Macintosh®

MacApp Cookbook

Apple Programmer's and Developer's Association

Contents

Figures and tables

Preface **About the Cookbook**

*****I write this after I know what I have written*****

About this book

The *MacApp Cookbook* is a manual for programmerswho wish to develop Macintosh applications using MacApp. **(and so on)**

Other materials you'll need

The software described in this book is part of MacApp, version 2.0, and requires Object Pascal, version X.X, and MPW, version 3.0. MacApp, MPW, and Object Pascal are available from APDA (Apple Programmer's and Developer's Association).

How to use this book

This book shows you how to accomplish certain programming tasks using MacApp. As its name implies, the book has been conceived of as a cookbook, which is usually a collection of step-by-step procedures that show you how to accomplish the task of cooking a certain food. The MacApp Cookbook thus also uses the term **recipe** to describe the set of actions necessary to accomplish a given task.

The recipes are described in a conceptual order, which of course is not the only possible order—you can actually write the steps in any order—but does give you a step-by-step way of approaching the task.

What it contains

mini-description of chapters

Visual cues

Certain conventions in this manual provide visual cues alerting you, for example, to the introduction of a new term or to especially important information.

When a new term is introduced, it is printed in boldface the first time it is used. This lets you know that the term has not been defined earlier and that there is an entry for it in the glossary.

Special messages of note are marked as follows:

- *Note:* Text set off in this manner—with the word *Note*—presents extra information or points to remember.

△ **Important** Text set off in this manner—with the word *Important*—presents vital information or instructions. △

Roadmap to the MacApp documentation suite

Essentially, the books in the MacApp suite divide themselves based on how much of the object-oriented/Macintosh/MacApp language you know, as follows:

- If you don't know anything about object-oriented languages, read *Introduction to Object-Oriented Programming*. This book contains Tutorial introduces object-oriented programming concepts in the context of a sample application. This book answers most of your questions that begin with "What are they talking about, anyway. . ."
- If you know something about object-oriented languages, but don't know MacApp, read *Introduction to MacApp*. This bookintroduces object-oriented programming concepts in the context of a sample application. This book answers most of your questions that begin with "What is *whatchamacallit*. . .", where *whatchamacallit* is replaced with any term from the MacApp world - **Ay, and there's the rub—how does the programmer know which book to look in for what?**
- If you have some experience with MacApp, but need help with a specific procedure, read the Cookbook. The Cookbook will answer most of your questions that begin with "How do I. . ."
- If you're looking up a method, see the Method reference and the MacApp source code.

Some examples of the questions that the books answer are shown in Table P-1.

- **Table P-1** Answers to questions in the MacApp suite

Question	Book
What is an object class? Or an object instance?	*Introduction to OOP*

What are the most important classes that I absolutely have to know about?	*Introduction to MacApp?*
What methods are available in the XXXX unit?	*Method Reference*
How do I debug my application?	*Some specific techniques in the Cookbook; the Debugger is described more completely in the General Reference Manual.*
How do I use the View Editor?	*TBD*
How do I install MacApp?	*Step-by-step in Tutorial; brief overview in Cookbook, complete information where?*
How do I find out about MPW?	*Specific techniques in Cookbook, complete information in MPW Reference suite*
What parameters does the XXX method have?	*Some in Cookbook, complete information in Encyclopedic reference*

****etc.*****

Chapter 1 Introduction to the Cookbook

*****I write this after I know what the book contains*****

A MacApp translation guide

The following table attempts to answer the question "What concepts do I need to understand in order to accomplish a Macintosh programming task, what does MacApp call that concept, and where is it documented?"

Possibly combine this table and the one in the Preface??

■ **Table 1-1** MacApp Cookbook topic guide

Macintosh terms	MacApp terms	Documented in:
Controls	Specialized types of views	Dialog chapter in MacApp Cookbook
Dialogs	Same as a window	Dialog chapter in Cookbook
Events	Applications, Mouse operations, Keyboard Handling	
Windows	Windows and Views	Windows and Views chapters in Cookbook
Undo	Undo, command objects	Undo chapter, Menu chapter
*****etc.*****		

Chapter 2 The Apple Menu

No recipes yet

2.1 Creating an "About Your Application" entry

2.2 Creating a separate "About ..." resource

2.3 Animating the "About ..." entry

Chapter 3 AppleTalk, Multiuser, and Network Considerations

No recipes yet

Using IO completion routines

(MacApp$, 5-27-88) Does this belong here?

Leaving files open for other users

(or maybe in a chapter called Multifinder and Multiuser considerations?)

Chapter 4 Applications

definition of an application-explanation of application object

basic getting started sectionTo build a MacApp program, you need the following five files:

Mappname.p	Pascal source to main program.
Uappname.p	Interface to the unit that will contain the object definitions.
Uappname.inc1.p Uappname.p.	Contains the implementation of the objects defined in
appname.r	The Rez file defining the resources of the application.
appname.make	Contains the build rules used by Make.

MacApp includes object classes with methods that handle events in a simple, general way. Your job when you are creating a MacApp application is to create new classes that override the methods you want to implement differently. In order to create a working MacApp application, you must override some classes and methods, and may override others if you desire to change the way that they function.

3.1 Creating objects: an overview

Should this section go here? Or maybe in "Object-Oriented Concepts?"

Whenever you create an object in MacApp, you must make an actual object instance of an object class. The object instance then exists in memory until you remove it. In order to create an object instance, you generally take the following steps:

1. Declare a variable that will reference the object. the object reference variable. (Remember that in MacApp object references are actually handles pointing to the object.)
2. Use the Object Pascal procedure New to allocate an object for it to reference. This works much the same way as using New to allocate memory for a pointer to reference. The New routine, however, can fail. For example, there might not be enough memory to allocate for your object. In this case, New returns nil as the value in your object reference variable.
3. After you have attempted to instantiate your application object using the New procedure, you should check to see whether it worked or whether nil was returned. If nil was returned, then your application should fail gracefully. MacApp provides all of this functionality in the routine FailNil. You should call FailNil every time you use the New procedure. If the New procedure failed, your application has run out of memory and your request cannot be satisfied.

Chapter X examines some methods of memory management to avoid running out of memory.

3.2 Creating an application

When you create an application in MacApp, you fill in the main routine in which you intialize MacApp, create an application object , and then call that object's Run method. The Run method then calls MacApp's code, which receives and analyzes events, such as mouse clicks and key presses. After each event has been analyzed, MacApp's code selects a particular object to handle the event, and then sends that object a message requesting that it handle the event.

What MacApp provides for you for at each step, the actions you must take, and when MacApp will call the methods you define or override are summarized in Table X-1. Each of the steps is explained in detail in the recipes that follow.

■ **Table 15-1** Overview: creating a ~~document~~ application

Step	Your action:	Because:
1.	In your Main routine, call InitToolbox and supply the number of master pointers your application needs.	MacApp provides routines that initialize the Macintosh toolbox.
2.	If your application supports printing, call InitPrinting in your main routine.	MacApp provides routines that initialize printing.
3.	Assign an appropriate value to the kSignature constant in your interface file.	MacApp provides methods that call for a string constant to determine the application signature for an application object.
4.	Declare your own subclass of TApplication.	MacApp provides a TApplication class that, as a subclass of TEvtHandler, assigns events to the instances that need to handle them.
5.	Define your own initialization method as part of your TApplication subclass. Normally, your initialization should also call the IApplication method.	MacApp provides a TApplication .IApplication initialization method that initializes the fields of application objects inherited from TApplication.
6.	Instantiate your application class by declaring a global reference variable, calling the Object Pascal New procedure, and calling the MacApp FailNil procedure.	MacApp provides a FailNil procedure that checks if memory was allocated for the instance of an object.
7.	Initialize your application object by calling your initialization method.	MacApp provides **(anything here???)**
8.	Call the TApplication Run method.	MacApp provides a TApplication Run method that starts the main event processing loop.

Step 1 Initializing the Toolbox

InitToolbox initializes the Macintosh Toolbox, and many parts of MacApp, including support for the debugger and memory management. You must call this procedure (it is a standard Pascal procedure, not a method) at the beginning of your application's main routine.

InitToolbox also calls the Macintosh Memory Manager routine MoreMasters to ensure that enough master pointers have been allocated. MoreMasters allocates space for an extra forty (???) master pointers. InitToolbox calls MoreMasters as many times as you specify in the parameter callsToMoreMasters. For more details about the MoreMasters routine, see Chapter X, "Memory Management."

```
PROGRAM IconEdit;

USES
  {$LOAD MacIntf.LOAD}
    MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf,
  {$LOAD UMacApp.LOAD}
    UMAUtil, UViewCoords, UFailure, UMemory, UMenuSetup, UObject, UList,
UAssociation, UMacApp,
  {$LOAD}
    UPrinting,

    UIconEdit;

VAR
  gIconEditApplication:    TIconEditApplication;

BEGIN
  InitToolbox(8);   { Initialize ToolBox & MacApp with 8 calls to MoreMasters. }
  InitPrinting;                  { Initialize printing. }

  New(gIconEditApplication);     { Create a new TIconEditApplication object. }
  FailNIL(gIconEditApplication);{ Make sure it didn't fail. }
  gIconEditApplication.IIconEditApplication(kFileType);

  gIconEditApplication.Run;      { Run the application.  When it's done, exit. }
END.
```

Step 2 Initializing printing

Since not all MacApp applications print, InitToolbox doesn't initialize the MacApp printing facilities. If you want to use these printing facilities, take the following steps:

1. Include MacApp's UPrinting unit.
2. Call the InitPrinting routine, which initializes the MacApp printing facilities.

Step 3 Assigning the application signature and the main file type

The IApplication method takes one parameter: an OSType named itsMainFileType. OSType is a type defined by the Macintosh toolbox as:

```
TYPE OSType = PACKED ARRAY[1..4] OF CHAR;
```

OSType is the Pascal type for application signatures and file types. Some examples of valid OSTypes are 'MyAp' and 'docu'. To ensure that the Finder properly recognizes your application's documents, you must specify an OSType when you call IApplication. MacApp handles the rest. For example, if you call

```
IApplication('myfl');
```

then your application's main file type would be 'myfl'. Using this identifier, MacApp ensures that the Finder can now keep track of your application's documents. Under normal circumstances, you declare this identifier as the kFileType constant in your interface file.

You will send this constant as the parameter to IApplication. MacApp then ensures that your application and its documents are properly identified for the Finder.

Another important identifier that the Finder requires is the application signature. Like kFileType, you will also make this identifier a constant in your program, this time called kSignature. kSignature should also be an OSType; that is, shoudl also be a packed array of four characters, such as 'ICED'.

Step 4 Declaring a subclass of TApplication

MacApp provides the predefined application class TApplication. You must define your own subclass of TApplication, and use that subclass to create your application object instance.

- Note: Defining a subclass in this way allows your application object to inherit all of the functionality that comes with the TApplication class, but also allows you to alter that functionality to suit the purposes of your specific application. For more information on subclasses and overrides, see Chapter X, "Object-Oriented Concepts," and *Introduction to MacApp.*

To define a subclass of TApplication, called TIconEditApplication, for example, you must at least include lines like the following in your interface file:

```
TIconEditApplication = OBJECT(TApplication)

END;
```

Step 5 Defining your application initialization method

To create an initialization routine for your application, take the following steps:

1. Declare your own initialization method
2. Define the behavior of your initialization method. Normally, you will call the initialization method for IApplication, to initialize the fields for that object.

The following sample code from UIconEdit.inc1.p illustrates these steps.

```
{------------------------------------------------------------------------}
{....in the interface unit}
TIconEditApplication = OBJECT(TApplication)
  procedure TIconEditApplication.IApplication(itsMainFileType : OSType);

{....in the implementation unit}
PROCEDURE TIconEditApplication.IIconEditApplication(itsMainFileType: OSType);
BEGIN
  IApplication(itsMainFileType);
END;
{------------------------------------------------------------------------}
```

Step 6 Instantiating your application class

In your main program, you must make an actual object instance of the application. To do so, take the following steps:

1. Declare an object reference variable. Since this variable needs to be accessed in the main routine, it must be a global variable.
2. Allocate an object for the variable to reference by using the Object Pascal procedure New.
3. Call the MacApp FailNil procedure to check if the New routine worked.

For more information about making an actual object instance, see Chapter X, "Where in heck do we want to put this information?"

Step 7 Calling your initialization method

Should this information be included in defining?

Step 8 Calling the Run method

When you call this method, you hand control of your program over to MacApp. The Run method basically checks for a few strange situations and then calls TApplication.MainEventLoop. From there, MacApp code takes over, polling for events, analyzing what type of event was found, determining which object should handle the event, and then passing the event to that object. This pattern then repeats until the Quit command is received.

You call the Run method of your application object as the last thing in your main routine. MacApp handles the flow of control after that. After you call Run, MacApp handles most events for you, and you only have to create objects to handle the events that MacApp's code cannot handle alone.

Continuing from here

*****Forward references to the rest of the chapter.*****

Define your menu resources.see Chapter X, "Menus,"

Create your documents and display views of those documents; see Chapter X, "Documents," Chapter X, "Views," and Chapter X, "Windows."

Building your application; see Chapter X, "MPW and MacApp," and what other *****MacApp documents?*****

3.3 Opening an application without opening a document

MacApp's default behavior is to open a document named "Untitled" whenever an application is opened. To change this default behavior, you need to override TApplication.HandleFinderRequest ***and do what????***

Chapter 5 Browser

Left just in case, or is it legitimate to recommend MacApp Developer's association, or can I give tips for using MPW as a browser?

Chapter 6 **The Clipboard and Cut, Copy, and Paste**

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.

