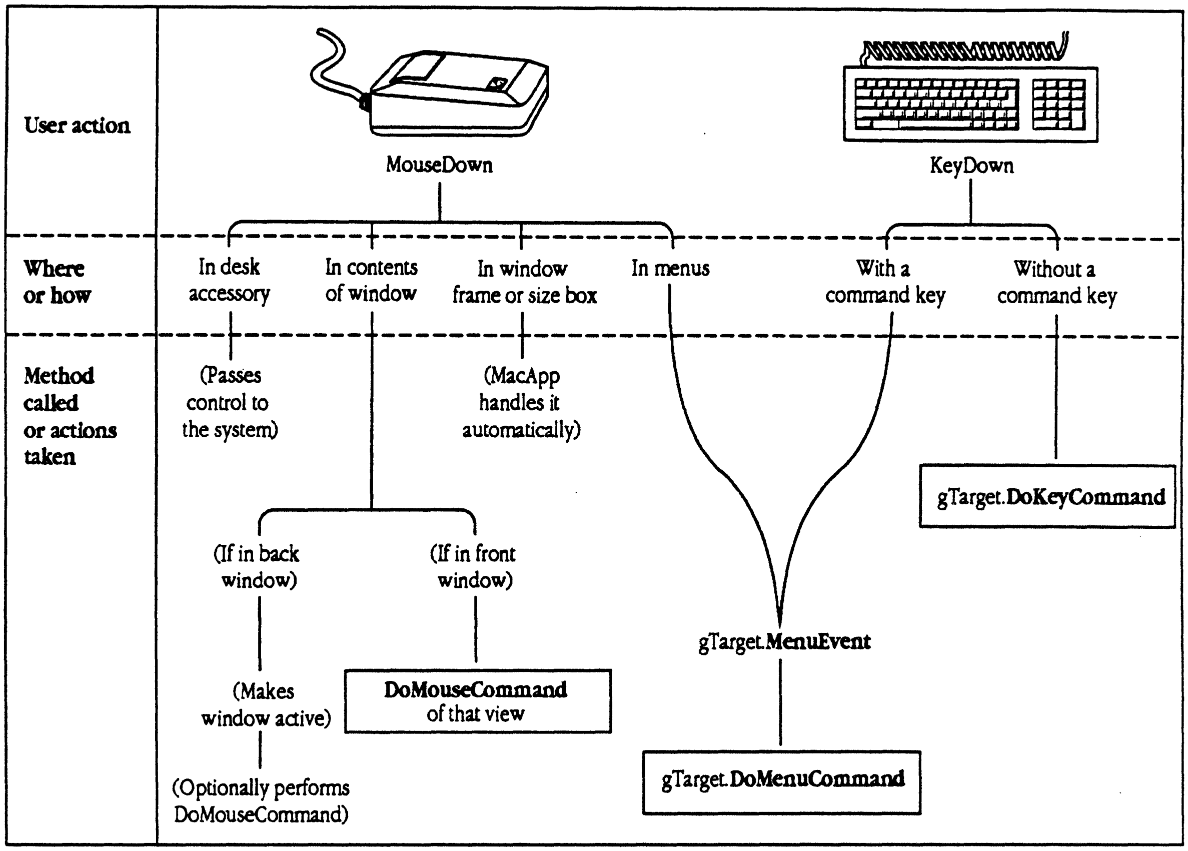


Figure 5-5
The events most important to you

As shown in Figure 5-5, there are a variety of ways mouse clicks can be handled, depending on the context of the click. If you are wondering how MacApp knows what the context was, it (again) comes from information provided by the ROM in the event record. By default, the MacApp code will handle many of the MouseDown clicks by itself, including clicks in the various window controls, such as the scroll bars and the zoom, size, and close boxes. MacApp will also handle clicks in any desk accessories by handing control over to the system. The two MouseDown events you should be concerned about are clicks in the content of a window and clicks in the menus. These may require customized action from your application, action MacApp could not anticipate. Figure 5-5 also shows two sorts of KeyDown events: ordinary keystrokes and Command-key combinations, both of which may require your customized code.

Because Command-key combinations are treated as equivalents of menu commands, there are only three sorts of MouseDown and KeyDown events that your code absolutely must handle: (1) clicks in the content of a window, (2) plain keystrokes, and (3) menu commands (or their keyboard equivalents). There is a fourth area that may be of concern to you: what happens if there is no event waiting? In that case, the program enters an Idle mode, treating the idling as if it too were an event (instead of a nonevent), optionally executing your customized code that may, for instance, maintain cursors or check for network activity. By default, when your application is in Idle mode, the application instance calls methods that can change the shape of the cursor depending on where the user has moved it, but even these methods are empty until you override them.

So these are the four truly interesting sorts of events—these are the places, in addition to the Draw methods of the TView instances, where much of your customized code will come in. How do you write code that MacApp will call? MacApp calls methods with certain names in certain circumstances. When MacApp responds to a click inside a window, it calls a method called DoMouseCommand. When MacApp responds to a menu command (or Command-key equivalent) it calls a method named DoMenuCommand; and a plain keystroke is handled with a DoKeyCommand method. Finally, when MacApp is idling, it calls DoIdle methods. You must override these methods to specify how these events will be handled.

But methods belong to particular instances. How does MacApp's code determine which instance should handle each event? All the events are handled by some descendant of TEvtHandler, but that doesn't narrow it down much: should a command be handled by a view, a window, a document, or the application instance? MacApp follows a set of protocols that invoke one of several chains of responsibility to determine which object is responsible for handling a particular event. There are three of these chains: the command chain takes care of plain keystrokes and menu commands (whether they are chosen by clicking menu items or typing Command-key combinations); the click chain takes care of mouse clicks in the content region of a window; and the idle chain takes care of idle "events."

The chains of responsibility

A chain is a linear connection of individual links. A chain of responsibility in MacApp is a linked list of instances, with a beginning and an end, which MacApp can traverse one link at a time, checking to see if each object along the way can handle the event. Once an object handles the event, the search may be called off. The connection between one link and the next is formed by way of **object reference fields.**

Because there is not necessarily a linear, chainlike relationship among the instances in a MacApp application, the method by which this chain is constructed from some underlying structure is a matter of importance, as are the method by which the head of the chain is chosen and the method by which the chain is traversed.

The command chain

The **command chain** is traversed whenever the user types at the keyboard or chooses an item from a menu, whether the item is chosen by pointing and clicking or by typing its keyboard equivalent. The command chain is the most complicated and the most interesting of the three chains because commands can be handled by almost any descendant of TEvtHandler, and it is not obvious where the search should start and how it should proceed. Certain commands should belong to a document, such as commands that save or print a particular document; others belong to a window, such as a zoom command; still others might belong to a particular view, such as the Cut and Paste commands. Still others might belong to the application as a whole, if they should be available all the time and do not concern a particular document, window, or view; for example, the New or About commands are always available even if no window is open.

But these instances do not have a linear relationship at all; if anything, they form a complex web of relationships, with the single application object at the center, and view and subview objects around the periphery.

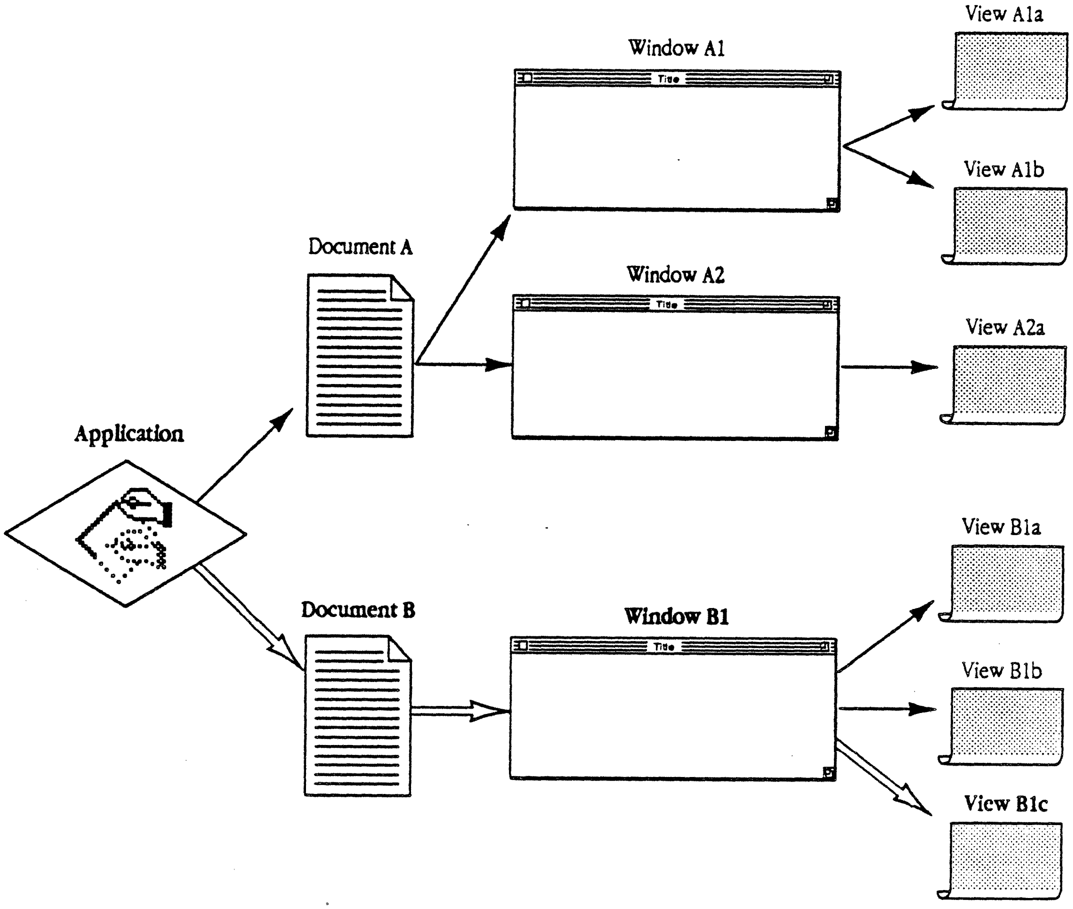


Figure 5-6
The command web

The different instances keep track of each other by way of object reference fields, represented by the arrows in the figure. The application object is an exception, because instead of having any of its information in fields, it stores its data in global variables (because it makes the code run faster). So the application instance has a list of all the documents associated with it; the windows record which document, if any, they are primarily associated with; and views have reference fields that point to their superview (which can be a window) and to a list of subviews (if any).

How the chain is built

Every descendant of TEvtHandler has a field called fNextHandler, which stores the next link in the chain. MacApp initializes each event-handling instance to have the correct fNextHandler. Subviews are initialized so that their fNextHandler points to their superview, which may or may not be a window. But since every subview is inside some window (which is itself a view object), this chain will eventually point to a window object.

Window objects are initialized with their fNextHandler field pointing to the document they are primarily associated with. (When you create the window, you provide this information by passing the document's handle as a parameter.) Documents in turn are initialized so their fNextHandlers point to the application. Some windows, like palette windows, are not associated with any document; these windows point directly to the application. The application points to NIL, so it is the end of the chain.If you look at the four objects connected by the open arrows in Figure 5-6 you can see one target chain.

The net result is a sensible chain of responsibility from specific to general. A subview gets first crack at any command or keystroke, followed by the window, the document, and finally the application. You may modify the chain by changing the fNextHandler field of any object.

How the head of the chain is determined

MacApp has a global variable named **gTarget** which always points to the head of the chain. The philosophy behind gTarget is that there is always some part of the application where the user fixes his or her interest, and that is where gTarget should point. There are only a few methods in MacApp that set gTarget, the most important of which is TWindow.Activate, a method belonging to window instances. This method is called whenever a window is activated or deactivated. When you, as the programmer, write the methods that create the window objects, you will specify which other instance should become the target when that window is activated. This will almost always be the most important or topmost view inside the window. That information is stored in a field of the window instance, a field that TWindow.Activate checks whenever the window becomes active.

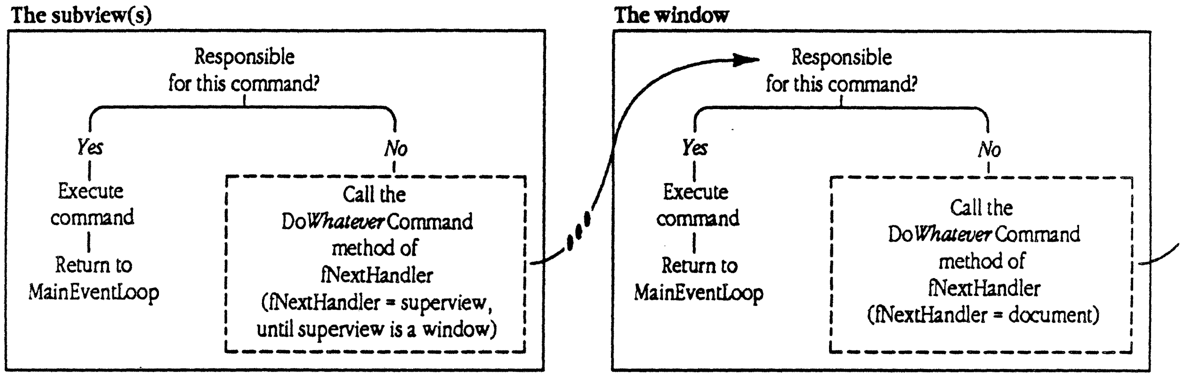
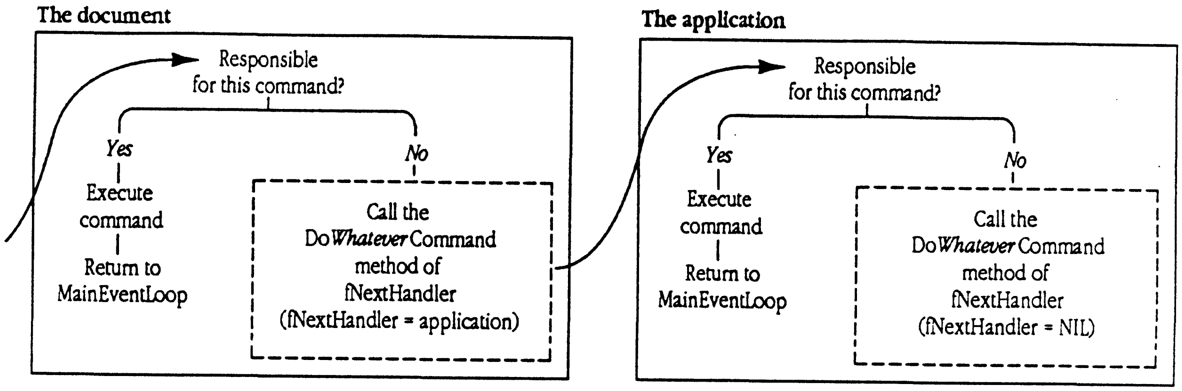

The subview(s)
Responsible for this command?
Yes
Execute command
Return to MainEventLoop
No
Call the DoWhateverCommand method of fNextHandler (fNextHandler = superview, until superview is a window)
The window
Responsible for this command?
Yes
Execute command
Return to MainEventLoop
No
Call the DoWhateverCommand method of fNextHandler (fNextHandler = document)

Figure 5-7
Diagram of fNextHandlers

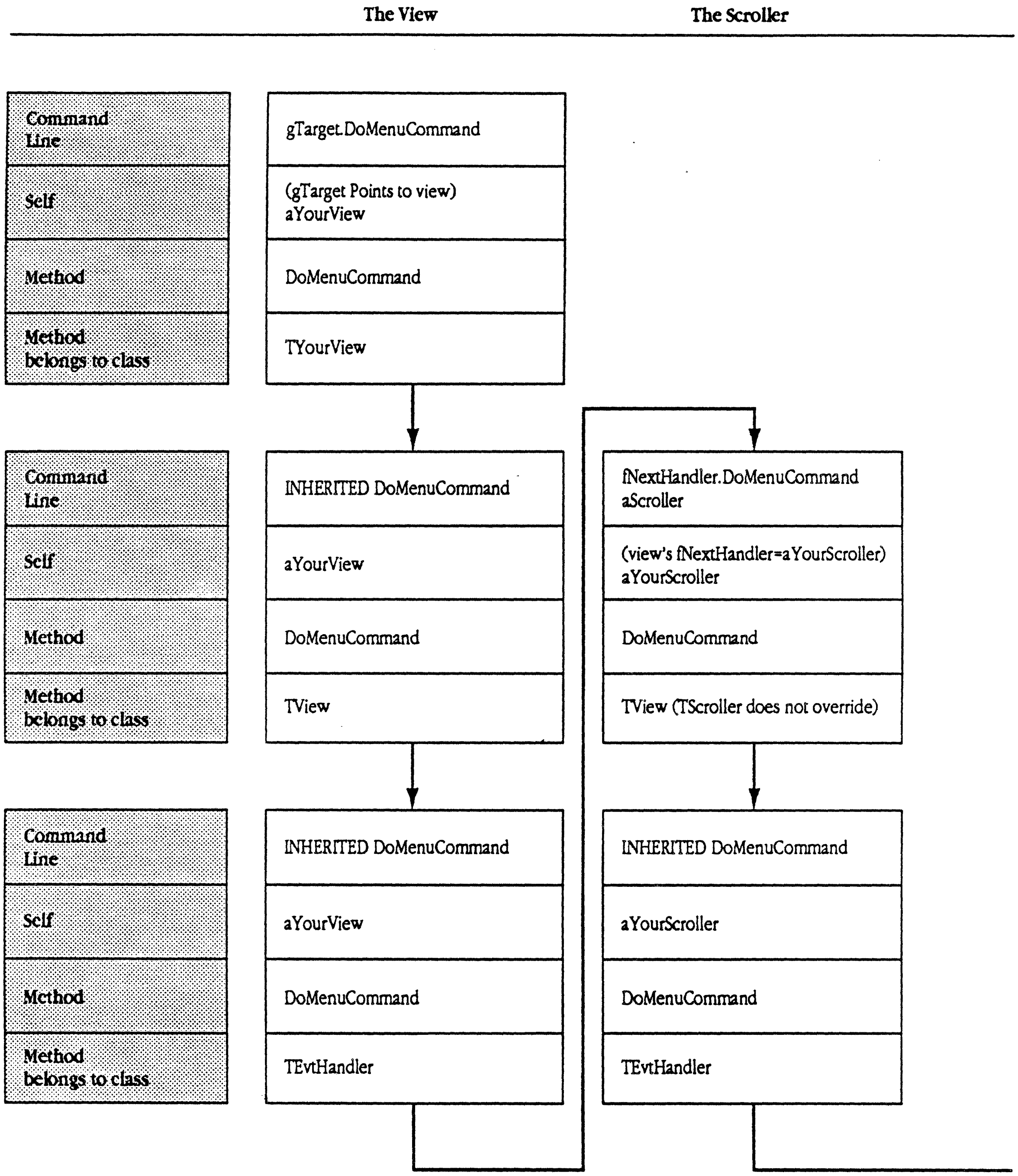
How the chain is traversed

When the user types a plain keystroke or chooses a menu command, MacApp calls a method of whatever is currently gTarget. In the case of keystrokes, it calls gTarget.DoKeyCommand; in the case of menu commands (or Command-key equivalents) it calls gTarget.DoMenuCommand. These methods are often CASE statements that call the INHERITED DoMenuCommand or DoKeyCommand if none of the specified conditions is met. Eventually, this INHERITED method will go through the class hierarchy and reach TEvtHandler, which will call a DoMenuCommand or DoKeyCommand belonging to whatever the fNextHandler instance is. This process continues on down the chain until one of the links handles the event or has an fNextHandler equal to NIL. (The application instance always has an fNextHandler that equals NIL.)


The document
Responsible for this command?
Yes
Execute command
Return to MainEventLoop
No
Call the Do*Whatever*Command method of fNextHandler (fNextHandler = application)
The application
Responsible for this command?
Yes
Execute command
Return to MainEventLoop
No
Call the Do*Whatever*Command method of fNextHandler (fNextHandler = NIL)

For example, imagine that a user is in a program that was written with MacApp. Imagine that when the user chooses the Quit command, there are four instances: a main content view called yourView; a window called yourWindow; a document called yourDocument; and an application called yourApplication. If the user were inside a window, gTarget would probably point to the view instance. Then, if the user chose Quit, MacApp would first call gTarget.DoMenuCommand. Remember that gTarget points to the item most likely to be of interest. When a window is activated, that is usually one of its views. Since the user was inside a window when he or she issued the command, gTarget would point to the view and yourView.DoMenuCommand would be called. Since Quit belongs to the application object, the view's method would not handle the event.

Remember that view instances have an fNextHandler that points to the superview, in this case a window. Consequently, the next link in the chain would be yourWindow.DoMenuCommand, which also would not handle the event, and go to the method of the fNextHandler. Next in the chain would be yourDocument.DoMenuCommand, which would also fail, but which would call yourApplication.DoMenuCommand, which was preprogrammed to know about the Quit command (since all Macintosh applications are assumed to have Quit commands).

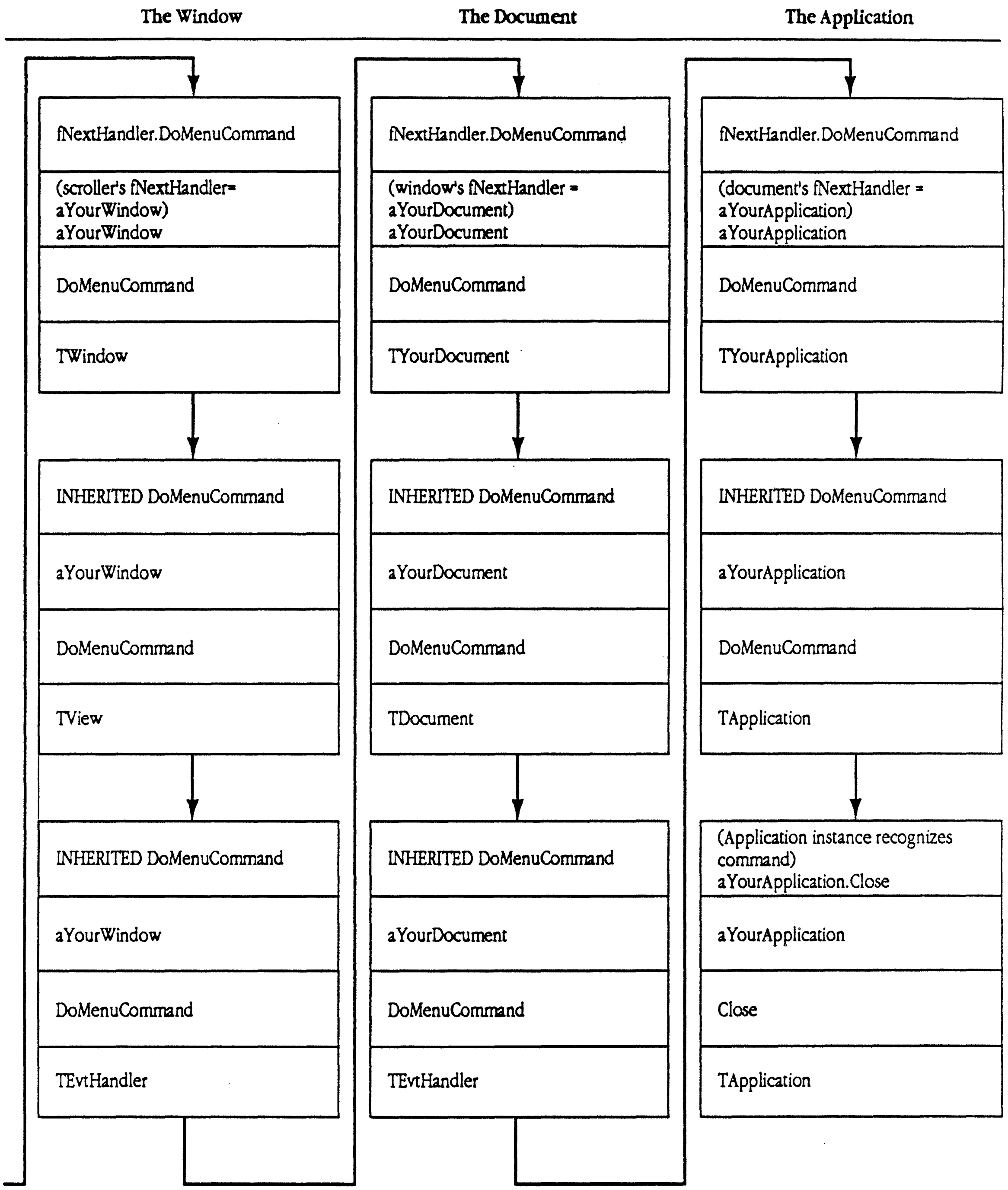


	The View	The Scroller
Command Line	gTarget.DoMenuCommand	
Self	(gTarget Points to view) aYourView	
Method	DoMenuCommand	
Method belongs to class	TYourView	

	The View	The Scroller
Command Line	INHERITED DoMenuCommand	fNextHandler.DoMenuCommand aScroller
Self	aYourView	(view's fNextHandler=aYourScroller) aYourScroller
Method	DoMenuCommand	DoMenuCommand
Method belongs to class	TView	TView (TScroller does not override)

	The View	The Scroller
Command Line	INHERITED DoMenuCommand	INHERITED DoMenuCommand
Self	aYourView	aYourScroller
Method	DoMenuCommand	DoMenuCommand
Method belongs to class	TEvtHandler	TEvtHandler

Figure 5-8
Example: The command chain when the user chooses Quit


The Window
The Document
The Application
fNextHandler.DoMenuCommand
(scroller's fNextHandler= aYourWindow)
aYourWindow
DoMenuCommand
TWindow
INHERITED DoMenuCommand
aYourWindow
DoMenuCommand
TView
INHERITED DoMenuCommand
aYourWindow
DoMenuCommand
TEvtHandler
fNextHandler.DoMenuCommand
(window's fNextHandler = aYourDocument)
aYourDocument
DoMenuCommand
TYourDocument
INHERITED DoMenuCommand
aYourDocument
DoMenuCommand
TDocument
INHERITED DoMenuCommand
aYourDocument
DoMenuCommand
TEvtHandler
fNextHandler.DoMenuCommand
(document's fNextHandler = aYourApplication)
aYourApplication
DoMenuCommand
TYourApplication
INHERITED DoMenuCommand
aYourApplication
DoMenuCommand
TApplication
(Application instance recognizes command)
aYourApplication.Close
aYourApplication
Close
TApplication

The click chain

The **click chain** is responsible for handling mouse clicks in the content region of the window. That means it handles all window clicks that aren't in the controls of the window, including palette clicks and clicks that select and drag objects. The click chain is much simpler than the command chain, partly because window clicks are always, by definition, handled by view objects, and the relationship of view objects to each other is much simpler than the relationship between all the possible command-handling objects. View objects form a simple tree, with the window as the root, and views and subviews as the branches and leaves.

How the chain is built

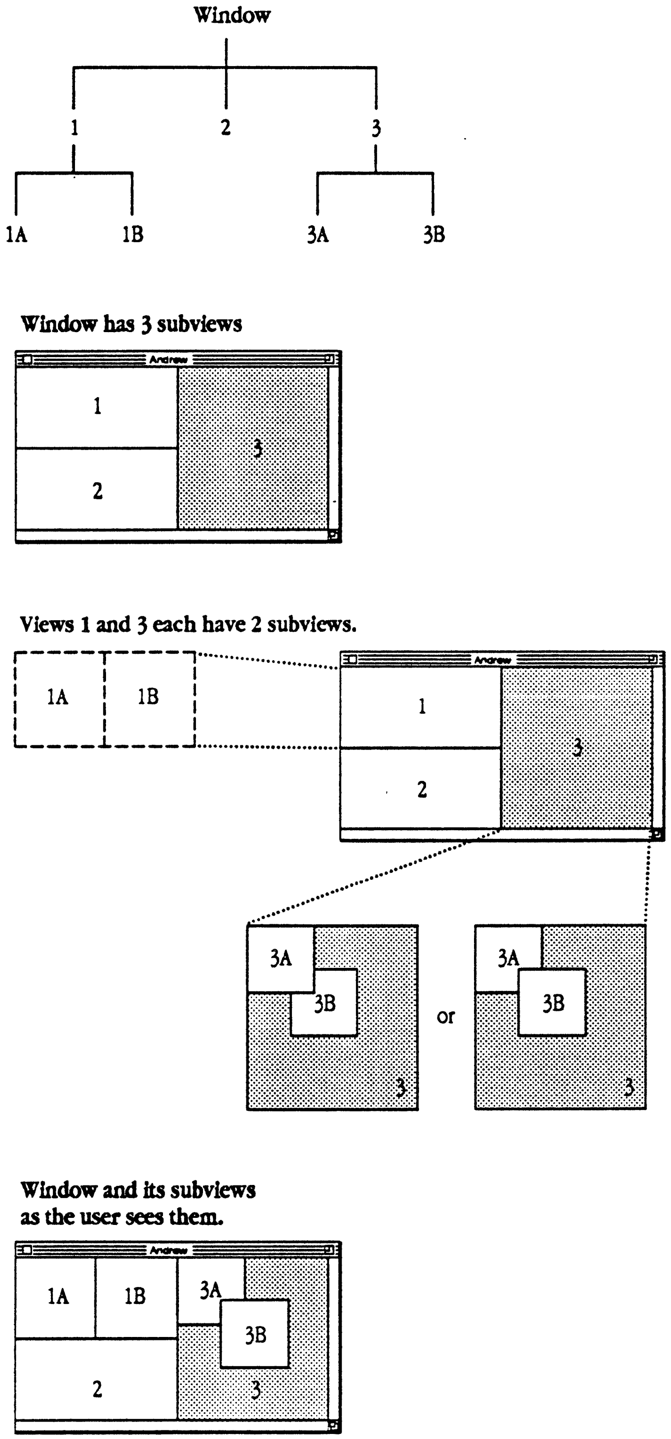
Every time you add a subview to a particular view object, MacApp places a reference to that subview in a list kept as a field of that object.

How the head of the chain is determined

Whenever a view is clicked, MacApp invokes a clever recursive routine, HandleMouseDown, that goes up each branch, looking until it finds the particular subview that contained the click. That view is treated as the head of the chain.

How the chain is traversed

MacApp expects each subview to handle the clicks in its own boundaries, so the event dies after the very first link in the chain. However, if you want to, you can write custom code that causes the event to work its way back up the chain by calling the appropriate method from this view's superview.



Window has 3 subviews

Views 1 and 3 each have 2 subviews.

Window and its subviews as the user sees them.

Figure 5-9
The view tree

The idle chain

While it is hard to imagine writing any useful application without using both the command chain and the click chain, you could easily write many marketable applications knowing nothing about the **idle chain**. The idle chain actually pulls double duty, providing a chain of responsibility for both idle-time events and alien events. This may seem like an odd combination, but it happens that many objects that handle alien events may also need to take advantage of idle time. In any case, if you will not be encountering any alien events and choose not to do anything fancy with the application's copious idle time, you can avoid the idle chain altogether.