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).

Your application 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 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, 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 you leave the application, it gets a chance to convert the information in your private scrap to the two common forms of the desk scrap. (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.

Creating a Clipboard view

*****definition of Clipboard view*****

Clipboard views commonly have documents to handle the data they show. However, that is not required. (cross-refer to views). 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.

Run-time summary of creating a clipboard view

You create a Clipboard view 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.

Figure 6-1 provides a summary of MacApp's actions at runtime when a new document is to be created.

- **Figure 6-1** MacApp's actions in relationship to this recipe

Figure TBD; for example see Chapter 14, "Documents"

Overview of your responsibilities

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 6-X. Each of the steps is explained in detail in the recipes that follow.

This section also assumes that you have already created an instance of an application object. For more information, see Chapter X, "Applications."

■ **Table 6-1** Overview: creating a document

Step	Your action:	Because:
1.	Define a handle type.	???
2.	Define a resource type.	???
3.	etc.	etc.

Table TBD; for example see Chapter 14, "Documents"

Step 1 Define a handle type

To define a handle type for your Clipboard data type, declare two pointers in your ? file. For example:

```
YourTypeOnClipboard = ^PYourTypeOnClipboard;
PYourTypeOnClipboard = ^YourClipType;
```

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.)

Step 2 Define a resource type

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 Apple Developer Technical Support to prevent duplication.

Step 3. Override MakeViewForAlienClipboard

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.

- Note: 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.

The interface for the MakeViewForAlienClipboard 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.

Step 4. Override methods for your Clipboard view type

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.

Step 5. Override the ContainsClipType method

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.

Step 6. Override the GivePasteData method

GivePasteData is called to get data from the Clipboard. If the data to be pasted is in your application's private scrap, you need to override this method.

- If you want to get data from the public scrap, you don't have to override this method, since it is declared and implemented for TView.

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.

The following sample code from UIconEdit.incl.p illustrates these steps.

```
{-------------------------------------------------------------------}
BEGIN
  IF gGotClipType THEN
    BEGIN
    dataType := gPrefClipType;

    err := gClipView.GivePasteData(aDataHandle, dataType);

    IF err < 0 THEN
      Failure(err, 0);
    END
  ELSE
    BEGIN
{$IFC qDebug}
    ProgramBreak('GetDataToPaste called when gGotClipType was FALSE');
{$ENDC}
    END;

  GetDataToPaste := err;
END;
```

{----------}

Step 7. Override TView.WriteToDeskScrap for your Clipboard view

To enable other programs to receive Clipboard data from your application, override TView.WriteToDeskScrap (which has no parameters) for your Clipboard view.

- *Note:* Generally, the Clipboard view is 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, MacApp ****right?**** calls WriteToDeskScrap 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.

Step 8. Create a Clipboard document, if desired

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.)

Step 9. Add a Show Clipboard menu item to your resource file

You need to have one item for the Clipboard in the resource file: the Show Clipboard menu item. ***Cross-reference to correct place?***

*****following code will be cut back into the individual steps*****

```
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;
```

Continuing from here

You will usually want to implement the Cut, Copy, and Paste commands in your application to allow the user to use the clipboard to transfer data from and to applications and desk accessories. The following sections discuss how you support the Cut, Copy, and Paste commands.

Supporting Cut and Copy commands

Cut and Copy commands are generally handled by a single type of command object. The next section 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.

To support the Cut and Copy commands, take the following steps:

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.

Supporting the Paste command

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).

To support the Paste commands, take the following steps:

Step 1 Call the global procedure CanPaste

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.

Step 2 Define and create a paste command object

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.

When your DoMenuCommand method finds the command number cPasteCreate, create a paste command object . 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.

Step 3 Retrieve the data to be pasted

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 "Supporting The Clipboard" earlier in this chapter for details on implementing that method.

Continuing from here

Cut, Copy, and Paste operations should almost always be undoable. See Chapter X, "Undoing" for more information.

See Chapter X, "Menus," for more information on commands.

Supporting a private scrap type

*****Anything for here?*****

Chapter 7 Collections

Invented terminology; premise is that tagging items to be operated on as a collection is an important part of the capabilities of MacApp.

*****No recipes yet*****

Chapter 8 Color

No recipes yet

Chapter 9 **Compatibility**

MacApp 2.0 does not support the Macintosh XL, but does support 64K ROMs and all Macintosh models from 512K on up through SE, II, and IIx.

Checking system software

Checking hardware

Future compatibility rules

Converting from MacApp 1.1 to MacApp 2.0

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

.i.converting MacApp 1.1 applications: 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 the *MacApp General Reference.*

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. For more information on the Fields method, see Chapter X, "Debugging."

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

.i.documents: changes from MacApp 1.1;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 X. 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 X.

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

.i.views: changes from MacApp 1.1;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 X, "Scrolling," for more information.

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 X, "Dialogs", and the sample programs.

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.

Chapter 10 Controls and Control Views

*****No recipes yet*****

Chapter 11 Cursors

*****No recipes yet*****

Changing the Cursor Shape

Cursor region

From Curt, what about the cursor region?

Chapter 12 Debugging in MacApp

This chapter describes some specific MacApp debugging techniques. For more complete description of how to to use the Debugger, see the *MacApp General Reference.*

When an application is running, an application object and any number of document objects exist in memory. MacApp also includes a facility called the Inspector that you can use to examine the contents of these objects as the application is running. The Inspector is described in the *MacApp General Reference.*

Is that where it is?

This chapter describes some specific MacApp debugging techniques. For more complete description of how to to use the Debugger, see the *MacApp General Reference.*

Writing a Fields Method

When the Inspector is listing the fields of an object instance, it does so by calling that object's Fields method. The classes that come with MacApp have Fields methods defined for them, and that is why the Inspector works for all MacApp-defined fields. If you want todefine your own classes (or subclasses like TIconEditApplication and TIconDocument) then

Important

Whenever you create an object class, you should always include a Fields method. This way, you will always be able to inspect all of the fields of your object instances.

Step 1 Declare a Fields method for your document class

In your interface file, declare a Fields method as one of the methods in your document class. Since it is an override method, you must use the predefined interface:

```
PROCEDURE TIconDocument.Fields (PROCEDURE DoToField (fieldname : Str255;
                                                     fieldAddr: Ptr;
                                                     fieldType: integer)); OVERRIDE;
```

Step 2 Define the Fields method

In your implementation file, define the Fields method by taking the following steps.

1. Using the DoToField routine, display the name of the class.
2. Then, also using the DoToField routine, display the name of each field unique to TIconDocument.
3. Call the inherited version of Fields.

When the Inspector calls the Fields method of an object, it sends one parameter—a routine that you can use to print the value of the fields. This routine is called DoToField.

When the Inspector window calls this Fields method for some particular document instance, it will send a DoToField routine as the parameter. It is your job to call this DoToField routine, which takes three parameters as follows:

- The name of the field. This is a Pascal string representing the name of the field. You can actually use any string that you like.
- The address of the field. You can find this by using the Pascal @ operator. DoToField uses this parameter to find the value of the field
- An integer representing the type of the field. DoToField uses this to decide how to display the contents of the field (for example, whether to display an integer or an address).

The body of your method should make these calls:

```
DoToField('TIconDocument', NIL, bClass);
DoToField('fIconBitMap', @fIconBitMap, bHandle);
INHERITED Fields(DoToField);;
```

Step 3 Call Inherited DoToFields

This method should always call the inherited Fields method to be sure that any inherited fields are displayed as well.

```
PROCEDURE TIconDocument.Fields (PROCEDURE DoToField (fieldname : Str255;
                                                     fieldAddr: Ptr;
                                                     fieldType: integer)); OVERRIDE;
BEGIN
   DoToField('fIconBitMap', @fIconBitmap, bHandle);

   INHERITED Fields(DoToField);
END;
```

MacApp has many predefined constants that you can use for this parameter: bInteger, bRect, ???. Each of these integers instructs the DoToField routine to display the value in a slightly different way. For example, rectangles are displayed like this: (12, 12, 40, 50), while integers are displayed like this: 57. In the first line of your fields method, you can display the class name of the object. For instance:

```
DoToField('TIconDocument', NIL, bClass);
```

Notice that the "field name" parameter is actually the name of the class, the address parameter is NIL, and the type parameter is bClass.

Step 4 Conditionally compile the Fields method

Heading sounds funny—any suggestions?l

Surrround both the Fields declaration (in the interface file) and the Fields definition (in the implementation file) with the conditional compilation flags:

```
{$IFC qDebug}
      { code to be conditionally compiled... }
{$ENDC}
```

Chapter 13 **Dialogs**

A dialog box in the traditional Macintosh sense is a specialized type of window designed to display information to and take input from the user. Such boxes can be **modeless**, meaning that the user does not have to respond to the box to continue, or **modal**, meaning that the user must respond to continue.

In MacApp 2.0, there is no distinction between dialog boxes and windows, and you construct a dialog box in the same way as you do any other MacApp window—by constructing a hierarchy of views. The UDialog unit provides a set of predefined view classes that can be used in *any* window. To create the illusion of a dialog box using MacApp, you use the UDialog unit to create objects that can request information from the user. Some of the elements traditionally found in dialog boxes and supported by the UDialog unit are as follows:

- modeless dialog boxes
- modal dialog boxes, including the features of tabbing between text fields and implementing default and cancel buttons
- views that can take in text and validate the text
- views that can take in numbers and validate the numbers, called **number text items**
- radio buttons, push buttons, and check boxes; that is, views that represent the Macintosh Toobox Control Manager controls
- views that represent pictures, icons, and static text—all of which can behave like buttons, if desired
- groups of radio buttons, called radio **clusters,** which define dependencies among the controls
- pop-up menus
- scrollable lists

To handle your application's dialogs, you can:

- Create a modeless dialog
- Create a modal dialog
- etc. **(***on through all the recipes provided by the chapter***)**

Creating a modeless dialog

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.

Run-time summary of creating a modeless dialog

MacApp asks your application to create a modeless dialog by *****doing what******

Figure 13-1 provides a summary of MacApp's actions at runtime when a modeless dialog is to be created.

- **Figure 13-1** MacApp's actions when creating a modeless dialog

Figure TBD; for example see Chapter 14, "Documents"

Overview of your responsibilities

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 13-1. Each of the steps is explained in detail in the recipes that follow.

This section also assumes that you have already created an instance of an application object. For more information, see Chapter X, "Applications."

Table 13-1 Overview: creating a modeless dialog

Step	Your action:	Because:
1.	Include UDialog in the USES statement of your main unit.	MacApp provides the special UDialog unit that contains the user interface objects.
2.	Call InitUDialog.	MacApp provides the initialization method for the UDialog unit.
3.	Define your dialog view as a subclass of TDialogView.	MacApp provides the TDialogView class to serve as a superview for dialog items.
4.	Create the dialog view and the dialog window	***Need more help in this step***
5.	Install the controls.	***Need more help in this step***
6.	Set the window title	***Need more help in this step***
7.	Launch the window	***Need more help in this step***
8.	Override the DoChoice method and provide the identity of the item selected by the user.	MacApp uses a parameter of DoChoice to find out which dialog item has been selected.
9	Implement a command object to take the action, if desired.	If you implement a command object that will take the desired action, the action the user takes can be undoable.

Step 1 Include UDialog

MacApp provides a special UDialog unit that contains the user interface objects. To take advantage of those objects, include UDialog in the USES statement of your main unit.

The following sample code from MDemoDialogs.p includes UDialog, among others

```
{-------------------------------------------------------------------------}
USES

   {$LOAD MacIntf.LOAD}
      MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf,
   {$LOAD UMacApp.LOAD}
      UMAUtil, UViewCoords, UFailure, UMemory, UMenuSetup, UObject, UList,
      UAssociation, UMacApp,
   {$LOAD}
      UTEView,
      UDialog,
      UGridView,
      UDemoDialogs;
{-------------------------------------------------------------------------}
```

Step 2 Call InitUDialog

You must initialize the UDialog objects before you use them by calling InitUDialog first.

Any rules about where this call must be made?

The following sample code from UDemoDialogs.inc1.p initializes UDialog:

```
{-------------------------------------------------------------------------}
BEGIN
   IApplication(itsMainFileType);
   InitUDialog;
   {other initialization methods}
END;
{-------------------------------------------------------------------------}
```

Step 3 Define your dialog view as a subclass of TDialogView

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. You do not have to override any other methods of TDialogView.

Step 4 Create the dialog view and the dialog window

*****Need more help in this step*****

Step 5 Install the controls.

*****Need more help in this step*****

Step 6 Set the window title

*****Need more help in this step*****

Step 7 Launch the window

*****Need more help in this step*****

Couldn't find an example in UDemoDialogs.incl.p. Is there one someplace?

```
{-------------------------------------------------------------------------}
```

```
PROCEDURE TYourType.PoseModelessDialog;
VAR   aDialogView: TDialogView;
   aWindow: TWindow;
BEGIN
   { Use this approach when using view resources.}
   aWindow := NewTemplateWindow(aRsrcID, NIL);
   { 'DLOG' is an arbitrary identifier you define in your resource file}
   aDialogView := TDialogView(aWindow.FindSubView('DLOG'));
   aWindow.Open;
END.
{-------------------------------------------------------------------------}
```

For a discussion of dialog items see the "Using Dialog Items" section.

Notice that the PoseModelessDialog method opens the dialog window and then returns without waiting for a response from the user.

Step 8 Override the DoChoice method

MacApp uses a parameter of DoChoice to find out which dialog item has been selected. Your job is to set the XXXX parameter to the identity of the item selected by the user.