On the other hand, the idle chain is very simple, and not much extra trouble to understand. Idle objects, including those that handle alien events, bear no underlying relation to each other, so the order of the chain is arbitrary; MacApp links the idle objects together like a breakfast sausage. When MacApp traverses the idle chain because of an Idle event, it calls the DoIdle method; when it traverses the chain because of an alien event, it searches for a DoHandleEvent method.

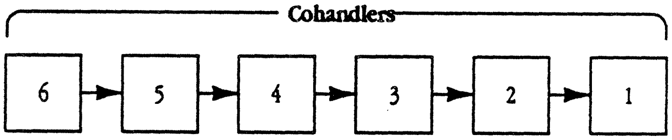


Figure 5-10
The idle sausage

How the chain is built

You call the TApplication.InstallCohandler method for each idle object instance you create. InstallCohandler simply links each piece of the chain to the previous piece.

Each idle instance has a field called fIdleFreq which you set to indicate how often its DoIdle method should be executed.

How the head of the chain is determined

MacApp sets the most recent link to be the head of the chain and the oldest link to be the end of the chain. A reference to the head of the chain is kept in a global variable called gHeadCohandler.

How the chain is traversed

The chain will be traversed both when it is time to idle and when an alien event is received. In both cases, the system of traversing the chain is similar to the method for traversing the command chain. In fact, unless you override MacApp's code, every time MacApp traverses the idle chain, it first traverses the command chain looking for DoIdle or DoHandleEvent methods. So the only time you need to install an object in the idle chain is when you want its DoIdle method to be executed even when it is *not* in the command chain. For instance, imagine you had a view with a flashing border around it. If you wanted the border to flash only when the view (or one of its subviews) was active, you would not have to place it in the idle chain. If you wanted it to flash no matter what, you *would* have to place it in the chain.

How often your application idles is determined by the fIdleFreq field of each instance in the idle and command chains. When there are no events pending, MacApp will check the fIdleFreq field of each instance in the idle chain. If enough time (measures in **ticks**) has elapsed since the object was last idled, then its DoIdle method is called. DoIdle is overridden for each type of object that requires processing at idle time. (For example, TTEView overrides DoIdle to make the cursor bar flash at the insertion point.) Unless you choose to program the idle chain otherwise, unlike the command chain, the whole idle chain is traversed, even after one object's DoIdle method executes successfully.

When MacApp encounters an alien event, it first traverses the command chain then calls gHeadCohandler.DoHandleEvent, which attempts to handle the event. Then, just like the command chain, it calls fNextHandler.HandleEvent until one of the methods handles the event or until the chain is over.

Summary

MacApp continually handles events as the user creates them, assigning instances to do the necessary work. Much of this work is accomplished behind the scenes, in such a way that you need not even know the event occurred. Other events will cause the MacApp code to execute methods that you should have overridden. By now, you should have some idea of what events will cause certain methods to be called, and to which instances those methods will belong.

As a practical matter, you can often tell when you should override a method by its name. The MacApp programmers christened the most important methods to override with names that start with "Do"—DoMouseCommand, DoKeyCommand, DoMenuCommand, and so on.

You should be aware that these event-handling methods are not the only methods you should be concerned about. There are also a great many methods that you will need to write to set up menus, draw the contents of views, create documents and windows, and handle the logic of your application. But this outline should give you a start.

Chapter 6

How to Install and Use MacApp

The MacApp package consists of:

- The source code library, including a variety of standard MacApp units and optional building blocks
- several MPW scripts and Make files to help you automate the compiling and linking of applications created with MacApp
- a number of sample programs

Before using MacApp, you must install it on your hard disk. Most people then build one or more of the sample programs into working applications. Finally, you can start working on your own application, either by starting from scratch or by modifying one of the samples.

Installing MacApp

This section describes how to install MacApp on your hard disk. It assumes that you have already installed MPW. See the *Macintosh Programmer's Workshop Reference* for details on installing MPW.

Follow these steps to install MacApp:

1. Create a folder named MacApp in your MPW folder. (If you prefer to place it elsewhere or name it something else, you should modify the UserStartup file, below, to accommodate this.)

2. Copy the contents of all the MacApp disks into your MacApp folder, and put the disks in a safe place.

3. For every folder with a name in the form `More XXX` copy its contents into the folder XXX. Then discard the empty folder `More XXX`.

   For example, you should take the contents of the folder `More MacApp Source Files` and put it into the folder `MacApp Source Files` Then discard the folder `More MacApp Source Files.`

4. Modify your UserStartup file in the MPW folder to contain the following lines:

   ```
   set MacApp "{MPW}MacApp:"
   export MacApp
   execute "{MacApp}MacAppStartup"
   ```

   If you've placed the MacApp folder in a different directory, be sure to modify the first line above.

If you are low on disk space, you can save space with these techniques:

- Delete any fonts, desk accessories, print drivers, utilities, or other elements you don't need.
- Don't copy all the MacApp sample programs onto your disk.
- Change the SeparateObjects flag to FALSE. See "Building a MacApp Program" later in this chapter.

When the installation is done, you should be able to build one of the sample programs or one of your own MacApp programs. Refer to the section "Building a MacApp Program," below, for details on how to build a program.

File naming conventions

MacApp filenames follow several conventions. Where *AppName* stands for the application name and *UnitName* stands for the name of a particular unit, the constituent files are named according to the following rules:

- U*UnitName*.p contains the Pascal interface for the unit U*UnitName*.
- U*UnitName*.*xxx*.p contains the implementation of the unit U*UnitName*. In most cases, the implementation of a unit is short enough to be put into a single file. In that case, 'xxx' is traditionally replaced with "inc1," which stands for "include file #1." However, when the implementation is too long to be managed easily in a single file, it is broken up across a number of files; each is given a mnemonic name. The most notable example of a unit broken into a large number of files is UMacApp, which is broken up into about a dozen files, each named after the most important thing defined in the unit, usually an object class.

- U*UnitName*.a contains supporting assembly source for the unit U*UnitName*, if any is needed.

- M*AppName*.p contains the main program for the application.

- *AppName*.r contains the Rez input for the application.

- *AppName*.make contains the make rules for the application.

If additional units are required, they are usually named in the form U*UnitName*, and the files containing the interface and implementation of the unit follow the conventions given above.

All the make files and USES statements conform to these conventions. The documentation refers to particular files in several ways:

- by filename; for example: UPrinting.p, UMacApp.TApplication.p, and DrawShapes.r

- by the unit name defined within the file; for example: UPrinting, UMacApp, UDrawShapes

- by what the file is; for example: the printing unit, the MacApp unit, and the DrawShapes program

Therefore, the printing unit is defined as UPrinting and can be found in the files UPrinting.p and UPrinting.incl.p. If you use this unit, your USES list would contain the line

```
UPrinting,
```

reflecting both the name of the file and the name of the unit defined in the file.

Building a MacApp program

The MacApp build process uses the MPW tool Make along with a MacApp command file MABuild, which together help automate the sometimes complicated process of compiling and linking MacApp programs.

* Note: You need to be familiar with the use of the MPW Shell to use these files. If there are any terms in this section that are not familiar, refer to the *Macintosh Programmer's Workshop Reference.*

In addition to automating the process of building debugging and nondebugging versions of any application that uses MacApp, MABuild allows you to use the method optimization feature of the MPW linker. It performs optimizations on method tables, making object-oriented code smaller and faster.

MABuild only compiles files when necessary. For example, if you specify nodebug and MacApp has already been compiled without debugging, it is not recompiled; if MacApp has been compiled with debugging on, it is automatically recompiled.

By default, MacApp automatically keeps two versions of MacApp's compiled code, each in its own directory: one with debugging code and the other without. If you are tight on disk space, you may want to keep only one version of the compiled code, which you can accomplish by modifying the MacAppStartup file to set the {SeparateObjects} flag to be FALSE. But while it *is* true, the MABuild program will maintain four directories:

- "{MacApp}MacApp Object Files:Non-Debug Files" for non-debug object files
- "{MacApp}MacApp LOAD Files:Non-Debug Files" for non-debug LOAD files
- "{MacApp}MacApp Object Files:Debug Files" for files with debugging code
- "{MacApp}MacApp LOAD Files:Debug Files" for LOAD files with debugging code

You must create a Make file for any application you want to build. Make files are provided for the sample applications. See "Creating a Make File," below, for instructions on writing one. If you want to build a sample program or you already have a make file, see "Using MABuild."

The file MacApp.make is used when you want to build just the MacApp library. The form of this command is

```
MABuild MacApp
```

This compiles and assembles all the MacApp units and places them in the file MacAppLib.o. You may specify debug or nodebug when you build MacApp; in this case the opt keyword would be synonymous with the nodebug keyword, since no application is being linked.

Creating a make file

Every program built with MABuild must have a make file with a name of the form *AppName*.make, where *AppName* is the name of the application. In this file you specify the application name, the signature, and what files your application depends on.

The file Application.make is included in the MacApp release and is a template for your program's make file. Most of the file is simply a set of assignments to Make variables. (Make is the MPW tool that is used to automate the build process.) The variables are as follows:

`AppName`	=	the name of the application
`Creator`	=	the application signature
`NeededSysLibs`	=	a list of required system libraries
`BuildingBlockIntf`	=	a list of the building block interface files the application uses
`BuildBlockObjs`	=	a list of the building block object files the application links with
`OtherInterfaces`	=	a list of other Pascal interfaces the application uses
`OtherLinkFiles`	=	a list of other object files the application links with
`OtherSegMappings`	=	a list of -sn commands for the linker
`OtherRezFiles`	=	a list of .r files that must be included in the Rez command, other than AppName.r
`OtherRsrcFiles`	=	a list of resource files that are included by one of your Rez files, other than the Debug.rsrc and MacApp.rsrc files

See the make files for the sample programs if you need additional guidance. In the simplest case, all you will have to do is perform a global search-and-replace on a valid make file, replacing the old application's name with the new one's name.

If you want to specify any additional files (such as assembly language files) that need to be linked with your application, you must provide dependency rules for them in your make file. For example,

```
OtherLinkFiles = MyApp.a.o
MyApp.a.o      ƒ MyApp.a FooInclude.a
```

You do not need to specify build rules for Pascal compiles or MPW assemblies because they are covered by the default rules specified in MacApp.make1. In this case you could leave out the filename MyApp.a on the second line, since the default rules say that MyApp.a.o must depend on a file MyApp.a if one exists. In fact, if the object file MyApp.a.o did not also depend upon the file FooInclude.a, you would not need the second line at all. (Note that the *ƒ* character is produced by typing option-f on the keyboard.)

The MABuild command file calls Make with your make file concatenated with the MacApp make files. The ordering is your file, MacApp.make1, AppName.make, MacApp.make2, and MacApp.make3. When running MABuild with optimization on, MacApp.make2 is replaced by MacApp.opt.make2.

Using MABuild

For program *AppName*, the command line for executing a build when the current directory contains the program's files is

```
MABuild AppName
```

Debugging is turned on by default (that is, debugging code is compiled in), although you can give the keyword debug after AppName, if you want. To build program AppName without debugging, you can use the command line

```
MABuild AppName nodebug
```

but generally you will also want to build an optimized version of program AppName, with the command line

```
MABuild AppName opt
```

The opt keyword invokes the method optimization feature of the MPW linker. This keyword also implies nodebug.

MacApp Rez files

To build a MacApp application, you must have these resources in your Rez (the MPW resource compiler) input file:

- MacApp Include files. See "Include Files," below.
- An Include line for your code. See the sample programs' resource files for examples.
- A cmnu resource for your menus. See "The cmnu Resource Type" below.
- An MBAR resource. See "MBAR Resources," below.
- An "About application" dialog. See the sample programs' resource files for examples.

There are three other resources not required for compilation, but they are strongly suggested nevertheless.

- A SIZE resource to tell MultiFinder how much space your application will need.
- The seg! and mem! resources to provide information for efficient memory management.

For complete examples of any of these elements, look at the Rez files of the sample programs.

Include files

Most of the standard MacApp resources are supplied as Rez input files which are compiled by MABuild. You include the compiled versions with lines like

```
include MacAppRFiles"FileName";
```

where FileName is the name of a MacApp file. If you don't need all the standard resources, you copy the files and remove the resources you don't want.

Some resources required by MacApp must be in your Rez input file. These include any view resources you may have created as well as any cmnu or MENU resources your application will require. (You would only use MENU resources in rare cases, if you want to bypass MacApp's menu handling.) Look at the sample programs for guidance.

The cmnu resource type

The cmnu resource type is provided so that menu items can be identified by a unique command number, independent of the item's placement within the menus.

You define resources as type cmnu instead of type MENU. These have the same syntax as resources of type MENU, except that menu items have one additional component, a command number (or the symbol nocommand). After running Rez, the make file runs the tool PostRez, which will generate the usual MENU resources, plus a single resource of type mntb and ID 128 which maps all command numbers for all cmnu resources encountered in the file into their corresponding menu and item numbers.

* **Important**: Because the PostRez tool changes cmnu resources to MENU resources and other information MacApp needs to handle menus properly, you cannot use DeRez to decompile cmnu resources for MacApp programs. In addition, you should not use the resource editor to change MENU resources, because that will not provide all the information MacApp needs. If you need to change your menus, you must edit the original resource file and allow MABuild to run Rez and PostRez again.

MBAR resources

MacApp requires one or two MBAR resources that tell it what to do with your menus.

You must have an MBAR resource with ID 128. This resource gives the ID's of all menus that should be read in and displayed initially. MacApp looks for this resource when it starts up and loads all menus whose ID's are listed here.

You can also have an MBAR resource with ID 129. This resource gives the ID's of menus that can be added to or removed from the menu bar by your application. The menus listed here are read in by MacApp but are not initially displayed.

You can also have menus that are not listed in either MBAR resource. Examples of these are

- Buzzword menus, which contain strings meant to be placed in other menus under certain conditions.

- The Debug menu, which has ID 900 and is treated as a special case by MacApp so that it is only read in if debugging code is included in the application.
- Nonstandard menus such as graphical menus. These must have ID's greater than mLastMenu (which is 63). You need to handle these menus by calling the Menu Manager directly.

Menus handled by MacApp must have ID's from 1 to mLastMenu, regardless of which MBAR resource refers to them.

Important

You should not use programmatically created menus; that is, your resource file should contain at least the menu title of every menu.

The structure of a MacApp program

MacApp programs are written with a certain structure. Some elements are mandated by the syntax and are so identified here, and others are simply conventions that make it easier to maintain the programs.

The elements of a MacApp program are the main program, the unit's public interface, and the unit's implementation. Each element is generally kept in a separate file, although you can combine them if you wish. See the sample programs for further detail on these files. (In particular, MacApp includes debugging features that require the use of some compiler switches.)

The main program, usually named M*AppName*.p where *AppName* is the name of the application, is generally very simple and small. A MacApp main program follows this structure:

```
PROGRAM YourName;
      USES
            {Units used by this program}
      VAR
            gYourApplication:   TYourApplication; {The application object}
BEGIN
      InitToolBox({Number of master pointer blocks to allocate});
      InitPrinting; {Only if you are using UPrinting}
      InitUTEView; {Only if you are using UTEView}
      InitUDialog; {Only if you are using UDialog}
      InitUGridView; {Only if you are using UGridView}
      SetResidentSegment(GetSegNumber(@AResDummy), TRUE);
      New(gYourApplication); {Allocate the application object}
      gYourApplication.IYourApplication; {Initialize the application}
      gYourApplication.Run; {Run the application; hand control to MacApp}
END.
```

* **Note:** If you use the UDialog unit, you must also include the UTEView unit. Make sure you place UDialog *after* UTEView in the USES statement and that you place InitUDialog *after* InitUTEView.

The rest of the application is generally run from the application's Run method. The basic type TApplication is defined for you by MacApp, and you customize it in one of the units to create your application type, called TYourApplication here. That unit is listed in the USES declaration along with any other units needed to compile its interface.

The unit's public interface file, usually called UAppName.p, defines the object types used by the program. The methods of object classes must be declared in the style of a FORWARD declaration (although you never use the reserved word FORWARD). Where you want to reimplement an inherited method, you write the full interface for the method and follow it with the OVERRIDE reserved word. A unit interface file usually follows this structure:

```
UNIT UnitName;
INTERFACE
       USES
              {Any units used by the unit}
       CONST
              {Any constants for the application}
       TYPE
              {Ordinary type declarations}
              {Object-type declarations}
IMPLEMENTATION
{$I UName.inc.p} {Name(s) of implementation file(s))
END.
```

The unit implementation file, usually named UAppName.inc1.p, can begin with a VAR part that declares any global variables for the unit followed by the implementations of the methods. You have the option of repeating the full interface declaration with the implementation of each method. There is nothing special about the design of this file; it consists simply of the global variable declarations, private constructs and type declarations, and the method implementations, which look like procedure and function declarations.

If an application is large, it can be broken up into a number of units.

Chapter 7
The Cookbook

This chapter features instructions for producing a MacApp program.

It is divided into ten sections covering documents, views and windows, handling mouse events, standard editing commands, menus and commands, using UPrinting, using UTEView, using UDialog, supporting the Clipboard, and failure handling. Each section begins with recipes every program needs and goes on to show how to implement more specialized and sophisticated behavior.

Any typical Macintosh program needs to implement the parts discussed in these recipes:

- "Creating a Document"
- "Initializing a Document"
- "Creating a View"
- "Initializing a View"
- "Drawing a View"
- "Creating a Window"

In addition, for most applications you need to implement the parts discussed in these recipes:

- "Saving and Restoring Data"
- "The Clipboard"
- All recipes under "Standard Editing Commands"
- Some recipes under "Menus and Commands"
- Some or all recipes under "Handling Mouse Events"

When you implement an application with these recipes, the application displays a standard window that may or may not have horizontal and vertical scroll bars, a size box, a zoom box, a close box, and the standard menus. The application can cut, copy, paste, undo, and save, and can open old and new documents. The application can handle any number of documents at a time, and its windows are refreshed correctly when necessary. The application can track the mouse when the mouse button is down.

Each recipe includes the following elements:

- **Purpose:** an explanation of why you use this feature.
- **How to do it:** step-by-step instructions for implementing this feature. These instructions often include object type or interface declaration samples and references to other recipes.
- **Template:** listings of sample implementations or implementation frameworks for the major methods needed to implement the feature described in the recipe.

The word *item* is used in this chapter to indicate any basic data element of your program. For example, EachItemDo is a method that you usually name according to the object type most basic to your data set. If your data set consists of different types of fruit, that method might be called EachFruitDo. Similarly, when an object is to be of some object type unique to your program, the type is called TItem in this chapter.

As in the rest of this manual, the word *Your* is included in variable and type names that your application declares. Replace the name with some appropriate word.

You can use the Nothing program, included in the Nothing Sample folder, as a base for your application. Nothing is an application that has the standard Macintosh interface and nothing else. It can display windows with scroll bars, a size box, a close box, and a title bar; has the standard menus; refreshes the window correctly; can show the Clipboard; and allows the use of desk accessories. It can even save and restore documents, but the documents and the windows have nothing significant in them. (The window actually shows a fixed text string.) The Nothing program's source code consists of the following files:

- MNothing.p, the main program
- UNothing.p, the interface file
- UNothing.incl.p, the implementation file
- Nothing.r, the resource compiler input file
- Nothing.make, the make file for building the program

The last two files are used only in the Macintosh Programmer's Workshop. If you are using a different development system you may have to create your own files to perform the equivalent functions.

If you begin with the Nothing program and modify it to create your application, you will make most changes either to the implementation file or to the interface file. In addition, some changes are made to the resource compiler input file. The main program is rarely changed. Many of the methods described in the basic recipes in this chapter are included in the Nothing program, because they must exist so that the Nothing application can run; you must modify those methods to do tasks specific to your application.

If you have a large program, you may want to break the implementation unit. into two or more files or add additional units. You may want to break your program into segments. See the "Segment Loader" chapter of *Inside Macintosh* for information on segmenting your program.

This chapter includes many references to *Inside Macintosh*. The chapter's title or its relevant section is given in quotation marks in these references.

Documents

Creating a document

Purpose

A document controls the data set of your application independently of how it is displayed or printed.

Unless you override TApplication.HandleFinderRequest, MacApp asks your application to create a new document when the user double-clicks on the application's icon.

How to do it

1. Declare the file type for your document, typically in the interface part of your object unit. The file type is generally stored as a constant yourFileType, and is always a four-character string, for example you might define kYourFileType to be 'TEXT'.

   If you use an existing file format, use the predefined file type. A file made up of strings of characters, where each line or paragraph is terminated by a return, is of type TEXT. A file consisting of QuickDraw pictures is of type PICT.

If you have your own file format, the file type is an arbitrary four-character string. File types should be registered with Developer Technical Support to ensure that they are and remain unique.

2. Implement TYourApplication.DoMakeDocument. MacApp calls this method when a new document needs to be created. You need the following declaration as part of the definition of TYourApplication:

```
FUNCTION TYourApplication.DoMakeDocument(itsCmdNumber: INTEGER): TDocument;
                                                                 OVERRIDE;
```

   A sample implementation is given in the template for this recipe.

   The command number is used primarily for applications with more than one document type. If your application provides more than one document type that the user can choose from a menu, use the command number to determine which type of document to create.

3. For each document type, implement IYourDocument as described in the "Initializing a Document" recipe. TYourApplication.DoMakeDocument should call IYourDocument after instantiating aYourDocument.

4. For each document type, if you have menu commands other than the standard File menu commands (New, Open, Save, Save As, Save Copy, or Revert) that apply to the document or its contents (regardless of which window is active or which view is selected), override TDocument.DoMenuCommand and TDocument.DoSetUpMenus. See the "Creating Menu Commands" recipe for details on DoMenuCommand.

Template

```
FUNCTION TYourApplication.DoMakeDocument(itsCmdNumber: INTEGER):TDocument;
      VAR
            aYourDocument: TYourDocument;
BEGIN
      New(aYourDocument);
      FailNIL(aYourDocument);
      aYourDocument.IYourDocument(yourFileType);
      DoMakeDocument := aYourDocument;
END;
```

Initializing a document

Purpose

After you create a new document you must initialize it, which you generally do by calling IYourDocument. (The call is made from your DoMakeDocument method, described in the "Creating a Document" recipe.)

How to do it

Implement TYourDocument. An example is shown in the template for this recipe. That template assumes you have a field fItemList, which is a TList-type object that stores your data. If your data is organized in another way, you should change the implementation of IYourDocument as appropriate.

Note that all fields that may be referenced by your document's (or any object's) Free method should be initialized to an appropriate value before the first point at which your initialization method can fail.

See the sample programs for more examples of this method. In particular, you may want to look at Puzzle and DrawShapes.

Template

```
PROCEDURE TYourDocument.IYourDocument(itsFileType);
BEGIN
       fItemList := NIL; { In case IDocument fails. }
       IDocument(itsFileType, itsCreator, kUsesDataFork, NOT kUsesRsrcFork,
                 NOT kDataOpen, NOT kRsrcOpen);
       { You may give different constant values in the above line.
         The values given are the most common commands and indicate a document
         that uses only the data fork of the file and is not disk-based
         (that is, the entire file is copied into memory).
         See the Cards sample program for how to create a disk-based document.     }

       fItemList := NewList;
       fSavePrintInfo := TRUE; { Used when saving the document.
                                 Set to FALSE if you don't want to save this. }

       { If you have fields for the views, which you generally do, }
       { set them to NIL here. }
END;
```

Saving and restoring data

Purpose

You save and restore data so the user can save documents, open document icons, and open documents using the Open command and the directory dialog box. In addition, you save the print state and the display state so the user does not have to reestablish them each time the document is opened.

How to do it

Objects contain data and also have pointers to methods. You save only the data, not the method pointers, in your document file. The way you do this depends on how your application's data is organized.

This recipe assumes your data consists of a list of objects of a single type. In such a case, you generally create records that are equivalent to the data parts of the objects you want to save, and you save those records in the file. In the templates, the record type is called TFiledItem. When you want to restore that document file (that is, when the user opens that document), you create a new set of objects, reading data from the file and transferring it from the filed records to the objects.

Important

Although, for simplicity, this recipe assumes that all the application's objects are of one class, that is relatively unlikely. If your file contains several classes of objects, you need to create a record type for each object class you want to save. In the file, each record should be preceded by a "identifier" value that indicates what type of record follows. You first read the identifier value and then read a record of the type indicated by the identifier value. You may read the record into a variable of the record type and then copy fields into the object or, if the field structure of the record and the object are identical, you can read directly into the object from the file.

For this recipe, assume the following interface declarations:

```
TItem = OBJECT(TObject);
       fIdentifier: INTEGER;
       fData1: DataType1;
       fData2: DataType2;
       { Any other fields... }
```

```
        FUNCTION TItem.WriteTo(aRefNum: INTEGER): OsErr;
        FUNCTION TItem.ReadFrom(aRefNum: INTEGER): OsErr;
        { Other methods would normally be included. }
    END;
```

1. Override TDocument.DoNeedDiskSpace. That method is called just before a document is saved. It should return the total amount of disk space, in bytes, needed to store the data and resources for the document. Don't worry about file overhead; MacApp adds that on.

   MacApp uses the value returned by DoNeedDiskSpace to check whether there is room on the disk to save the new file without first deleting the old. What happens if there isn't enough room depends on the value of the document's fSaveInPlace field. The possible values are

   - sipNever, to indicate that the original file should never be overwritten
   - sipAlways, to indicate that the original file should always be overwritten when there is not enough space for a copy
   - sipAskUser, to indicate that the user should be asked whether the original file should be overwritten when there is not enough space for a copy.

   IDocument sets the value of fSaveInPlace in this way:

   ```
   IF keepSDataOpen OR keepSRsrcOpen THEN
       fSaveInPlace := sipNever
   ELSE
       fSaveInPlace := sipAskUser;
   ```

   You can change the value of fSaveInPlace in your IYourDocument method.

   Take care that your DoNeedDiskSpace method returns the correct value or overestimates slightly; if DoNeedDiskSpace returns too large a value, the old file may be deleted unnecessarily, or DoNeedDiskSpace may erroneously inform the user that the file cannot be saved; if DoNeedDiskSpace returns too small a value, you may get an I/O error when the application tries to save the document, which could be particularly serious if MacApp has deleted the old file.

The interface for this method is

```
PROCEDURE TYourDocument.DoNeedDiskSpace(VAR dataForkBytes,
                                             rsrcForkBytes: LONGINT);
                                        OVERRIDE;
```

A sample implementation is given in the templates for this section. It begins with a call to INHERITED DoNeedDiskSpace so that MacApp can calculate the space needed to save the print state, the overhead for the file, and if you are using the resource fork, the overhead for the resource fork. (Set document.fSavePrintInfo to TRUE in IYourDocument if you want TDocument.DoNeedDiskSpace to incorporate the print state in its calculations.) Notice that this method adds to the initial values of dataForkBytes and rsrcForkBytes. MacApp sets the initial values of these parameters, and you should not reset them to 0. The rest of the implementation is very general, because the amount of space needed depends entirely on your application. See the sample programs for more specific examples.

2. Override TDocument.DoWrite, called to write the document to a file. The interface for this method is

```
PROCEDURE TYourDocument.DoWrite(aRefNum: INTEGER; makingCopy: BOOLEAN);
                                OVERRIDE;
```

A sample implementation is given in the templates for this section.

The sample begins by saving the print state. MacApp does that for you when you call INHERITED DoWrite. (You need to set document.fSavePrintInfo to TRUE in IYourDocument or TDocument.DoWrite won't save the print state.) Finally, the data is saved.

The makingCopy parameter is primarily for disk-based documents. It indicates that DoWrite is being called to make a new copy of the file.

3. Override TDocument.DoRead, called when an existing document is opened. The interface for this method is

```
PROCEDURE TYourDocument.DoRead(aRefNum: INTEGER;
                               rsrcExists, forPrinting: BOOLEAN);
                              OVERRIDE;
```

The implementation is given in the templates for this section. The sample begins by calling INHERITED DoRead to restore the print state if fSavePrintInfo is TRUE. Notice that the display state is not restored at this time. That is done when the window is created, with DoMakeWindows, or DoMakeViews.

4. Assuming you have several different types of items, each a descendant of TItem and differentiated by a value that indicates the kind of item, add the following method to your document

```
FUNCTION TYourDocument.MakeItem(kind: ItemKind): TItem;
```

The implementation is given in the templates.

You also need to define a set of constants for the different kinds of items.

5. Give each object type in your document's data set a WriteTo method for the TYourDocument.WriteTo method and a ReadFrom method for the TYourDocument.ReadFrom method. The interfaces for TItem.WriteTo and TItem.ReadFrom could be as follows:

```
FUNCTION TItem.WriteTo(aRefNum: INTEGER): OsErr;
FUNCTION TItem.ReadFrom(aRefNum: INTEGER): OsErr;
```

The implementations are given in the templates for this section.

If you have several different classes of items, each a descendant of TItem, assign an identifier value to each of them, and create a method for each item class that returns the identifier value for the item. Before you write each item's data, write its identifier value into the file.

If your different item classes have different sizes, you may also want to add a method to your item classes that calculates the size of the item. You'll need to call this method when you are calculating the amount of disk space you need to save a document to disk.