Step 9 Implement a command object, if desired

If you implement a command object that will take the desired action, the action the user takes can be undoable. For more information on command objects, see Chapter X, "Undoing."

Closing a modeless dialog

*****Is this recipe needed? There was some debate on MacApp.Tech$, followed by suggestions from Curt, Russ Wetmore, and Mike Cremer. What of the following should I**

include? (the stuff hasn't been rewritten very much)***

(from Curt Bianchi)

MacApp automatically closes windows when the window's CloseByUser method is called. If you wanted to do this in your application, you can call CloseByUser in the appropriate DoChoice to close the window in response to clicking a button. When CloseByUser returns, the window and its subviews are free'd but the methods are not. Thus, you can't refer to any of the fields or instance variables of any of the views after the CloseByUser call but the code will still execute and DoChoice will return to its caller and so on. The important point is to not refer to any fields of objects that will have been free'd as a result of calling CloseByUser.

Another thing you may want to do is call the dialog view's CanDismiss method first, and call CloseByUser if CanDismiss returns true.

(from Russ Wetmore)

It's up to the method that opens the dialog to close it (that is, close the dialog window). In most cases, the sequence usually looks something like:

```
aWindow := NewTemplateWindow(aRsrcID, NIL);
dismisser := TDialogView(aWindow.FindSubView('DLOG')).PoseModally;
CASE dismisser OF
    { handle dismisser here }
END; { CASE }
aWindow.Close;
```

Now, if you want a button to dismiss a dialog (in other words, end a PoseModally session) then all you have to do is call TDialogView.DismissDialog in the button's DoChoice method. The next time PoseModally tries to field an event, it will then exit back to its caller.

The reason the window is not closed automatically is in case you want to be able to re-enter the dialog, based on the dismisser or on actions you take based on a dismissal.

from Mike Cremer

Another solution is to not actually close the window at all. Specify "dontFreeOnClosing" in the Rez template of the window (or in ViewEdit, leave the "Free on closing" checkbox empty). The DialogView in the window should have an "ok" and "cancel" option, both of which dismiss the dialog. And the DismissDialog method should call GetWindow.Close. If you look at the TWindow.Close method, you'll note that it does the following: ...

```
    Show(FALSE, kRedraw);
    Activate(FALSE);
    IF fFreeOnClosing THEN
        FREE;
...
```

So as long as your window is not "freeOnClosing," MacApp won't attempt to free all the subviews, including the TDialogView where you are handling the mousedown. So, make sure you keep an application-level or global-level reference to the dialog (say, gFindDialog). Initialize it to NIL, then whenever the user selects a command that causes the dialog to pop into existence, check the global objRef to see if it is NIL. If it is, create the object. If it isn't, perform and cleanups (clearing out dialog edit text fields, resetting controls, etc.) and Select/Show the window (both are necessary).

In this case, the object still exists even when it may no longer be needed. However, the window appears exactly where the user put it last. So, If you are using modeless dialogs for such things as find/replace type commands, this may be worth trying.

Creating a modal dialog

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.

Does this mean the user or the application,can't deactivate the dialog?.

- Note: Because modal dialogs force the user to take a specific action, you should use them sparingly in your application. See *Human Interface Guidelines: the Apple Desktop Interface*, for more information.

Run-time summary of creating a modal dialog

MacApp asks your application to create a modal dialog by ***doing what****

Figure 13-2 provides a summary of MacApp's actions at runtime when a modeless dialog is to be created.

■ **Figure 13-2** MacApp's actions when creating a modal dialog

figure TBD

Overview of your responsibilities

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 13-2. Each of the steps is explained in detail in the recipes that follow.

This section also assumes that you have already created an instance of an application object. For more information, see Chapter X, "Applications."

Table X-1 Overview: creating a modal dialog

Step	Your action:	Because:
1.	Include UDialog in the USES statement of your main unit.	MacApp provides the special UDialog unit that contains the user interface objects.
2.	Call InitUDialog.	MacApp provides the initialization method for the UDialog unit.
3.	Define your dialog view as a subclass of TDialogView.	MacApp provides the TDialogView class to serve as a superview for dialog items.

Step 1 Include UDialog

The dialog-handling capabilities of MacApp or in the UDialog unit, so include UDialog in the USES statement of your main unit.

Step 2 Define a method that displays the dialog

This display method should be for the object type that issues 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.

- *Note:* Because you rarely customize TWindow, the display method is rarely a window method.

The interface to the display 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.

Step 3 Implement PoseModalDialog

.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 with 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).

The following sample code from UDemoDialogs.incl.p illustrates this step:

```
{------------------------------------------------------------------}
   PROCEDURE MakeSaveDialog;

   VAR
```

```
      aWindow:          TWindow;
      dismisser:        IDType;

   BEGIN
      aWindow := NewTemplateWindow(aCmdNumber, NIL);
      dismisser := TDialogView(aWindow.FindSubView('DLOG')).PoseModally;

{$IFC qDebug}
      IF dismisser = 'yes ' THEN
            WRITELN('The user said yes.')
      ELSE IF dismisser = 'no  ' THEN
            WRITELN('The user said no.')
      ELSE IF dismisser = 'cncl' THEN
            WRITELN('The user cancelled the dialog.')
      ELSE
            WRITELN('I don''t know how the user responded');
{$ENDC}
      aWindow.Close;
   END;
{-------------------------------------------------------------------------}
```

Using dialog items

Every dialog view is made up of a list of dialog items

The items don't have to completely make up the view, do they? That is, can the view contain other things?

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.

The predefined dialog item types from UDialog are as follows:

- 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. 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.

Is there an algorithm that can be presented?

Samples needed from DemoDialogs or IconEdit.

Creating buttons

*****Recipe needed?*****

Creating radio buttons

Recipe needed?

Making a default button

Recipe needed?

Creating check boxes

Recipe needed?

Handling scroll bars

Scroll bars and scrolling views in dialogs are no different than any other scroll bars or scrolling views. For more information, see Chapter X, "Scrolling."

Handling double clicks in scrolling list in a modal dialog

Recipe needed?

from Richard Rodseth, in response to a MacAppTech$ question

If you subclass TTextListView and override DoMouseCommand, you can detect the double click by looking at info.theClickCount. I would then call SELF.DoChoice with some constant you define, eg. mDoubleClickInList. This message will get forwarded to your DialogView's DoChoice method, which can respond appropriately. I'm not sure what that is in this case. You might just call FlashControl to highlight the button, and DismissDialog, as is done in TDialogView.HandleCR.

In any case, I think it's good style to isolate the interaction between controls/views in a dialog in the DialogView's DoChoice method. You could argue that all views should call DoChoice, with special constants for things like single and double clicks, perhaps even passing the event info. This could function sort of like Smalltalk's dependency/self-changed mechanism. Packages of reusable dialog elements could be produced, which could be used in composing dialogs, without any subclassing necessary, other than DoChoice, to manage the side-effects. In your case, you wouldn't have to subclass TTextListView. On the other hand, views would always pay the price, even when they didn't have a side-effect, or weren't even in a dialog. Comments anyone?

Continuing from here

Because MacApp 2.0 treats a dialog exactly as a window, you can use any of the techniques descriibed in Chapter X, "Views," and Chapter X, "Windows."

Chapter 14 **Documents and Files**

In the MacApp world, a **document** is used to 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. This data is stored on disk in the form of **files**.

In MacApp applications, this data is read from disk and stored in memory by a particular instance of a document object. To help you create that document object, MacApp provides a class called TDocument, which is an immediate descendant of TEvtHandler. As a descendant of that class, TDocument can respond to all events that affect the entire document, such as Save, Save as, Save a copy, and Revert to saved. In addition to inherited **event-handler** fields, it has fields that record the title, creator, type, and date of the document.

Your application can have two (or more) different types of documents, can allow more than one document to be open at a time, and can allow documents of different types to be open at the same time.

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.

To handle your application's documents, you can:

- Create a new document
- Save the data into a disk file and read it back
- Open an existing document
- Close a document
- etc. **(***on through all the recipes provided by the section***)**

This chapter describes in detail the steps you need to take to accomplish these tasks.

Creating a document

To create a document, you never instantiate the TDocument class directly. Instead you create one descendant of TDocument for each different type of document that your application can handle. You can then customize each descendant class with fields for the 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 dynamically creates new instances of these classes.

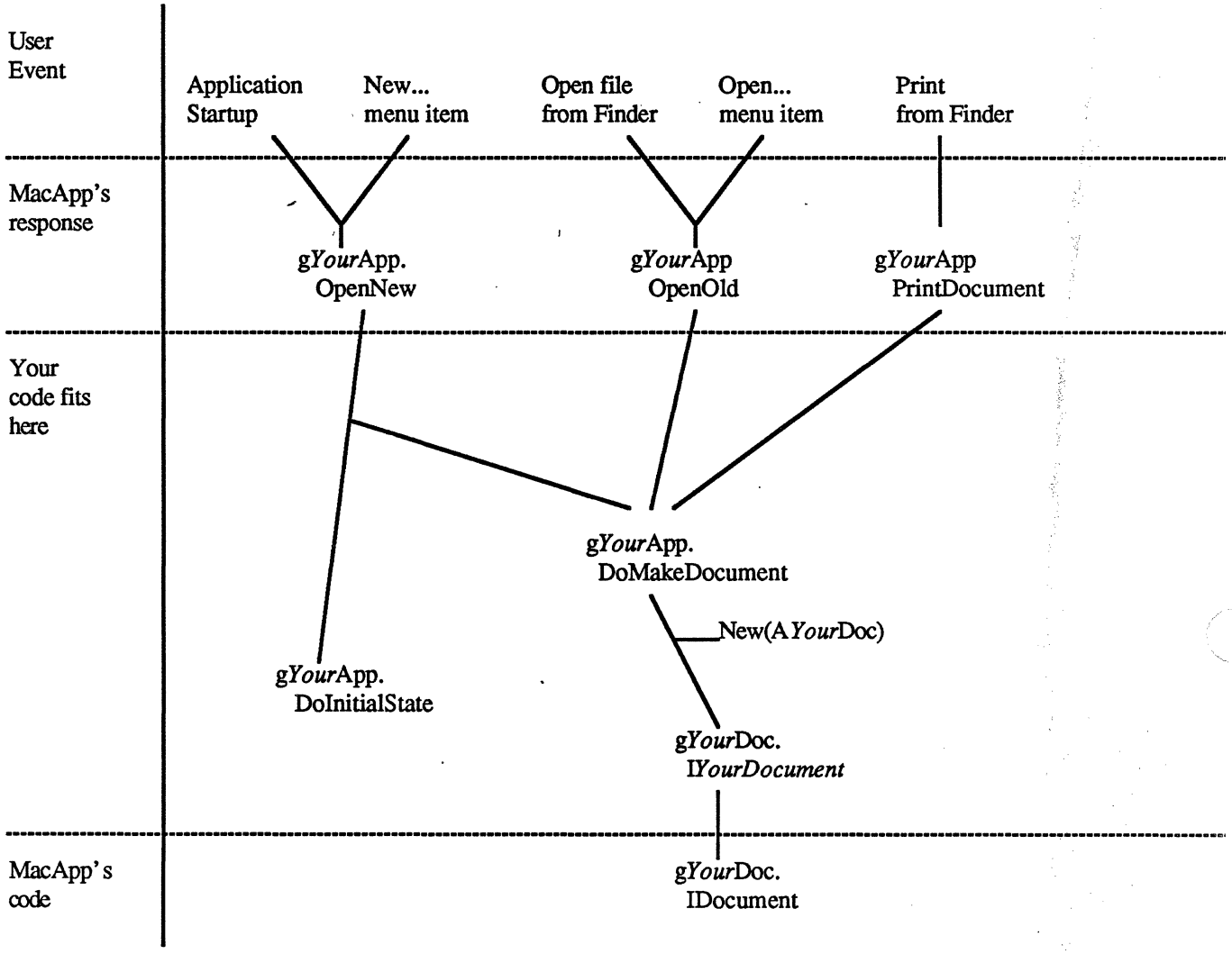
Run-time summary of creating a document

MacApp asks your application to create a new document by sending a message to the TApplication.DoMakeDocuments method in the specific instance of your application. As default behavior, this happens when the user double-clicks on the application's icon or gives a New command.

- *Note:* To change MacApp's default behavior when the user double-clicks on an icon, you need to override TApplication.HandleFinderRequest. See the "Opening an Application Without Opening a Document" section in Chapter X, "Applications" for more information.

Figure 15-X provides a summary of MacApp's actions at runtime when a new document is to be created.

■ **Figure 15-1** MacApp's actions in relationship to this recipe



Overview of your responsibilities

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 15-X. Each of the steps is explained in detail in the recipes that follow.

This section also assumes that you have already created an instance of an application object. For more information, see Chapter X, "Applications."

■ **Table 15-1** Overview: creating a document

Step	Your action:	Because:
1.	Override TApplication.DoMakeDocuments and supply a method to create a document object.	MacApp calls this method whenever the application creates a new object, and the default TApplication.DoMakeDocuments method is empty.
2.	Declare a subclass of TDocument.	MacApp provides a TDocument class that handles the invocation of the New, Open, Close, Save, Save as, Save a copy, and Revert to saved commands.
3.	Define your own initialization method for your subclass. Your application object must call this method whenever the application creates a new document or opens an existing document. Normally, your initialization should call the IDocument method to initialize the inherited data.	MacApp doesn't know about your subclass and therefore can't initialize it. MacApp does provide a TDocument.IDocument initialization method that initializes the fields of document objects inherited from TDocument.
4.	Override the Free method to free any allocated data belonging to your document. Include a call to the inherited Free method.	MacApp calls the document object's Free method whenever a document is to be closed. MacApp provides a TDocument.Free method that closes any disk files and sends messages to associated window objects to free themselves.
5.	Override the DoInitialState method and supply the methods necessary to set the state of the new, "blank" document.	MacApp calls that method whenever the application object creates a new, unsaved document, and the default TDocument.DoInitialState method is empty.