Templates

```
PROCEDURE TYourDocument.DoNeedDiskSpace(VAR dataForkBytes, rsrcForkBytes: LONGINT);
       PROCEDURE AddSize(item: TItem);
       BEGIN
              dataForkBytes := dataForkBytes + { The number of bytes needed to store the
                                                 data in this item. Don’t forget the kind
                                                 value, if there is one. };
       END;
BEGIN
       { get space needed to save print state }
       INHERITED DoNeedDiskSpace(dataForkBytes, rsrcForkBytes);

       fItemList.Each(AddSize); { This assumes your data is stored in a TList-type object
                                  in the field TYourDocument.fItemList }

       { Add a value to rsrcForkBytes when you want to store some resources
         for the document. }
END;


PROCEDURE TYourDocument.DoWrite(aRefNum: INTEGER; makingCopy: BOOLEAN);
       PROCEDURE WriteItem(item: TYourObject);
       BEGIN
              err := item.WriteTo(aRefNum);
       END;
BEGIN
       INHERITED DoWrite(aRefNum, makingCopy); { Save print info record. }
       fItemList.Each(WriteItem);
END;


PROCEDURE TYourDocument.DoRead(aRefNum: INTEGER; rsrcExists, forPrinting: BOOLEAN);
{ This method assumes you have a number of data object types, each a descendant of TItem.
  The types are differentiated by a kind value. }
VAR
       newItem: TItem;
       kind, i, nItems: INTEGER;
       size: LONGINT;
BEGIN
       FailOsErr(GetEOF(aRefNum, eof)); { See the “Failure Handling” recipe. }

       INHERITED DoRead(aRefNum, fRsrcExists, forPrinting); { Read print info record. }

       FailOSErr(GetFilePos(aRefNum, fPos));
       nItems := (eof - fPos) DIV Sizeof(TFiledItem);
       FOR i := 1 TO nItems DO BEGIN
              size := 2;
              FailOSErr(FSRead(aRefNum, size, @kind));
              newItem := MakeItem(kind);
              FailNIL(newItem);
              FailOSErr(newItem.ReadFrom(aRefNum));
              fItemList.InsertLast(newItem);
       END;
END;
```

```
FUNCTION TYourDocument.MakeItem(kind: TItemKind): TItem;

VAR     firstTypeItem: TFirstItem;
        secondTypeItem: TSecondItem;

BEGIN
        CASE kind OF
                kFirstType:  BEGIN
                        New(firstTypeItem);
                        MakeItem := firstTypeItem;
                END;
                kSecondType:  BEGIN
                        New(secondTypeItem);
                        MakeItem := secondTypeItem;
                END;
        END;
END;


FUNCTION TItem.WriteTo(aRefNum: INTEGER): OSErr;
BEGIN
        { Here write the item's kind and then the item's data to the file.
          Return 0 or any nonzero error code in WriteTo.
          See the "Failure Handling" recipe. }
END;


FUNCTION TItem.ReadFrom(aRefNum: INTEGER): OSErr;
{ This method assumes your object has an IItem method that initializes the object.
  Item loads the data fields with the data given in the parameter list.
  Note that fNextItem and fPreviousItem should be initialized to NIL so that
  TYourDocument.AddItem can work correctly. }

VAR     savedItem: TFiledItem;
        lengthOfFiledItem: LONGINT;

BEGIN
        lengthOfFiledItem := Sizeof(TFiledItem);
        IItem({parameters});
        ReadFrom := FSRead(aRefNum, {length of this kind's data}, @fData1);
END;
```

Saving the display state

Purpose

When opening an old document, the user usually likes to find the window and view the way they were left when the document was saved; that is, with the window in the same position and at the same size and displaying the view in the same scroll position. This recipe shows how to implement that capability for a single window with one view.

How to do it

Each step in this recipe has some code in the template section. This recipe assumes that you only have one window per document. See the Puzzle sample program for an example of saving the states of more than one window per document.

1. You need a data type to save the state information. In the template, it is a record called DisplayState. Define this as a global data type, so you can refer to it in different methods.

2. You need a place to save display-state data read from a document file and a Boolean variable that indicates whether or not the display state has been read from a document file. In the template, both are fields of the document. The Boolean field, fUseDisplayState is set to FALSE for a new document. It is set to TRUE when a document is read from a file. When the document is read from a file, the saved display state is read and stored in fDisplayState. Otherwise, that field has no meaning. (Both fields are only used immediately after a document object is created.)

3. In TYourDocument.IYourDocument, set fUseDisplayState to FALSE. (This value is reset to TRUE in DoRead.) You also may need to initialize fDisplayState to the default arrangement. (That is not done in the template, because the items stored in the display state already have default values.)

4. In DoWrite, load the display state into a display-state record, and then write the record to the document file.

5. In TYourDocument.DoRead, read the display state (if there is one) into a display state record, transfer the data to fDisplayState, and set fUseDisplayState to TRUE if there was a display state.

6. In TYourDocument.DoMakeWindows, if fUseDisplayState is set, use fDisplayState to position the scroll bars and size and position the window.

7. In TYourDocument.DoNeedDiskSpace, add in the amount of space needed to save the display state.

Templates

```
{Add as a global type definition:}
DisplayState = RECORD
       theWindowRect: Rect;
       theScrollPosition: VPoint;
       { If you want to save the print record for each view add a field here. }
       { (Normally, you save one for the entire document.) }
END;


{Add as fields of your document:}
fDisplayState: DisplayState;
fUseDisplayState: BOOLEAN;


{Add to TYourDocument.IYourDocument:}
fUseDisplayState := FALSE; { Always set to FALSE here. If you are now restoring a saved
                             document, set this to TRUE in DoRead }


{Add to TYourDocument.DoWrite:}
VAR
       aDisplayState: DisplayState;
       theWindow: TYourWindow;
       theScroller: TScroller;

{To save the state of the first window created for a document, add this to the block,
between saving the print state (the call to INHERITED DoWrite) and saving the data: }
theWindow := TWindow(fWindowList.First);
theScroller := fMainView.GetScroller(FALSE);
WITH aDisplayState DO BEGIN
       theWindow.GetGlobalBounds(theWindowRect);
       IF theScroller <> NIL THEN
              theScrollPosition := theScroller.fTranslation;
END;
count := Sizeof(DisplayState);
FailOSErr(FSWrite(aRefNum, count, @aDisplayState));


{ Add to TYourDocument.DoRead: }
VAR    count: LONGINT;
       aDisplayState: DisplayState;
       aScroller: TScroller;

{ In the block after calling INHERITED DoRead: }
count := Sizeof(DisplayState);
FailOSErr(FSRead(aRefNum, count, @aDisplayState));
fDisplayState := aDisplayState;
fUseDisplayState := TRUE;
```

```
{ Add to TYourDocument.DoMakeWindows: }
VAR     vhs: VHSelect;
        aDisplayState: DisplayState;

{ In the block, after you've created the window: }
IF fUseDisplayState THEN BEGIN
        aDisplayState := fDisplayState;
        WITH aDisplayState.theWindowRect DO BEGIN
                aWindow.Resize(right-left, bottom-top, FALSE)
                aWindow.Locate(left, top, FALSE);
        END;
        aWindow.ForceOnScreen;

        aScroller := fMainView.GetScroller(FALSE);
        IF aScroller <> NIL THEN
                aScroller.ScrollTo(aDisplayState.theScrollPosition, FALSE);
END;


{ Add to TYourDocument.DoNeedDiskSpace: }
dataForkBytes := dataForkBytes + Sizeof(DisplayState);
```

Windows and views

MacApp allows you to create and initialize views (and therefore windows) in two different ways: with procedures and with templates. Because creating a view template is quite a bit of work in itself, it is occasionally more convenient to create your views procedurally. However, you gain many benefits by creating view templates, and that is the recommended method for creating views. The first six entries in this section examine procedural view creation. These sections are "Creating a View", "Initializing a View", "Creating a Window", "Creating a Palette Window", "Creating a Window with Two or More Main Views", and "Creating a Document with Two or More Windows."

The final two entries of this section, "Creating View Templates" and "Creating and Initializing Views with Templates", explain the template method for creating views and windows.

Creating a view

Purpose

Views are usually used to display data associated with documents, although it is possible to have a view that has no associated document. However, everything displayed by a document must displayed in a view. MacApp translates between the view and the screen or a printer. All you have to do is create the view and provide it with certain methods. Applications can offer one or more views of each document.

How to do it

1. Define a view type that is a descendant of TView. Your view type must have the following methods:

```
PROCEDURE TYourView.Draw(area: Rect); OVERRIDE;
{ Called by MacApp to draw the view. See the "Drawing a View" recipe. }

PROCEDURE TYourView.IYourView;
{ Usually called from DoMakeViews or DoMakeWindows after creating a view.
  See the "Initializing a View" recipe. }
```

If you have more than one view type, create equivalent methods for each type.

If a mouse click, press, or drag in the view can do something, you must also implement the following method:

```
FUNCTION TYourView.DoMouseCommand(VAR theMouse: Point;
                              VAR Info: EventInfo;
                              VAR hysteresis: Point): TCommand;
                                          OVERRIDE;
{ See "Handling Mouse Events." }
```

If parts of the view can be selected by the user, you usually also override TView.DoHighlightSelection. See the "Selecting" recipe.

If there are menu commands that apply to the view (such as the "Reduce to Fit" command in MacDraw®), override TView.DoMenuCommand and TView.DoSetUpMenus. See "Menus and Commands" in this chapter.

If the view can be a Clipboard view, you need additional methods. See "The Clipboard" in this chapter.

2. Declare a field for each view in your subclass of TDocument. For example:

   ```
   fYourView: TYourView;
   ```

   The fields referencing your views will be used by your methods, not by MacApp, so you are free to organize them as you wish. If it makes sense in your document, you may want to use a list instead of individual fields. The object type TList provides a convenient list type (actually, it implements a dynamic array).

3. Override TDocument.DoMakeViews to create your views. The interface is

   ```
   PROCEDURE TYourDocument.DoMakeViews(forPrinting: BOOLEAN); OVERRIDE;
   ```

   The parameter forPrinting is TRUE when the user is printing the document from the Finder™. In that case, you may not need to create all your document's views.

   The template in this recipe shows a sample implementation of DoMakeViews.

Template

```
FUNCTION TYourDocument.DoMakeViews(forPrinting: BOOLEAN);

VAR     yourView: TYourView;

BEGIN
        { The forPrinting parameter is TRUE only when printing from the Finder.
          You can use this value to optimize performance by creating only views
          that need to be printed. }

        New(yourView);
        FailNIL(yourView); { See the "Failure Handling" recipe. }
        { Send SELF to IYourView so that the view can reference the document. }
        yourView.IYourView(FALSE {means not for Clipboard}, SELF);
        fYourView := yourView;

        { If you have more views, create, initialize, and install them
          into fields of your document here. }
END;
```

Initializing a view

Purpose

After you create a view, you call IYourView to initialize it. The initialization routine sets the initial size for the view and, if the view is printable, creates a print handler for the view.

How to do it

Implement TYourView.IYourView, as shown in the template for this section. Call IYourView from TYourDocument.DoMakeViews after you create your view.

Template

```
PROCEDURE TYourView.IYourView(forClipboard: BOOLEAN; itsYourDocument: TYourDocument);
{ In this case, you don't need the forClipboard parameter. It is used so that a print
       handler object is not created for a Clipboard view. }

VAR    viewSize: VPoint;

BEGIN
       SetVPt(viewSize, 1000, 1000);
       { The size of the view. Set to values appropriate for your view.
         These values can be changed later. }

       fYourDocument := itsYourDocument;
       { Most views have documents, but some may not. }

       IView(itsYourDocument, NIL, gZeroVPt, viewSize, sizeFixed, sizeFixed);
       { The enumerated constant value sizeFixed is from the predefined SizeDeterminer
         enumerated type.  The significance of these parameters can be found in the
         "Creation/Destruction Methods" under "The TView Class" section of the
         "Display Architecture ERS." }


       { If the view can be printed, more is included.
         See "Using UPrinting" in this chapter for more information. }
END;
```

Creating a window

Purpose

This and the next few entries of this section explain how to create windows with procedures instead of templates. To create windows from view resource templates, see the "Creating View Templates" and "Creating and Initializing Views with Templates" entries in the previous section.

The TWindow class, a descendant of TView, represents a Window Manager window. It responds to mouse clicks outside the window's content region, draws the window's size box, and overrides other view methods where appropriate. Since TWindow objects represent windows, they never have superviews, but they must have subviews or nothing will be drawn in the window's content region.

Windows allow a portion of their subviews to be seen—the portion that lies within the content region of the window. If any of these subviews is scrollable, that subview must have a scroller object as its superview. Scrollers (not windows) are scrollable, but windows can be resized, opened, closed, and moved around the screen.

This recipe deals with creating a simple resizable window that contains a single view, which may or may not be scrollable. If you want a window with more views, read this recipe and then proceed to the "Creating a Palette Window" and "Creating a Window With Two or More Main Views" recipes.

How to do it

1. In your resource file, you define a resource for your window. The way you define it depends entirely on the resource compiler you use. Here is an example of one for the MPW Resource Compiler, Rez:

```
resource 'WIND' (1005) {
    {50, 40, 250, 450},
    zoomDocProc,
    invisible,
    goAway,
    0x0,
    "<<<Untitled>>>"
};
```

   The first line has the required resource type WIND and an arbitrary resource number (1005 in this case).

   The second line defines the default initial size of the window, in screen coordinates. (Note that you often modify these values before displaying the window.)

   The third line indicates that this window should have a zoom icon. (If you don't want a zoom icon, use documentProc here. The constant documentProc is defined in the standard MPW Rez types file.)

   The fourth line tells the Window Manager that this window should be initially invisible. You always tell the Window Manager not to display MacApp windows, even if you want them to be initially visible, because they are displayed (if appropriate) by TApplication.ShowWindows.

   The fifth line indicates that the window is to have a close box. The alternative is noGoAway.

   The sixth line is the window refCon. It doesn't matter what you put here, because MacApp always replaces it.

   Finally, the last line defines the initial window title. Note that the triple brackets shown here are not displayed. When an existing document is opened, the text enclosed in brackets is replaced by the document name.

When a new document is opened, the text is replaced by the word *Untitled*, followed by a number. If you want the window to have a fixed title, don't use the brackets. You can also give text outside the brackets, and that text is concatenated to the document name.

See the sample programs' resource files for more examples. See the "Window Manager" chapter of *Inside Macintosh* for complete information.

2. In your unit interface file, define a constant for the resource number of your window resource. For example:

```
kIDYourWindow = 1005;
```

3. Implement DoMakeViews, as described in the "Creating a View" recipe. MacApp calls DoMakeViews immediately before DoMakeWindows, and DoMakeWindows needs to have the view object available. This recipe assumes that the view is stored in yourDocument.fView.

4. Override TDocument.DoMakeWindows for your document type. The interface of this method is

```
PROCEDURE TYourDocument.DoMakeWindows; OVERRIDE;
```

The implementation is discussed in the rest of this recipe.

5. The window object, along with the required Window Manager structure, is created by the MacApp global function NewSimpleWindow. The interface of that method is

```
FUNCTION NewSimpleWindow(itsRsrcID: INTEGER;
                         wantHScrollBar, wantVScrollBar: BOOLEAN;
                         itsDocument: TDocument; itsView: TView): TWindow;
```

The itsRsrcID parameter gives the ID of the window resource.

The next two parameters, wantHScrollBar and wantVScrollBar indicate whether or not you want scroll bars for the frame in this window. Use the kWantHScrollBar and kWantVScrollBar predefined constants here, preceded by NOT if you don't want the scroll bars.

The itsDocument parameter is the document whose data is displayed in this window.

The itsView parameter is the view shown in the window.

The template shows NewSimpleWindow called with parameter values that result in a scrollable window.

Template

```
PROCEDURE TYourDocument.DoMakeWindows; OVERRIDE;

VAR aWindow: TWindow;

BEGIN
        aWindow := NewSimpleWindow(kIDYourWindow, kWantHScrollBar);
        { If you want a nonscrollable window, precede kWantHScrollBar and kWantVScrollBar
          with NOT. You can keep one scroll bar and not the other, if you want to. }

        aWindow.AdaptToScreen;
        { This adapts the window size to a different screen size if necessary. }

        aWindow.SimpleStagger(kHStagger, kVStagger, gStaggerCount);
        { SimpleStagger is a TWindow method that staggers the application's windows
          so they do not completely cover each other. If you use this, you must define
          constants such as kHStagger and kVStagger, which are the number of pixels the
          window should be staggered in the horizontal and vertical dimensions, and
          gStagger, which is an INTEGER global variable used by SimpleStagger to keep
          track of how many windows have been staggered. Initialize gStagger to 0 in
          IYourApplication. If you have multiple windows per document, you may want to
          have multiple global variables like gStagger so the windows can be staggered
          in groups. }
END;
```

Creating a palette window

Purpose

Some applications require a window that contains two views. The DrawShapes sample program is an example; the palette is one view and the drawing area is another. Other applications require windows with two equal areas or with three or more areas.

The areas within windows that allow subviews to be scrolled are called scrollers and are objects of type TScroller. In general, a simple palette window will have two subviews not counting scroll bars: the palette view and a scroller view. The scroller view will have one subview—the view that contains the picture that the scroller will scroll. However, you can have as many subviews as you wish within a window, of class scroller or not, and these may each have their own subviews. Typically, all the views in a window share the same document object.

If you want a simple window with a palette (or any nonscrollable and nonresizable area) and a display area, follow the directions in this recipe.

The characteristics of a window created using the NewPaletteWindow global function used in this recipe are as follows:

- The window contains two subviews: a main view and a palette view.
- The main view may be scrollable, depending on the values passed. (In the template version, the main frame is scrollable.) It is resized along with the window.
- The palette view is not scrollable and is of a fixed size in one direction, while it takes up the width or height of the window in the other direction.
- The palette can be vertically or horizontally oriented, depending on the value of the last parameter of NewPaletteWindow. If you need to create a window of a different form, see the "Creating a Window With Two or More Main Views" recipe.

◆ *Note:* You need to have a window resource in your resource file. See the sample program's resource files for examples of window resources.

How to do it

1. Create a view object type for the main view and the palette view as described in the "Creating a View" recipe. You do not have to worry about creating the scroller superview of your main view or the scroll bar views if your main view is to be scrollable—MacApp does that for you.
2. Add two fields to your document object type to store references to the view objects for the document. The template for this recipe assumes the fields are fMainView and fPaletteView. (If you have additional views, you may want to store them in a list object instead of individual fields.)
3. Define a constant for the fixed dimension of the palette. In the template, it is called kPaletteWidth.
4. Override TYourDocument.DoMakeViews to create your views. In that method, create and initialize the views, and then store them in the fields you've added to your document. (See the "Creating a View" recipe.) This method is called by MacApp just before it calls DoMakeWindows.

5. Override DoMakeWindows for your document. The interface of this method is

   ```
   TYourDocument.DoMakeWindows; OVERRIDE;
   ```

   In your implementation, you first call NewPaletteWindow, the MacApp global function that is the key part of this method. NewPaletteWindow creates a Window Manager window with the requested characteristics, creates two frames, installs your views in the frames, and installs the window object in the document.

   The interface of NewPaletteWindow is

   ```
   FUNCTION NewPaletteWindow(itsRsrcID: INTEGER;
                             wantHScrollBar, wantVScrollBar: BOOLEAN;
                             itsDocument: TDocument
                             itsMainView: TView; itsPaletteView: TView;
                             sizePalette: INTEGER;
                             whichWay: VHSelect): TWindow;
   ```

   The itsRsrcID parameter gives the resource ID used to determine the window template for the window. (The window template defines the window's general appearance, including whether or not the window has a size icon, a close box, and a zoom box, and the appearance of the title bar.)

   The next two parameters, wantHScrollBar and wantVScrollBar, tell whether or not you want scroll bars for the main part of the window. The palette portion never gets scroll bars. Use the kWantHScrollBar and kWantVScrollBar predefined constants here, preceded by NOT if you don't want the scroll bars.

   The itsDocument, itsMainView and itsPaletteView parameters are self-explanatory.

   The sizePalette parameter gives the size of the palette view (not counting borders) in the direction specified by the whichWay parameter (see the next paragraph). In other words, if the palette is at the left of the window, this is the width of the view; if the palette is at the top of the window, this is the height of the view. This size is fixed. (If the window is made smaller or larger in the specified direction, only the main view gets larger.) The size of the palette in the other direction is the full size of the window and can vary.

   The whichWay parameter tells where in the window the palette frame is located. There are two choices: kLeftPalette and kTopPalette.

Template

```
PROCEDURE TShapeDocument.DoMakeWindows; OVERRIDE;

VAR     aWindow: TWindow;

BEGIN
        aWindow := NewPaletteWindow(kIDStdWindow,
                                   kWantHScrollBar, kWantVScrollBar,
                                   SELF, fMainView, fPaletteView,
                                   kPaletteWidth, kLeftPalette);
END;
```

Creating a window with two or more main views

Purpose

Some applications require a window that contains two main views. The DrawShapes sample program is an example; the palette is one view and the drawing area is another. Other applications require windows with two equal areas or with three or more areas.

If you want a simple window with a palette (or, more precisely, with any nonscrollable and nonresizable area) and a display area (which may or may not be scrollable and resizable), you can probably use the NewPaletteWindow global function provided by MacApp. See the "Creating a Palette Window" recipe for details on creating a window using that function. This recipe describes how to create a window with two views in a more general way that can be adapted to any number of views, any of which may be scrollable and resizable.

◆ *Note:* You need to have a window resource in your resource file. See the sample program's resource files for examples of window resources.

How to do it

You know how to create a simple window that contains a single main view before you use this recipe. See the "Creating a Window" recipe.

1. To create more than one view, you usually have a view object type for each kind of view. See the "Creating a View" recipe.

2. Create a field or a number of fields in your document to store references to the view objects for the document. The templates for this recipe assume that

there are two views, stored in the document fields fFirstView and fSecondView.

3. Implement the TYourDocument.DoMakeViews method to create your views. In that method, create and initialize the views, and then store them in the fields you've added to your document. See the "Creating a View" recipe. This method is called by MacApp immediately before it calls DoMakeWindows.

4. Implement DoMakeWindows for your document. The interface of this method is

   ```
   TYourDocument.DoMakeWindows; OVERRIDE;
   ```

   In this method, you will need to create scroller views to be superviews of any scrollable views, but subviews of the window. The scroll bars will be created by MacApp when you create the scrollers. The example in the template shows how to implement this method.

5. To implement this method, you must create the window you need. For every scrollable view in a window, you need to create a scroller and associate each view with its scroller. (You can also associate different views with a single scroller at different times.) When you initialize each scroller, you give a point that defines the initial size of the scroller. Depending on the size determiners passed to IScroller, the scroller may automatically change size when the window changes size. (You can have views that do not have associated scrollers, but they cannot be scrolled.)

   The example in the template shows how to implement this method.

   Your resource file must contain a window template for use by this method. See the "Creating a Window" recipe for a discussion of window resources.

6. Though MacApp may change the *size* of scrollers automatically (when one of its size determiners is sizeSuperView or sizeRelSuperView), MacApp never changes the *location* of a view automatically. If you want the top-left corner of a scroller to move when the window is resized, you must override the scroller's SuperViewChangedSize method. There you would compute the scroller's new location and size and call its Locate and Resize methods to move the scroller and set its size. See the MacApp source code and the Display Architecture ERS for further details.

Templates

```
FUNCTION TYourDocument.DoMakeWindows;

VAR    aWmgrWindow: WindowPtr;
       aWindow: TWindow;
       firstScroller,
       secondScroller: TScroller;
       canResize: BOOLEAN;
       canClose: BOOLEAN;
       tempLocation: VPoint;
       tempSize: VPoint;

BEGIN
       aWmgrWindow := gApplication.GetRsrcWindow(NIL, kYourWindowRsrcID,
                                                canResize, canClose);

       FailNIL(aWmgrWindow);
       { The NIL is in place of a pointer to a space to hold the Window Manager
         window definition.  When a NIL is passed, MacApp uses a pointer to
         a heap block it has allocated. canResize and canClose are returned by
         GetRsrcWindow according to the specifications of the window resource.}

       New(aWindow);
       FailNIL(aWindow);
       aWindow.IWindow(SELF, aWmgrWindow, canResize, canClose, TRUE);
       {Among other actions, installs window in the document.}

       { Create the first scroller. }
       SetVPt(tempLocation, left, top); {upper left corner of first view}
       SetVPt(tempSize, width, height); {dimensions of first view}
                                         {you should supply width and hieght}
       New(firstScroller);
       FailNIL(firstScroller);
       firstScroller.IScroller(aWindow, tempLocation, tempSize,
                             sizeFixed, sizeFixed, 0, 0,
                             kWantHScrollBar, kWantVScrollBar);
       {If you don't want scrolling or resizing, precede the constants with NOTs.}
       firstScroller.AddSubView(fFirstView);

       { Create the second scroller. }
       SetVPt(tempLocation, left, top); {upper left corner of second view}
       SetVPt(tempSize width, height); {dimensions of second view}
                                       {you should supply width and hieght}
       New(secondScroller);
       FailNIL(secondScroller);
       secondScroller.IScroller(aWindow, tempLocation, tempSize,
                              sizeFixed, sizeFixed, 0, 0,
                              kWantHScrollBar, kWantVScrollBar);
       {If you don't want scrolling or resizing, precede the constants with NOTs.}
       secondScroller.AddSubView(fSecondView);
       {Follow the same pattern for each view.}
       aWindow.SetTarget(fFirstView); {The target might be a different view.}
       { You may have additional code here to restore a saved window state. See the
         “Creating a Window” recipe. }
END;
```

Creating a document with two or more windows

Purpose

Some applications display two or more views of a document's data at one time. When you want to display two separate views, whether of a single set of data or of separate data sets, you can display them in two subviews of a single window or in two separate windows. This recipe describes how to display two different views of the same data in separate windows. (If you want to display two or more main subviews in a single window, see the "Creating a Window With Two or More Main Views" recipe.)

How to do it

You should be familiar with the "Creating a Window" recipe, which describes how to create the simplest kind of window.

1. You must have at least one view for each window. The views are normally of different types, although they can be of the same type. See the "Creating a View" recipe.

2. Create a field or a number of fields in your document to store references to the view objects for the document. You may want to use individual fields for each view, or use a list object to hold all the views. The template for this recipe assumes that there are two views stored in the document, named fFirstView and fSecondView.

3. Implement a TYourDocument.DoMakeViews method to create your views. In that method, create and initialize the views, and then store them in the fields you've added to your document. See the "Creating a View" recipe.

4. If you want the windows to be spread evenly around the screen, create a window resource for each of your windows and define a constant for each resource. In the template, the constants are kWindow1Kind and kWindow2Kind. Part of the window resource definition defines the four corners of the window in screen coordinates. After you create the window, you can move it around the screen using the Window Manager procedure MoveWindow; similarly, you can resize the window using the TWindow.Resize method. You may also want to use SimpleStagger and AdaptToScreen. See the template for the "Creating a Window" recipe for more information.

5. Override TDocument.DoMakeWindows for your document. The interface is

   ```
   PROCEDURE TYourDocument.DoMakeWindows; OVERRIDE;
   ```

   In your implementation, begin by creating a window for each view. In the template code for this method, this is done with calls to NewSimpleWindow.

Template

```
PROCEDURE TYourDocument.DoMakeWindows; OVERRIDE;

VAR    window1, window2: TWindow;

BEGIN
       window1 := NewSimpleWindow(kWindow1Kind,
                                  kWantHScrollBar, kWantVScrollBar,
                                  fFirstView);
       { See the "Creating a Window" recipe for details on this call.}

       window2 := NewSimpleWindow(kWindow2Kind,
                                  kWantHScrollBar, kWantVScrollBar,
                                  fSecondView);

       { You may have additional code here to restore a saved window state.
         See the "Creating a Window" recipe.}
END;
```

Creating view templates

Purpose

Since your application may require windows and views in complex hierarchies, you may want to design and correct your view hierarchies many times. View templates allow you to design your views and their hierarchical relationship in a file separate from the rest of your code. The resource compiler will make your view templates become resources that your program can access during runtime to create actual view instances. This gives MacApp's views the same flexibility that other Macintosh resources have.

How to do it

1. Design your view hierarchy. Decide what types of views you want to define, and what the hierarchy should be. Remember that a window view will typically be at the top of your view hierarchies.

If your view hierarchy is going to include scroller views and scroll bar views, and if you are not overriding MacApp's scroll bar methods, you do not need to include scroll bar views in your view templates. MacApp will create those for you automatically.

2. Add a view resource for each view hierarchy (usually one per window) to your .r file. The view resource format is defined in the file MacAppTypes.r, and you might want to examine that definition for details later. For now, this recipe will give you the necessary details to create simple view resource templates.

   View resources are formatted like this:

```
resource 'view' (1001, purgeable) {
    {
       /* Each view of the hierarchy has an entry here */
       }
};
```

   Let's take an example view resource:

```
resource 'view' (1001, purgeable) {
   {
   root, 'WIND', {50, 20}, {260, 430},
   sizeVariable, sizeVariable, shown, enabled,
   Window { "class name", <fields specific to window views go here> };

   'WIND', 'MAIN', {0, 0}, {140, 240},
   sizeFixed, sizeFixed, shown, enabled,
   View { "class name", <fields specific to views go here> }
   /* Note there is no ';' after the right brace above. */
   }
 };
```

   As you can see in this example, each view entry has a format like this:

```
ParentViewsID, ThisViewsID, LocationInParentView, SizeOfView,
VerticalSizeDeterminer, HorizontalSizeDeterminer, ShownDeterminer, EnabledDeterminer,
TypeOfView { "class name", <more fields depending on type of view> }
```

   Let's look at each of these fields.

   - **ParentViewsID.** This field contains the ID of the parent view. You can use the word **root** (with no quotes) to specify that this view has no parent view. You will always want to specify **root** for your window views; otherwise you must use a four character string that is the ID of the parent view..

- **ThisViewsID.** The identifier of this view. Identifiers are four-character strings, such as 'WIND', 'MAIN', 'SCRL',or 'MYVW'. You create this identifier to use in theParentViewsID field of any view that is a subview of this view. You can also use this identifier in your program to obtain a reference to a subview in a view hierearchy. See the “Creating and Initialization Views with Templates” section. View identifiers do not have to be unique, and not all views need an identifier. For example, if you have a view that has no subviews, and that you won’t need a reference to in your program,you can use the constant **noID** (with no quotes) for its identifier.

- **LocationInParentView.** This specifies the offset of the upper left-hand corner of this view from its parent view, in pixels.

- **SizeOfView.** This field specifies the initial size of the view.

- **VerticalSizeDeterminer and HorizontalSizeDeterminer.** These fields determine how the view is to be sized. The possible values are: sizeSuperView, sizeRelSuperView, sizePage, sizeFillPages, sizeVariable, sizeFixed.

- **ShownDeterminer.** This determines whether the view is displayed initially. The value choices are: shown and notShown.

- **EnabledDeterminer.** This determines whether the view is enabled (whether it responds to mouse clicks). The possible values are: disabled and enabled.

- **TypeOfView.** This determines the format of the rest of the view’s data. You can think of this as indicating the type of view to create, though it doesn’t actually indicate a view’s class. (See the “Class Name” entry below.) This field is followed by the view’s class name and a list of fields specific to the class of the view instance. The predefined choices for this field are: View, Window, Scroller, DialogView, Control, Button, CheckBox, Radio, ScrollBar, Cluster, Icon, Picture, Popup, StaticText, EditText, NumberText, TEView, GridView, TextGridView, TextListView. You can define your own view types to have resource entries, if you like, by either changing the ViewTypes.r Rez file or creating your own view types Rez file. However, because your views will probably be descendants of one of these classes of views, you will most likely be able to use the ancestor class for your template, and do any extra initialization in your own code.

- **Class Name.** This determines what class the view instance belongs to. It is the name of the class, in double quotes, as defined the program. For instance, if the view is an instance of a standard MacApp window then the class name would be "TWindow". However, if the view is an instance of a subclass of TWindow, for example TYourWindow, then the class name would be "TYourWindow."

Each of these view class entries has its own list of fields. You can see examples of these in the samples below and in the sample programs. If you want a complete enumeration of these fields, see the ViewTypes.r rez file.