6. Provide a Fields method that is conditionally compiled and calls the DoToField routine for each field in the document object.	You can then use the Inspector debugging tool to show the names and values of the fields of an object instance.

Step 1 Declare the file type as a constant

MacApp determines the file type for each different type of document for your application by examining a constant you declare in the interface part of your object unit. The file type is generally stored as a constant k*Your*FileType, is always a four-character string, and should be of type OSType, which is a packed array of four characters.

*****How and when does MacApp examine this constant????******

For example, you might define k*Your*FileType 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 Apple Computer Developer Technical Support to ensure that they are unique.

Step 2 Overriding DoMakeDocuments

MacApp calls your application's DoMakeDocument method whenever a new document needs to be created (typically when the user double-clicks on the application icon or gives a New or Open command). Because MacApp cannot know what kind of document object is to be created, the default DoMakeDocument method is empty. Therefore, for your application to respond to the request for a new document, it must create a document object and return a reference to it.

- *Note:* The DoMakeDocument method belongs to the application object because, when DoMakeDocument is called, no document object exists.

To override the DoMakeDocuments method, take the following steps:

1. In the interface file, declare a function method for the appropriate document type.

   If the documents for the application are all of one type, you can ignore the cmdNumber parameter. If the application has more than one document type, you can use the command number to indicate which type of document needs to be created. See "Creating Documents of More Than One Type" in this chapter.
2. In the implementation file, for the function method, define a variable to serve as a reference to the document object.
3. Create an instance of a T*Your*Document object by calling the New routine with the document reference as the parameter.
4. Use the global FailNil procedure to check for errors (see method [7.11]).
5. Initialize the instance of the document object using your own initialization method. For the details on how to write that initialization method, see "Defining your document initialization method" later in this section.
6. Return a reference handle to the object.
7. Save the reference in a global variable called gDocList, which is a list of all the open document objects. That way, at any point in the future, your application object can send messages to any document object.

The following sample code from UIconEdit.inc1.p creates and initializes an instance of a document object named anIconDocument:

```
{------------------------------------------------------------------------}
FUNCTION  TIconEditApplication.DoMakeDocument
       (itsCmdNumber:  CmdNumber):TDocument;  OVERRIDE;
  VAR
    anIconDocument:  TIconDocument;
BEGIN
  New(anIconDocument);      { Create a TIconDcoument object }
  FailNIL(anIconDocument);        { Make sure it succeeds }
  anIconDocument.IIconDocument;   { Initialize the document }
  DoMakeDocument  :=  anIconDocument;      { Return a reference to it }
END;
{------------------------------------------------------------------------}
```

Step 3 Declaring a subclass of TDocument

To help you build your specific document objects, MacApp provides a general TDocument class that automatically handles the New command. To take advantage of that capability, but still allow the flexibility to add your document's unique behavior, you must declare your document as a subclass of TDocument. MacApp can then provide the capabilities of TDocument and add your methods to the behavior of T*Your*Document.

To declare T*Your*Document as a subclass of TDocument, take the following steps:

1. In your interface file, declare each of your document classes as a subclass of TDocument, and add any fields that you need to access the data of each document, such as a handle to the data.
2. In the declaration of each document class, declare four methods: your initialization method, an override of the DoInitialState method, an override of the Free method, and a Fields method.

The following sample code from UIconEdit.incl.p illustrates these steps:

```
{------------------------------------------------------------------}
  TIconDocument = OBJECT(TDocument)
    fIconBitMap:          Handle;       { A handle to the icon's bit map. }
    PROCEDURE  TIconDocument.IIconDocument;     { Initializes the document. }
    PROCEDURE  TIconDocument.Free; OVERRIDE;    { Frees the fIconBitMap handle. }
    PROCEDURE  TIconDocument.DoInitialState; OVERRIDE;
      { Sets the document's data to represent a "new" document. }
    ...       (Other methods for the document)
{$IFC qDebug}
    PROCEDURE  TIconDocument.Fields (PROCEDURE DoToField (fieldName: Str255;
                                                          fieldAddr: Ptr;
                                                          fieldType: INTEGER)); OVERRIDE;
{$ENDC}
    ...       (Other methods for the document)
    END;
{------------------------------------------------------------------}
```

The definition of the initialization method, the override definitions of the DoInitialState and Free methods, the definition of the Fields method are discussed in the following steps.

Step 4 Defining your document initialization method

Like most objects, documents should always be initialized before being used. You must define your document initialization method that your application object calls ***correct?*** whenever your DoMakeDocuments method creates a new document or opens an existing one (typically when the user has given a New or Open command).

- *Note:* See "Opening an Existing Document" later in this chapter for more details about that process. ***if that reference is there***

You define your own initialization method, instead of overriding IDocument, so that you can initialize anything else the document needs. To do so, take the following steps:

1. Initialize the state of any fields required to free the object, which usually means setting handles to NIL or ***what else***
2. Initialize the inherited data by calling IDocument and supplying the appropriate constants. The values you can supply for the constants are as follows:

`kFileType`	`???????`
`kSignature`	Already determined in the application object's methods
`kUsesDataFork`	Document uses the data fork
`NOT kUsesDataFork`	Document doesn't use the data fork
`kUsesRsrcFork`	Document uses the resource fork
`NOT kUsesRsrcFork`	Document doesn't use the resource fork
`kDataOpen`	Document keeps the data fork open; that is, the document is disk-based and does not keep the entire file in memory
`NOT kDataOpen`	Document doesn't keep the data fork open; that is, the document is not disk-based and keeps the entire file in memory
`kRsrcOpen`	Document keeps the resource fork open
`NOT kRsrcOpen`	Document doesn't keep the resource fork open

3. Perform any initialization that may fail, so that you can fail gracefully if it doesn't succeed.

The following sample code from UIconEdit.incl.p illustrates these steps.

```
{-------------------------------------------------------------------}
PROCEDURE TIconDocument.IIconDocument;
BEGIN
  fIconBitMap:= NIL;        { In case IDocument fails}

  IDocument( kFileType,     { TIconDcoument.Free will work OK. }
      kSignature,
      kUsesDataFork,        { Dcoument uses the data fork, }.
      NOT kUsesRsrcFork,    { but does not use the resource fork.}
      NOT kDataOpen,        { Don't keep the data fork open.}
      NOT kRsrcOpen);       { nor the resource fork.}

  fIconBitMap:= NewPermHandle(kIconSizeInBytes);{ Allocate handle for bitmap}
  FailNil(fIconBitMap)      { Fail if the handle won't allocate}
END;
{-------------------------------------------------------------------}
```

Note the use of the MacApp utility call FailNil, which fails if the specified handle cannot be allocated, and the use of the MacApp memory management call NewPermHandle to allocate the fIconBitMap handle instead of NewHandle. NewPermHandle is part of MacApp's memory-management support and differs from NewHandle in that it attempts to ensure that there is enough space for code resources and so on before allocating the handle. For more information about memory management, see Chapter X, "Memory Management" in this book.

Step 5 Defining your DoInitialState method

MacApp calls your DoInitialState method whenever a document is created with a New command. In your implementation file, override the DoInitialState method and supply any methods necessary to set the state of the document to be an unsaved, "blank" document.

Note that blank does not necessarily mean "empty", but rather means that the document comes up in a reasonable blank state. For example, in the case of an icon editor, you'll have to decide what a "new" icon means; that is, should the screen be blank or should a basic icon be given as a template.

The following sample code from UIconEdit.incl.p sets the value of the document's icon bit map to that of a seed icon in a resource file.

```
{-------------------------------------------------------------------}
PROCEDURE TIconDocument.DoInitialState; OVERRIDE;
```

```
{ This method is called to set the document's data to the "new" state, as when the
user}
{ chooses to open a new document instead of an existing one.  We set the value of
the    }
{ document's icon bit map to that of a "seed" icon in our resource file.
VAR
  seedIcon:          Handle;
BEGIN
  seedIcon := GetIcon(kSeedIconId);      { Get the seed icon resource.}

  IF seedIcon <> NIL THEN          { If we got the seed icon resource}
    BlockMove(seedIcon^, fIconBitMap^,{then copy it into the document's }
              kIconSizeInBytes) { …icon bitmap.                    }
  ELSE
    BEGIN                                  { Else report error to the debugger.}
{$IFC qDebug}
    ProgramBreak('Unable to get the seed icon resource.');
{$ENDC}
    END;
END;
{-------------------------------------------------------------------------}
```

By placing the initial state of the document in a resource, your application can change that state simply by changing the resource. For more information about using resources in MacApp, see Chapter X, "Resources," and Chapter X, "ViewEdit."

Step 5 Defining your Free method

Whenever a document is closed, such as when the user clicks in the close box or quits from the application when a document is open, MacApp calls the document object's own Free method. The TDocument.Free method knows how to close disk files and send a message to the associated windows to free themselves, but the method cannot know what memory you have allocated for the document. Therefore, you must take the following steps:

1. Set up your Free method to dispose of any memory that the document itself has allocated. For example, the following code fragment disposes of a handle to the fIconBitMap if the handle is not a nil object.
2. Call the inherited Free method to allow MacApp to free up other resources.

The following sample code from UIconEdit.incl.p illustrates these steps.

```
{-------------------------------------------------------------------}
PROCEDURE TIconDocument.Free; OVERRIDE;
BEGIN
  DisposIfHandle(fIconBitMap);    { Dispose of the bitmap if non-Nil.}
  INHERITED Free;
END;

{-------------------------------------------------------------------}
```

Note the use of the MacApp utility call DisposIfHandle. That call checks if the specified handle is NIL before disposing of it. For more information about memory management, see Chapter X in this book.

Step 7 Defining your Fields method

When the Inspector is listing the fields of an object instance, it does so by calling that object's Fields method. Whenever you create an object class, you should always include a Fields method that will be conditionally compiled in debug versions of your application. This way, you can inspect all of the fields of your object instances when debugging.

◆ *Note:* This step quickly reviews the Fields method. For more information, see the section "Writing a Fields Method" in Chapter X, "Debugging."

When the Inspector window calls this Fields method for some particular document instance, it sends a DoToField routine as the parameter. Your method call this DoToField routine for each field in the TIconDocument object, and then call the inherited version of the Fields method.

To add a Fields method to your document class, take the following steps:

1. Declare a Fields method for your document class.

   In your interface file, declare a Fields method as one of the methods in your document class.

2. Define your Fields method.

   In your implementation file, define the Fields method.First, using the DoToField routine, supply the following parameters for each field unique to TIconDocument:

   - The name of the field. This is a Pascal string representing the name of the field. You can use any string that you like.

- The address of the field. You can find this by using the Pascal @ operator. DoToField uses this parameter to find the value of the field.
- An integer representing the type of the field. DoToField uses this to decide how to display the contents of the field (for example, whether to display an integer or an address).

Then, call the inherited version of Fields.

3. Make sure that the Fields method is conditionally compiled.

   Surrround both the Fields declaration (in the interface file) and the Fields definition (in the implementation file) with the conditional compilation flags.

The following sample code from UIconEdit.incl.p illustrates these steps.

```
{------------------------------------------------------------------------}
{$IFC qDebug}
PROCEDURE TIconDocument.Fields (PROCEDURE DoToField (fieldName: Str255;
                                                     fieldAddr: Ptr;
                                                     fieldType: INTEGER)); OVERRIDE;
BEGIN
  DoToField('TIconDocument', NIL, bClass);
  DoToField('fIconBitMap', @fIconBitMap, bHandle);
  INHERITED Fields(DoToField);
END;
{$ENDC}
{------------------------------------------------------------------------}
```

Continuing from here

Other recipes later in this chapter give you the capabilities to deal with the rest of the documents things, such as saving and restoring data.

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. For more information on those definitions, see Chapter XX, "View," and Chapter XX, "Windows."

In addition to reading and writing the disk files and creating and maintaining the views and windows associated with them, each document type'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.

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), you need to override TDocument.DoMenuCommand and TDocument.DoSetUpMenus. . For more information about menu commands, see Chapter XX, "Menus," and Chapter X "Undoing" recipe for details on DoMenuCommand.

Saving and restoring data

You save and restore data to and from files so that the user can save documents, open document icons, and open documents using the Open command and the directory dialog box.

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.

MacApp provides a general TDocument class that implements a framework for saving files and opening those files to read their contents . What TDocument doesn't implement is the code to read and write the data unique to a particular document. You do that by overriding TDocument methods.

In addition, you can save the print state and the display state so the user does not have to reestablish them each time the document is opened.

Following from Interim cookb still

The 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.

Run-time summary of saving and restoring data

Figure 15-X provides a summary of MacApp's actions at runtime when data is to be saved or restored.

■ **Figure 15-2** MacApp's actions when saving or restoring a document

TBD

Overview of your responsibilities

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 15-2. Each of the steps is explained in detail in the recipes that follow.

This recipe also assumes that you have already successfully created a document object. For more information, see the section "Creating a Document" earlier in this chapter.

■ **Table 15-2** Overview: saving and restoring data

Step	Your action:	Because:
1.	If desired, change the value of fSavePrintInfo.	MacApp checks fSavePrintInfo to determine whether to include space for the print record in its disk space calculations. By default, MacApp does not include space for the print record.
2.	If desired, change the value of fSaveInPlace.	MacApp checks the value of the fSaveInPlace field to decide whether or not to overwrite the original file when there isn't enough space for a copy. By default, MacApp doesn't overwrite the file if either the `keepSDataOpen` or the `keepSRsrcOpen` flags are true; otherwise it asks the user whether or not to overwrite the file.

3. Override DoNeedDiskSpace. You will usually call inherited DoNeedDiskSpace to save any information from the parent of your TDocument subclass.	MacApp uses DoNeedDiskSpace to estimate the amount of disk space required to save a document's data. MacApp calls ***correct?**** DoNeedDiskSpace just before a document is saved and uses the value returned by that method to check whether there is room on the disk to save the new file.
4. Override DoWrite for your document object.	MacApp calls document.DoWrite whenever the data needs to be saved.
5. Override DoRead for your document object.	An empty TDocument.DoRead method. MacApp calls this method whenever.

Step 1 If desired, change the value of fSavePrintInfo

By default, MacApp does not calculate the space needed for the print information record in its calculations for the amount of space needed for the document. If you want DoNeedDiskSpace to allow space for that record in its calculations, set the fSavePrintInfo field to TRUE in the I*Your*Document method.

Is this the style we want to use "your" methods?

Step 2 If desired, change the value of fSaveInPlace

MacApp also checks the value of the fSaveInPlace field to decide whether or not to overwrite the original file when there isn't enough space for a copy. By default, MacApp doesn't overwrite the file if either the `keepSDataOpen` or the `keepSRsrcOpen` flags are true; otherwise it asks the user whether or not to overwrite the file.

In the preceding paragraph, should the flags be replaced with English equivalents?

Rather than text, should we use the following code fragment?

MacApp uses the IDocument method to set the value of fSaveInPlace as follows:

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

To change that default behavior, set the value of the document's fSaveInPlace field in the *IYourDocument* method to one of the following

`sipNever`	don't overwrite original file
`sipAlways`	always overwrite original filethere is not enough space for a copy
`sipAskUser`	ask user whether to overwrite the original file when there is not enough space for a copy

Step 3 Override DoNeedDiskSpace

MacApp uses the DoNeedDiskSpace method to allow it to estimate the amount of disk space required to save a document's data. MacApp calls ***correct?**** DoNeedDiskSpace just before a document is saved and uses the value returned by that method to check whether there is room on the disk to save the new file without first deleting the old. Therefore, your override of the DoNeedDiskSpace method must return the total amount of disk space, in bytes, needed to store the data and resources for the document. . MacApp will add the number of dataForkBytes and rsrcForkBytes to the number of bytes rreturned to determine the total number of bytes needed to store the document.

△**Important** 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.. △

To override the document's DoNeedDiskSpace method, take the following steps:

1. In your override of DoNeedDiskSpace, call the document's inherited DoNeedDiskSpace so that MacApp can calculate the space needed to save the print state (if fSavePrintInfo is true), the overhead for the file, and, if you are using the resource fork, the overhead for the resource fork.
2. Include the necessary methods to calculate the disk space needed to store the number of bytes in the data fork and the resource fork of the document.

The following sample code from UIconEdit.inc1.p illustrates a typical DoNeedDiskSpace method

```
{-------------------------------------------------------------------------------}

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

BEGIN
  INHERITED DoNeedDiskSpace(dataForkBytes,      { In case parent class saves data.}
                            rsrcForkBytes);

  dataForkBytes := dataForkBytes +          { Add the size of the icon in bytes to }
                   kIconSizeInBytes;        { the number of bytes in the data fork.}
END;

{-------------------------------------------------------------------------------}
```

Notice that this method adds to the initial value of dataForkBytes. MacApp sets the initial values of the dataForkBytes and rsrcForkBytes parameters, and you should not reset them to 0.

Old interim Cookbook had following paragraph. Is it still valid? If so, does it go here?

△**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.. △

Step 4 Override DoWrite

When MacApp calls DoWrite, the data file has already been opened, so all you need to do is write the data. To write data to a file, override DoWrite for your document object and take the following steps:

1. Call inherited DoWrite to ensure that the print information is read if necessary, and pass the following:
   - The file reference number ***had this already been provided, and, if so, where?***
   - The makingCopy parameter, which is primarily for disk-based documents and indicates that DoWrite is being called to make a new copy of the file.***need more explanation of makingCopy***
2. Use the Macintosh File System routine FSWrite, which is passed a file reference number, the number of bytes to write, and a pointer to the buffer to be written.

Preceding adapted from DoRead; please check.

The following sample code from UIconEdit.inc1.p calls the inherited DoWrite method and sets the numberof bytes to be equal to the size of the icon in bytes:

```
{-------------------------------------------------------------------------------}
PROCEDURE TIconDocument.DoWrite (aRefNum: INTEGER; makingCopy: BOOLEAN); OVERRIDE;

VAR
  numberOfBytes:          LONGINT;

BEGIN
  INHERITED DoWrite(aRefNum, makingCopy); { In case parent class writes data.}

  numberOfBytes := kIconSizeInBytes;
  FailOSErr(FSWrite(aRefNum,numberOfBytes,{ Use Mac File System to write data }
                    fIconBitMap^));       { ...directly from the bitmap handle,}
                                          { ...fail if FSWrite returns an error}
END;
{-------------------------------------------------------------------------------}
```

Old interim Cookbook had following paragraph. Is it still valid?

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.

Step 5 Override DoRead

When MacApp calls DoRead, the data file has already been opened, so all you need to do is read the data. To read data from a file, override DoRead for your document object and take the following steps:

1. Call inherited DoReadto ensure that the print information is read if necessary, and pass the following:
   - The file reference number.
   - The rsrcExists parameter, which indicates***need more explanation of** rsrcExists ***
   - The forPrinting parameter, which indicates***need more explanation of** forPrinting ***
2. Use the Macintosh File System routine FSRead and pass the following:
   - The file reference number.
   - The number of bytes to read.
   - A pointer to the buffer in which to place the bytes.

The following sample code from UIconEdit.incl.p calls the inherited DoRead method and sets the numberof bytes to be equal to the size of the icon in bytes:

```
{-------------------------------------------------------------------------------}
PROCEDURE TIconDocument.DoRead (aRefNum: INTEGER; rsrcExists, forPrinting: BOOLEAN);
OVERRIDE;

VAR
  numberOfBytes:            LONGINT;

BEGIN
  INHERITED DoRead(aRefNum, rsrcExists,{ In case parent class reads data.}
                   forPrinting);

  numberOfBytes := kIconSizeInBytes;
  FailOSErr(FSRead(aRefNum, numberOfBytes,{ Use Mac File System to read data}
                   fIconBitMap^));        { ...directly into the bitmap handle,}
                                          { ...fail if FSRead returns an error.}
END;
{-------------------------------------------------------------------------------}
```

Notice that this case dereferences the pointer once, which results in a pointer to the 128 bytes reserved for the icon's bit map. This is a safe dereference of a handle because FSRead does not cause heap object relocation.

Is there a list somewhere of calls that don't cause heap object relocation?

Old interim Cookbook had following paragraph. Is it still valid and/or relevant?

Notice that the display state is not restored at this time. That is done when the window is created, with DoMakeWindows, or DoMakeViews. See "Saving the Display State" later in this chapter.

All the following stuff is from old cookbook. Still valid?

Saving different typs of items

Salvaged from old Cookbook. Is it still valid?

Assuming you have several different types of items, each a descendant of TItem and differentiated by a value that indicates the kind of item, take the following steps:

1. Add the following method to your document

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

2. Define a set of constants for the different kinds of items.
3. Give each object type in your document's data set a WriteTo method for the T*Your*Document.WriteTo method and a ReadFrom method for the T*Your*Document.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;
```

4. Assign an identifier value to each item, 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.

5. 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.

Opening an existing document

*****Is a recipe needed, given the information in Saving and Restoring Data?*****

Closing a document

*****Recipe needed?*****

Saving the display state

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.

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. Construct a data type to save the state information. Define this as a global data type, so you can refer to it in different methods.

   The following example defines a record called DisplayState.

```
{------------------------------------------------------------------}
```

```
{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;
{------------------------------------------------------------------------}
```

The leading between the steps is not big enough in the following steps. How should it look?

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 following example, both are fields of the document. The Boolean field fUseDisplayState is set to FALSE for a new document, or 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.)

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

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.)

   The following example sets fUseDisplayState to FALSE:

```
{------------------------------------------------------------------------}
{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 }
{------------------------------------------------------------------------}
```

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

```
{------------------------------------------------------------------------}
{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));
{-------------------------------------------------------------------}
```

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.

```
{-------------------------------------------------------------------}
{ 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;
{-------------------------------------------------------------------}
```

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

```
{-------------------------------------------------------------------}
{ 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;
{-------------------------------------------------------------------------}
```

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

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

{----------}

Importing and exporting data

Recipe needed?

Caching data

Recipe needed?

Handling multiple file types

Recipe needed?

Chapter 15 **Drawing and Highlighting**

Whenever you call QuickDraw routines, usually in your view's Draw method, you must tell QuickDraw where you want the drawing to occur. Before MacApp calls your view's Draw routine, it first tells QuickDraw that all subsequent drawing should be done relative to the upper-left corner of your view. This is called **focusing the view**. This way, when you call QuickDraw routines in your Draw method, the drawing always happens at the right place within your view. When a window needs to be redrawn, MacApp starts by focussing on the view instance at the top of the hierarchy—the window object, and then calling its Draw method. When the window has drawn itself, it in turn focusses and then draws each of its subviews. This process recurses until every view in the hierarcy has been drawn.

Drawing an object in a view

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.

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

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.
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;
```

Chapter 16 Error and Failure Handling

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. 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.)

Checking for failures

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.

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.

There's more algorithms ~~luring~~ in here somewhere!

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.

Step 2 Post your exception handler

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.

Step 3 Call FailNil, FailOSErr, FailMemErr, or FailResERr

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 global routines are as follows:

- FailNIL: This procedure tests whether the given pointer (or handle) is NIL and, if it is, calls Failure(memFullErr, 0). The interface is as follows:

```
PROCEDURE FailNIL(p: UNIV Ptr);
```

The p parameter is any pointer or handle (including object references).

- FailOSErr : This procedure checks whether the given OS error code signals an error and, if it does, calls Failure. The interface is as follows:

```
PROCEDURE FailOSErr(error: INTEGER);
```

The error parameter is an OS error code, presumably returned by an *Inside Macintosh* or language routine.

The FailOSErr procedure 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));
```

- FailMemError : This procedure checks whether there was a memory error and, if there was, calls Failure. The interface is as follows:

```
PROCEDURE FailMemError;
```

You generally call FailMemError after you attempt to allocate a new pointer or handle. It tests the value of MemError. If MemError <> noErr, it calls Failure(MemError, 0).

Step 4 Set the error message in your exception handler

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.

Step 5 Handle errors during the creation and initialization of objects

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:

1. Initialize any variables that your Free method needs to operate successfully. No action that can fail may be taken in this step.
2. Call the immediate ancestor's initialization method (if any). This may fail, in which case your Free method will be called.
3. 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.

Chapter 17 **Event Handling**

Translation from Macintosh events to MacApp events, and that MacApp takes care of low-level events for you

Every menu command is handled by some object. For example, the application object generally handles the Quit command, document objects handle the Save, Save As and Revert to Saved commands, and view objects handle commands such as Zoom In and Zoom Out. This set of objects, those that are eligible to handle menu commands, are called **event handlers**. The application object, document objects, and view and window objects are all event handlers—that is, they can respond to menu commands.

For further information on mouse-down events, see Chapter X, "Mouse operations." For further information on key-down events, see Chapter X, "Keyboard handling."

The event-handling classes

All of the event-handling classes (TApplication, TDocument and TView) are descendant classes of TEvtHandler. The TEvtHandler class defines the fields and methods necessary for an object to handle menu commands. Therefore, application, document, and view objects all inherit these fields and methods. A much-reduced view of the TEvtHandler class is given here:

```
TEvtHandler = OBJECT(TObject)
        fNextHandler : TEvtHandler;
        ...{other fields}...