Here is an example of a complete view resource:

```
resource 'view' (1001, purgeable) {
   {
   root, 'WIND', {50, 20}, {260, 430},
   sizeVariable, sizeVariable, shown, enabled,
   Window { "TWindow", zoomDocProc, goAwayBox, resizeable, modeless,
             ignoreFirstClick, freeOnClosing, disposeOnFree, closesDocument,
             openWithDocument, dontAdaptToScreen, stagger, forceOnScreen,
             dontCenter, 'NOTH', "" };

   'WIND', 'SCLR', {0, 0}, {260-kSBarSizeMinus1, 430-kSBarSizeMinus1},
   sizeRelSuperView, sizeRelSuperView, shown, enabled,
   Scroller { "TScroller", vertScrollBar, horzScrollBar, 0, 0, 16, 16,
                vertConstrain, horzConstrain, {0, 0, 0, 0} };

   'SCLR', 'NOTH', {0, 0}, {140, 240},
   sizeFixed, sizeFixed, shown, enabled,
   View { "TNothingView" }
   }
};
```

This view template defines a window view with one immediate subview, a scroller, which, in turn, has its own subview. The size of the scroller is relative to the size of the window—it leaves space for the horizontal and vertical scroll bars. The 'NOTH' view is of a fixed size, and is not offset from the scroller.

As you can see, the fields in the view class template entries basically mirror the fields of the the view classes.

3. It is possible to divide a view hierarchy into several view resources. For example,

```
resource 'view' (1001, purgeable) {
   {
   root, 'WIND', {50, 20}, {260, 430},
   sizeVariable, sizeVariable, shown, enabled,
   Window { "TWindow", zoomDocProc, goAwayBox, resizeable, modeless,
             openWithDocument, freeOnClosing, disposeOnFree, closesDocument,
             openWithDocument, dontAdaptToScreen, stagger, forceOnScreen,
             dontCenter, 'NOTH', "" };

   WIND, 'SCLR', {0, 0}, {260-kSBarSizeMinus1, 430-kSBarSizeMinus1},
   sizeRelSuperView, sizeRelSuperView, shown, enabled,
   Scroller { "TScroller", vertScrollBar, horzScrollBar, 0, 0, 16, 16,
               vertConstrain, horzConstrain, {0, 0, 0, 0} };

   'SCLR', IncludeViews {1002}
   }
};

resource 'view' (1002, purgeable) {
   {
   root, 'NOTH', {0, 0}, {140, 240},
   sizeFixed, sizeFixed, shown, enabled,
   View { "TNothingView" }
   }
};
```

Template

```
/* A complete view hierarchy in one resource. */
   resource 'view' (1001, purgeable) {
       {
      root, 'WIND', {50, 20}, {260, 430},
      sizeVariable, sizeVariable, shown, enabled,
      Window { "TWindow", zoomDocProc, goAwayBox, resizeable, modeless,
               openWithDocument, freeOnClosing, disposeOnFree, closesDocument,
               openWithDocument, dontAdaptToScreen, stagger, forceOnScreen,
               dontCenter, 'MAIN', "" };

      'WIND', 'SCLR', {0, 0}, {260-kSBarSizeMinus1, 430-kSBarSizeMinus1},
      sizeRelSuperView, sizeRelSuperView, shown, enabled,
      Scroller { "TScroller", vertScrollBar, horzScrollBar, 0, 0, 16, 16,
                 vertConstrain, horzConstrain, {0, 0, 0, 0} };

      'SCLR', 'MAIN', {0, 0}, {140, 240},
      sizeFixed, sizeFixed, shown, enabled,
      View { "TYourView" }
      }
   };
```

```
/* A view hierarchy spread out over two resources. */
   resource 'view' (1001, purgeable) {
       {
       root, 'WIND', {50, 20}, {260, 430},
       sizeVariable, sizeVariable, shown, enabled,
       Window { "TWindow", zoomDocProc, goAwayBox, resizeable, modeless,
                openWithDocument, freeOnClosing, disposeOnFree, closesDocument,
                openWithDocument, dontAdaptToScreen, stagger, forceOnScreen,
                dontCenter, 'MAIN', "" };

       WIND, 'SCLR', {0, 0}, {260-kSBarSizeMinus1, 430-kSBarSizeMinus1},
       sizeRelSuperView, sizeRelSuperView, shown, enabled,
       Scroller { "TScroller", vertScrollBar, horzScrollBar, 0, 0, 16, 16,
                  vertConstrain, horzConstrain, {0, 0, 0, 0} };

       'SCLR', IncludeViews {1002}
       }
   };

   resource 'view' (1002, purgeable) {
       {
       root, 'MAIN', {0, 0}, {140, 240},
       sizeFixed, sizeFixed, shown, enabled,
       View { "TYourView" }
       }
   };
```

Creating and initializing a view with templates

Purpose

Once you've created your view resources, you must still call the correct routines to create the actual view instances.

How to do it

1. Register your view classes. MacApp creates views from resources by cloning prototype views which you must create when your program begins. For that reason, you must "register" each of your view classes that will be read from view resources, Typically you register your view classes in IYourApplication. Registering a view class looks like this:

```
NEW(aYourView);
FailNil(aYourView);
RegisterType('TYourView', aYourView);
```

where aYourView is an instance of TYourView local to IYourApplication. The view classes defined by MacApp are already registered: you only need to register view classes that you define.

2. In the DoMakeViews method of your document class, you can create your view hierarchy using the global function NewTemplateWindow. For example:

   ```
   aWindow := NewTemplateWindow(kYourWindowID, SELF);
   ```

   In this example, kYourWindowID should be defined to be 1001 to match the resource definition given in the last section.

   To get a reference to your TYourView instance in the view hierarchy, you can use the FindSubView method. FindSubView will return a reference to a particular view type in a view hierarchy, given that subview's four-character identifier. For instance,

   ```
   aYourView := TYourView(aWindow.FindSubView('MAIN'));
   ```

   will put a reference to the 'NOTH' subview instance of aWindow into aYourView. If more than one subview of aWindow was given 'NOTH' as an identifier in the resource file, then a reference to the first 'NOTH' subview found will be returned—so be careful if you don't use unique identifiers in your view templates.

   Since FindSubView returns a reference to the generic TView class, you will need to cast the result to the class of your view.

   If you want to create a view hierarchy that does not have a window as the root, you can call the TEvtHandler method DoCreateViews. For example,

   ```
   aYourView := TYourView(DoCreateViews(SELF, NIL, kYourViewID));
   ```

   If kYourView has been defined to be 1002 (to correspond to our resource template in the previous section), then aYourView will now reference the view at the top of the view hierarchy defined in view resource 1002.

3. Initialize your views. MacApp calls the IRes initialization method of view objects created from resources. If you have created a descendant class of a MacApp-defined view class, you may want to do some specialized initialization of your own. To do this, you must override IRes to initialize your descendant class's fields. However, remember that you cannot add any parameters to an override method, so you may still wish to create your own initialization method, and ensure that it is called.

Template

```
PROCEDURE TYourApplication.IYourApplication(itsMainFileType: OSType);

VAR    aYourView: TYourView;

BEGIN
    IApplicaiton(itsMainFileType);

    NEW(aYourView);
    FailNil(aYourView);
    RegisterType('TYourView', aYourView);
END;


PROCEDURE TYourDocument.DoMakeViews(forPrinting: BOOLEAN);

VAR
    aView:     TView;
    aWindow:   TWindow;

BEGIN
    { The forPrinting parameter is set to TRUE when the user has requested }
    { printing from the Finder.  In this case, you only need to create the }
    { view that is actually being printed. }
    IF forPrinting THEN BEGIN        { You don't need the whole window. }
        aYourView := TYourView(DoCreateViews(SELF, NIL, kYourViewID));
        fYourView := aYourView;
    END
    ELSE BEGIN                        { You do need the whole window. }
        aWindow := NewTemplateWindow(kYourWindowID, SELF);
        fYourView := TYourView(aWindow.FindSubView('MAIN'));
    END;
```

Drawing a view

Purpose

When one of your application's windows needs to be updated, MacApp calls the DrawContents method for the view representing the window. DrawContents sends a DrawContents message to each subview, and then calls the window's Draw method. (Draw is defined for TView; the default method, TView.Draw, does nothing.) TYourView.Draw translates between the data stored in the document and the screen (or printed page).

How to do it

This recipe assumes your data is organized into a list of instances that draw themselves. In other words, your data consists of objects organized into a list (generally stored in an object of class TList), and each object type has a TItem.Draw method. TItem.Draw actually draws the object. If your application cannot be organized like that, have TYourView.Draw do the drawing itself. Note that you rarely, if ever, call any Draw method yourself; you call TView.InvalidRect to invalidate the part of your view that has changed or call TView.DrawContents if you need to redraw the view and its subviews immediately. When there is nothing else for the application to do, MacApp calls the Draw methods for all views that have invalidated areas that are actually displayed in the window.

See the sample programs for other examples of Draw.

Implement TYourView.Draw. The interface of that method is

```
PROCEDURE TYourView.Draw(area: Rect);
```

A sample method is given in the template for this recipe. That sample assumes your objects draw themselves and their Draw methods take no parameters. The sample also makes no use of the area parameter, which is a rectangle containing all invalid areas. You can use the area parameter to optimize your drawing. See the "Optimizing Drawing" recipe.

If you use filtered commands, this method is often coded so it draws items that are not in the document or skips some items that are in the document. See the "Creating Filtered Commands" recipe for more information.

Template

```
PROCEDURE TYourView.Draw(area: Rect);
{ See the "Optimizing Drawing" recipe for a discussion of the area parameter. }

        PROCEDURE DrawItem(item: TItem);
        BEGIN
                item.Draw(area); { See the "Drawing an Object in a View" recipe. }
        END;

BEGIN
        fItemList.Each(DrawItem);
END;
```

Drawing an object in a view

Purpose

MacApp calls TYourView.Draw when drawing is required. You often pass the actual drawing on to the objects that make up your view. You generally do that so that the view has no need to know the form of the objects.

How to do it

1. Implement TYourView.Draw as described in the "Drawing a View" recipe.
2. Draw the object in your view's coordinate system. MacApp has already set up the drawing environment so that drawing takes place in your view.
3. If you want to optimize drawing by only drawing what has changed and is visible, see the "Optimizing Drawing" recipe.

Because drawing is so application-specific, no template is given for this recipe, but it might be helpful to look at the Draw methods in the Nothing, DrawShapes and Calc sample programs.

Optimizing drawing

Purpose

When MacApp calls TYourView.Draw, it passes a rectangle (the area parameter) that gives the invalid area of the view, which is the only part that needs to be redrawn. Whenever you call one of the invalidating routines, the rectangle you give is added to the invalid area. In addition, whenever the user scrolls the frame, the strip that appears is added to the invalid area. MacApp automatically adjusts the invalid area so that only parts actually displayed in the frame are included. Therefore, the maximum invalid area is the size of the content rectangle of the frame, even if you have invalidated other areas.

Note that moving a window does not invalidate its contents, unless it was partly off the screen, because the system automatically moves the window's contents along with its borders. Also, covering a window does not invalidate the contents of the covered window. Uncovering a window invalidates the newly revealed parts. Similarly, when a view is scrolled, only the part that newly appears in the frame is invalidated. The part that was already displayed in the view but has now been moved is not invalidated.

The area parameter is always the smallest displayed rectangle that encloses all invalidated areas.

This recipe describes how to use the invalid area so that you only draw the part of the view that needs to be drawn.

How to do it

If your data set consists of separate objects that are not spatially ordered, you must check each object to see if it is in the invalid area. There are two places in which you can check: in TYourView.Draw, before calling item.Draw, or in TItem.Draw. The templates section of this recipe shows examples of both.

You need a way of identifying the rectangle containing a particular item. In the template methods, there is a field of TItem called fExtentRect that is a Rect with the bounds of the item. You could replace fExtentRect with a functional method that returns the same value. Note that using a rect for fExtentRect works only for views no larger than 30,000 pixels. For larger views, or views located in a larger space, you would use a VRect for fExtentRect.

(The methods in the template call RectIsVisible. If you look at the MacApp source code, you'll find that RectIsVisible tests whether the given Rect is in the window's visRgn. The Window Manager sets the visRgn to the intersection of the visRgn and the update region before the update cycle begins.)

If your data set is organized spatially (for example, in rows and columns or in paragraphs) you can avoid examining parts that are definitely not in the invalid area. You can do this in an application displaying rows and columns, for example, by finding the first and last row and the first and last column that intersect the invalid area. Then, only the rows and columns between those limits need to be drawn. The templates contain an example.

Templates

```
{ The following procedure shows how you can optimize TYourView.Draw. }
PROCEDURE TYourView.Draw(area: Rect);
       PROCEDURE DrawItem(item: TItem);
       BEGIN
              IF RectIsVisible(item.fExtentRect) THEN
                     item.Draw; {See the "Drawing an Object in a View" recipe.}
       END;
BEGIN
       fItemList.Each(DrawItem);
END;


{ The following procedure shows how you can optimize TItem.Draw. }
PROCEDURE TItem.Draw(area: Rect);
BEGIN
       IF RectIsVisible(fExtentRect) THEN
              {Draw the object}
END;


{ The following procedure shows how you can optimize drawing in spatially organized views. }
PROCEDURE TYourView.Draw(area: Rect);

VAR    firstRow, firstCol, lastRow, lastCol: INTEGER;
       rowIndex, colIndex: INTEGER;

BEGIN
       GetDrawLimits(area, firstRow, firstCol, lastRow, lastCol);
       FOR rowIndex := firstRow TO lastRow DO
              FOR colIndex := firstCol TO lastCol DO
                     DrawItemAt(rowIndex, colIndex);
                     { The method DrawItemAt is not specified here. Its implementation
                       depends on how you structure your data. }
END;

PROCEDURE TYourView.GetDrawLimits(area: Rect;
                                  VAR firstRow, firstCol, lastRow, lastCol: INTEGER);

       PROCEDURE PtToRowCol(aPoint: Point; VAR row, column: INTEGER);
       BEGIN
              row := aPoint.v DIV cRowHeight;   { You define cRowHeight. }
              column := aPoint.h DIV cColWidth; { You define cColWidth.  }
       END;

BEGIN
       PtToRowCol(area.topLeft, firstRow, firstCol);
       PtToRowCol(area.botRight, lastRow, lastCol);
       lastRow := Min(lastRow, fNumRows);
       lastCol := Min(lastCol, fNumCols);
       { The preceding two statements assume that you maintain the current number of rows
         and columns in fields of the view. }
END;
```

Handling mouse events

You often need to track the pointer after the mouse button goes down and take some action while the mouse moves or when the mouse button comes up. (You also occasionally need to track the mouse when the button is up and take action when the button goes down.) Mouse actions normally fall into four groups:

- selecting
- manipulating buttons and other controls
- dragging
- drawing

When MacApp detects a mouse-down event, it first checks the location of the mouse when the button was pressed. If the mouse button was pressed when the pointer was not in one of the window's subviews, the event is handled by MacApp, which may call your code. For example, the user may choose a menu item, which results in a call to yourView.DoMenuCommand. If the pointer was in a scroll bar, it causes scrolling to take place, which results in a call to yourView.Draw.

However, if the pointer was in one of your views when the mouse button was pressed, TYourView.DoMouseCommand is called.

The DoMouseCommand method is a function that returns either a handle to a command object or the global variable gNoChanges. If the mouse event requires tracking or indicates that the user is beginning an undoable command, DoMouseCommand will create a command object; otherwise, if there is a command, it will execute the command and return gNoChanges.

This section includes a recipe for handling each of the four general types of mouse actions. Each recipe assumes that only that type of action can occur. These four recipes are followed by a recipe for tracking the mouse, which contains detailed information about the mouse trackers used for the preceding four recipes. Then there is a recipe that shows how to differentiate among several possible mouse actions.

The last recipe in this section covers the relatively rare need to track the mouse after the mouse button comes up. (The most common situation in which this is necessary is when drawing a polygon, as in MacDraw.)

Selecting

Purpose

The user can select all or part of an object displayed in your view in preparation for performing some action. You need to detect when the user is attempting to select something, figure out what was selected, and mark it as selected in your data set and also in the view by highlighting it in some way.

For simplicity, this recipe assumes that a mouse-down event indicates either a selection or nothing. In general, a mouse-down event indicates the beginning of one of a number of possible actions, and your program uses a number of criteria to figure out which action the user wants. See the "Handling Several Types of Mouse Events" recipe for an example of integrating different types of mouse actions.

How to do it

1. Write TYourView.DoMouseCommand so that it detects selections. The user should be able to select a single item and should be able to make multiple selections.

   There are several ways to handle multiple selections, generally depending on the kind of data being selected.

   Applications generally handle selections in one of two ways:

- If your application, like the sample program DrawShapes, has discrete independent objects scattered around the view, the user should be able to select individual objects by clicking them. The user should also be able to make multiple selections by drawing a selection rectangle around several objects, and add objects to the group of selected objects by holding down the Shift key and clicking a new object. (Similarly, the user should be able to remove selections from the group by holding the Shift key and clicking a selected object.) Selections don't have to be contiguous—selecting two objects using Shift-click does not automatically select everything between the two objects.

- If your application, as text or spreadsheet applications do, has data organized in a contiguous list, selections should be contiguous arbitrary portions of the data. If the application deals with text, the amount selected usually depends on the number of clicks (that is, a single click places an insertion point, a double click selects a word, and a triple click selects a paragraph). In addition, the user should be able to select blocks of text by holding the mouse button down and dragging the pointer across the text, as well as by holding the Shift key down and clicking to extend the selection. Extending the selection generally selects everything up to the new selection. Selection in cell-based applications, such as spreadsheet programs, are similar.

Some applications (such as MacDraw) fall partially into both categories, depending on the mode chosen by the user.

Text selections are usually handled by UTEView (see the "Using UTEView" recipe). If you need to handle text selections yourself, see the UTEView source code.

If you have discrete objects displayed in your view, DoMouseCommand can follow this plan:

- Scan through your set of objects and check each to see whether the mouse pointer was within its area. If you find the mouse pointer was over an object, check whether the Shift key was down. If it wasn't, mark the identified object as selected and deselect the previous selection. If the Shift key was down, toggle the selection status of the identified object. See step 2 of this recipe for a discussion of how the selection status of objects may be stored.

- If the pointer was not over any object, the user may have been trying to select or deselect a group of objects. Create a selector object, which is a type of mouse tracker. (See the "Tracking the Mouse" recipe for a description of mouse trackers and their methods.) MacApp calls the methods command.TrackMouse, command.TrackFeedback, and command.TrackConstrain while the button is down. You can find all the selected objects and mark them in TrackMouse when the trackPhase is trackRelease. (See step 2 of this recipe for a discussion of marking selections.)

  A sample of a TrackMouse method for a selector object is given in the templates for this recipe.

  A template for DoMouseCommand is also given in the templates for this recipe.

2. Create a DoHighlightSelection method for your view. MacApp calls yourView.DoHighlightSelection after it calls yourView.Draw. The interface for DoHighlightSelection is

```
PROCEDURE TYourView.DoHighlightSelection(fromHL, toHL: HLState);
OVERRIDE;
```

HLState is an enumerated type with values hlOff, hlDim, and hlOn. The value hlOff indicates that no highlighting should take place; hlOn indicates that the selection should be highlighted when the window is active; hlDim indicates that the selection should be also highlighted when the window is inactive. Dim highlighting (which is not part of the user interface standard and is an optional enhancement) can be used instead of no highlighting when the window is not active. If your application doesn't do highlighting you can treat hlDim and hlOff as the same thing.

DoHighlightSelection finds all selections and turns highlighting on, off, or to dim. MacApp calls it when the window showing the view is activated or deactivated or when the view is updated. The values of the parameters are as follows:

- Updating the active window: hlFrom is hlOff and hlTo is hlOn.
- Updating an inactive window: hlFrom is hlOff and hlTo is hlDim.
- Activating a window: hlFrom is hlDim and hlTo is hlOn.
- Inactivating a window: hlFrom is hlOn and hlTo is hlDim.

You call DoHighlightSelection yourself when the selection changes. A sample implementation is given in the templates for this section. The template allows multiple selections, with each object marked as selected or not selected.

Unlike most methods that draw in the view, DoHighlightSelection can be called from other methods. When the selection changes, you can remove highlighting from the old selection by calling DoHighlightSelection(hlOn, hlOff) and then calling DoHighlightSelection(hlOff, hlOn) to highlight the new selection. Note that your view must be focused before calling DoHighlightSelection. If you're not sure if it's focused you can insert code to say

```
IF yourView.Focus THEN ....
```

When MacApp calls your Draw, DoHilightSelection, or DoMouseCommand methods, your view has been focused.

3. Record what objects (or parts of objects) are selected. There are many ways you could record this information. Some common ways are listed here:

   - If there is always only one selection, the selection is somehow indicated separately from the list of objects (probably stored in a field of the document, or if that is not meaningful, of the view), and DoHighlightSelection simply highlights the current selection.

   - The document (or view, if necessary) has a list of selected objects separate from the list of all objects. DoHighlightSelection scans through that list and highlights all of them.

   - Each object is marked as selected or not selected. One way to mark them is to have a Boolean field, fIsSelected, in each object. When the object is initialized, you set that field to FALSE. DoHighlightSelection scans through the list of objects and highlights any that have fIsSelected TRUE.

   - There is a Boolean function that decides whether or not an object is selected. DoHighlightSelection can scan through the list of all objects and highlight those for which this function returns TRUE.

4. Define and implement a command object to handle selection.

Templates

```
TYourSelector = OBJECT(TCommand);

        fYourDocument: TYourDocument;
        fYourView: TYourView;
        fDeltaH: INTEGER;
        fDeltaV: INTEGER;

        PROCEDURE TYourSelector.IDragger(view: TYourView);

        FUNCTION TYourSelector.TrackMouse(aTrackPhase: TrackPhase;
                                VAR anchorPoint, previousPoint, nextPoint: VPoint;
                                mouseDidMove: BOOLEAN): TCommand; OVERRIDE;

        PROCEDURE TYourSelector.DoIt; OVERRIDE;

        PROCEDURE TYourSelector.UndoIt; OVERRIDE;

        PROCEDURE TYourSelector.RedoIt; OVERRIDE;
```

```
       PROCEDURE TYourSelector.TrackFeedback(anchorPoint, nextPoint: VPoint;
                                           turnItOn, mouseDidMove: BOOLEAN); OVERRIDE;

       PROCEDURE TYourSelector.FixSelection;

       PROCEDURE TYourSelector.MoveBy(moveIt: BOOLEAN);

END;


FUNCTION TYourView.DoMouseCommand(VAR theMouse: Point; VAR info: EventInfo;
                               VAR hysteresis: Point): TCommand;

VAR    hitItem: TItem;
       aSelector: TYourSelector;

       PROCEDURE CheckHit(item: TItem);
       BEGIN
              IF {for example} PtInRect(theMouse, item.fBoundsRect) THEN
                     hitItem := item;
       END;

BEGIN
       hitItem := NIL;
       IF NOT info.theShiftKey THEN
              DeSelect;
              { This is a method you must design and add to your view to remove marking
                from the current selection or selections. }

       fItemList.Each(CheckHit);
       { This TList-type field of the view holds all the application's items. The code
         here assumes that the list is ordered back-to-front and the frontmost object
         is the one the user selects. }

       IF hitItem = NIL THEN BEGIN { begin a selection rectangle }
              New(aSelector);
              FailNIL(aSelector);
              aSelector.ISelector(fYourDocument, SELF, info.theShiftKey);
              DoMouseCommand := aSelector;
       END
       ELSE BEGIN { one object selected or toggled }
              DoMouseCommand := gNoChanges;
              hitItem.fIsSelected := NOT hitItem.fIsSelected;
       END;
END;
```

```
PROCEDURE TYourSelector.ISelector(ItsDocument: TYourDocument;
                               itsView: TYourView; shiftKey: BOOLEAN);

BEGIN
       { Call ICommand to set the command's fView to the view in which tracking
         takes place and to set the scroller used for automatic scrolling during
         selection.  cSelect is command number constant for this command that can
         be used to distinguish one kind of selection from another.  After calling
         ICommand it is necessary to set fCausesChange and fCanUndo to false, as
         a selection neither changes a document or is undoable. }

       ICommand(cSelect, itsView, itsView.GetScroller(TRUE));
       fCausesChange := FALSE;
       fCanUndo := FALSE;
       fYourDocument := itsDocument;
END;


FUNCTION TYourSelector.TrackMouse(aTrackPhase: TrackPhase;
                                VAR anchorPoint, previousPoint, nextPoint: VPoint;
                                mouseDidMove: BOOLEAN): TCommand;

       PROCEDURE CheckHit(item: TItem);
       BEGIN
       { Here check if the item is in the rectangle marked by the mouse between
         anchorPoint and nextPoint. If it is, mark it selected or deselected or
         add it to the list or remove it from the list of selected items, depending
         on the state of the Shift key stored in the selector object. }
       END;

BEGIN
       TrackMouse := SELF;
       IF aTrackPhase := trackRelease THEN
       BEGIN
              fView.DoHighlightSelection(hlOn, hlOff);
              fYourDocument.Each(CheckHit); {assumes items are in a TList list}
              fView.DoHighlightSelection(hlOff, hlOn);
              TrackMouse := gNoChanges;
       END;
END;


PROCEDURE TYourView.DoHighlightSelection(fromHL, toHL: HLState);

       PROCEDURE HighlightItem(item:TItem);
       BEGIN
              IF item.fIsSelected THEN
                     item.Highlight(fromHL, toHL);
       END;

BEGIN
       fItemList.Each(HighlightItem);
END;
```

Dragging

Purpose

Many applications have discrete objects that can be moved around the view. They are often moved by the user who can drag such objects with the mouse.

For simplicity, this recipe assumes that a mouse press indicates that the user wants to drag an object or has no meaning. In general, a mouse press may indicate a number of possible actions, and your program uses a number of criteria to figure out which action the user wants. See the "Handling Several Types of Mouse Events" recipe for an example of integrating different types of mouse actions.

How to do it

1. Implement DoMouseCommand so that it creates a dragger object if the mouse has been clicked on an object. (If the mouse has not been clicked on an object, nothing should be done and DoMouseCommand should return gNoChanges to indicate that no valid action has occurred.) The next step discusses dragger objects.

   The DoMouseCommand in the templates assumes that the object located under the mouse pointer need not be marked as selected and any previous selection should not be deselected, a choice of action that is rarely appropriate but is used here for simplicity because this recipe ignores all selection issues. See the "Selecting" recipe for a full discussion of selection.

2. Implement a dragger object. Here is a sample interface of a dragger type:

```
TYourDragger = OBJECT(TCommand);

        fYourDocument: TYourDocument;
        fYourView: TYourView;
        fDeltaH: INTEGER;
        fDeltaV: INTEGER;

        PROCEDURE TYourDragger.IDragger(view: TYourView);

        FUNCTION TYourDragger.TrackMouse(aTrackPhase: TrackPhase;
                                         VAR anchorPoint,
                                             previousPoint, nextPoint: VPoint;
                                         mouseDidMove: BOOLEAN): TCommand; OVERRIDE;

        PROCEDURE TYourDragger.DoIt; OVERRIDE;

        PROCEDURE TYourDragger.UndoIt; OVERRIDE;
        PROCEDURE TYourDragger.RedoIt; OVERRIDE;

        PROCEDURE TYourDragger.TrackFeedback(anchorPoint, nextPoint: VPoint;
                                     turnItOn, mouseDidMove: BOOLEAN); OVERRIDE;

        PROCEDURE TYourDragger.FixSelection;

        PROCEDURE TYourDragger.MoveBy(moveIt: BOOLEAN);

    END;
```

You need to override TrackFeedback because you generally need to give feedback other than the standard flickering rectangle, which is only appropriate for making certain kinds of selections If you want to constrain the mouse (for example, to conform to a grid), also override TrackConstrain. TrackFeedback and TrackMouse are discussed later in this recipe. TrackConstrain is discussed in the "Tracking the Mouse" recipe.

A dragger object is a type of mouse tracker. See the "Tracking the Mouse" recipe for details on mouse trackers.

3. Add the following field to your view:

```
fDragging: BOOLEAN;
```

This field is used to determine if the mouse is actually moving. It is used for a number of optimizations, but is primarily necessary so that TrackMouse can determine when the mouse has first moved. See the discussion of TrackMouse later in this recipe for an explanation.

In your IYourView method, initialize fDragging to FALSE.

4. Define a command constant for the dragging command. Although dragging is not a menu command, it must have its own unique constant, such as

```
cDragCommand = { Use numbers above 1000 for your application's commands.
                 Building blocks can use numbers above 500. };
```

5. In your TYourView.Draw method, before drawing each item, you may want to test whether fDragging is TRUE and the item is currently selected. If both conditions are TRUE, you might not draw the item in Draw. Instead, you may draw it in its current position in TrackFeedback. (Whether or not you do this depends on what you want the user to see during a dragging operation.) Similarly, you may want to prevent highlighting in your DoHighlightSelection method if the item is being dragged.

6. Implement IDragger. Note that fView.fDragging should be set to FALSE here because at the time the dragger object is created, you cannot assume

that dragging will actually occur, only that it is possible. Also, you ordinarily call ICommand to initialize the command. (In some cases, you may call another method which itself calls ICommand.) No template is given for this method; see the previous section or the sample program PatView for an example.

7. Implement TrackFeedback so that it shows the dragged item or items as they move. The feedback should, of course, be chosen as appropriate for your application, but to prevent unnecessary drawing you should gate your feedback by checking whether fView.fDragging is TRUE and whether mouseDidMove is TRUE. (The parameter mouseDidMove is passed to your TrackFeedback method by MacApp. It indicates whether or not the mouse moved since the last time TrackFeedback was called.)

8. Add the following method to the interface of your view type:

```
PROCEDURE TYourView.PrepareToTrack;
```

   This method prepares the view for dragging. To do so, it should erase any selected items (unless you don't want your application to do that) and set fView.fDragging to TRUE. If your view contains items that might overlap—in which case, when you erase the selected items, you might also erase unselected items that overlap the selected items—call DrawContents. A sample of this method is shown in the templates for this recipe.

9. Implement TrackMouse. When aTrackPhase is trackMove, check the value of fYourView.fDragging. If fYourView.fDragging is FALSE, this is the first time that TrackMouse has been called in the trackMove phase, and it is time to prepare for tracking. First, call the view's DoHighlightSelection(hlOn, hlOff) to remove highlighting from the selection. Then, call the view's PrepareToTrack method. (See the previous step of this recipe.) Finally, focus on your view, because PrepareToTrack may have changed the focus. If fYourView.fDragging is TRUE, you don't have to do anything, unless your application has actions that should be performed at this time.

   When aTrackPhase is trackRelease, if fYourView.fDragging is still FALSE, you should return gNoChanges, because the user has done nothing. If fYourView.fDragging is TRUE, it is time to set up for moving the items that were dragged. (If this drag doesn't change the document, you can carry out the action of the command here, and then return gNoChanges. This recipe assumes that a dragging action changes the document.) To set up for moving, calculate the change in position and store those values in fDeltaH and fDeltaV. Finally, reset fYourView.fDragging to FALSE.