        FUNCTION TEvtHandler.DoMenuCommand(aCmdNumber: CmdNumber) : TCommand;

        PROCEDURE TEvtHandler.DoSetupMenus;
        ...{other methods}...
END;
```

Therefore, all application, document, and view objects have an fNextHandler field and a DoMenuCommand method. This chapter discusses the significance of the fNextHandler field; for the significance of the DoMenuCommand method, see Chapter X, "Menus."

The Command Chain

Following two sections need to be combined

The command chain is the list of event handlers starting with the current target object and following the fNextHandler field of that current target object until you reach NIL . Since the target object changes as the user activates and deactivates windows, the command chain also changes. MacApp maintains the fNextHandler field for you in most cases. For view objects, the fNextHandler is usually the view's superview. For windows, this field is usually the window's associated document object. For documents, this field normally references the application object. The application object normally stores a NIL in this field.

When a MacApp application is running, there are usually many event-handling objects. For example, there is always an application object, and there could be a number of open documents, open windows and displayed views. Theoretically, however, the user's attention is focused on only of these event handlers at a time. This event handler is called the target object. MacApp keeps track of the target object by storing a reference to it in the global variable gTarget: VAR gTarget : TEvtHandler;

MacApp defines a global variable, gTarget, that refers to the event handler that initially receives menu commands and keystrokes. 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, called the command chain

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 *Introduction to MacApp* for a more complete description of how the command chain works.

Changing the command chain

By default, MacApp always makes gTarget reference the frontmost window object. If no windows are open, MacApp makes the application object the target object.

In many MacApp applications, however, it isn't necessarily the entire active window that is the focus of the user's attention, but rather the main view of that window. You can override this behavior to specify a different target object.

When a window has an important view, for example, you generally want to tell MacApp not to set gTarget to reference the window object when it is active, but rather to reference the main view inside the window—in this case, the TIconEditView view. This main view is called the target view of a window.

The following sample code from the IconEdit resource file illustrates this step

```
{-------------------------------------------------}
resource 'view' (kSampleWindowID, purgable) {
{
        root, 'WIND', {50, 20}, {100, 100}, sizeVariable, sizeVariable, shown, enabled,
        Window          {"TWindow", zoomDocProc, goAwayBox, resizable, modeless, ignoreFirstClick,
                         freeOnClosing, disposeOnFree, closesDocuemnt, openWithDocument,
                         dontAdaptToScreen, stagger, forceOnScreen, dontCenter,
                         noID,    /* The target view field. */
                        ""};

        'WIND', 'SMPL', {0, 0} {100, 100}, sizeVariable, sizeVariable, shown, enabled,
        View            {"TSampleView"}
        }
};
{-------------------------------------------------}
```

This view template defines a window object with one subview of class TSampleView. The view ID of the window is 'WIND' and the view ID of the subview is 'SMPL'. Notice that the target view field is set to noID, a predefined MacApp constant meaning no view—in this case specifying no target view.

When the window object described by the above template becomes the active window, MacApp looks at the contents of this field, and finds noID. Since there is no target view specified, MacApp will set gTarget to reference the window object. However, you can set this field to the ID of some subveiw of the window, for example 'SMPL':

```
resource 'view' (kSampleWindowID, purgable) {
{
        root, 'WIND', {50, 20}, {100, 100}, sizeVariable, sizeVariable, shown, enabled,
        Window          {"TWindow", zoomDocProc, goAwayBox, resizable, modeless, ignoreFirstClick,
                         freeOnClosing, disposeOnFree, closesDocument, openWithDocument,
                         dontAdaptToScreen, stagger, forceOnScreen, dontCenter,
                         'SMPL', ""};    /* Notice the target view field. */
```

```
        'WIND', 'SMPL', {0, 0} {100, 100}, sizeVariable, sizeVariable, shown, enabled,
        View            {"TSampleView"}
        }
};
```

When the window object defined by this template becomes the active window, MacApp again examines the contents of this target view field. Since MacApp finds the value 'SMPL' in this field, it sets gTarget to reference the subview of this window with view ID 'SMPL', which in this case is the TSampleView subview. In the example above, the window object will become the second object in the command chain, since MacApp initializes the fNextHandler field of views to reference their superviews. In this manner, you can exert some control over which object becomes the target object. Of course, when no windows are open, MacApp will still automatically make the target object be the application object.

Is this important? Does it belong here or in menus?

(Answer from Curt Bianchi on MacAppTech$)

In MacApp 2.0 we got rid of the global variable gEventInfo in order to better support nested event handling. Unfortunately this exposed the fact that the event info isn't parametrically passed to all the methods that may need it (e.g. DoMenuCommand). We'll add the parameter to the next release of MacApp.

In the meantime you can override a TApplication method like DispatchEvent and record the info in your own global variable. Another possibility, depending on what you're doing, would be to override DoCommandKey, which is a TEvtHandler method that handles keystrokes when the command key is down.

Chapter 18 Grids, Lists, and Palettes

translation from MacApp-speak about Grids and Lists into Macintosh terms

Chapter 19 Icons

*****No recipes yet*****

Creating an icon resource

Retrieving an icon resource into memory

Chapter 20 Keyboard Handling

*****No recipes yet*****

Refer different language keyboards to localization.

Handling DoKeyCommand

Recipe needed!

****Because of following type of question on MacAppTech$, obviously must include this recipe.****

Ihave a window containing a view that itself contains subviews subclassed from TView. The topmost view gets a DoKeyCommand message, but its subviews dont seem to get the message. The window & views are created from a resource. Should I be doing something to add the subviews of the main view to the linked list of TEvtHandlers ??, or do I use the main views' DoKeyCommand method to call a 'DoKey' method of the subviews so that they get a chance to handle the keypress themselves, which is the object of the exercise.

From: SCHMUCKER1 Schmucker, Kurt

The view that gets the keystrokes first is the one pointed to by gTarget. Certain subviews (like TEViews) make themselves the target when they are clicked upon. They then get the first crack at processing the keystrokes. You might want to put such a DoMouseCommand method in your subview.

Chapter 21 Languages

*****No recipes yet*****

Object Pascal and its unit dependencies?

Anything about TML or Lightspeed Pascal?

Cross-reference to C++ if timing appropriate?

Chapter 22 Localization

*****No recipes yet*****

Chapter 23 Memory Management

The Macintosh memory management system uses handles instead of pointers, so that dynamically allocated blocks of memory can be relocated without dangling pointers. In order to implement this, each dynamically allocated block has exactly one non-relocatable master pointer that references it. Any number of handles can then reference the master pointer.

Since master pointers can never be relocated, a Macintosh application should allocate plenty of space for master pointers immediately after startup. Allocating space for master pointers later may result in a fragmented heap.

The Macintosh Toolbox routine MoreMasters allocates space for an extra forty (???) master pointers. For example, the MacApp InitToolbox routine calls MoreMasters as many times as you specify in the parameter callsToMoreMasters.

Any of the following situations can cause an application to stop with a System Error alert, without any chance for that application to gain control:

- Not enough memory to load a code segment.
- Not enough memory to load a PACK resource.
- Not enough memory to save the bits under a menu which is pulled down.
- Not enough memory to load a defproc (WDEF, MDEF, CDEF, LDEF).
- Not enough memory for Standard File to create its file list.
- Add your personal favorite here.

Not all commercial Macintosh applications deal correctly with these issues, in large part because doing so involves some rather complicated code. MacApp provides a memory management mechanism that helps keep a MacApp application from getting into these critical memory situations.

Permanent and temporary memory

MacApp divides available heap space into permanent memory and temporary memory (also known as code reserve). **Permanent memory** is the space occupied by data which your application allocates: objects and any subsidiary data structures you may create. **Temporary memory** is reserved for your code segments plus any resources and/or memory needed by the Macintosh Toolbox for a short period of time.

MacApp always reserves enough space for temporary memory requests to be satisfied. You tell MacApp how much memory needs to be reserved, and it does the rest. It only needs to know whether a given memory request is permanent or temporary. Objects created via New (as well as TObject.ShallowClone and all other MacApp methods which allocate memory) are automatically taken from permanent memory. You can ask for a new handle from permanent memory by calling NewPermHandle instead of NewHandle; for other kinds of requests you set the "permanent" flag by calling PermAllocation(TRUE), make the request, then set the flag back (PermAllocation returns the previous value of the flag as its result). Any other requests (such as those made by the ROM) default to temporary memory.

It is dangerous to have the permanent flag TRUE for any length of time, as any toolbox call which allocates memory or any procedure call which could load a segment will then operate just as it would with no memory management mechanism. For example, calling a procedure in another segment which isn't loaded when the flag is TRUE and there is no permanent memory available will cause a segment loader bomb, even if plenty of temporary memory is available. When debugging is on, MacApp checks for the flag being TRUE in the main event loop or when a segment is loaded and drops into the debugger if it is.

Allocating both permanent and temporary memory

If you need to call a routine which allocates both permanent and temporary memory (such as TextEdit, which allocates permanent data structures and which also loads temporary resources such as fonts), take the following steps:

1. Set the perrmanent flag to FALSE

Do you set the flag, or does somebody else do it?

2. Call CheckReserve. CheckReserve is a BOOLEAN function which returns TRUE if there is still enough temporary memory. If CheckReserve returns FALSE, you should undo the memory allocation. For convenience, there is also a procedure named FailNoReserve, which calls Failure(memFullErr, 0) if CheckReserve returns FALSE.

Reserving temporary memory

To reserve the correct amount of temporary memory, you need to know the sum of the sizes of these items at the point in your application where the largest number of them are in use simultaneously. Some of the Toolbox items mentioned above usually need not be considered in this calculation. For example, the saved bits underneath a menu are only allocated when MacApp calls MenuSelect, which is only done in the main event loop, at which time all the non-resident code segments are unloaded. This is also the case for MDEFs. These will seldom be larger than the code segments which you can load.

There are some situations you do have to watch out for. For example, putting up a standard Open or Save dialog can use up quite a bit of memory: PACK 3 (Standard File), PACK 2 (Disk Initialization), plus the list of files in the current folder which Standard File creates. Also, although the Font Manager will not fail if there is insufficient memory to load a desired font, it will substitute a less suitable font, and your program's screen appearance will degrade. Thus, you should include enough memory to load the largest font you will use in your temporary memory figure. Remember, it will only be used when your Draw method is called, or when you perform text measurement.

Printing is another situation where you can run low on memory. Strictly speaking, it's OK to run out of memory while printing: an alert will come up saying the document could not be printed because there was not enough memory. However, users can find it frustrating if they create a document which is too large to print. If you want to make sure that any document a user can create can be printed, you should factor in the memory taken up by the Print Manager while printing—along with any of your code segments which may be present—when calculating the amount of memory to reserve for temporary allocations.

It is usually easier to split your temporary memory size up into several pieces which you calculate independently. For your code, you may want to use the MacApp 'seg!' resources, which let you list the code segments you want considered (see below). That way, you can just determine which segments are loaded at the time of maximum temporary memory use, and let MacApp figure out their size at execution time. For Toolbox use, you can use some fixed constants. For variable resources like fonts, you can actually alter the temporary reserve size while your application is running.

Depending on how careful you want to be about your application's memory use, you can just pick a comfortably large number (which wastes memory), or you can watch your program in action using the MacApp debugger and MacsBug and figure out the smallest safe number (which is a fair amount of work but gives the best use of available memory).

Using the debugger's high water mark

The MacApp debugger helps you figure out which set of resources takes the most room by keeping a high water mark for loaded resources. Under the heap & stack command (H), the I subcommand now gives the maximum amount of memory used by loaded code segments, PACKs, defprocs, and so forth, and the R subcommand resets this number to zero. Under the toggle flags command (X), the R flag reports whenever a new high water mark is reached, and if the B (memory management break) flag is set, MacApp will enter the debugger. In calculating this number, MacApp only considers resources on its resource lists (see below).

Note that this won't take into account memory allocated by such things as Standard File or the Print Manager. You should always check the amount of memory used in these situations and any other situation where the Toolbox can allocate large amounts of memory. The best way to do that is to

1. Break before and during such a situation
2. use the M subcommand of the H command to see how much permanent memory is available (the number labelled "(permanent) FreeMem"). The difference (call it "extra") will be the sum of the sizes of objects you've allocated and those that the toolbox has allocated but which MacApp doesn't track specifically (see "MacApp Resource Lists," below).
3. Note the set of segments loaded. The sum of "extra" and the "locked resources" number displayed in the debugger, minus any permanent memory you allocated, is the actual amount of temporary memory in use. You can use "extra" to reserve more temporary memory with a mem! resource (see below).

Remember, though, that you are only observing the memory in use at one point in time; memory usage can be greater for a brief period of time, and you won't necessarily catch it in the debugger. Sometimes a trial and error approach is necessary to determine the exact amount of memory being used.

Currently, the most space intensive print driver is the LaserWriter driver. For version 3.1 of the LaserWriter driver, we have empirically determined that "extra" is about 40K. In release 4.0 of the LaserWriter driver, this amount varies, and can be as high as 56K. Moreover, this may well increase in future releases. Fortunately, recent releases of the LaserWriter driver recover gracefully from out of memory conditions, giving the appropriate error message. If you do not need to insure that it is always possible to print, you can eliminate this from your permanent memory reserve.

There is a sporadic bug in LaserWriter driver 3.1 which can cause heap space to be permanently lost. This only occurs when a bitmap font is downloaded to the LaserWriter. Bitmap fonts are only downloaded when font substitution is off in Page Setup (font substitution is always off if you set FractEnable to TRUE or turn off the driver's line layout algorithm) and the user selects Geneva, New York, or Monaco, or if the user selects any other font which is not available in PostScript form (such as Athens or Mobile). The driver finds the largest available size of that font, makes it unpurgeable, then downloads it. Occasionally the driver will not make the font purgeable again, and it remains in memory until the application quits. Since the font is the largest size the driver could find, it takes a significant amount of space (8K for Geneva 24). This bug was fixed in release 3.3 of the LaserWriter driver.

Using the low space reserve

MacApp keeps a special handle around which it will dispose of in order to satisfy a permanent memory request. This handle is called the **low space reserve** (it's also sometimes called the permanent memory reserve). You can test if your application is running low on memory by calling the BOOLEAN function MemSpaceIsLow. MacApp makes this test periodically and will call the method TApplication.SpaceIsLow, which you can override to take any action you want. The default version will periodically put up an alert advising the user that memory is low.

Another important issue which the reserve handle helps with is making sure that users don't lose data. The Macintosh Memory Manager does not guarantee that a particular set of objects which was once allocated in a heap of a given size can again be allocated in a heap of the same size (although it will come pretty close). As a result, do not allow user documents to grow until they fill the entire heap, since it is possible that you would not be able to read them back in again. Also, if space becomes so scarce that there is no room to create a new command object, users won't be able to decrease their document size, even with a command like Clear.

You can use the low space reserve to handle these problems, as follows:

1. If MemSpaceIsLow returns TRUE, don't enable any commands that increase document size, such as Paste, drawing, typing, and so on.
2. Always enable commands which allow the user to decrease document size (such as Clear, backspace, and so on).
3. Commands such as Cut and Copy require more thought. On the one hand, they can increase memory use, which can result in a situation where command objects can't be created. On the other hand, if a user wants to decrease a document's size, having Cut and Copy available prevents having to simply throw data away. What you do here depends on your application. For example, UTEView allows Cut when space is low, but not Copy.
4. If you have a command which allocates additional memory, you should check for space being low at the end of your DoIt method, since otherwise it's possible for the reserve to be in place at the start of the command and completely gone by the end. You should treat running out of low space reserve the same as running out of memory. You can call FailSpaceIsLow, which calls Failure with an error of memFullErr if MemSpaceIsLow returns TRUE. Your command's failure handler should back out any changes it made. Only commands which decrease or don't change memory use should be allowed to eat into the low space reserve.

MacApp itself calls FailSpaceIsLow in several places, including the end of TApplication.OpenNew and TApplication.OpenOld to make sure that a new document doesn't decrease available free space too much. For TApplication.OpenOld, however, MacApp temporarily halves the low space reserve so that existing documents have a little "breathing room" to deal with the nondeterministic behavior of the Memory Manager.

Using the seg! and mem! resource types

MacApp initially sets the size of the temporary memory reserve by looking at all resources of type seg! and type mem!. The sizes of all code segments whose names are listed in any seg! resource are added up as part of the temporary memory reserve. Note that the segment names are the those generated after segment mapping. Each mem! resource has three long integer quantities, as follows:

- An amount to add to the temporary memory (code) reserve
- An amount to add to the low space reserve
- An amount to add to the size of the stack.

MacApp calculates the size of the temporary memory reserve by adding up the sizes of all segments in all seg! resources and the first number from all mem! resources. It calculates the low space reserve by adding up the second number from all mem! resources, and it calculates the size of the stack by adding up the third number from all mem! resources.

This approach allows a great deal of flexibility. For example, the Debug.r file adds the debugging segments to the list of segments, so that if debugging is turned on there will be room to load them. MacApp defines an initial set of segments, reserves 8K of stack space, reserves 4K of low space reserve, and reserves 4K extra of temporary memory. You can easily add to (or even subtract from, except for the stack) MacApp's values by supplying your own mem! and seg! resources. Remember, however, that these resources only govern the initial value of these numbers. If you wish to change any of them while your program is running (except the stack size, which can't be changed), you will have to call the routine SetMemReserve (see the MacApp source code for details).

Remember to factor in temporary memory taken up by things other than resources (such as the memory taken by the Print Manager while printing). For example, the MacApp debugger may tell you that the largest set of resources loaded at one time occurs when opening a document and totals, for example, 130K. When printing, your resources may only total 110K. However, the LaserWriter driver uses another 40K of temporary memory, so your maximum temporary memory use is 150K while printing. In that case you would list the segments you use during printing in your seg! resource, and use a mem! resource to reserve an additional 40K for the Print Manager.

MacApp resource lists

MacApp keeps a list (actually several) of the resources which it considers "temporary:" that is, they are considered as not taking up permanent memory. It constructs this list at application startup time, and uses it whenever it calculates how much temporary memory is in use and how much more to reserve. These resources are the ones that the MacApp debugger describes in the H command. MacApp will also purge these resources when trying to satisfy a temporary memory request unless they are locked.

By default, this list contains all resources of type CODE, PACK, LDEF, CDEF, WDEF, and MDEF (except those which come from ROM, or reside in the System heap). Normally, you don't need to worry about this list at all. It's OK for temporary resources to not be on the list, although they will be purged frequently when space is low, and they won't be figured into the resource statistics in the debugger. If you have such resources (fonts, for example), and if you want to reserve some memory for them, you may want to consider adding them to the list. Look at the UMemory unit to see how the lists are managed.

If you do put fonts on the list, you should move them high (with MoveHHi) and lock them when you do so to prevent MacApp from purging them; the ***Macintosh, I assume*** Font Manager expects that fonts marked nonpurgeable won't be purged.

Segmenting Your Application

In order for the memory management mechanism to work properly, your application must be segmented properly. The more code which is unnecessarily dragged into memory, the larger your temporary memory reserve must be, and the less space is available for your user's data in any given memory configuration. MacApp defines the following code segments for your application:

ARes	Resident application code. Anything that gets called frequently in the main event loop (such as your DoSetupMenus methods), drawing, or typing.
ADebug	Your debugging code; that is, any procedures or methods only present when debugging is on.
AFields	All of your Fields methods should go here.
AInit	Code which you only use at application start-up time (IYourApplication).
ATerminate	Code which you only use at application shut-down time.
ASelCommand	Code used to select the next command (DoMenuCommand, DoMouseCommand, DoKeyCommand, and IYourCommand methods).
ADoCommand	Code used for executing commands (all other TYourCommand methods except IYourCommand).
AClipboard	Clipboard code that is not part of a command (MakeViewForAlienClipboard, GivePasteData, WriteToDeskScrap, but NOT TYourPasteCommand).
AOpen	Code used when opening (DoMakeViews, DoMakeWindows, IDocument, DoMakeDocument, IDocument, IView).
AClose	Code used when closing (TYourDocument.Free, FreeData).
AReadFile	Code used when reading or reverting (DoRead, ShowReverted, DoInitialState).
AWriteFile	Code used when writing (DoWrite, DoNeedDiskSpace, SavedOn).
AFile	Code used when reading or writing (typically you won't have anything in this segment).
ANonRes	Catch-all non-resident segment, for infrequently used methods (e.g. ResizeWindow).

The default segment mapping established by MacApp combines these segments (except for ARes) with the corresponding ones for MacApp. You can override some of MacApp's mappings by specifying your own mappings in your make file, and you can override MacApp's mappings entirely by placing a definition for the Make variable SegmentMappings in your make file (Make will issue a warning, but your definition will override MacApp's). Look at MacApp.make1 for guidance.

Even using the suggested segment mappings, you may overflow the 32K bytes limit on the size of a segment. You can override MacApp's segment mappings in your application's .make file by providing your own -sn mappings; the Linker gives your segment mappings priority over the ones MacApp specifies.

Generally, there is a trade-off involved in segmentation. A few large segments make your program run faster, but increase its memory requirements. More, smaller segments decrease memory requirements (and are a necessity to run in a small Switcher partition), but make your program slower to start up. This latter problem can be alleviated by using the JumpStart utility included in the Macintosh Development Utilities product, available from APDA.

If you're really desperate to decrease segment sizes, you can change MacApp's segmentation by editing the source code and increasing the number of segments. This is not recommended. Be extremely careful about which segments are resident if you do this.

I don't think this last suggestio
a good idea. Can I delete

Using the res! resource type

The 'res!' resource type is used to identify code segments that are to be made resident (that is, never unloaded). Its format is similar to the 'seg!' resources in that each 'res!' resource consists of a list of segment names. When MacApp initializes the application it makes resident any segment listed in any 'res!' resource found. 'res!' resources are already included for MacApp's resident segments. You can add a 'res!' resource to your application's resource file to list resident segments you've defined. Note that as with 'seg!' resources the segment names in the 'res!' resources are the names after segment mapping.

Chapter 24 Menus and Menu Commands

A menu command on the Macintosh occurs whenever the user chooses a menu item. In MacApp, any class that descends from TEvtHandler can handle menu commands. If your objects don't override the TEvtHandler method, then they inherit the default behavior, which is simply to pass the DoMenuCommand message to the next event handler in the command chain. TApplication, TDocument, TView, and TWindow all descend from TEvtHandler, so it follows that the application or any document, window, or view can handle menu commands.

In fact, these classes are implemented such that they already handle some standard Macintosh commands. For example, TApplication handles the desk accessories, the About command, New, Open and Quit in the File menu, and the Show/Hide Clipboard command. TDocument handles Save, Save As, Save A Copy In, and Revert. In most applications your application, document, and view classes will handle menu commands specific to your application, document, or view.

To handle your application's documents, you can:

- Implement simple menu commands
- Maintain the menu bar and enable your menu items
- Change menu appearance and function
- Handle negative command numbers

This chapter discusses only those features that deal with how your code should handle the command when it is invoked, and those commands that are classified as simple commands. For the details on complex commands, see Chapter X, "Undoing." For the details on mouse tracking, see Chapter X, "Mouse Operations."

Implementing simple menu commands

The particular manner in which the DoMenuCommand does its work depends on whether the user chooses a simple command or a complex one. The terms *simple* and *complex*.i.commands: complex; have a specific meaning in this case. **Simple commands** are the commands that do not change a document's data nor require the mouse to be tracked. This type of menu commands should not be undoable. **Complex commands** are either undoable or require mouse tracking. Most commands change the document in one way or another, and thus they should be undoable. Many others require the mouse to be tracked.

Run-time summary of implementing simple menu commands

The application object sends ***automatically, or do you make your application object do this?*** a DoMenuCommand message to the target object. This allows the target object a chance to handle the command, if it can. If not, the target object sends the DoMenuCommand message to the next handler in the command chain. This process continues until each event handler in the command chain has a chance to handle the menu command.

Figure X-X provides a summary of MacApp's actions at runtime when a new document is to be created.

- **Figure X-1** MacApp's actions in relationship to this recipe

Figure TBD; for example see Chapter 14, "Documents"

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 24-1. Each of the steps is explained in detail in the recipes that follow.

This section also assumes that you have already created an instance of an application object. For more information, see Chapter X, "Applications."

Table 24-1 Overview: implementing a simple menu command

Step	Your action:	Because:
1.	Add the menu items to your 'cmnu' menus in your resource file and define command numbers and constants for the items.	MacApp constructs the menu items from the resource file, and uses the command numbers and constants to link menu items
2.	Decide which object should handle the command	Many classes descend from TEvtHandler, and thus can handle menu commands.
3.	Implement the DoMenuCommand method for the appropriate object. If the object cannot handle the current command, it must call the inherited version of DoMenuCommand.	Whenever the application object receives a menu command, it sends a DoMenuCommand message to the current target object.
4.	Override DoSetupMenus to enable the new menu items.	Whenever a mouse-down event is detected in the menu bar, ***MacApp?*** calls DoSetupMenus for the application, document, window, and view before the menus are displayed.

Step 1 Add the menu items and the command numbers to your resource file

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.

To start the menu process, you add the menu items as a 'cmnu' resource in your application's resource compiler input file and give the commands a command number. Also, In the implementation of your unit, define a constant for the command number you gave for the menu command in the resource file; by convention, such constants start with a lowercase "c".

You can make a new menu for commands specific to your application, or you can add commands to existing menus. The following example from UIconEdit.p adds a new menu with a Zoom In and a Zoom Out command:

```
{------------------------------------------------------------------------}
/* new command constants */
#define cZoomIn             1000
#define cZoomOut            1001


/* the view template resources */

/* the icon resource */

/* other 'cmnu' resources */

/* the new Icon menu resource */
resource 'cmnu' (4) {
  4,
  textMenuProc,
  allEnabled,
  enabled,
  "Icon",
  {
  /* [1] */  "Zoom In",   noIcon, "M", noMark, plain, cZoomIn;
  /* [2] */  "Zoom Out", noIcon, "L", noMark, plain, cZoomOut
  /* [3] */  "Zoom Out", noIcon, "L", noMark, plain, cZoomOut
  /* [4] */  "Zoom Out", noIcon, "L", noMark, plain, cZoomOut
  }
};

/* the 'MBAR' resource must be changed to include the new menu */
resource 'MBAR' (128) { {1; 2; 3; 4} };
{------------------------------------------------------------------------}
```

In this example, the Zoom In and Zoom Out commands are placed in a new Icon menu Therefore, this menu must be defined in a 'cmnu' resource, and included in the 'MBAR' resource.

Step 2 Decide which object should handle the command

When you add a menu command, you must decide which object should handle it. Sometimes it is difficult to decide whether a command should be implemented by a view or a document. There is no strict rule for this, but a general heuristic is this: if the command changes the document's data, then the command should be handled by the document; if the command only changes the appearance of that data in some view, then the command should be handled by a view.

Can this section become more general, or do I only need to deal with views and windows?

Step 3 Implement the DoMenuCommand method for the appropriate object

The DoMenuCommand method is the center of focus when MacApp applications respond to menu commands. Whenever the application object receives a menu command, it sends the DoMenuCommand message to the current target object. The whichCommand parameter is the command number corresponding to the menu item chosen (which the application object finds in the command number table.)

When one of your event-handling objects receives a DoMenuCommand message, it should take the following steps:

1. In response to simple commands, your objects should always return the predefined MacApp global variable gNoChanges. This result informs the application object that the command was a simple command—basically letting the application object know not to worry about enabling and renaming the Undo menu item in the Edit menu.

   - *Note:* The return value will be important for implementing undoable commands. See Chapter X, "Undo," for more information.

2. Decide whether or not the object can respond to the command by examining the whichCommand parameter. If it can respond to the command, it should take the appropriate action.

- *Note:* To prevent your DoMenuCommand overrides from growing too large become enormous, it is often best to have have them call other methods of your object to actually handle the command.

If the object cannot handle the command, it must call the inherited version of DoMenuCommand, so that the next event handler in the command chain gets a chance to handle the command.

The following example from UIconEdit.inc1.p implements the DoMenuCommand to trap a Zoom In and a Zoom Out command, and then passes the command back to the event-handling chain:

```
{-------------------------------------------------------------------}
FUNCTION TIconEditView.DoMenuCommand (aCmdNumber: CmdNumber): TCommand; OVERRIDE;

VAR
  anIconEditCommand:      TIconEditCommand;
  anIconPasteCommand:     TIconPasteCommand;

BEGIN
  DoMenuCommand := gNoChanges;          { Prime result of DoMenuCommand.      }