A sample TrackMouse method is shown in the template.

10. If a dragger changes the document, the action of the dragger is not performed in TrackMouse (although TrackFeedback may make it appear to the user that the action of the dragger has been carried out); instead, the action is performed by the DoIt method. MacApp calls DoIt after the mouse button comes up. A sample DoIt method is given in the templates. The sample assumes that you have a MoveBy method, which actually moves the object. A sample MoveBy method is shown in the templates.

    You should also implement UndoIt and RedoIt for your dragger type. Using MoveBy rather than actually moving the object in DoIt makes implementing UndoIt and RedoIt easier. Samples of UndoIt and RedoIt are given in the templates. The next step of this recipe includes further discussion of what is necessary to properly undo and redo this command.

    Notice that MoveBy checks all objects and moves any that are selected. The sample assumes that the objects are marked as selected or not selected. Your application may maintain its selections differently or may allow only a single selection. See the "Selecting" recipe for details on marking selections.

    MoveBy as shown in the templates invalidates the original position of the object. The DrawShapes sample program handles that invalidation differently, and also generally handles dragging items differently. You may want to examine DrawShapes to get a different perspective on this operation.

11. When you undo and redo this command, you must be sure that the selections are set correctly. Because selections do not change the document, the dragger command is not committed just because the user changes the selection. Thus the user might change the selection before choosing Undo. You must therefore have a record of what was selected when the dragger command was executed, and you must restore the selection when Undo and Redo are chosen.

    Implement TYourDragger.FixSelection so that it restores the selections in effect when the command was first executed. You can record the old selections in any of the ways that you can record current selections. The sample in the templates gives a field fWasSelected to every object, as well as a field fIsSelected. The current selection is indicated in fIsSelected; the selection at the time of the command is in fWasSelected. When the command is undone or redone, fIsSelected has its value replaced by fWasSelected.

Have Undo and Redo call FixSelection before calling MoveBy.

Templates

```
FUNCTION TYourView.DoMouseCommand(VAR theMouse: Point; VAR info: EventInfo;
                               VAR hysteresis: Point): TCommand;

VAR     hitItem: TItem;
        dragger: TYourDragger;

        FUNCTION CheckHit(item: TItem): BOOLEAN;

        BEGIN
               CheckHit := {test location for hit};
        END;

BEGIN
        hitItem := fYourDocument.fItemList.FirstThat(CheckHit);
        IF hitItem <> NIL THEN BEGIN
               { You should mark the item as selected }

               { Create a dragger command object }
               New(dragger);
               FailNIL(dragger);
               dragger.IDragger(fYourDocument, SELF);
               DoMouseCommand := dragger;
        END
        ELSE
               DoMouseCommand := gNoChanges;
END;


PROCEDURE TYourView.PrepareToTrack;

        PROCEDURE PrepareItem(item: TItem);

        VAR    r: Rect;

        BEGIN
               IF item.fIsSelected THEN BEGIN
                      r := item.fExtentRect;
                      InsetRect(r, -2, -2);
                      fYourView.InvalidRect(r);
               END;
               WITH item DO
                      fWasSelected := fIsSelected;
        END;

BEGIN
        fYourDocument.fItemList.Each(PrepareItem);
        fDragging := TRUE;
        fYourView.Update;
END;
```

```
PROCEDURE TYourDragger.IDragger(ItsDocument: TYourDocument;
                             itsView: TYourView; shiftKey: BOOLEAN);

BEGIN
       { Call ICommand to set the command's fView to the view in which tracking
         takes place and to set the scroller used for automatic scrolling during
         selection.  cMoveItem is the command number constant for this command.
         After calling ICommand it is necessary to set fCausesChange and fCanUndo
         to true, as dragging an object both changes a document and is undoable. }

       ICommand(cMoveItem, itsView, itsView.GetScroller(TRUE));
       fCausesChange := TRUE;
       fCanUndo := TRUE;
       fYourDocument := itsDocument;
END;


FUNCTION TYourDragger.TrackMouse(aTrackPhase: TrackPhase;
                                 VAR anchorPoint, previousPoint, nextPoint: VPoint;
                                 mouseDidMove: BOOLEAN): TCommand;

BEGIN
       TrackMouse := SELF;
       IF aTrackPhase = trackMove THEN BEGIN
              IF NOT fYourView.fDragging THEN BEGIN  { This is the first move. }
                     fYourView.DoHighlightSelection(hlOn, hlOff);
                     fYourView.PrepareToTrack;
                     IF fYourView.Focus THEN ; { PrepareToTrack changes the Focus }
              END;
       END
       ELSE IF aTrackPhase = trackRelease THEN BEGIN  { Set up for moving the items(s). }
              IF fYourView.fDragging THEN BEGIN  { Actually did move. }
                     fDeltaH := previousPoint.h - anchorPoint.h;
                     fDeltaV := previousPoint.v - anchorPoint.v;
                     fYourView.fDragging := FALSE;
              END
              ELSE
                     TrackMouse := gNoChanges;
       END
END;


PROCEDURE TYourDragger.DoIt;
BEGIN
       MoveBy(TRUE);
END;


PROCEDURE TYourDragger.UndoIt;
BEGIN
       FixSelection;
       MoveBy(FALSE);
END;
```

```
PROCEDURE TYourDragger.RedoIt;
BEGIN
        FixSelection;
        MoveBy(TRUE);
END;


PROCEDURE TYourDragger.MoveBy(moveIt: BOOLEAN);

        PROCEDURE MoveItem(item: TItem);
        BEGIN
                IF item.fIsSelected THEN BEGIN
                        { Invalidate the item's old image. }
                        { Move the item's definition. }
                        { Invalidate the item's new position. }



BEGIN
        fYourDocument.fItemList.Each(MoveItem);
END;


PROCEDURE TYourDragger.FixSelection;

        PROCEDURE FixItem(item: TItem);
        BEGIN
                item.fIsSelected := item.fWasSelected;
                IF item.fIsSelected THEN
                        { Invalidate the item in the view. }
        END;

BEGIN
        fYourDocument.Deselect; { This method removes the selection. You should implement
                                  it so that it removes all selections and updates the
                                  view, either by calling DoHighlightSelection(hlOn,hlOff)
                                  or by invalidating the selected areas of the view. }
        fYourDocument.fItemList.Each(FixItem);
END;
```

Drawing with the mouse

Purpose

Many applications allow the user to draw using the mouse. This recipe shows how to implement that operation.

For simplicity, this recipe assumes that a mouse press indicates the user wants to draw or the mouse press has no meaning. In general, a mouse press may indicate a number of possible actions, and your program uses a number of criteria to figure out which action the user wants. See the "Handling Several Types of Mouse Events" recipe for an example of integrating different types of mouse actions.

How to do it

1. When DoMouseCommand detects that a drawing operation has started, it should create a sketcher object instance, because drawing changes the document. You should create command object instances in two situations: when the command is undoable and when the command requires abilities of command objects, such as mouse tracking. The templates give the structure of DoMouseCommand.

2. Use a sketcher command object to track the mouse, to provide appropriate feedback as the mouse moves, and when the mouse button comes up and a valid item has been drawn, to add the new item to the document. Here is a sample interface for a sketcher type:

```
TYourSketcher = OBJECT(TCommand);

        fYourView: TYourView;
        fItem: TItem; { The new item. }

        PROCEDURE TYourSketcher.IYourSketcher(document: TYourDocument;
                                              view: TYourView);

        FUNCTION TYourSketcher.TrackMouse(aTrackPhase: TrackPhase;
                                          VAR anchorPoint,
                                          previousPoint, nextPoint: VPoint;
                                          mouseDidMove: BOOLEAN): TCommand; OVERRIDE;

        PROCEDURE TYourSketcher.DoIt; OVERRIDE;

        PROCEDURE TYourSketcher.UndoIt; OVERRIDE;

        PROCEDURE TYourSketcher.RedoIt; OVERRIDE;

END;
```

If you want to give feedback other than the standard flickering rectangle (which you will usually want to do), also override TrackFeedback. If you want to constrain the mouse—to stay in the bounds of the view, to draw a circle or a square, or to conform to a grid, for example—also override TrackConstrain. TrackFeedback and TrackConstrain are discussed in the "Tracking the Mouse" recipe.

The templates give the structure of TrackMouse, DoIt, UndoIt, and RedoIt.

3. If you want to, you can continue drawing when the mouse button comes up. This is done, for example, when a polygon is drawn in MacDraw or MacPaint. See the recipe, "Tracking the Mouse When the Mouse Button Is Up."

Template

```
FUNCTION TYourView.DoMouseCommand (VAR theMouse: Point; VAR info: EventInfo;
                                VAR hysteresis: Point): TCommand;

VAR    sketcher: TYourSketcher;

BEGIN
       New(sketcher);
       FailNil(sketcher);
       sketcher.ISketcher(fYourDocument, SELF);
       DoMouseCommand := sketcher;
END;


PROCEDURE TYourSketcher.ISketcher(ItsDocument: TYourDocument;
                               itsView: TYourView);

BEGIN
       { Call ICommand to set the command's fView to the view in which tracking
         takes place and to set the scroller used for automatic scrolling during
         sketching.  cNewItem is the command number constant for this command.  After
         calling ICommand it is necessary to set fCausesChange and fCanUndo to true,
         as dragging an object both changes a document and is undoable. }

       ICommand(cNewItem, itsView, itsView.GetScroller(TRUE));
       fCausesChange := TRUE;
       fCanUndo := TRUE;
       fYourDocument := itsDocument;
END;
```

```
FUNCTION TYourSketcher.TrackMouse(aTrackPhase: TrackPhase;
                                  VAR anchorPoint, previousPoint, nextPoint: VPoint;
                                  mouseDidMove: BOOLEAN): TCommand;

VAR     anItem: TItem;

BEGIN
        TrackMouse := SELF;
        IF aTrackPhase = trackRelease THEN
                IF {not a legal item} THEN
                        TrackMouse := gNoChanges
                ELSE BEGIN
                        New(anItem);
                        fItem := anItem;
                        { You can't use fItem in New because the heap might compact. }

                        { Extract the information you need from the anchorPoint
                          and nextPoint and initialize the new item. }
                END;
END;


PROCEDURE TYourSketcher.DoIt
BEGIN
        fYourDocument.fItemList.InsertFirst(fItem);
END;


PROCEDURE TYourSketcher.UndoIt;
BEGIN
        fYourDocument.fItemList.Delete(fItemList.First);
END;


PROCEDURE TYourSketcher.RedoIt;
BEGIN
        fYourDocument.fItemList.InsertFirst(fItem);
END;
```

Tracking the mouse

Purpose

After you've created a command object called yourMouseCommand and returned it through DoMouseCommand (see the discussion at the beginning of this section), MacApp tracks the mouse, calling yourMouseCommand.TrackMouse repeatedly.

TrackMouse has a parameter aTrackPhase. MacApp calls TrackMouse once with a track phase of trackPress, then calls repeatedly with a track phase of trackMove, and then calls once with a track phase of trackRelease.

If you do not want to take any action depending on the track phase and the action of the mouse command changes the document, you do not have to override TrackMouse.

You generally do have to override TrackMouse. If you want to take some other action depending on the trackPhase or the mouse location, override TrackMouse. In addition, if the command may not change the document, override TrackMouse. The default version of TrackMouse returns the command object itself as the function return value, which results in always marking the document as changed after the mouse button is released.

If you want to give nonstandard feedback as the mouse moves, override TrackFeedback, as described below. If you want to constrain mouse movement in some way, override TrackConstrain, also described below.

How to do it

1. Add the following to your mouse command object type definition:

```
FUNCTION TYourMouseCommand.TrackMouse(aTrackPhase: TrackPhase;
                                  VAR anchorPoint, previousPoint, nextPoint: VPoint;
                                  mouseDidMove: BOOLEAN): TCommand; OVERRIDE;
```

   In your implementation of TrackMouse, you should return SELF so that MacApp continues to call yourMouseCommand.TrackMouse. You can also return another command object, in which case that command object takes over tracking the object. (MacApp frees the old command object for you.) On trackRelease, if no changes have been made to the document, you can return gNoChanges, which tells MacApp to free the command object. It also tells MacApp to not commit and free the last command object. (If gNoChanges is not returned, MacApp automatically calls Commit for the last command and frees that command. The result is that the last command can no longer be undone, which may not be appropriate.)

   No template is given for TrackMouse because its form varies greatly. See the other recipes of this section and the sample programs for samples.

2. If you return a command object, MacApp calls yourMouseCommand.DoIt when the mouse is released. If the command can be undone, or if it changes the document, you normally perform the action of the mouse command in DoIt. If the command cannot be undone, and if it does not change the

document, you can perform the action in TrackMouse when the track phase is trackRelease. In that case, return gNoChanges instead of your own command object. No template is given for TYourMouseCommand.DoIt.

3. MacApp calls the method yourMouseCommand.TrackFeedback as the mouse moves. TCommand.TrackFeedback produces a shadowy (black pen, XOR mode) box between the point where the mouse button was pressed and the current mouse position. If you want different feedback, add the following to your definition of TYourMouseCommand:

```
PROCEDURE TYourMouseCommand.TrackFeedback(anchorPoint, nextPoint: VPoint;
                                          turnItOn, mouseDidMove: BOOLEAN);
                                          OVERRIDE;
```

You can, for example, change the pen state or mode and then call INHERITED TrackFeedback, or you can provide completely different feedback.

No template is given for this method.

4. If you want to constrain mouse movement in some way, as is done when the grid is on in MacPaint:

- Set yourMouseCommand.fConstrainsMouse to TRUE. (You can do that in IYourMouseCommand.) fConstrainsMouse is a field of TCommand. fConstrainsMouse defaults to FALSE. When that field is TRUE, MacApp calls yourMouseCommand.TrackConstrain.

- Override TCommand.TrackConstrain. Here is the interface for that method:

```
PROCEDURE TYourMouseCommand.TrackConstrain(anchorPoint, previousPoint: VPoint;
                                           VAR nextPoint: VPoint); OVERRIDE;
```

This is called only if fConstrainsMouse is TRUE. In your implementation, change the value of nextPoint according to your program's requirements. No template is given for this method. See the sample programs for examples.

Handling several types of mouse events

Purpose

The preceding recipes in this section assume that only one type of mouse event is possible. Few applications are so limited. In general, your aView.DoMouseCommand method must differentiate between possible types of events and take appropriate action.

There are two basic ways to differentiate between possible mouse events: based on mode and based on location. Programs generally use a combination of these methods. For example, the DrawShapes sample program has two modes: when the arrow pointer is displayed and when a drawing pointer is displayed. In the arrow pointer mode you can select individual shapes, select an area, and drag shapes, and the program determines which you want to do basically by where the mouse button went down.

When one of your application's view.DoMouseCommand methods is called, indicating a mouse-down event in one of your application's views, the application must determine what kind of action is beginning and (generally) it must create an appropriate type of command object, which then tracks the mouse and carries out the action of the command. This recipe generally covers the needed steps up to the point of creating a command object for the mouse command. See the individual recipes in this section for details on implementing those command objects.

How to do it

1. Implement DoMouseCommand for each view type that needs to respond to a mouse-down event. DoMouseCommand is a function that returns a TCommand-type object. The interface for doMouseCommand is

```
FUNCTION TYourView.DoMouseCommand(VAR theMouse: Point; VAR info: EventInfo;
                                  VAR hysteresis: Point): TCommand; OVERRIDE;
```

   A sample skeleton for DoMouseCommand is given in the template for this recipe. That sample is very sketchy because the form of DoMouseCommand depends on what your particular application does.

2. Your DoMouseCommand method must first determine if the user made a selection or is indicating some other action.

   The sample in the template then checks for certain conditions and creates an appropriate command object depending on the conditions.

3. The details of what happens once you have determined the type of action needed are not given in this recipe. See the first five recipes of this section for details.

   YourView.DoMouseCommand often creates a mouse command object. There may be several types of mouse command objects. If the event is handled entirely by DoMouseCommand (which should be the case only for mouse events that do not change the document), or if the event does not produce an action, your view's DoMouseCommand method should return gNoChanges, a global variable of type TCommand that indicates no changes to the document have occurred. (You can also return gNoChanges later, if it turns out that no changes have been made. See the discussion of TrackMouse in the "Tracking the Mouse" recipe.)

Command objects returned through DoMouseCommand are expected to have different methods than other command objects. The "Tracking the Mouse" recipe explains what is required of those methods.

Template

```
FUNCTION TYourView.DoMouseCommand(VAR theMouse: Point; VAR info: EventInfo;
                              VAR hysteresis: Point): TCommand;

VAR     firstMouseCommand: TFirstMouseCommand;
        secondMouseCommand: TSecondMouseCommand;

BEGIN
        DoMouseCommand := gNoChanges;  { in case no action found that changes the document
}

        { Check for selections here. See "Selections" in this chapter. }

        IF { the action indicates a firstMouseCommand } THEN BEGIN
               New(firstMouseCommand);
               FailNIL(firstMouseCommand);
               firstMouseCommand.IFirstMouseCommand(SELF, theMouse);
               { Those parameters are only an example. }
               DoMouseCommand := firstMouseCommand;
        END
        ELSE IF { the action indicates a secondMouseCommand } THEN BEGIN
               New(secondMouseCommand);
               FailNIL(secondMouseCommand);
               secondMouseCommand.ISecondMouseCommand(SELF, theMouse);
               { parameters only examples }
               DoMouseCommand := secondMouseCommand;
        END;
END;
```

Tracking the mouse when the mouse button is up

Purpose

Some applications must occasionally track the mouse and possibly provide feedback when the mouse button is up. An example of this occurs in MacDraw, when you draw a polygon: you mark the end of the first side of the polygon by letting the mouse button up and draw the second side with the button up. The second side is marked when the mouse button goes down again.

You track the mouse when the mouse button is down with DoMouseCommand and TrackMouse, as described in the "Tracking the Mouse" recipe. MacApp does not call either of these methods when the mouse button is up. This recipe describes what you have to do to track the mouse with the button up.

How to do it

1. Override DoSetCursor. The interface to DoSetCursor is

```
FUNCTION TYourView.DoSetCursor(localPoint: Point;
                               cursorRgn: RgnHandle): BOOLEAN; OVERRIDE;
```

   DoSetCursor for the view that contains the mouse is called repeatedly during idle time, that is, when the user is doing nothing but moving the mouse. The default version of DoSetCursor contains only one line of code:

```
DoSetCursor := FALSE;
```

   This line simply informs MacApp that the pointer should be the arrow pointer. To track the mouse, you need to add your tracking and feedback functions to this method.

2. Implement DoMouseCommand so it recognizes that you were tracking the mouse while the mouse button was up and takes appropriate action. You can add a field to your view that keeps track of this. The interface of DoMouseCommand is

```
FUNCTION TYourView.DoMouseCommand(VAR theMouse: Point; VAR info: EventInfo;
                              VAR hysteresis: Point): TCommand; OVERRIDE;
```

Standard editing commands

Undo

Purpose

The Undo menu command should be implemented for any user action that changes the document. In other words, it is not usually desirable to implement Undo for actions like scrolling or selections that do not actually change the data, but you should implement Undo for actions like adding or deleting objects from the document's data set.

The Undo menu command is automatically enabled by MacApp when there is a command object that has had its DoIt method executed and has its fCanUndo field set to TRUE and has not been superceded by another command object. (Note that MacApp cannot tell if the command object actually has an implemented UndoIt or RedoIt method.) As long as you return gNoChanges from DoMenuCommand and from TrackMouse when the track phase is trackRelease, the Undo command remains enabled for the last undoable command. When a different type of command object is returned, MacApp calls command.Commit for the previous command object (unless that command was undone and not redone) and enables or disables Undo depending on whether the new command object can be undone. If, however, the new command object has both fCanUndo and fChangesDocument equal to FALSE, MacApp does not commit the previous command. Instead, it simply calls the DoIt method of the new command.

How to do it

The Undo command is handled by the current command object. Whenever a user action (a mouse action, a menu choice, or typing) will change the document, a command object should be created. (See "Handling Mouse Events" and "Menus and Commands" in this chapter.) When the command is initially executed, MacApp calls the command object's DoIt method. When the user chooses Undo the first time (or any odd number of times), MacApp calls the command object's UndoIt method. When Undo is chosen a second time (or any even number of times), MacApp calls the command object's RedoIt method.

If the command is simple, you normally change the document's data with DoIt, UndoIt, and RedoIt, and these methods invalidate any affected portions of the view or views.

If the command has results that are too complicated to undo directly, DoIt and RedoIt apply a filter that makes the view appear as if the data had actually been changed, but do not actually change the data. UndoIt simply removes the filter. (All three methods must still invalidate the changed parts of the view. When a filter is applied, it is usually implemented with a flag that indicates a filtering method should be called from the drawing methods, which are themselves called during the update cycle.) To change the document's data, override TCommand.Commit so that Commit changes the data. Commit is called before the command is freed, usually just before another command object is created or when the document is saved. (Note that Commit is not called if the command was in undo phase.)

The default TCommand methods Commit, DoIt, RedoIt, and UndoIt do nothing.

See the "Creating Filtered Commands" recipe for more information about filtering.

Cut and Copy

Purpose

The Cut, Copy, and Paste commands should be implemented in all Macintosh applications to allow transfer of data among and within applications and desk accessories. In order for this recipe to work, you must also implement the recipe under "The Clipboard," below.

This recipe deals with the Cut and Copy commands, which are generally handled by a single type of command object. The next recipe deals with the Paste command.

The Cut command removes the selected information from the view (and generally also from the document) and places the information in the Clipboard. The Copy command copies the selected information to the Clipboard but does not remove the original.

How to do it

1. In the appropriate DoMenuCommand method (usually belonging to the view but possibly to the document), create a cut/copy command object of a type that is a descendant of TCommand. (Some programs may need separate

command objects for cut and copy, although generally a copy is identical to a cut except that the information is not removed from the document.)

2. In the IYourCommand method of your cut/copy command object, set the fChangesClipboard field to TRUE after calling ICommand.

3. In the DoIt method of your cut/copy command object, create a view for the cut or copied data. The view is typically of the same type as the one holding the selection and, again typically (but not universally), you must create a document object to go with the view object.

4. After you initialize this view, call TApplication.ClaimClipboard to install the view in the Clipboard. The interface for that method is

   ```
   PROCEDURE TApplication.ClaimClipboard(clipView: TView);
   ```

   ClaimClipboard automatically preserves a reference to the old Clipboard view, in case this command is undone.

   If this is a Cut command, cut the data from your document and invalidate the representation of the data in the view.

   You must not call ClaimClipboard in your UndoIt or RedoIt methods. MacApp automatically replaces the old Clipboard contents when Undo is picked and automatically replaces the new Clipboard when Redo is picked.

   In the case of a Copy command, UndoIt need do nothing except, if you wish, restore the selection state at the time the command was originally executed (MacApp restores the old Clipboard view for you). RedoIt needs to do everything DoIt does, except create the Clipboard view and call ClaimClipboard. It may also restore the last selection.

   No template is given for this recipe and the next, because the methods depend too much on application-specific conditions.

Paste

Purpose

The Cut, Copy, and Paste commands should be implemented in all Macintosh applications to allow transfer of data among and within applications and desk accessories.

The Paste command pastes data from the Clipboard into the application's document. The Clipboard may contain data cut or copied from your application or from another application. In the second case, the data is usually available as TEXT data (a string of ASCII characters) and/or PICT data (PICT is a QuickDraw picture).

How to do it

1. In the DoSetupMenus method for the object whose DoMenuCommand method handles Paste (usually the view but possibly the document), tell MacApp what kind of data you can paste. You do this by calling the global procedure CanPaste. The interface of that routine is

   ```
   PROCEDURE CanPaste(aDataType: ResType);
   ```

   Call this procedure once for each Clipboard data type you can handle. (See the "The Clipboard" recipe for more about Clipboard data types.) If you can paste more than one kind of data (you should, ideally, be able to handle PICT and TEXT data as well as your own types), make the calls in inverse order of preference: from the least preferred to the most preferred.

   Note that you never call Enable or EnableCheck for the Paste command. MacApp tests the contents of the Clipboard for the Clipboard data types you specify in your CanPaste calls (by calling clipboardView.ContainsClipType) and enables or disables the command accordingly.

2. Create a paste command object (discussed in the next step) when your DoMenuCommand method finds the command number cPaste. Given the CanPaste calls made in DoSetupMenus, you can be certain that information of some type you can handle is present in the Clipboard any time you get a cPaste command number.

3. Define a paste command object type that is a descendant of TCommand. The object should be created and initialized in DoMenuCommand when a cPaste command number is received. The action of the command is carried out in the pasteCommand.DoIt and RedoIt methods.

   To get the data to be pasted, allocate an empty handle and pass the handle to the application's GetDataToPaste method. The interface of this method is

   ```
   FUNCTION TApplication.GetDataToPaste(aDataHandle: Handle;
                                        VAR dataType: ResType): LONGINT;
   ```

If you only want to find out the size of the data (probably to determine whether there is enough memory to carry out the requested paste operation), pass NIL as aDataHandle. When the data is in the public scrap (also called the desk scrap), this call is equivalent to the Scrap Manager routine GetScrap. Do not call GetScrap directly, because the data may be in the private (application) scrap.

You do not choose the data type here; that is determined by your CanPaste calls in DoSetupMenus. The data type passed to you is the most preferred type available. If you can paste more than one type, you probably need to use IF statements to branch according to the type; note that MPW Pascal does not allow CASE statement branches on four-byte quantities.

The data referred to by the handle is a copy of the data in the Clipboard. You can do anything you want with that data or the handle.

GetDataToPaste (which you rarely need to override) calls the method gClipView.GivePasteData. See "The Clipboard" in this chapter for details on implementing that method.

Paste operations should almost always be undoable (see the "Undo" recipe, above). See "Menus and Commands," below, for more information on commands.

Menus and commands

Creating menu commands

Purpose

In general, all commands that are not mouse actions or typing are menu commands.

This section also applies to commands given using Command-key combinations, which are equivalent to menu commands, as defined in the resource file for the application.

How to do it

1. Add a cmnu resource to your application's resource compiler input file to add the menu command. When you put the command in the resource file, you give the command a command number. See the sample programs' resource files for examples of menu commands.

Important

Menu commands are defined differently for MacApp resource files than for the resource files of standard Macintosh applications. MacApp menu resources are defined as cmnu resources in the resource input file. The Build command file that builds MacApp programs runs the PostRez tool to convert the cmnu resources to MENU resources plus the additional information MacApp needs. Therefore, you cannot use a resource editor to add menus or menu items and you cannot use DeRez to decompile your menus.

---

2. In the implementation of your unit, define a constant for the command number you gave for the menu command in the resource file.

3. Override the appropriate DoMenuCommand method. If the command has the same effect regardless of which view of the document is active or which view contains the selection, then override TDocument.DoMenuCommand for your document. If the command is view-specific, override TView.DoMenuCommand for your view. Similarly, if the command applies to a particular window or the application as a whole, override the DoMenuCommand for your descendants of TWindow or TApplication.

   MacApp defines a global variable, gTarget, that refers to the event handler that initially receives menu commands and keystrokes. (Views, windows, documents, and applications are all event handlers.) MacApp also defines a field of TWindow calledfTarget. MacApp automatically sets gTarget to window.fTarget whenever the window is activated. The window's fTarget is set to itself in IWindow. NewSimpleWindow and NewTemplateWindow set fTarget to the window's main view.

   In addition, MacApp defines a field TEvtHandler.fNextHandler, which puts the event handlers in an application in a linked list.

   When MacApp receives a menu event, it passes it to gTarget.DoMenuCommand. If the target cannot handle the command, it calls INHERITED DoMenuCommand. That method, usually part of

MacApp, first checks whether the command is one it can handle, and then again calls INHERITED DoMenuCommand. This chain eventually leads to TEvtHandler.DoMenuCommand. That method calls fNextHandler.DoMenuCommand. The chain continues through the list until the application object is reached. At that point, there is no next event handler, and TEvtHandler.DoMenuCommand reports an error.

See "The Command Chain" in Chapter 5 for a more complete description of how the command chain works.

Here is the interface for TYourType.DoMenuCommand:

```
FUNCTION TYourType.DoMenuCommand(aCmdNumber: CmdNumber): TCommand; OVERRIDE;
```

The form for the implementation is given in the template part of this recipe.

The template is for the simplest DoMenuCommand method. In this case, an IF statement could have been used in place of a CASE statement to create the proper type of command object. Whatever structure you use, though, if the command number is not one of yours, you must call INHERITED DoMenuCommand so MacApp can handle its commands.

If the command changes the document, create a command object and pass the command object to MacApp as the return value of DoMenuCommand. If the command does not change the document, perform the command immediately and return gNoChanges.

4. If you return a command object, MacApp calls command.DoIt using the command object you return. You should override TCommand.DoIt to execute your command. If the command can be undone, you should also override TCommand.UndoIt and TCommand.RedoIt (see the "Undo" recipe).

Template

```
FUNCTION TYourType.DoMenuCommand(aCmdNumber: CmdNumber): TCommand;

BEGIN
      CASE aCmdNumber OF
            { Here give one of your command numbers. }: BEGIN
                  { Here create and initialize an appropriate command object or,
                    if there are no changes to the document, do the command. }
                  DoMenuCommand := { your command object or gNoChanges };
            END;
            OTHERWISE
                  DoMenuCommand := INHERITED DoMenuCommand(aCmdNumber)
      END;
END;
```

Changing menu appearance and function

Purpose

You need to change menu appearance and function to

- disable a menu command (draw the text in gray)
- enable a menu command (draw the text in black)
- add or remove a check mark (usually for a toggle command)
- change the text of a command (either for a toggle command such as Undo/Redo, or for a more variable command)
- add or remove a menu
- add or remove a menu command
- change the font style of a menu command

How to do it

1. Every application that defines its own menu commands must override DoSetupMenus. Whenever a mouse-down event is detected in the menu bar, DoSetupMenus is called for the application, document, window, and view before the menus are displayed. You can override the DoSetupMenus methods for any of these to change the text for any menu item or to enable, disable, or check menu items. You must override DoSetupMenus for any object type for which you override DoMenuCommand.

◆ *Note:* MacApp actually calls DoSetupMenus only when some change has occurred. MacApp calls it after processing all available events, so it is usually not called when the user clicks in the menu bar. Therefore, the user usually does not have to wait for DoSetupMenus to execute before seeing a menu.

Begin your override method with INHERITED DoSetupMenus. You then call the MacApp and Menu Manager routines described in the rest of this recipe to change the appearance of the menus.

Although MacApp has global procedures for the most common menu operations, you must use Menu Manager routines for much of what is described in this recipe. Menu Manager routines use menu handles, menu ID's, and item ID's to refer to menus and commands. Convert the command number to a menu handle and item number using the following MacApp global procedure:

```
PROCEDURE CmdToMenuItem(aCmd: CmdNumber; VAR menu, item: INTEGER);
```

This procedure returns the Menu Manager menu and item ID associated with the given command number. If you need a menu handle (which you generally need for Menu Manager routines) use the following MacApp global function:

```
FUNCTION GetResMenu(menuID: INTEGER): MenuHandle;
```

The Menu Manager contains routines that are not discussed here because they are rarely used in MacApp programs. See the "Menu Manager" chapter of *Inside Macintosh* for complete information.

2. For each command number you handle in DoMenuCommand, call either Enable or EnableCheck in a version of DoSetupMenus defined for the same object type. You must enable all commands that you want the user to be able to choose, even if the status of the command hasn't changed since the last time DoSetupMenus was called, because all menu items start out unchecked and disabled.

   Enabling a command draws the command name in black; disabling it draws the name in gray. You enable and disable commands that never have check marks with this procedure:

   ```
   PROCEDURE Enable(aCmd: CmdNumber; canDo: BOOLEAN)
   ```

   This procedure enables or disables the given command depending on the value of the parameter canDo. If canDo is FALSE, the command is disabled and is displayed in gray. Since commands are always disabled before calling DoSetUpMenus, it is only necessary to enable commands.

If a command may have a check mark, use this procedure:

```
PROCEDURE EnableCheck(aCmd: CmdNumber; canDo: BOOLEAN; checkIt: BOOLEAN)
```

EnableCheck places or removes a check mark next to the menu item, depending on the value of checkIt. It also draws the text in gray or black, depending on the value of canDo.

You do not use Enable or EnableCheck to enable the Paste command. Instead, use

```
PROCEDURE CanPaste(aDataType: ResType)
```

This procedure tells MacApp what data types you can paste at this point. Call it once for each data type you can handle, in inverse order of preference. MacApp checks the contents of the Clipboard and enables the Paste command if pasting is possible. See the "Paste" recipe for more information.

3. If you want to change the text of a menu item, you should use the following routine:

```
PROCEDURE SetCmdName(aCmd: CmdNumber; menuText: Str255);
```

This routine changes the text of the menu item with command number aCmd to menuText.

Important

You must never use Menu Manager routines directly in DoSetupMenus to take the actions performed in steps 3, 4, and 5 of this recipe.

4. If you want to change the font style of a menu command, printing it in bold, italic, subscript, superscript, condensed, or expanded, or returning it to plain text, use the following MacApp global procedure:

```
PROCEDURE SetStyle(aCmd: CmdNumber; aStyle: Style)
```

This is typically used only for the menu items that change font style.

5. Some menus have icons displayed to the left of the item text. If you want to set such an icon, use the following MacApp global procedure:

```
PROCEDURE SetCmdIcon(aCmd: CmdNumber; menuIcon: Byte);
```

This procedure changes the icon shown in the menu for the menu item with command number aCmd to the icon represented by menuIcon.

Handling negative command numbers

Purpose

When you have a menu command that cannot be assigned a command number when you write your application or that does not fit into the normal menu structure, your DoMenuCommand method receives a negative command number. This happens if you have a custom menu with commands depicted as icons, if you add menu items using Menu Manager routines, or if menu items cannot be determined until runtime. It happens most commonly with the Font menu, which always returns negative command numbers because the number of fonts cannot be predetermined.

How to do it

1. Implement DoMenuCommand for the appropriate target, which depends on whether you want the command to affect one view, one window, one document, or the entire application. When you have a negative command number, you have two choices:

   - Make a case statement directly on the negative values. The values are equal to –(256 * menu + item).

   - Call CmdToMenuItem to convert the number to the menu ID and item ID for the item the user picked. Then take action depending on those values.

   A sample DoMenuCommand is shown in the templates for this recipe. Note that the sample handles only negative command numbers. See the "Creating Menu Commands" recipe for more general information about DoMenuCommand.

2. Implement DoSetupMenus for the same target so that it handles the menus and menu items that return negative command numbers. As with ordinary menu items, you must explicitly enable all enabled items and check items that have checks. (All items start out disabled and unchecked.) There are several possibilities, depending on your application. You can use Menu Manager routines to enable or check these custom menu items. However, to change the text, style, or icon of a custom menu item you must call the

routines discussed in the previous section, passing the appropriate negative command number.

If you have menus (such as the Font menu) in which all items return negative numbers, use code that follows the pattern shown in the templates.

If you have menus that may include negative command numbers because menu items are added by calls to Menu Manager routines while the application is running, use the Menu Manager function CountItems (used in the template for DoSetupMenus in this recipe) to find out how many items are actually in the menu. Then, if there are menu items that return negative numbers, set up those items in DoSetupMenus and handle those items in DoMenuCommand the same way as shown in the template.

3. If you are implementing a Font menu, you need to use the menu ID and item ID in DoMenuCommand to get the font number. The font number is used in calls to SetFont, a QuickDraw procedure. To find the font number, use the following code sequence:

```
CmdToMenuItem(aCmdNumber, menu, item);       { MacApp global procedure }
IF menu = mFont THEN BEGIN
    GetItem(GetResMenu(menu), item, aName);  { Menu Manager procedure }
    GetFNum(aName, theFontNumber);           { Font Manager procedure }
END;
```

The value of mFont (a constant you should define) depends on the order of your menus. The variable aName, returned by GetItem, is a value of type Str255. The font number is an INTEGER. You should store the number somewhere and use it to set the font whenever you draw text in that font. Also store the menu item corresponding to the currently selected font, for use in DoSetUpMenus. Note that your drawing methods should never assume that the font (or any other characteristic, for that matter) has been set. If you care what the font is, always set it yourself.

Templates

```
FUNCTION TTarget.DoMenuCommand(aCmdNumber: INTEGER):TCommand;

VAR     menu, item: INTEGER;

BEGIN
        IF aCmdNumber < 0 THEN BEGIN
                CmdToMenuItem(aCmdNumber, menu, item);
                { Take action depending on the menu and item values. }
                DoMenuCommand := {a command object or gNoChanges }
        END
        ELSE
                DoMenuCommand := INHERITED DoMenuCommand(aCmdNumber);
END;


PROCEDURE TTarget.DoSetupMenus;

VAR     item: INTEGER;
        aMenuHandle: MenuHandle; { a Menu Manager type }

BEGIN
        { All procedure and function calls are to Menu Manager routines. }
        INHERITED DoSetupMenus;
        aMenuHandle := GetMHandle(mNumber1);
        { mNumber1 is a constant you define. It is the menu ID for a menu that
          only returns negative command numbers. }

        IF aMenuHandle <> NIL THEN
                FOR item := 1 TO CountMItems(aMenuHandle) DO BEGIN
                        EnableItem(aMenuHandle, item); { or use DisableItem }
                        { If this is a font menu, and the menu item corresponding
                          to the currently selected font is stored in fCurrFontItem, add: }
                        CheckItem(aMenuHandle, item, item = fCurrFontItem);
                END;

        aMenuHandle := GetMHandle(mNumber2);
        { mNumber2 is a constant you define. It is the menu ID for a menu that may have
          menu items added. The constant cRegularItems, used below, is another constant
          you define which defines the number of permanent items in this menu. It is
          assumed here that those menu items are handled by ordinary command numbers.
          Handle the setup for the ordinary menu items in the menu here or elsewhere in
          this method. See the "Changing Menu Appearance and Function" recipe for more
          information. }

        IF CountMItems(aMenuHandle) > cRegularItems THEN
                FOR item := (cRegularItems + 1) TO CountMItems(aMenuHandle) DO BEGIN
                        EnableItem(aMenuHandle, item); { or use DisableItem }
                END;
END;
```

Creating filtered commands

Purpose

With commands that make large or complex changes to a document, it may be inefficient to actually make the changes when you may have to undo them later. Instead, you may apply a filter to the view. Conceptually, a filter makes the view appear as if the data has actually changed, when in reality the data set remains as it was. That way, if the user chooses Undo you simply remove the filter, and if the user chooses Redo you apply the filter again. You don't actually change the data (commit the command) until the command can no longer be undone.

How to do it

1. You must somehow record which items in the document's data set were changed by the command. If the items are separate objects, this is usually done with a Boolean flag in each object, although some applications maintain a separate list of changed items, probably as part of the view or in the command object. In addition, you need a flag that tells whether or not the command is currently in effect.

2. In DoIt, mark the changed items and set the flag indicating that the changes are in effect. Then invalidate the images of the changed items in the view. In the UndoIt and RedoIt methods, set the flag that indicates whether or not the command is in effect and invalidate the items' images.

3. In the Draw and DoHighlightSelection methods, check the flags (or list of changed items) and appropriately alter the way the data is displayed. It is easiest to see how this is done if the command deletes selected items. In that case, you can simply not draw any items that were selected when the command was initially executed. In more complex cases, you may call a separate drawing method, possibly part of the command object, that draws the changed items.

4. In Commit, make the actual changes to your document's data set. In the example of deleting selected items, you can actually delete the corresponding objects.

Because the actual implementation of filtered commands varies significantly between applications, no recipe is given here. However, the DrawShapes sample program uses filtered commands for its Cut, Copy and Paste commands.

Using UPrinting

Purpose

The printing unit, UPrinting, provides standard printing capability that is sufficient for most applications. This recipe shows how you can use the unit in the simplest possible way. See the sample programs for examples of more extensive use of the capabilities of UPrinting, and the Printing ERS for further discussion of MacApp printing.

How to do it

1. You need to include UPrinting in the USES statement at the beginning of your unit.
2. Define a new local variable, aStdHandler, of the type TStdPrintHandler, for your TYourDocument.DoMakeViews method. (Alternatively, you can define this variable in TYourView.IYourView.)
3. Insert the lines shown in the template at the end of TYourDocument.DoMakeViews. (You can also do this in TYourView.IYourView.)
4. Insert the following line in your main program before calling application.Run:

   ```
   InitPrinting;
   ```

   That initializes UPrinting.

Template

```
{ The next two lines make the view printable. }
New(aStdHandler);
FailNIL(aStdHandler);
aStdHandler.IStdPrintHandler(SELF, aView, FALSE, TRUE, TRUE);
```

Using UTEView

Purpose

The text-editing unit, UTEView, implements more than the text-editing features of the toolbox TextEdit package. Using this unit, you can have simple text editing of series as long as 32,767 characters. TextEdit capabilities include the following:

- inserting new text
- deleting characters that are erased with backspacing
- translating mouse activity into text selection
- implementing the Cut, Copy, Clear and Paste commands and Clipboard support
- ability to undo typing and the Cut, Copy, Clear and Paste commands

See the "TextEdit "chapter of *Inside Macintosh* for details of TextEdit's actions.

This recipe shows essentially how to implement a limited version of the DemoText sample program. You may want to build and run that program to get a better idea of what UTEView can do for you. Also, you can find more information about the UTEView unit in the "MacApp 2.0 TTEView ERS."

How to do it

1. Include UTEView in the USES statement at the beginning of your unit.
2. As with any application, you must create your own descendant of TApplication and override DoMakeDocument for that type. Here is a sample interface:

```
TYourApplication = OBJECT(TApplication)
   ...
   FUNCTION TYourApp.DoMakeDocument(itsCmdNumber: CmdNumber): TDocument; OVERRIDE;
END;
```

   The implementation of DoMakeDocument is similar to any DoMakeDocument method. A sample is in the template for this recipe.

3. Create your own document type. It must have certain fields to hook into TextEdit. Here is a sample interface:

```
TTextDocument = OBJECT(TDocument)

    fText: Handle; {handle to the actual text belonging to the document}
    fTEView: TTEView; {the TEView object that manages the text}

    PROCEDURE TTextDocument.ITextDocument;

    PROCEDURE TTextDocument.Free; OVERRIDE;

    PROCEDURE TTextDocument.FreeData; OVERRIDE;

    PROCEDURE TTextDocument.DoNeedDiskSpace(VAR dataForkBytes,
                                             rsrcForkBytes: LONGINT); OVERRIDE;

    PROCEDURE TTextDocument.DoRead(aRefNum: INTEGER;
                                   rsrcExists, forPrinting: BOOLEAN); OVERRIDE;

    PROCEDURE TTextDocument.DoWrite(aRefNum: INTEGER; makingCopy: BOOLEAN); OVERRIDE;

    PROCEDURE TTextDocument.DoMakeViews(forPrinting: BOOLEAN); OVERRIDE;

    PROCEDURE TTextDocument.DoMakeWindows; OVERRIDE;

END;
```

The implementation of these methods is discussed in the rest of this recipe.

4. Create a new handle for the text with the IDocument method. You do that with the MPW Pascal function NewHandle. At this time, fTEView, which will later hold a reference to the TTEView object that handles the text, is set to NIL.

5. Provide a FreeData method to get rid of the document's data when the document is reinitialized. You can do that by just setting the handle size to 0.

6. Provide a Free method. Call DisposHandle(fText) and then call INHERITED Free.

7. Implement DoMakeViews so that it makes a TTEView object.

   The view object is central to a UTEView application, because the text-edit view is what handles the text by communicating with the toolbox TextEdit package. The method's implementation in the templates is self-explanatory.

Notice that DoMakeViews creates a print handler and thus can be printed. (UPrinting must also be in a USES statement for printing to work. See the "Using UPrinting" recipe in this chapter.)

8. Implement DoMakeWindows. The implementation is shown in the template.

9. Implement DoNeedDiskSpace, DoRead, and DoWrite so that you can save and restore documents. See the sample implementations in the template. Notice the failure handler used, HdlDoRead. See the "Failure Handling" recipe for more information.

Templates

```
FUNCTION TYourApplication.DoMakeDocument(itsCmdNumber: CmdNumber): TDocument;

VAR     aTextDocument: TTextDocument;

BEGIN
        New(aTextDocument);
        FailNIL(aTextDocument);
        aTextDocument.ITextDocument;
        DoMakeDocument := aTextDocument;
END;


PROCEDURE TTextDocument.ITextDocument;

BEGIN
        fText := NIL;
        IDocument(kFileType, kSignature, kUsesDataFork, NOT kUsesRsrcFork,
                    NOT kDataOpen, NOT kRsrcOpen);
        fText := NewPermHandle(0);
        FailNIL(fText);
        fTEView := NIL;
END;


PROCEDURE TTextDocument.Free; OVERRIDE;

BEGIN
        IF fText <> NIL THEN
               DisposHandle(fText);
        INHERITED Free;
END;


PROCEDURE TTextDocument.FreeData; OVERRIDE;

BEGIN
        SetHandleSize(fText, 0);
END;
```

```
PROCEDURE TTextDocument.DoMakeViews(forPrinting: BOOLEAN); OVERRIDE;

VAR     aWindow:  TWindow;
        aHandler: TStdPrintHandler;
        aTEView; TTEView;

BEGIN
        aWindow := NewTemplateWindow(kWindowRsrcID, SELF);

        aTEView := TTEView(aWindow.FindSubView('TEVW');
        fTEView := aTEView;

        New(aStdHandler);
        FailNIL(aStdHandler);
        aStdHandler.IStdPrintHandler(SELF, aTEView, FALSE, TRUE, TRUE);

        fTEView.StuffText(fText);     { Put in the text. }
END;


PROCEDURE TTextDocument.DoNeedDiskSpace(VAR dataForkBytes, rsrcForkBytes: LONGINT);

BEGIN
        INHERITED DoNeedDiskSpace(dataForkBytes, rsrcForkBytes);
        dataForkBytes := dataForkBytes + GetHandleSize(fText);
END;


PROCEDURE TTextDocument.DoRead(aRefNum: INTEGER; rsrcExists, forPrinting: BOOLEAN);

VAR     numChars: LONGINT;
        fi: FailInfo;

        PROCEDURE HdlDoRead(error: INTEGER; message: LONGINT);
        BEGIN
                SetHandleSize(fText, 0);
                Failure(error, message);
        END;

BEGIN
        CatchFailures(fi, HdlDoRead);
        FailOSErr(GetEOF(aRefNum, numChars));
        SetHandleSize(fText, numChars);
        FailMemError;
        FailOSErr(FSRead(aRefNum, numChars, fText^));
        Success(fi);
END;
```

```
PROCEDURE TTextDocument.DoWrite(aRefNum: INTEGER; makingCopy: BOOLEAN);

VAR     numChars: LONGINT;

BEGIN
        numChars := GetHandleSize(fText);
        FailOSErr(FSWrite(aRefNum, numChars, fText^));
END;
```

Using UDialog

The dialog unit, UDialog, provides an extensive group of capabilities for creating dialog boxes and other windows made up of groups of controls (such as radio buttons) and other items that can request information from the user. To do this, the dialog unit makes extensive use of nested views.

The dialog unit provides support for these features:

- views that can take in text and validate the text
- views that can take in numbers and validate the numbers, called **number text items**
- views that represent Control manager controls: radio buttons, push buttons, and check boxes
- views that represent pictures, icons, and static text—all of which can behave like buttons, if desired
- groups of radio buttons, called radio **clusters**
- modal dialog boxes
- modeless dialog boxes

The recipes in this section show how to create simple dialog boxes. See the sample programs, the "MacApp 2.0 UDialog ERS", and the source code for further information. In particular, you may find the DemoDialogs sample program helpful.

Creating a modeless dialog

Purpose

A modeless dialog is similar to an ordinary window, except that the dialog's main view object is an instance of TDialogView and its view is an instance of a TDialogView. Modeless dialog boxes usually exist to request some sort of information from the user, and any information they convey to the user is generally simple. Modeless dialog boxes can be deactivated just like ordinary windows, so the user does not have to respond to them before continuing work with the application.

How to do it

1. Include UDialog in the USES statement at the beginning of your unit.

2. Define a method that will display the dialog. This method should be for the object type that will issue the command. If the dialog has to do with the operation of the application as a whole, the method should belong to TYourApplication, and similarly with your document or view. (Because you rarely customize TWindow, this is rarely a window method.)

   The interface of this method can be something like

   ```
   PROCEDURE TYourType.PoseModelessDialog;
   ```

   The details of the method's interface depend entirely on the use your application makes of the dialog.

   Some applications (those that use the dialog more like an ordinary window) do not have a PoseModelessDialog method. Instead, the dialog view and window are created in DoMakeViews and DoMakeWindows along with the document's other windows.

3. In the PoseModelessDialog method, create the dialog view and the dialog window, install the controls, set the window title, and launch the window. See the template for details. For a discussion of dialog items see the "Using Dialog Items" recipe.

   Notice that the PoseModelessDialog method opens the dialog window and then returns, without waiting for a response from the user.

4. For some types of modeless dialogs, you may need to create your own descendant of TDialogView. You need to do this if you want to override

one of the standard dialog methods. For example, if you want to save some information pertaining to your dialog after it is closed, you can override TDialogView.DismissDialog. The interface of your new object type might be

```
TYourModelessDialogView = OBJECT(TDialogView)
    FUNCTION TYourModelessDialogView.DismissDialog(dismisser: IDType); OVERRIDE;
END;
```

The dismisser parameter contains the four-character identifier of the item that actually dismissed the dialog.

Notice that you do not have to override any other methods of TDialogView.

5. If you have overridden any TDialogView methods, implement the override version so it takes an appropriate action or returns a command object that will take the action. No template is given for this method because it depends so much on your application.

Template

```
PROCEDURE TYourType.PoseModelessDialog;

VAR     aDialogView: TDialogView;
        aWindow: TWindow;

BEGIN
        { Use this approach when using view resources.  See the
          UDialog ERS for example of such resources. }
        aWindow := NewTemplateWindow(aRsrcID, NIL);
        { 'DLOG' is an arbitrary identifier you define in your resource file. }
        aDialogView := TDialogView(aWindow.FindSubView('DLOG'));
        aWindow.Open;
END.
```

Creating a modal dialog

Purpose

A modal dialog requires a response before the user can continue with the application. As with modeless dialogs, a modal dialog view is an instance of a TDialogView. Modal dialogs usually exist to alert the user to some condition and force the user to make some sort of response. Modal dialogs cannot be deactivated, but can only be dismissed.

How to do it

1. You need to include UDialog in the USES statement of your unit.

2. Define a method that will display the dialog. This method should be for the object type that will issue the command. If the dialog has to do with the operation of the application as a whole, the method should belong to TYourApplication, and similarly with your document or view. (Because you rarely customize TWindow, this is rarely a window method.)

   The interface of this method can be something like

   ```
   PROCEDURE TYourType.PoseModalDialog;
   ```

   The details of the method's interface depend entirely on the use your application makes of the dialog.

3. Implement PoseModalDialog as shown in the template. After you set up the dialog, you call its PoseModally method. That method requires a response from the user before continuing. TDialogView.PoseModally processes events until selecting one of the dialogs items causes the dialog to be dismissed.When one of the items returns the value TRUE for the done parameter, PoseModally returns, returning a message from the item that returned TRUE. You should then interpret the value of this return value and take appropriate action (which might, if the user chose Cancel, mean taking no action).

Template

```
PROCEDURE TYourApplication.PoseModalDialog(aCmdNumber: INTEGER);

VAR     aWindow: TWindow;
        dismisser: IDType;

BEGIN
        aWindow := NewTemplateWindow(someRsrcNumber, NIL);
        dismisser := TDialogView(aWindow.FindSubView('DLOG')).PoseModally;

        { In this case, 'yes' is the identifier of the yes button. }
        if dismisser = 'yes ' then
               { react to this response }
        else { next possible response }
               { react to next possible response }
        { continue for all possible responses }
END;
```

Using dialog items

Purpose

Every dialog view is made up of a list of dialog items. Dialog items are objects of type TControl or its descendants TStaticText, TCtlMgr, TCluster, TIcon, TPopup, and TPicture. Descendants of TCtlMgr include TScrollBar, TButton, TCheckBox, TRadio, TEditText, and TNumberText. You may also define your own descendants of any of these types or of TControl itself.

Here are the predefined dialog item types from UDialog, and what they are used for:

- TButton implements a simple Control Manager button.
- TCheckBox implements a simple Control Manager check box control.
- TRadio implements a simple Control Manager radio button control.
- TCluster implements a "holding" view for radio buttons or other objects. It has two intrinsic functions—it understands an mRadioHit message from a subview, and can be used to contain other controls with a graphic label.
- TIcon implements an icon item that can serve as a basic form of button if enabled.
- TPicture implements a picture item that can also serve as a basic form of button if enabled.
- TPopup implements a simple pop-up menu selector, following the guidelines for pop-up menus established by the Apple Human Interface Group.
- TScrollBar implements scroll bars as simple dials not associated with any scroller object.
- TStaticText implements a static text item that can serve as a basic form of button if enabled. The text cannot be edited.
- TEditText implements a simple editable text item. It is implemented as a subclass of TStaticText. When the item needs to be edited, the parent DialogView places a floating TEView over the view.

- TNumberText can take in numbers. (Any characters other than 0 through 9 are ignored.) It can validate the numbers to make certain they are within a given range, after the tab key is pressed or another control is selected.

The number and position of dialog items in the dialog view is defined in the dialog's template in the resource file. The distinction between ordinary windows and dialog boxes is virtually gone in MacApp 2.0. Controls are now views which can be subviews of any views, not just dialog views. However, using dialog views does ensure that tabbing and editing of editable text items works properly.

See the "MacApp 2.0 UDialog ERS" for more information, and the DemoDialogs sample program for examples.

The Clipboard

Purpose

The Clipboard and the desk scrap are the Macintosh computer's standard mechanisms for copying and pasting selections within or between applications and desk accessories.

MacApp maintains a private scrap for the running application, so when you cut or copy information from a document, the information is placed in the Clipboard in a form particular to the application. The Clipboard window is represented by the TWindow object referred to by gClipWindow.

When the user cuts or copies data from your application, your application creates a view to display and possibly otherwise handle the data. Normally, that view is of the same view type as the one that originally displayed the data. The data local to your application (and typically stored in objects) is in your application's private scrap.

When your application begins running, the desk scrap contains data from the last cut or copy operation. (The desk scrap will be empty if there has been no cut or copy operation since the Macintosh started up.) This is the public scrap, and the data it contains is in one or both of two forms common to most Macintosh applications: TEXT (ASCII strings) or PICT (a QuickDraw picture). It may also contain data in the form preferred by your application, if that data was cut or copied from a previous instance of your application or another application that uses compatible data types. When it is time to display the

Clipboard and the desk scrap contains no private scrap yet, you can create a view of one of your application's types (typically because there is data in a form used by your application), or you can allow MacApp to create a view that will display the common data types.

When you leave the application, it gets a chance to convert the information in your private scrap to the two common forms. (Leaving the application can mean quitting, switching to another application with MultiFinder™ using the Switcher™, or starting a desk accessory.)

See the Scrap Manager chapter of *Inside Macintosh* for more information on the desk scrap and the Clipboard.

How to do it

Clipboard support implies support of the Cut, Copy, and Paste commands. The implementations of those commands are described under "Standard Editing Commands" in this chapter. This recipe implements only the Clipboard support necessary for those commands.

In particular, this recipe deals largely with what is required of a Clipboard view. The view is the main controlling object. The Clipboard view is created in one of two ways. First, when the application starts up, a view is created to handle the initial contents of the Clipboard, as taken from the public scrap. Second, when data is cut or copied from your application, a view of some type originating in your application must be created. In either case, the view must be able to handle certain calls from other methods. This recipe deals with those calls.

Clipboard views commonly have documents to handle the data they show. However, that is not required. A view showing the desk scrap, for example, may simply read and display the desk scrap directly. In implementing Cut and Copy, however, the most common situation is that the data the user has cut or copied is handled by instances of the same objects that handled them in the application itself: document, view, and data objects. The methods described here are typically implemented for any view types that can have data cut or copied, because instances of these view types may be Clipboard views.

1. Define a handle type for your Clipboard data type by declaring two pointers. For example:

   ```
   YourTypeOnClipboard = ^PYourTypeOnClipboard;
   PYourTypeOnClipboard = ^YourClipType;
   ```

   You must create a handle type because the desk scrap cannot use object-oriented data structures. Instead, it uses handles to ordinary Pascal data structures. Also, a single handle thus refers to all the data of a particular type in the Clipboard.

As with the data structure created to save your data in a file, the details of your Clipboard structure depend entirely on your application. (You can use a common structure to save data in a file and to write to the desk scrap, although you'll probably want to add fields when saving to a file so you can save state information.)

2. Define a resource type for your Clipboard data type. The value is an arbitrary four-letter string, usually stored in a constant (kClipDataType in the template). Unless your information is of the same type used by other applications, you should make this string unique, as it is used to identify data in the public scrap as data your application can understand. If the Clipboard information is simply a sequence of ASCII characters, kClipDataType should be 'TEXT'; if the Clipboard information is a QuickDraw picture (a saved sequence of drawing commands), kClipDataType should be 'PICT'. If you have a number of different possible Clipboard data types, define several constants. You should register the type identifiers you've chosen with Developer Technical Support to prevent duplication.

3. You don't have to do anything to display PICT or TEXT data from the public scrap. MacApp automatically creates an object of type TDeskScrapView when necessary. If you want to be able to display the public scrap data in your own type of view (usually because the data is of some type preferred by your application), override MakeViewForAlienClipboard for your application type. The interface for that method is

   ```
   FUNCTION TYourApplication.MakeViewForAlienClipboard: TView; OVERRIDE;
   ```

   In the implementation of this method, call GetScrap (a Scrap Manager routine) once for each Clipboard data type you can handle. (GetScrap takes a handle for the data. Pass NIL in this case, because you don't need to actually read the data now.) If you find data of one of your types, create an appropriate view object, and return it. If you don't find one of your types, you should call INHERITED MakeViewForAlienClipboard so that the MacApp method can create and return a TDeskScrapView object.

   You need to override this method to create views for your application's scrap types.

   A sample implementation is given in the templates for this recipe. The sample begins with a call to GetScrap. The first parameter of GetScrap is ordinarily a handle used as the destination of the scrap data. In the templates, the destination is NIL, so nothing is passed to the application.

4. Override the necessary methods for your Clipboard view type as shown:

```
FUNCTION TYourView.ContainsClipType(aType: ResType): BOOLEAN; OVERRIDE;
FUNCTION TYourView.GivePasteData(aDataHandle: Handle; dataType: ResType): LONGINT;
                                OVERRIDE;
PROCEDURE TYourView.WriteToDeskScrap; OVERRIDE;
```

The implementations are discussed in the following steps.

5. ContainsClipType is called by other methods to find out whether the Clipboard contains a particular type of data. The default implementation (as defined in TView) calls GetScrap to find out if the requested type is in the public scrap. You should override this method for a view that can display a private scrap. (Note that this is always the case when the data in the Clipboard got there through a cut or copy in this instance of your application.)

   The interface of this method is

```
TYourView.ContainsClipType(dataType: ResType): BOOLEAN; OVERRIDE;
```

   A sample is given in the templates.

6. GivePasteData is called to get data from the Clipboard. If you want to get data from the public scrap, you don't have to override this (it is declared and implemented for TView). If the data to be pasted is in your application's private scrap, you need to override this method. Its interface is

```
TYourView.GivePasteData(aDataHandle: Handle; dataType: ResType): LONGINT;
```

   GivePasteData has two purposes. First, it returns the length of the data of the given resource type in bytes (or, if there is some problem, returns a negative number, which is an error code). Second, if aDataHandle is not NIL, the method places the data in the space referred to by the handle.

   Your version of GivePasteData should follow this logic:

- Check whether the data type requested matches the data types your program can handle. This should always be TRUE (because the request comes from one of your paste methods). If it is not TRUE, return noTypeErr, a predefined constant.

- If the data has been written to the desk scrap, call INHERITED GivePasteData. TView.GivePasteData uses GetScrap to put the information in the handle.
- Otherwise, the data in the Clipboard originated from your application, and you must extract the required information.

  A sample of this method is given in the templates.

7. To enable other programs to receive Clipboard data from your application, override TView.WriteToDeskScrap (which has no parameters) for your Clipboard view. (Generally, the Clipboard view is of the same type as your ordinary application view; it becomes a Clipboard view when an instance is created to display the Clipboard. Therefore, you usually need to override WriteToDeskScrap for every customization of TView in your application that allows a cut or copy operation.)

   When your application terminates or the user uses MultiFinder™ or starts a desk accessory, WriteToDeskScrap is called to convert the Clipboard's contents to the desk scrap.

   See the "Scrap Manager" chapter of *Inside Macintosh* for details of writing data to the scrap.

   After you write the data in your application's preferred type, you should, if possible, write it as PICT or TEXT data or both.

8. If you want to have a Clipboard document, create it before making the Clipboard view. When you call TYourDocument.IYourDocument, you can pass in TRUE to indicate to IYourDocument that you are creating a Clipboard document, although that may not matter to IYourDocument. (You do not have to have a Clipboard document, although applications usually do.)

9. You need to have one item for the Clipboard in the resource file: the Show Clipboard menu item.

   See the sample programs' resource files for guidance.

Templates

```
FUNCTION TYourApplication.MakeViewForAlienClipboard: TView;

VAR    offset: LONGINT;
       clipYourView: TYourView;
       aHandle: Handle;
       clipDoc: TYourDocument;

BEGIN

{ Test whether your preferred data type is in the scrap.
  If you can understand other types, test for them here. }
       IF GetScrap(NIL, kClipDocType, offset) > 0 THEN BEGIN
              New(clipDoc);
              clipDoc.IYourDocument(TRUE); { The TRUE is only needed if IYourDocument
                                              cares if this is a Clipboard document. }
              New(clipYourView);
              clipYourView.IYourView(clipDoc);
              WITH clipYourView DO BEGIN
                     fInformBeforeDraw := TRUE;
                     fWrittenToDeskScrap := TRUE; { Tells MacApp it is not necessary to
                                                    write this view to the desk scrap if the
                                                    application quits because the Clipboard
                                                    view was derived from data in the
                                                    desk scrap. }
              END;
              MakeViewForAlienClipboard := clipYourView;
       END
       ELSE
              MakeViewForAlienClipboard := INHERITED MakeViewForAlienClipboard;

END;


FUNCTION TYourView.ContainsClipType(aType: ResType): BOOLEAN;
BEGIN
       ContainsClipType := (aType = kYourClipType);
END;
```

```
FUNCTION TYourView.GivePasteData(aDataHandle: Handle; dataType: ResType): LONGINT;

VAR     aSize: LONGINT;
        err: OSErr;

BEGIN
        { The following test checks whether the requested data type is your program's
          type. You may have several types, in which case this would be a multiple test. }
        IF dataType <> kYourClipDataType THEN
                GivePasteData := noTypeErr
        ELSE
                IF fWrittenToDeskScrap THEN
                        GivePasteData := INHERITED GivePasteData(aDataHandle, dataType)
                ELSE BEGIN
                        { Copy the data in the Clipboard and accumulate the size in aSize.
                          If aDataHandle is not NIL, then by exit time its size must be
                          equal to the ultimate value of aSize, and the Clipboard data must
                          be in the data area referred to by aDataHandle. }
                        GivePasteData := aSize;
                END;
END;
```

Failure handling

Purpose

Any time you access devices, failures may occur. In addition, unanticipated code problems may cause failures. MacApp includes a failure-handling mechanism that is intended to allow applications to clean up debris left by the failure and continue running from the point before the failure. You can also use it to display alert boxes for the user.

This recipe, unlike the others, is more descriptive than prescriptive. The failure-handling mechanism is really a set of routines that you can call, along with routines that you write, to provide minimally disruptive handling of failures. This recipe describes the architecture of the mechanism and tells how each routine can be used. See the sample programs for specific examples of failure handling.

How to do it

The failure-handling mechanism is built around **exception handlers.** An exception handler is a routine, generally local to some method, that is called when a failure occurs and takes action to handle the failure. See the sample programs for examples of exception handlers.

References to exception handlers are kept on a stack. When an error is likely to occur (generally because of I/O or memory allocation) and cleanup needs to be done, MacApp posts exception handlers to the stack; application routines should post exception handlers when an error the application should handle might occur. (Applications post exception handlers sometimes because an error MacApp can't anticipate may occur. Other times exception handlers are used to supplement the MacApp exception handler with application-specific action.)

Whenever a failure occurs, the Failure global procedure is called. Failure is never called automatically. You must check for a failure, and call Failure when you find that it is needed. That check is most often done by calling the MacApp global procedures FailNIL, FailOSErr, FailMemError, or FailResError, which check for specific kinds of errors. Here is the interface to Failure:

```
PROCEDURE Failure(error: INTEGER; message: LONGINT);
```

Failure pops the handler at the top of the stack and calls it. That handler generally does any cleaning up it can do (such as freeing temporary objects or handles), possibly sets up the error alert box, and sometimes calls Failure again to invoke the next handler on the stack with a new error. Returning from the handler automatically passes the same error to the next handler on the stack.

1. The exception handlers that MacApp posts handle errors in a generalized way, usually by displaying an alert box telling the user what happened (to the best of MacApp's ability to tell) and then branching around the code that caused the error. (That generally means abandoning the command that resulted in the error.) When an error that MacApp can anticipate may occur, you may want to post your own exception handler to set up your own alert box or to handle the failure in your own way. The mechanism allows you to take the action you want, set up certain values to produce a useful message, and then invoke MacApp's exception handler. You should always post your own exception handler when a failure that MacApp can't anticipate is possible.

   An important part of the failure-handling mechanism is the ability to give the user a useful alert message. MacApp provides several ways to do that, all working through the same routines. When a failure occurs, the exception handler that is initially called (which may be a MacApp or an application handler) usually calls Failure (directly or by returning) to invoke another failure handler. Failure's error and message parameters are used to build the alert box that informs the user of the error. Handlers usually set those values only if the method that called Failure hasn't set them. (In other words, handlers should assume that the routine that called Failure has more specific knowledge about the error, and thus, if it gave values for error and message, those are the most appropriate values.) Typically, there is a chain of Failure calls that leads to an exception handler (defined by MacApp) that calls TApplication.ShowErrors. (If you want to change what happens next, you can override ShowErrors.) ShowErrors calls the global routine ErrorAlert. ErrorAlert builds the alert message in different ways depending on the message value that you passed. The standard alert strings defined in the standard resource files are

```
phGenError      =     could not ^2, because ^0. ^1.

phCmdErr        =     could not complete the "^2" command because ^0. ^1.

phUnknownErr    =     could not complete your request because ^0. ^1.
```

The alert string is chosen and the placeholders ^0, ^1, and ^2 are filled by ErrorAlert based on the error and message values that are passed to Failure. MacApp uses the error parameter to Failure to find a string to replace ^0. That string identifies the kind of error that occurred. It also uses the error value to find a string to replace ^1, if appropriate. ^1 is used for a string that gives the user advice on what to do, and is only given if that isn't clear from the error identifier.

The message parameter of Failure determines what replaces ^2 and what alert message is used. The message parameter is a LONGINT that is treated as a pair of numbers. The first integer, or high word, of a message determines how the second integer, or low word, is interpreted. There are five possibilities:

- If the high word is equal to msgCmdErr, the low word is a command number. ErrorAlert translates that command number into a command name, and substitutes it for ^2. The phCmdErr alert is used.

- If the high word is equal to msgAlert, the low word is an alert number (that is, a resource number). This generally is an alert that you have defined. That alert message is then displayed.

- If the high word is equal to msgLookup, the low word is a positive integer that is an index into an operation table in the resource file. This is rarely used.

- If the high word is not any of those values, it is a resource ID for a string list and the low word is an index into that list. This string is then substituted for ^2. The phGenErr alert is used.

- If message is equal to zero, the phUnknownErr alert is used.

2. There are two global routines provided to post exception handlers to the stack and remove them when the chance for failure is past: CatchFailures and Success. The interface to CatchFailures is

   ```
   PROCEDURE CatchFailures(VAR fi: FailInfo; PROCEDURE Handler(e: INTEGER; m: LONGINT));
   ```

   The fi parameter is a variable of type FailInfo that you must provide. You don't have to set it to anything.

   Call CatchFailures to set up an exception handler. This pushes your handler onto a stack of exception handlers. If MacApp has already pushed a handler on the stack, yours is above it, so a call to Failure results in a call to your handler.

   The interface to Success is

   ```
   PROCEDURE Success(VAR fi: FailInfo);
   ```

   The fi parameter is a variable of type FailInfo that you must provide. You don't have to set it to anything.

   Success removes your handler from the stack.

   Any calls to Failure within the limits of the CatchFailures and subsequent Success calls result in the execution of your exception handler. If a routine calls CatchFailures, it must call Success (unless there was an error). Also, you must not call Success unless you called CatchFailures earlier in the same routine.

3. You usually don't call Failure directly. Instead, you use one of the four global routines that are provided to test for different kinds of errors: FailNIL, FailOSErr, FailMemErr, or FailResErr. In each case, they call Failure with appropriate error and message values if a failure occurred. If a failure did not occur, they simply return.

   The interface to FailNIL is

   ```
   PROCEDURE FailNIL(p: UNIV Ptr);
   ```

   The p parameter is any pointer or handle (including object references).

This procedure tests whether the given pointer (or handle) is NIL and calls Failure(memFullErr, 0) if it is.

The interface to FailOSErr is

```
PROCEDURE FailOSErr(error: INTEGER);
```

The error parameter is an OS error code, presumably returned by an *Inside Macintosh* or language routine.

This procedure checks whether the given OS error code signals an error and, if it does, calls Failure. This is most often used with functions whose return value is an error code, and you use it with a statement such as

```
FailOSErr(functionCall(parameters));
```

The interface to FailMemError is

```
PROCEDURE FailMemError;
```

This procedure checks whether there was a memory error and, if there was, calls Failure. You generally call this after you attempt to allocate a new pointer or handle. It tests the value of MemError. If MemError <> noErr, it calls Failure(MemError, 0).

4. In your exception handler, you usually want to set the message parameter only if it has not already been set. To do that, you can use the global procedure FailNewMessage in place of Failure.

```
PROCEDURE FailNewMessage(error: INTEGER; oldMessage, newMessage: LONGINT);
```

This procedure calls Failure and passes the error and newMessage or oldMessage parameters. FailNewMessage passes the oldMessage parameter to Failure unless it is 0, in which case newMessage is passed. This is used in an error handler so that the error handler can provide a message (newMessage) only if a message was not provided already. You would use this routine instead of calling Failure when you want to set the message value but do not want to override a message value established by a lower-level handler.

5. You must take special care to handle failures carefully during creation and initialization of objects. You should always call FailNIL after calling New. However, it is also possible to encounter failures when calling the initialization method of an object that you have just successfully created. This case occurs frequently, so all MacApp code follows a helpful convention: if the initialization method for an object fails, the method frees

the partially initialized object. This convention, which relieves you from freeing the object, makes it easier to write code that creates new objects.

Although convenient, this scheme has a potentially damaging side effect: when you call the initialization method of an object and it fails, your reference to the object may become invalid. If the code calling the initialization method has a failure handler, the handler must be prepared for this situation. Because most MacApp objects (such as views and windows) will be freed automatically if they are successfully initialized, you normally don't need to have a failure handler.

Initialization methods that can signal failure (or that call ancestral methods that do so) must be written carefully. Because its Free method may be called, the object must be put into a state that allows Free to succeed *before* any action that can fail is taken. Thus, the sequence of actions in your initialization method should be:

- Initialize any variables that your Free method needs to operate successfully. No action that can fail may be taken in this step.
- Call the immediate ancestor's initialization method (if any). This may fail, in which case your Free method will be called.
- If you do any initialization that can fail, set up a failure handler, do the initialization, and then remove the failure handler. The failure handler should do any specific cleaning up you need done, and then call Free.

Your notes

Chapter 8

MacApp Debugging Facilities

MacApp provides four debugging tools:

- **The Debug window.** This window is available when the MacApp code used with the application was compiled with debugging on. Output from Write or WriteLn statements are directed to this window. In addition, this window allows you to use the Debug menu and the Interactive Debugger.

- **The Debug menu.** This menu appears in the menu bar of any MacApp application (that was compiled with debugging on) that has the Debug menu in its resource file. Using this menu, you can control certain features of your application and cause certain information to be printed instantly or periodically in the Debug window.

- **Inspector windows.** These windows provide an easy way to display the fields of any object instance. An Inspector window contains a list of every object class that has at least one instance. When you choose one of these classes, a list of every instance of that class appears. You can then choose to see the fields of any of these instances. Since you can have multiple Inspector windows, you can examine the fields of more than one instance at a time.

- **The Interactive Debugger.** This is a high-level debugger that runs in the Debug window. It takes control whenever there is an error as well as the beginning and end of most MacApp methods and routines. Unless you specify otherwise, it also takes control at the beginning and end of methods and routines in your application.

To use the MacApp debugging facilities, the version of MacApp and your application that you use must have been compiled with debugging on. See "How to Install and Use MacApp" for information on how to turn off debugging. (The easiest way to tell whether or not a version has been compiled with debugging on is by its size: debugging code typically doubles the size of the MacApp object file. Note, though, that additional building block units such as UTEView, UPrinting, and UDialog also make the object file significantly larger, so you can only compare files with the same units.)

None of the debugging facilities are available when MacApp is compiled without debugging code. (Optimized code never includes debugging code.) However, you can always use the low-level MPW debugger, MacsBug, with your MacApp application—even if it's compiled with debugging off. Some information about using MacsBug with a MacApp application is included under "Using MacsBug With MacApp," later in this chapter. See the *Macintosh Programmer's Workshop Reference* for complete information on MacsBug.

You should always recompile MacApp and your application without debugging when you want to produce a production version of your application. For information, see “What Controls Debugging Code” in this chapter.

* *Note:* To have the full set of debugging facilities, you must link your application with UTrace.p.o, UTrace.a.o, UWriteLn.p.o, UGridView.p.o, UInspector.p.o, and WWDriver.c.o. These units are normally linked if you use the default build files. To have the Debug menu appear, you must have that menu in the resource file. It is included in the MacApp debugging resource file. Applications don’t need to have USES UTrace, UWriteLnWindow, UGridView, or UInspector themselves unless they call the routines in these units directly. (Note that you can call WriteLn without USES UWriteLnWindow.)

What controls debugging code

Several factors control what you can do with the debugging facilities. They are described in this section.

Compiler variables

There are a number of conditional compilation variables defined in MacApp that are always TRUE when the code is compiled with debugging on and always FALSE when the code is compiled with debugging off. (You can also set their values individually from the Compiler command line, although that is rarely done.) The variables are

```
qInspector
qTrace
qDebug
qNames
qWritelnWin
qRangeCheck
```

To insert code that will be compiled only when one of these switches is TRUE, precede the code with a line such as

{$IFC *variable*}

where *variable* is replaced by the identifier of one of the Compiler variables. Follow the code with

```
{$ENDC}
```

Everything between these switches will be compiled only when the variable is TRUE. In general, you will use the qDebug switch to delimit your debugging code.

The $D switches

The $D compiler switches {$D++} and {$D+} cause the Compiler to insert additional debugging information in the compiled code.

When you use {$D+} or {$D++}, the Compiler inserts an 8- or 16-character identifier after the return instruction of every method or other procedure. (16 characters are used for a method; 8 are used for an ordinary routine).

In addition, when you use {$D++}, the Compiler generates a call to the Interactive Debugger at the entry and return of every routine and at every EXIT statement. When the Debugger is called in this way, you can stop the program with the attention keys (see "Entering Debugger Mode") or the Interactive Debugger can stop the program because of a breakpoint or because you had run the program with the Single Step command (see "Entering Debugger Mode," below).

When you use {$D+}, the Compiler inserts the identifier but does not generate any calls to the Interactive Debugger.

In either case, the identifier allows the Interactive Debugger to display the name of the routine when it displays the stack.

Each switch affects every routine until the other switch or {$D–} is encountered.

When you are debugging, you generally want every routine in your code to be covered by {$D++}. UObject includes the following switch:

```
{$IFC qTrace}{$D++}{$ENDC}
```

Because of this switch, every routine in a unit that names UObject in its USES clause (including all of MacApp) is automatically covered by {D++} unless you use {$D–} or {$D+}. (See the *Macintosh Programmer's Workshop Pascal Reference* for an explanation of these switches.)

If you want a particular routine to be excluded, embed it as follows:

```
{$IFC qTRACE}{$D+}{$ENDC}

PROCEDURE Example;
.
.
.
BEGIN
.
.
.
END;
{$IFC qTRACE}{$D++}{$ENDC}
```

You can similarly embed a group of routines in the same switches.

Note that the use of {$D-} alone is not recommended. Even with thoroughly debugged routines, it is better to have the routine's name available. (In production code, the names are not included because the qNames compiler variable is set to FALSE.)

Important

Assembled routines (such as those described in *Inside Macintosh*) or routines from units that do not use UObject (such as Paslib) are never governed by the $D switches.

Including debugging code

An application often includes code that is used only for debugging. MacApp uses the compiler variable qDebug to control compilation of debugging code. To insert code that will be compiled only when debugging is desired, surround the code with

```
{$IFC qDebug}
```

and

```
{$ENDC}
```

All code within these markers is compiled only when qDebug is TRUE.

> *Note:* You can also write code so you can, when debugging, enable or disable features. See "The Experimenting Flag," in "The Toggle Command Flag" subsection of " The Interactive Debugger" section below.

The Debug window

The Debug window appears when you start up any application that is linked with a version of MacApp compiled with debugging on. It is a scrollable, movable, resizable window. It is initially inactive. If you intend to use it often, you may want to position it and your other windows so the Debug window is always visible.

The Debug window can hold, by default, 52 lines of 80 characters each. (This value is set in UMacApp.inc2.p, in a call to WWInit. You can change it if you want to.)

Application mode and debugger mode

When debugging code is included in a MacApp program, the program has two distinct modes: **application mode** and **debugger mode.** In application mode, the application runs normally (except for any special behavior you insert with debugging code), and the Debug window can be activated, resized, moved, and scrolled like any window. The application may use WriteLn to write information in the Debug window, and information may be written there in response to Debug menu commands or Interactive Debugger commands.

Using WriteLn statements with the Debug window

There is no special trick to using WriteLn statements and Write statements to print in the Debug window. MacApp uses UWriteLnWindow, a unit provided with MacApp, to set up the debugging window so that all WriteLn text goes to that window. (Note that, except in debugging, a WriteLn is never used to print text in a MacApp program. Instead, you use UTEView or the QuickDraw text calls.)

You always enclose WriteLn statements with conditional compilation switches so that they are not included in production code. See "Including Debugging Code," above.

Reading debugging information in application mode

If you want to type in debugging information, you can use Read and ReadLn. As with WriteLn, you normally surround Read and ReadLn statements with conditional compilation switches so they will not be included in production code. See "What Controls Debugging Code," above.

The Debug menu

Any MacApp application that was compiled with debugging on and includes the MacApp debugging resources the Debug menu displayed in its menu bar. The

Debug menu lets you control certain features of your application and print certain information in the Debug window. This section describes the commands in that menu.

The rest of this section details the standard Debug menu commands. You can add additional commands by editing Debug.r and adding support for your commands. You may want to add your own commands in an additional Debug menu, rather than adding them to the MacApp Debug menu. See the sample programs for examples.

New Inspector window

This command creates a new Inspector window titled "Inspector Window *N*." You will want one Inspector window whenever you want to examine the fields of object instances. You might want multiple Inspector windows to examine fields of more than one instance simultaneously.

Allow trace of menu setups

This is a toggle command. When it is checked and the Interactive Debugger is in trace mode (see "The Trace Command, T," below), {$D++} program points are printed in the Debug window even during the menu setup cycle.

Allow trace during idle

This is a toggle command. When it is checked and the Interactive Debugger is in trace mode (see "The Trace Command, T," below), {$D++} program points are printed in the Debug window even during the idle cycle.

Make Front Window Modal

This is a toggle command. It controls the setting of the front window's fIsModal flag. If the front window is already modal, this command reads "Make Front Window Modeless".

Do first click for This Window

This is a toggle command. Choosing it toggles the front window's fDoFirstClick field. If fDoFirstClick is true, this reads "Don't Do First Click for

This Window" and the mouse click that activates the window is also (assuming it is in the content area of the window) passed to gTarget.DoMouseCommand for action. When fDoFirstClick is false, the first mouse click activates the window but is not passed to DoMouseCommand.

Scale pictures in Clipboard to window

This is a toggle command. When it is checked and the Clipboard window is displayed, any PICT data (pictures) in the Clipboard is scaled to the size of the window before being displayed.

Show Debug window

When you choose this command and the Debug window is closed, it is opened and made the active window.

Show software version

Choosing this command results in a call to the method gTarget.IdentifySoftware. That method normally prints the compilation data and time in the Debug window.

Refresh front window

Choosing this command invalidates the entire front window, resulting in an update of the contents (as well as the borders) of the window.

Show page breaks

This is a toggle command. When it is checked, lines are drawn to show page breaks of your views. When it is not checked, page breaks of your views are invisible. The page breaks shown by this command are for debugging purposes only.

Inspector windows

Inspector windows provide an easy way to display fields of your program's instances, as well as a few other types of information. Inspector windows are created by choosing the New Inspector Window command of the Debug menu. You can create multiple Inspector windows, which allows you to examine the fields of more than one instance at once.

Using Inspector windows

The upper-left pane of a Inspector window is a list of classes that have at least one instance. Clicking one of the class names fills the upper-right pane with a list of instances of that class. Clicking one of the objects causes the lower pane to display the fields of that object. Once an object is displayed in the lower pane, it is also possible to click a field that is one of the following types: an object reference field, a GrafPtr, a WindowPtr, a ControlHandle, a TEHandle, or a RgnHandle. After you click one of these fields, it will be displayed in the lower pane.

The lower pane is not automatically updated when the data being displayed changes. The new data is displayed whenever the Inspector window is redrawn. (Remember that redrawing occurs only when the displayed data has been covered and is uncovered. Therefore, you may have to do some window resizing to update the displayed data.)

Figure 8-1 shows a sample session with a Inspector window.

Figure 8-1 is currently located in the MacApp 2.0 ERS.

Figure 8-1
A sample session with a Inspector window

There are two methods that you will need to override for each of your object classes if you wish to fully employ the Inspector window: the Fields method and the GetInspectorName method. Both of these methods have been added to the TObject class.

The Fields method

The Fields method returns information about each field of an object. It has already been implemented for all of the classes defined by MacApp, so you will only need to implement it for those classes you define. You should replace any Inspect methods you may have defined by Fields methods.

The declaration line of Fields looks like this:

```
PROCEDURE TObject.Fields (PROCEDURE DoToField(fieldName:  Str255;
                                              fieldAddr:  Ptr;
                                              fieldType:  integer));
```

In general, a Fields method should call the DoToField procedure on each field defined by its class. Typically, a Fields method will first call DoToField to report the class name, then call DoToField for each field in the class, and finally call the inherited Fields method (which will report the inherited data).

For example, imagine that you've defined a TShape class like this:

```
TShape = OBJECT(TObject)
       fRect:   rect;
       fColor:  RGBColor;

       {$IFC qDebug}
       TShape.Fields(PROCEDURE DoToField(fieldName:  Str255;
                                         fieldAddr:  Ptr;
                                         fieldType:  integer); OVERRIDE;
       {$ENDC}
END;
```

You should implement its Fields method like this:

```
PROCEDURE TShape.Fields (PROCEDURE DoToField(fieldName:  Str255;
                                             fieldAddr:  Ptr;
                                             fieldType:  integer);

BEGIN
       DoToField('TShape', NIL, bClass);       { First report the class name  }
       DoToField('fRect', @fRect, bRect);      { Then report the fields  }
       DoToField('fColor', @fColor, bRGBColor);
       INHERITED Fields(DoToField);            { Finally report the inherited fields. }
END;
```

As you can see, DoToField has three parameters: fieldName, fieldAddr, and fieldType.

- The fieldName parameter is a string containing the name of the field (or class) that you want reported.
- The fieldAddr parameter is a Ptr containing the address of the field that you want reported. When you are reporting a class name, this parameter should be NIL.
- The fieldType parameter is an integer containing the value corresponding to the type of the field that you are reporting. MacApp has defined a number of constants to make filling in this parameter easy. These are: bInteger, bHexInteger, bLongInt, bHexLongInt, bString, bBoolean, bChar, bPointer, bHandle, bPoint, bRect, bObject, bByte, bCmdNumber, bClass, bOSType, bWindowPtr, bControlHandle, bTEHandle, bLowByte, bHighByte, bPattern, bFixed, bRgnHandle, bRGBColor, bTitle, bGrafPtr, bSTyle, bVCoordinate, bVPoint, bVRect, bFontName.

The GetInspectorName method

Hex addresses of object instances are displayed in the upper-right pane of the Inspector window. If you want your objects to display more than their instances names, you need to override the GetInspectorName method.

The GetInspectorName method returns a string in its VAR parameter. This string will be displayed after the hex address in the upper-right pane of the Inspector window. GetInspector name is declared as:

```
PROCEDURE TObject.GetInspectorName (VAR inspectorName:  Str255);
```

As an example, the class TWindow uses GetInspectorName to return the window instance's title:

```
{$IFC qInspector}
{$IFC MAInspector}
PROCEDURE TWindow.GetInspectorName (VAR inspectorName:  Str255); OVERRIDE;

BEGIN
        { Make sure that a window exists }
        IF fWMgrWindow <> NIL THEN
                { If it does, return its name. }
                GetWTitle(fMgrWindow, inspectorName)
END;
```

The Interactive Debugger

The MacApp Interactive Debugger is a high-level command-line–operated debugger available to MacApp applications compiled with debugging on. It operates in the Debug window. The Interactive Debugger is displayed when your application is in debugger mode.

When you are in debugger mode, the Interactive Debugger prompts you with a ≥. In addition, the Interactive Debugger displays a square block cursor. You cannot use the mouse when in the interactive debugger (with a few exceptions) and the mouse pointer is hidden.

Entering debugger mode

You can enter debugger mode by pressing and holding the attention keys, Shift-Option-Command. You then enter debugger mode at the next {$D++} program point. The Debugger prints "Stopped at" followed by the name of the routine containing the program point.

In addition, debugger mode is automatically entered at the next {$D++} program point when

- a breakpoint has been set at program points with this point's identifier (see "The Breakpoint Command, B," below), or
- the Interactive Debugger's Single Step command is in effect (see "The Single Step Command, Space Bar," below)

Also, debugger mode is entered automatically when

- the procedure ProgramBreak is called in the program,
- SysError (the operating system trap that usually displays a "bomb" alert box) has been called. In this case, the program will not be able to continue, but you may be able to use the Debugger to examine the circumstances of the error. (Some 68000 exceptions are included in this group.)

For this last group, the Debugger can be entered at any point, and not just at {$D++} program points.

What the Interactive Debugger prints when it starts

When you enter debugger mode, the Interactive Debugger prints a line of text that identifies what routine was executing and some information about the circumstances under which debugger mode was entered. These messages are explained below.

* *Note:* When the Debugger prints the messages described below, the placeholders *RoutineName* and *SegmentNumber* shown here will be replaced by the appropriate information. In each case, if the routine is a method, *RoutineName* has the form ObjectType.Method.

`BEGIN` *RoutineName #SegmentNumber*

The routine *RoutineName* has just been entered. This could be in response to the attention keys, a breakpoint, or the Single Step command.

`END` *RoutineName #SegmentNumber*

The routine *RoutineName* is about to return. This could be in response to the attention keys, a breakpoint, or the Single Step command.

`EXIT` *RoutineName #SegmentNumber*

An EXIT statement in routine *RoutineName* is about to execute. This could be in response to the attention keys, a breakpoint, or the Single Step command.

`BREAK` *RoutineName #SegmentNumber*

The MacApp global debugging procedure was called from routine *ROUTINENAME.*

`SYSER` *RoutineName #SegmentNumber*

A 68000 exception (such as divide-by-zero) or a SysError error has occurred.

Using the Debug window in debugger mode

In debugger mode, the MacApp Interactive Debugger is in control, and the Debug window cannot be activated, resized, moved, or scrolled in the normal manner (with the mouse). However, you can use the following commands to control the window:

- WF brings the Debug window in front of any other windows.

- WB moves the Debug window behind any other windows.
- WR allows you to resize the Debug window with the mouse. After you've given the WR command, the mouse pointer becomes a small upper-left corner symbol. You should click at the point where you'd like the upper-left corner of the Debug window to be, and then drag to where you'd like the lower-right corner to be. The window will be resized and the pointer will disappear. You won't be able to use the mouse for anything else.
- The Backspace key scrolls the window up.
- The Return key scrolls the window down.

There is no way to scroll horizontally in debugger mode.

Interactive Debugger commands

You can enter Interactive Debugger commands only when in debugger mode and when the ≥ prompt is displayed. Each command is a single character, generally the first letter of the command name. Some of the commands require parameters. In those cases, a message and a ? prompt are printed after the command is entered. In entering parameters in response to the ? prompt, terminate your input with Return.

When the ≥ prompt is displayed, an incorrect or inappropriate response is the same as the Help command. When the ? prompt is displayed, an inappropriate or incorrect response is ignored.

The rest of this section describes the action of each command. The character you type for each command is given after its command name.

The Help command

The Help command displays a list of the Interactive Debugger commands. This list is printed whenever you type a character that is not a command.

The Status command, ?

The Status command, ?, prints the list of Debugger commands and also includes some status information. The status information includes:

- the value of A5 and thePort

- whether trace mode is on (see "The Trace Command, T," below)
- whether a breakpoint is set and, if so, where
- the identifier and segment for the current {$D++} program point

The Recent History command, R

The Interactive Debugger logs the 63 most recent {$D++} program points encountered, whether or not the program stopped at them.

The Recent History command lists all the saved {$D++} program points using a three-column format. The current {$D++} program point is at the bottom of the left column, the preceding point is above that, and so on. The oldest recorded {$D++} program point is at the top of the rightmost column.

Important

This command may not work properly (that is, you may get an incorrect name) if automatic segment unloading is on (the default). Segments are generally unloaded in the main event loop, so, the results of the Recent History command are generally accurate if you have not entered that loop. You can turn off automatic segment unloading by giving the Toggle Flag command (X) followed by a U.

The Parameters command, P

The Parameters command displays the parameters of a procedure frame. Its output is in hexadecimal form.

When you give this command, you are prompted for a stack frame number. (See "The Stack Crawl Command, S," below.) You give the stack frame number as a decimal number and press Return. (Pressing Return alone is equivalent to typing 0.) Level 0 is the routine where the program stopped.

The Locals command, L

The Locals command displays the local variables of a procedure frame. Its output is in hexadecimal.

When you give this command, you are prompted for a stack frame number. (See "The Stack Crawl Command, S," below.) You give the stack frame number as a decimal number and press Return. (Pressing Return alone is equivalent to typing 0.) Level 0 is the routine where the program stopped.

The Fields command, F

The Fields command displays the fields of an object in hexadecimal form, and shows the object's type and all ancestor types (except TObject, as explained below).

When you give this command, you are prompted for the object you want to know about. You can give an object identifier, a hexadecimal handle , or you can give a decimal stack frame number, and then press Return. The stack frame number is interpreted as referring to SELF for the method in that frame. If the routine in that stack frame is not a method, you are told there is no object at that level. (See "The Stack Crawl Command, S," below.)

For certain objects that are referred to by global variables, you can give the global variable identifier. Those global variables are

```
gTarget
gLastCommand
gDocument
gApplication
gDocList
gFreeWindowList
gClipView
gClipUndoView
gPrintHandler
gFocusedView
```

In the output of the Fields command, the fields of the object are divided into groups determined by the object type that declared the field. That is, inherited fields are listed separately from fields declared for the object's type. The name of each type is printed before the field. Here is an example:

```
TEVTHAND
10456: 0001   5643   0001
TDOCUMEN
1045C: 0645   8987   1087   9876   7575   9876   7563   7565
1046C: 9876   7565   0645   8987   1087   9876
TYOURDOC
10478: 7865   4387   FEA6   BCA4
```

In this example, the object occupies locations 10456–1047F. Its type is

```
TYourDocument = OBJECT(TDocument)
```

The field declarations define four words (8 bytes or 16 nibbles) of data, which here occupy locations 10478–1047F. The immediate ancestor, TDocument, declares fields accounting for 14 words of data, shown here in locations 1045C–10466. The next ancestor, TEvtHandler, declares fields accounting for three words of storage from 10456–1045B. The ultimate ancestor, TObject, declares no fields and is always omitted from the Fields command output.

The Display Memory command, D

The Display Memory command displays the contents of any part of memory.

It prompts for a hexadecimal address and then displays 16 bytes beginning at the specified address. The memory contents are displayed in both hexadecimal and ASCII form.

If you want to see the next 16 bytes, type the Display More command (M) at the next ≥ prompt.

The Stack Crawl command, S

The Stack Crawl command displays a list of the current stack frames. For each stack frame, the command displays the frame number (which can be used in the F, P, L, and I commands), the pointer to the frame in hexadecimal, the name of the procedure or function, and the number of the segment. If the frame belongs to a method, the value and type of SELF are also displayed.

The Display More command, M

This command can be given only immediately after the Display Memory command or the Stack Crawl command. It displays the next 16 bytes of memory or the next group of stacked procedure calls. See "The Display Memory Command, D," and "The Stack Crawl Command, S," above.

The Trace command, T

The Trace command turns trace mode on and off. When trace is on and you give the Go command (to restart your application), the identifier for every {$D++} program point is printed.

The program will slow greatly. Performance is especially slow if the Debug window is partially hidden by other windows.

Once you restart the program, you are in application mode, and you can stop the program with the attention keys (see "Entering Debugger Mode," above). Any errors or breakpoints will also stop the program.

Note that you may be able to use the Recent History command to print a trace of the last 63 {$D++} program points even if trace mode was not on.

The Single Step command, Space bar

The Single Step command, given by pressing the Space bar, restarts the program until the next {$D++} program point. The program point is identified.

The Go Command, G

The Go command ends debugger mode and starts the application again. The application continues until the user quits or the Debugger is given some reason to take control again.

The Breakpoint command, B

The Breakpoint command allows you to set a breakpoint at a {$D++} program point. When you next give the Go command, the program continues until the breakpoint is encountered.

Only one breakpoint can be set at a time. Setting a new breakpoint cancels the old one.

When you give the Breakpoint command, you are prompted for a routine name. Only the first eight characters are significant in a routine name. If you also give an object type, only the first eight characters are significant. (If the first eight characters of the name are not unique, a breakpoint is set for all appropriate routines.) Case is insignificant.

If you want to set the breakpoint at a method, you have two choices:

- Give the name of the object type for the method. This will have the form ObjectType.Method, where ObjectType is the name of the object type that implemented the method. (Note that you cannot give an object-reference variable identifier.) The break will occur only when that particular method is invoked. (That method is invoked either from an object of the type named or from an object of a descendant type, if the method named is not overriden by that particular descendant type or any intervening descendant types.)

- Give the method name without a qualifying object type. The break will then occur on any method (or ordinary routine) with that name, regardless of what object type implemented it.

The name you give is not checked to verify that it actually exists.

You can set a breakpoint at any routine, regardless of whether it is currently in memory, but note that the break will not occur unless the named routine contains a {$D++} program point.

The Clear Break command, C

The Clear Break command clears any breakpoint set with the Breakpoint command.

The Scroll Up command, Delete (or Backspace)

Pressing Delete when the ≥ prompt is displayed scrolls the Debug window up one line.

The Scroll Down command, Return

Pressing a Return when the ≥ prompt is displayed scrolls the Debug window down one line.

The Debug window commands, WF, WB, and WR

These commands are used to manipulate the Debug window:

- WF places the Debug window in front of any other windows and, if it is inactive, activates it.

- WB places the Debug window behind other windows.

- WR activates the mouse so you can resize the Debug window. The mouse cannot be used for anything else in debugger mode.

* *Note:* Application windows cannot be updated in debugger mode, so if you use the WB command to move the Debug window behind other windows or resize the Debug window so that new parts of other windows

are revealed, those windows will not be properly drawn until you reenter application mode and the windows are updated.

The Output Redirect command, O

The Output Redirect command redirects all Debug window output into a file.

When you enter this command, you are prompted for a filename. Enter any filename. (If the file already exists, it must be of type TEXT.) If you do not specify a volume or folder, the output will be written to the default volume in the default folder (that is, the folder containing the application).

If you precede the filename with >>, the Debug window output is appended to the file. (Spaces between >> and the filename are ignored.)

If the file exists and you do not include >>, the contents of the file are erased.

If the file does not exist, it is created. Its type is TEXT and its creator is MPS.

To cancel this command, give the command again, and press a Return in response to the ? prompt. The file will be closed.

The Quit command, Q

The Quit command executes an ExitToShell.

The Heap and Stack command, H

The Heap and Stack command is actually the entry into a group of subcommands. After you give this command, you are prompted with a ? for a subcommand.

All the subcommands print information about the heap and the stack.

The Help command, ?: Typing ? gives you a list of the subcommands.

The Procedure Stack Usage Breakpoint command, +: This command allows you to set a breakpoint on stack usage by a single routine.

The stack usage of the routine is computed to be the difference between registers A7 and A6. This value will take into account the local variables declared in the procedure, other local variables used internally by compiler-generated code, and registers that are saved on the stack. It does not take into account any stack usage for passing parameters to routines, stack usage by

Inside Macintosh routines (including QuickDraw), and stack usage for routines that do not contain a {$D++} program point.

Setting breakpoints on total stack depth and individual procedure usage will help you to gauge how much stack space your application requires and to identify routines that have excessively high stack usage. (These may have several string parameters that are passed by value rather than by reference.)

The Stack Usage Breakpoint command, B: This command allows you to set a breakpoint on total stack usage. Total stack usage is the difference between register A7 and the bottom of the heap.

Setting breakpoints on total stack depth and individual procedure usage will help you to gauge how much stack space your application requires and to identify routines that have excessively high stack usage. (These may have several string parameters that are passed by value rather than by reference.)

The Reset Stack Usage command, D: This command allows you to reset the maximum stack usage value maintained by MacApp to determine if a stack usage breakpoint has been reached. Typing D resets that value.

The Show Heap/Stack Information command, I: This command displays information about the heap and the stack.

The Print MaxMem command, M: This command prints the value of MaxMem. See *Inside Macintosh* for information.

The List Loaded Segments command, S: This command lists all segments currently loaded. An asterisk (*) next to the segment number that indicates that the segment is resident.

The Toggle Flag command, X

The Toggle Flag command allows you to toggle a number of flags that govern the behavior of the Interactive Debugger.

After you give this command, you are prompted for more information. Typing ? gives you a list of the flags. Typing the letter for one of the flags toggles the flag's value.

You can add new items to the list of flags by calling the MacApp global debugging procedure TRCFlag, found in UTrace. The interface is

```
PROCEDURE TRCFlag(flagAddr: Ptr; flagChar: CHAR; flagDesc:
TRCFDescription);
```

You pass this procedure a pointer to a Boolean, a character (used to toggle the flag), and a short description. Only twenty flags are allowed. If you do add flags, be careful not to assign the same letter to two different flags.

The current flags are listed below.

The Report Memory Management Information flag, M: When this flag is on and the number of master pointers changes, the old and new numbers are printed in the Debug window. When the Break flag is also on, the program is stopped, and you enter debugger mode.

The Report Segments flag, S: When this flag is on, a routine name is printed in the Debug window when a segment is loaded. Usually this is the routine that caused the segment to be loaded (unless that routine has no {$D++} program point). When the Break flag is also on, the program is stopped, and you enter debugger mode.

The Break flag, B: When this flag is on and the Report Segment flag, S, or the Report Memory Management flag, M, is also on, a break is generated whenever M or S causes something to be printed in the Debug window.

The Automatic Segment Unloading flag, U: This flag, when on, turns off automatic segment loading by the UnloadAllSegments routine. You might want to try this flag if you find that your program is mysteriously jumping off into an unexpected routine; you may have saved a pointer to a routine in a segment that MacApp unloaded.

The Ask About Allocation flag, A: When this flag is on and an object is created, the name of the routine and the type of object are printed in the Debug window. Debugger mode is entered, and you are given a chance to replace the object reference with NIL. This is used to check handling of memory management.

The Ask About Failures flag, F: When this flag is on, every call to FailOSErr, FailResError, and FailMemError will give you a chance to enter an error code and make your application fail. This allows you to check that you recover properly from these kinds of errors.

The Report Menu Commands flag, C: When this flag is on and the user chooses a menu command, the command number is printed in the Debug window.

The Report Events flag, E: When this flag is on, details of any events (except Debug window events) are printed in the Debug window.

The Intense Debugging flag, I: The Intense Debugging flag toggles the value of the global variable gIntenseDebugging. By enclosing code in an IF structure beginning with the statement

```
IF gIntenseDebugging THEN
```

you can make code have effect only when gIntenseDebugging is TRUE.

The Debug Printing flag, P: When this flag is on and the user does any UPrinting operation, information about printing is printed in the Debug window.

The Experimenting flag, X: The Experimenting flag toggles the value of the global variable gExperimenting. By enclosing experimental code in an IF structure beginning with the statement

```
IF gExperimenting THEN
```

you can make features work only when gExperimenting is TRUE.

The Inspect Object command, I

This command allows you to inspect an object.

When you give this command, you are prompted for the identifier of the object you want to inspect. You can identify it with a hexadecimal handle, an object name, or a decimal stack frame number. If you give a stack frame number, the request is interpreted as being for the object used to call the method at that level. (If it is not a method, you are told there is no object at that level.)

For certain objects that are referred to by global variables, you can give the global variable identifier. Those global variables are

```
gTarget
gLastCommand
gDocument
gApplication
gDocList
gFreeWindowList
gFocusedView
gClipView
gClipUndoView
gPrintHandler
gFocusedView
```

This command then calls the Inspect method for the given object. Since the INHERITED Inspect method will call the object's Fields method, you do not need to override the Inspect method for your objects if you have created a Fields method for them.

The Enter MacsBug command, E

When you give this command, MacApp enters MacsBug. You can return to the MacApp debugger by giving MacsBug the Go command, G.

This command is equivalent to pressing the interrupt button on the Macintosh programmer's switch.

Using MacsBug with MacApp

Although the MacApp debugging facilities are powerful, you may sometimes need to use MacsBug, the lower-level debugger shipped with the MPW Shell. See the *Macintosh Programmer's Workshop Reference* for details on using MacsBug; this section has some additional information you may find useful for using MacsBug with MacApp programs.

You can set breakpoints on methods as you do with any routine. If you want to set a breakpoint on all method calls, you need to know something about how method calls are implemented.

Method calls are implemented in different ways, depending on whether or not the code has been optimized using the optimization option of the MPW Linker. When the code is not optimized (as is normally the case until the code has been debugged and finalized), method calls are routed through the method dispatch routine, %_METHOD. If you want to set a breakpoint on all methods, you can do so by setting a breakpoint on %_METHOD.

To find %_METHOD, look at the linker map produced when you built your application; it's named *YourApp.map*. The map shows all routines and their locations as an offset from register A5. Find %_METHOD in that map and in MacsBug give the command

```
dm ra5+offset
```

where offset is the value shown for %_METHOD in the map. The second and third word of the line produced by this command give the location of the method display routine. You can use this value to set a breakpoint on %_METHOD.

When the code has been optimized %_METHOD is not used and it is not possible to make a general statement about how methods are called under those circumstances, although many will be called by a direct JSR.

If you've stopped at a {$D++} program point, you can use a different method to invoke MacsBug in the routine containing the program point:

1. While stopped at the {$D++} program point (which you can reach by setting a MacApp breakpoint or in any other way), invoke MacsBug from the MacApp Interactive Debugger.

2. Give the MacsBug SC (Stack Crawl) command. A line of this form is displayed:

   ```
   SF @address FR fromAddress Method.Object +offset
   ```

   where *address* and *fromAddress* are two addresses, *Method* is the routine containing the {$D++} program point, *Object* is the type that defined that method, and *offset* is the offset into the routine.

3. Add 4 to the *fromAddress,* and using that number type

   ```
   GT address
   ```

   where *address* is *fromAddress* plus 4. This sets a temporary breakpoint just beyond the {$D++} program point.

4. Return to the MacApp Debugger and type G. You will enter MacsBug in the routine with the {$D++} program point.

Appendix A

Changes Since MacApp 1.1

MacApp 2.0 differs from MacApp 1.1 in many ways. This appendix explains the architectural differences and details how the implementation has changed. Despite all these changes, you should still find that you will be able to convert your old MacApp applications into MacApp 2.0 very quickly. Once you know how MacApp 2.0 works, you should be able to convert even rather large applications in less than a day. Appendix B describes the method for converting applications.

Changes to the architecture

The architecture, or overall design, of MacApp has changed in a great many ways.

- **Views.** The view architecture has been completely overhauled. All classes responsible for displaying images on the screen are now descendants of TView. The TFrame class no longer exists.

  Views now offer a larger coordinate system: 2 billion pixels both horizontally and vertically.

- **TextEdit.** The new TextEdit supports views that allow multiple fonts, styles, and sizes.

- **Dialogs.** MacApp 2.0 has a whole new design for implementing dialogs, allowing you to create dynamic, sophisticated dialogs with smart controls that you can customize.

- **Grids.** MacApp 2.0 has a new unit containing code that will allow you to manipulate data grids, as in spreadsheet applications.

- **Inspector.** The debugging capabilities now include an inspector, which you can use to examine the fields of any instance while the application is running.

- **AppleTalk.** MacApp no longer contains a special AppleTalk® support unit.

- **MultiFinder.** New support for MultiFinder features..

These changes are described more fully below.

Views

MacApp 2.0 offers two major improvements over MacApp 1.1 in the display architecture: it greatly simplifies the architecture and it provides a larger coordinate system.

The old display architecture

MacApp 1.1 provided three basic classes for displaying information on the screen: TWindow, TFrame, and TView. TWindow provided a Window Manager window, TFrame managed the scroll bars and the coordinate translation for scrolling, and TView was responsible for drawing the contents of the window and for handling mouse commands inside the view. This architecture made it difficult to nest views and frustrating to provide dialogs.

Figure A-1 is currently located in the MacApp 2.0 ERS.

Figure A-1
The display architecture from MacApp 1.1

The new display architecture

Whereas the old architecture had two separate lineages (TFrame and TView) that used different schemes for nesting views, the new 2.0 architecture has only one: TView. All display classes descend from TView and they share the following abilities:

- nesting
- focusing
- drawing
- handling events

A view is still an imaginary construct that displays information. On the screen, views are rectangular. One view may be inside another view; also, views may overlap.

Nesting. Each instance of the TView class (or its descendants) may be inside another view, called the superview; and it may have any number of other views, called subviews, inside it. This relationship between superviews and subviews forms a tree. Note that these terms are relative: a single view may be both a superview in relation to one view and a subview in relation to another.

On the screen, nesting is a matter of layering the views. Superviews are at the bottom. A subview always covers the superview: you cannot usually see the superview through one of its subviews. However, superviews do clip subviews. Only that portion of the subview that intersects with the superview is visible.

If a superview has more than one subview, the subviews are also arranged in layers. If two subviews overlap, one of them will be on the top and the other will be on the bottom.

In Figure A-2, there are eight views arranged into a tree. View A has no superview, but it does have three subviews. Note how the layering appears on the screen, and how superviews clip their subviews.

Figure A-2 not available for this draft.

Figure A-2
Tree of views

Focusing. The concept of focusing has not changed. Each view has its own local coordinate system. Focusing translates between the different coordinate systems, and prepares the view so that your drawing will appear in the correct place on the screen.

Drawing. Each instance of the view class has a Draw method, which you are expected to override. Whenever the view draws itself, it tells its subviews to draw themselves. The process continues iteratively until all the subviews are drawn.

Handling events. TView is still a descendant of TEvtHandler, and thus it inherits the ability to handle events including keystrokes, menu commands, and other mouse clicks from the user. Because views can be nested, MacApp needs an algorithm to determine which particular view instance will handle a particular mouse command. Based on the metaphor of subviews layered on top of superviews (and on top of their sibling views), the algorithm is simple: the command is handled by the topmost view at the location of the mouse click. Mouse clicks are handled by the method HandleMouseDown.

The most important classes in the new architecture

The major players in the new MacApp 2.0 display architecture are TWindow, TSScrollBar, TScroller, and specialized content views. These are all descendants of TView.

- TWindow displays the window frame and size box and handles clicks in the title bar, zoom box, and size box..
- TSScrollBar displays the scroll bars and communicates with TScroller to carry out the user's commands.
- TScroller does the actual calculations for scrolling.
- Specialized content views display whatever you want them to. MacApp provides several of these classes: TEView for displaying text; TGridView for displaying spreadsheets and other information that fits well into grids; and a number of others defined in the UDialog unit.. To display other sorts of information you must create your own descendants of TView.

Everything that appears on the screen (except menus) appears inside a window. The window then becomes the root of the view tree—it has no superview. If some portion of the contents can be scrolled, the window instance will have a subview that is a scroller instance and perhaps one or two instances of TSScrollBar (vertical or horizontal or both). Anything that can be scrolled should be a subview of the scroller. For an example, see Figure A-3.

Figure A-3 not available for this draft.

Figure A-3
View tree for a MacApp window

Large coordinates

MacApp 2.0 provides a large coordinate system so you can implement bigger views. The old MacApp was limited to QuickDraw's 30,000 by 30,000 pixels. QuickDraw hasn't changed that limitation, but MacApp 2.0 provides types to store these large 32-bit coordinates: VCoordinate, VPoint, and VRect. With these types, you can store views up to 2 billion pixels by 2 billion pixels.

All drawing is still done using QuickDraw, in 16-bit coordinates. If you want to use 32-bit coordinates, the burden is on you to convert between 32-bit and 16-bit coordinates, using routines and methods provided by MacApp's UViewCoords unit and the TView class.

Text

The TTEView optional building block (in the UTEView unit) has been enhanced to support the new display architecture and to allow your TEViews to have multiple fonts, styles, and sizes. The building block defines the same classes that it used to, but it has additional methods to deal with the different commands; and the old methods have been changed to record and deal with information related to font, style, and size.

TTEView is still based on TextEdit in ROM, so it has the same limitations as TextEdit, namely a maximum of 32,000 characters per view and a maximum height and width of 30,767 pixels each.

Dialogs

MacApp no longer makes use of the ROM's Dialog Manager. Instead, dialogs are now constructed the same way as any other MacApp window—by constructing a hierarchy of views. The new UDialog unit simply provides a set of predefined view classes that can be used in *any* window. One class, TDialogView, can provide modal dialog behavior such as tabbing between text fields and implementing default and cancel buttons. Here are some of the elements defined by MacApp which were traditionally found in dialog boxes:

- static text
- editable text
- icons
- pictures
- check boxes
- radio buttons
- plain buttons
- pop-up menus
- scrollable lists
- clusters, which define dependencies among the controls

The two major advantages this new scheme gives you are: (1) you can use any of the classes of UDialog in any window; and (2) there is no need to distinguish between "windows" and "dialogs" as there was in MacApp 1.1 and as there continues to be when using the Toolbox directly.

Note that the new display architecture blurs the distinction between windows and dialog boxes. Any window can be modal or modeless. To create the illusion of a dialog box, you will use the UDialog unit to create either a modal or a modeless dialog with a variety of controls.

MacApp defines a large number of classes associated with dialogs, most of which can be found in the UDialog unit:

Figure A-4 not available for this draft.

Figure A-4
The Classes defined in UDialog

TEntry. Instances are metaphorical text items, to be used as entries in a dictionary, managed by an instance of TAssociation. Instances of TEntry are used mainly for substituting text in dialog windows.

TAssociation. Instances manage lists of instances of TEntry.

TDialogView. Instances duplicate the function of a dialog window. An instance of TDialogView acts as the topmost view in a template-driven dialog and in some ways replicates the function of the ROM Dialog Manager.

TButton. Instances implement a Control Manager button.

TCheckBox. Instances implement a Control Manager check-box.

TRadio. Instances implement a Control Manager radio button.

TScrollBar. Instances implement a scroll bar, which is one of the more popular types of **dials** available in Macintosh dialogs.

TCluster. Instances implement a container for a cluster of radio buttons or other objects. The cluster can have a label. Clusters are used to group radio buttons and to localize parts of a window.

TIcon. Instances implement an icon which can serve as a static image, or work as a button.

TPicture. Instances implement a picture, which can serve as a static image or work as a button.

TPopup. Instances implement a pop-up menu.

TStaticText. Instances implement a text item, which can serve as a static image, or work as a button.

TEditText. Instances implement a simple editable text item. When the item needs to be edited, the parent DialogView places a floating TEView over the top of the view.

TNumberView. A specialized version of TEditText which accepts only integer numbers.

TDialogTEView. This class is intended for use with TDialogView and not intended to be used for your views.

Grids

MacApp 2.0 offers a new optional building block, UGridViews, which implements several classes that make it easier to display and manipulate gridlike information, such as spreadsheets. The widths and heights of the individual cells can be adjusted dynamically in response to the user's input. GridViews do not dictate how the data is displayed; the representation of that data is completely up to you.

The TGridView unit includes definitions for the following classes:

TGridView. Instances display a one- or two-dimensional grid of cells. The cells can contain anything you wish to draw: text, PICTs, other views, etc.

TCellSelectCommand. A command class that implements the command that selects a cell.

TRowSelectCommand. A command class that implements the command that selects a row divider.

TColumnSelectCommand. A command class that implements the command that selects a column divider.

TVertexSelectCommand. A command class that implements the command that selects a row divider and a column divider.

TTextGridView. Instances display a one- or two-dimensional grid of cells that can display only text.

TTextListView. Instances display a one-dimensional list of cells that can display only text.

Inspector

MacApp 2.0 features an Inspector for examining the contents of any instantiated object's fields. When you are running a debug version of your application, you can select the New Inspector Window command in the Debug menu. See Figure A-5 for an example of a inspector window.

Figure A-5 not available for this draft.

Figure A-5
An inspector window

Using the Inspector, you can select any instantiation of any class and then examine the contents of its fields, as described in Chapter 5, "MacApp Debugging Facilities."

AppleTalk

MacApp 2.0 contains no UAppleTalk unit. Though some Macintosh application programs have some AppleTalk functionality in their applications, it remains to be seen whether there is enough in common among all these programs to warrant an optional AppleTalk building block.

MultiFinder and network support

While programs built with MacApp 1.1 work under MultiFinder, they don't really take advantage of it. MacApp 2.0 makes use of the WaitNextEvent trap and provides a mechanism for identifying the cursor region and length of time before MultiFinder needs to wake up the application.

Documents will be changed to work better in a shared environment (which aids your application not only by making it more compatible with MultiFinder, but by making it more suitable for use on a network). Files are no longer opened only when reading them into memory or writing them back to disk, but are kept open for as long as their document object exists. Also, MacApp 2.0 allows for read-only files.

Appendix B

How to Convert Your MacApp 1.1 Application

MacApp 2.0 has changed significantly since MacApp 1.1. The display architecture has been reorganized. The implementation of dialogs has been completely rewritten. Debugging facilities have been greatly enhanced. The text edit views have been modified for the new display architecture and to support styled TextEdit. New building blocks (like UGridView) have been added. New enhancements (like large views, view resource templates, and MultiFinder support) have been added.

With all of these changes, from enhancements to underlying architecture changes, you might think that converting your MacApp 1.1 application would be laborious. Certainly, recoding a substantial MacApp 1.1 application to take full advantage of MacApp 2.0 is quite a task. However, making only the changes to a MacApp 1.1 application necessary for it to run correctly with MacApp 2.0 is not so bad. In fact, you should be able to convert even relatively large applications in only a day or so.

The reason for this ease of conversion is that most of the commonly used procedures and methods have not changed their interface or function, and some of those that have changed have done so only slightly. Of course, some have changed significantly—those relating to dialogs, for example. These you are better off reimplementing entirely. This chapter steps through each of necessary changes, detailing them where appropriate, and pointing to sources for more information for the others.

Global changes

Much of the global level of your application will stay the same. For example, you probably needn't touch your application object, or any of its methods. There are, however, two changes that affect your program globally.

Unit dependencies

MacApp 2.0 brings with it a whole new set of units. In your main program, as well as in your interface file, you will need a USES statement similar to the following:

```
USES    {$LOAD MacIntf.LOAD}
        MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf,

        {$LOAD UMacApp.LOAD}
        UMAUtil, UViewCoords, UFailure, UMemory, UMenuSetup, UObject, UList,
        UAssociation, UMacApp,

        {$LOAD}
        UPrinting,
        UYourUnit;
```

Debugging

The debugging facilities of MacApp have also changed. The Inspect method used to be the way that your code communicated with the Interactive Debugger. This has been replaced by the Fields method and the Inspector window. For a more complete discussion of the new debugging facilities, see Chapter 8, "MacApp Debugging Facilities."

You should override the Fields method for every object class that you might want information about while debugging, or in other words for all your object classes. You should replace all of your Inspect methods with Fields methods.

As an example, imagine that you've defined a TShape class like this:

```
TShape = OBJECT(TObject)
        fRect:    rect;
        fColor:   RGBColor;

        {$IFC qDebug}
        TShape.Fields(PROCEDURE DoToField(fieldName:   Str255;
                                          fieldAddr:  Ptr;
                                           fieldType:  integer); OVERRIDE;
        {$ENDC}
END;
```

You should implement the corresponding Fields method like this:

```
PROCEDURE TShape.Fields (PROCEDURE DoToField(fieldName:   Str255;
                                             fieldAddr:  Ptr;
                                             fieldType:  integer);

BEGIN
        DoToField('TShape', NIL, bClass);        { First report the class name. }
        DoToField('fRect', @fRect, bRect);       { Then report the fields. }
        DoToField('fColor', @fColor, bRGBColor);
        INHERITED Fields(DoToField);              { Finally report the inherited fields.
}
END;
```

Document changes

For most applications, the document instances and their methods will remain largely unchanged. The most significant exceptions to this are the DoMakeWindows and DoMakeViews methods. If you are not using a simple or a palette window, then your DoMakeWindows will probably have to be rewritten to include Scroller views. See the "Creating a Window" section of Chapter 7. If you want to use the new view templates, you can use DoMakeViews to create a hierarchy of views. See the "Creating Views with Templates" section of Chapter 7.

For simple windows and palette windows, the code in DoMakeViews will remain the same. DoMakeWindows will change slightly, as windows are now considered more like real object classes than in MacApp 1.1. For example, some routines that used to be global procedures are now methods belonging to window objects, such as ForceOnScreen, AdaptToScreen, SetResizeLimits, and SimpleStagger.

You can now use the function NewTemplateWindow to create your windows from resource templates. See the sample programs and the view and dialog ERS documentation for examples.

View changes

The view architecture has changed radically. Yet you can get by with only a minimum number of changes to your old code if you were using fairly standard views before.

One significant change to TView is that it no longer has an fCanSelect field, which you might have used in TYourApplication.MakeViewForAlienClipboard, TYourView.DoMouseCommand, or TYourCutCopyCommand.DoIt. References to TYourView.fCanSelect can usually be replaced by

```
(TYourView <> gClipView)
```

depending on the circumstances.

You will have to replace globally the Focus method. Focus used to be a procedure method of the TFrame class, which is now gone. Focus is now a function method of theTView class. You can usually replace calls to focus by

```
IF yourView.Focus THEN ;
```

if nothing else seems appropriate. Focus returns FALSE if it is not possible to focus the view. See the sample programs for examples.

Finally, the call to IView has changed significantly. The new IView interface is:

```
PROCEDURE  TView.IView(itsDocument:  TDocument;
                       itsSuperView:  TView;
                       itsLocation,  itsSize:  VPoint;
                       itsHSizeDet,  itsVSizeDet:  sizeDeterminer);
```

This procedure initializes the view by calling IEvtHandle(itsSuperView), setting its fSuperView, fLocation, fSize, fSizeDeterminer fields, initializing its fHLDesired fields to hlOff, and adding the view to its superview by calling its superview's AddSubView method.

For further discussion of the new view implementation or a description of the new VPoint type, see the "MacApp 2.0 Display Architecture ERS."

Windows

As before, window methods are rarely overridden. If you used simple or palette windows, you will probably not have to make any window-related changes other than calling the routines listed earlier as methods instead of as global procedures.

Your views

How you should change views specific to your application depends strongly on how you used them. If you had multiple scrolling views per window, you will have to rewrite a bit of your code using the new Scroller architecture. See Chapter 7, “The Cookbook” and the display architecture ERS for details.

If you used fairly standard views and windows, your job will be much easier. Among the things to look out for are:

- The CalcMinExtent has been replaced by CalcMinSize. CalcMinExtent dealt with rect types. CalcMinSize uses the new VPoint type. You will need to do the proper translation before you change the call.

```
PROCEDURE TView.CalcMinSize (VAR minSize: VPoint);
```

- The interface to DoMouseCommand has changed slightly. (Of course, this will apply to all event handlers—so check your application’s and document’s DoMouseCommand methods if you have them.) The only difference is the first parameter. Here is the new declaration:

```
PROCEDURE TView.DoMouseCommand (VAR theMouse: Point; VAR info: EventInfo;
                                VAR hysteresis: Point): TCommand;
```

TEViews and Dialog Boxes

Both TTEView and TDialogView have been substantially rewritten. You will probably have to rewrite any code using TDialogView. If you do not want to add support for style TextEdit, you can probably leave your TTEView code alone. For examples of how to use them now, see Chapter 7, “The Cookbook”, the sample programs, and the ERS documentation.

Command objects

You will probably not have to alter your command objects much, as this part of MacApp was not extensively rewritten.

ICommand

The parameters of ICommand have changed. ICommand used to be declared:

```
PROCEDURE  TCommand.ICommand(itsCmdNumber:  CmdNumber);
```

but now a ICommand sets the command's fView to the view in which the command is taking place, and also sets the scroller used for automatic scrolling during the command:

```
PROCEDURE  TCommand.ICommand(itsCmdNumber:  CmdNumber;
                             itsView:  TView;
                             itsScroller:  TScroller);
```

Tracking methods

The point parameters of the tracking methods TrackMouse, TrackConstrain, and TrackFeedback are now VPoints. You will have to do the necessary conversions before storing these points in rects, and so forth.

Also, now that frames are gone, you may have to replace calls to UpdateEvent with something like this:

```
fYourView.GetWindow.DrawContents;
```

Editing commands

Editing commands also stay the same for the most part. The only differences will occur as they reference views. For example, your Cut/Copy command might have referenced the fCanSelect field of gClipView. For a list of possible problems, see the "View Changes" section, above.

Index