  CASE aCmdNumber OF                    { Decide if this command is ours...   }

    cZoomIn:
      SetMagnification(fMagnification + 2);{ Increase size of each bit by 2 pixels.}

    cZoomOut:
      SetMagnification(fMagnification - 2);{ Decrease size of each bit by 2 pixels.}

  ...{Other commands that the object can handle}
    OTHERWISE                           { Otherwise, let someone else handle it.}
      DoMenuCommand := INHERITED DoMenuCommand(aCmdNumber);
    END;
END;
{-------------------------------------------------------------------}
```

The DoZoomIn method will be called when the user selects the Zoom In menu command. In your DoZoomIn method you will actually have to get the icon to appear to grow in the window, which changes the size of the icon edit view, and then you will have to tell that view to redraw itself.

Step 4 Override DoSetupMenus to enable the new menu items

Can this step be included in the next major section?

Every application that defines its own menu commands must override DoSetupMenus. Whenever a mouse-down event is detected in the menu bar, ***MacApp?*** calls DoSetupMenus 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.

Override the appropriate DoSetupMenus method to enable the new menu items by taking the following steps:

does the first step have to be included in every step?.

1. Add the interface line for the appropriate object to your interface file.
2. Define your override method of DoSetupMenus in your implementation files.The implementation of DoSetupMenus consists of the following:
   - A call to INHERITED DoSetupMenus.
   - A series of calls to the Enable routine followed by a call to the inherited version of DoSetupMenus. Enable takes two parameters: the command number corresponding to the menu item to be enabled, and a Boolean parameter specifying whether to enable the item, or leave it disabled.
   - Calls to the MacApp and Menu Manager routines described in the rest of this chapter 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 the following sections. 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.

Maintaining the menu bar and enabling your menu items

Whenever any event occurs, MacApp disables every menu item in every menu. Then, MacApp sends a DoSetupMenus message to the target object, requesting that it enable the menu items that it handles. When one of your event handling objects receives the DoSetupMenus message, it should enable the menu items that it handles in its DoMenuCommand method and then pass the message on to the next event handler in the command chain. Eventually, each object in the command chain receives the chance to re-enable the menu items that it can handle.

- *Note:* MacApp enables the menu items that it can handle. You only need to enable the menu items specifically handled by your objects and then call the inherited DoSetUpMenus so that the rest of the command chain objects can enable their menu items.

When this process is finished, MacApp redraws the menu bar if necessary. This way, at any point during the execution of the application, only those menu items that can be handled by an object currently in the command chain are enabled.

Step 1 Enable appropriate commands

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)
```

Enable takes two parameters: the command number corresponding to the menu item you want to enable, and a Boolean parameter specifying whether to enable or disable the item.

Is the Boolean always canDo, or is defined by the programmer?

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 Chapter X, "The Clipboard and Cut, Copy, and Paste" for more information.

Step 2 Override DoMenuCommand to check for the new menu items

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.

To override DoMenuCommand, take the following steps:

1. Include the interface line in your interface file.
2. Define your override method of DoSetupMenus in your implementation files.

In the implementation of DoMenuCommand, remember these points: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.

Call another method to actually implement the commands this object handles

Call the inherited version of DoMenuCommand for commands this object doesn't handle.

```
{------------------------------------------------------------------------}

FUNCTION TIconEditView.DoMenuCommand (aCmdNumber: CmdNumber): TCommand; OVERRIDE;

VAR
  anIconEditCommand:       TIconEditCommand;
  anIconPasteCommand:      TIconPasteCommand;

BEGIN
  DoMenuCommand := gNoChanges;    { Prime result of DoMenuCommand. }

  CASE aCmdNumber OF       { Decide if this command is ours…       }

    cZoomIn:
      SetMagnification(fMagnification + 2);    { Increase size of each bit by 2
pixels.}

    cZoomOut:
      SetMagnification(fMagnification - 2);    { Decrease size of each bit by 2
pixels.}
```

```
  cCut,
  cCopy,
  cClear:
    BEGIN    { Return a TIconEditCommand object.      }
    NEW(anIconEditCommand);
    FailNIL(anIconEditCommand);
    anIconEditCommand.IIconEditCommand(aCmdNumber,  SELF);
    DoMenuCommand  :=  anIconEditCommand;
  END;

  cPaste:
    BEGIN    { Return a TIconPasteCommand object.     }
    NEW(anIconPasteCommand);
    FailNIL(anIconPasteCommand);
    anIconPasteCommand.IIconPasteCommand(SELF);
    DoMenuCommand  :=  anIconPasteCommand;
    END;

  OTHERWISE  { Otherwise, let someone else handle it.}
    DoMenuCommand  :=  INHERITED DoMenuCommand(aCmdNumber);
  END;
END;

{-------------------------------------------------------------------------------
--------}
```

In this example, DoMenuCommand sorts commands into two major kinds: those this view handles, and those it doesn't. For those commands that it does handle, another method, SetMagnification, is called to actually handle the command.

what does it do for cut and paste?

Step 3 Override DoIt

Is there a reason to return a command object if you are not Undoing?

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 Chapter x, "Undoing").

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

You need to change menu appearance and function to

- 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

Step 1 Changing the text of a menu item

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** Never use Menu Manager routines directly in DoSetupMenus to take ***which?*** actions.. △

Step 2 Changing the font style of a menu item

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.

Step 3 Displaying an icon in a menu item

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

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.

To handle negative command numbers, take the following steps:

Step 1 Implement DoMenuCommand

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.

Step 2 Implement DoSetupMenus

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.

Step 3 Implement a Font menu, if desired

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.

```
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;
```

Dynamically changing a popup or pulldown menu

*****Recipe needed?*****

(from Larry Rosenstein, in a response to MacApp.Tech$ question)

There are traps InsMenuItem and DelMenuItem, as well as SetItem to simply change the text. (Note that inserting or deleting items from the middle of a menu means that experienced users won't be able to predict where an item will appear on the screen, which is sometimes a useful shortcut.) ****

Creating menus outside of MacApp

*****Recipe needed?*****

*****An idea from Andy Swartz, as follows:*****

There is a way to create your own menus outside of MacApp's structure. The main thing you have to do (besides providing your own menu support) is set gRedrawMenuBar to TRUE in your DoMenuCommand override.

Continuing from here

This chapter has discussed how to implement simple commands; that is, those commands that do not require command objects (and are therefore not undoable) and do not require mouse-tracking commands. For more information on those subjects, see Chapter X, "Undoing" and Chapter X, "Mouse Operations."

You can also change the order in which events are handled in the command chain (see Chapter X, "Event Handling").

I moved that topic, and also moved the target view topic to "Creating a Window", since that is where you specify it. Is that OK?

Chapter 25 **Mouse Operations**

The user can click the mouse in a variety of situations: in a desk accessory window, in the window of another application under MultiFinder, in the title bar of a window, in the grow box of a window, or in the content area of a window. MacApp handles most of these for you. Only when the click is in the content area of a window, that is, in one of your special views, do you have to handle it.

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 chapter 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.

Command objects and mouse tracking

If you want the mouse click to be followed by some action while the mouse is being dragged, create and return a command object. MacApp then handles mouse-tracking by calling the appropriate methods of the mouse command object.

Command objects have three methods that are useful for implementing mouse tracking. TCommand has a few fields that will be important for mouse tracking, as follows:

- The fView field contains a reference to the view in which the mouse-tracking is taking place—this should be the view that created the command object (in its DoMouseCommand method). If that view is in a scroller, then the fScroller field of the command object will reference the that scroller. This allows MacApp to auto-scroll the view when tracking the mouse
- The fConstrainsMouse field is explained in the section on TrackConstrain, below. Also, the three new methods of TCommand are presented: TrackConstrain, TrackFeedback, and TrackMouse. These methods are called by MacApp to implement mouse-tracking.

Tracking the Mouse

In response to a mouse click, MacApp sends a DoMouseCommand to the frontmost view in which the mouse was clicked. (Remember that subviews lie in front of superviews, so the DoMouseCommand message is always sent to the view lowest in the view hierarchy.).

The standard DoMouseCommand that comes with all TView objects simply ignores the mouse click altogether. When you have a view subclass that should respond to mouse clicks, you will need to override DoMouseCommand for that class. In your DoMouseCommand override, you will typically respond to the mouse click in one of three ways:

Ignore the mouse click

Handle the mouse click immediately, and then return to MacApp

Create a command object to handle the mouse command. As with DoMenuCommand, your DoMouseCommand method would then return a reference to this command object. MacApp is responsible for calling the appropriate methods of this command object.

Run-time summary of tracking the mouse

When the mouse is clicked in your view, MacApp calls the DoMouseCommand of that view. In your DoMouseCommand override, you should create a command object, and return a reference to it. As long as the mouse button is held down, MacApp will repeatedly call the TrackConstrain, TrackFeedback, and TrackMouse methods of that command object.

Figure 27-1 provides a summary of MacApp's actions at runtime when the mouse is to be traked.

■ **Figure 27-1** MacApp's actions in relationship to this recipe

Overview of your responsibilities

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 27-1. Each of the steps is explained in detail in the recipes that follow.

This section also assumes that you have already created an instance of ~~an application object.~~ For more information, see Chapter X, "Applications."

■ **Table 27-1** Overview: tracking the mouse

Step	Your action:	Because:
1.	tcol1text	tcol2text

Create a subclass of TCommand

In your interface file, define a new object class as a subclass of TCommand. The class should have a field referencing the related document object and a field referencing the related view object: You must also override three methods of that class, as described in the following sections.

The following sample code from UIconEdit.shows the definition of a TIconDrawCommand class and the two fields that should exist.

```
{------------------------------------------------------------------------}
  TIconDrawCommand = OBJECT(TCommand)

    fIconDocument: TIconDocument;{ The document affected by this command.}
    fIconEditView: TIconEditView;{ The view in which this command draws.}
    ...
{------------------------------------------------------------------------}
```

To implement drawing, you must define a drawing command object class, and instantiate
that class when a mouseclick is received in a TIconView object.

Initialize the command object

This method should call
the ICommand
method to initialize inherited fields, and then initialize the fields unique to TIconDrawCommand. This method has one parameter: a reference to the related view object, itsIconView.
The call to ICommand must give the following:
the command number
a reference to the related document
a reference to the related view
a reference to the related scroller, if any

These values can all be found using the

itsIconView parameter:

```
ICommand(cDrawCommand,                                  { the command number }
      itsIconView.fIconDocument,              { the related document object }
     itsIconView,                                       { the related view object }
               itsIconView.GetScroller(true));  { a method which returns the related scroller }
```

In this example, notice that drawing has been given a command number constant, cDrawCommand. You should declare this constant with your other command constants in your implementation file.

Next, IIconDrawCommand must initialize the fields unique to TIconDrawCommand:

```
fIconView := itsIconView;
fIconDocument := itsIconView.fIconDocument;
```

Finally, you must reinitialize any inherited fields that ICommand doesn't initialize in the way you want. In this case, fConstrainsMouse and fCanUndo are both initialized incorrectly by ICommand. You can change their initialization here:

```
fConstrainsMouse := true;
fCanUndo := false;
PROCEDURE TIconDrawCommand.IIconDrawCommand(itsIconView:TIconView);

BEGIN
        ICommand(cDrawCommand,                          { Initialize the command...    }
                 itsIconView.fIconDocument,  { Associate it with a document. }
                itsIconView,                            { Associate it with a view.    }
                 itsIconView.GetScroller(TRUE);         { Associate it wiht a scroller. }

        fConstrainsMouse := TRUE;                       { TrackConstrain will be called. }
        fCanUndo := FALSE;                              { Can't undo drawing yet.      }

        fIconView := itsIconView;                       { Initialize the new fiels...  }
        fIconDocument := itsIconView.fIconDocument;
END;
```

Override DoMouseCommand

The general purpose of DoMouseCommand is to create a command object and return a reference to it. MacApp can then call the mouse-tracking methods of that command object.

. 1. Create a command object

2. Return a reference to that object

```
FUNCTION TIconView.DoMouseCommand (VAR theMouse: Point; VAR info: EventInfo;
                                   VAR hysteresis: Point): TCommand; OVERRIDE;
NEW(anIconDrawCommand);
FailNil(anIconDrawCommand);
anIconDrawCommand.IIconDrawCommand(SELF); { In this case, SELF is the related view.
}
DoMouseCommand := anIconDrawCommand;
```

Change the visual feedback, if desired

The TrackFeedback method allows you to give simple feedback while the mouse is being dragged. The default version of TrackFeedback draws a grey rectangle from the point where the mouse was clicked to the current mouse location. This is similar to the feedback for the selection rectangle tool in MacPaint. You can override that behavior in your mouse-tracking command objects to provide different types of feedback.

```
PROCEDURE TCommand.TrackFeedback(anchorPoint, nextPoint: VPoint;
                                 turnItOn, mouseDidMove : BOOLEAN);
```

If your view creates a command object in response to a mouse click, then MacApp will repeatedly call the TrackFeedback method of that command object while the mouse button is still down.

MacApp updates the four TrackFeedback parameters to let TrackFeedback know the state of the mouse, as follows:

- The anchorPoint parameter is always set to be the point where the mouse button was originally pressed. This parameter is a VPoint in the local coordinates of the view. VPoint is a special point type declared to allow views to be larger than the QuickDraw point system allows. For small views, you can always convert between VPoints and QuickDraw points using the MacApp utility ???VPtToQDPoint.

What about for large views?

- The nextPoint parameter is always set to be the point where the mouse is currently. Again, it is as VPoint in the local coordinate system of the view. The anchorPoint and nextPoint parameters can be used to give feedback similar to the selection rectangle tool—that is, feedback that relies only on the initial point and the current point.

- The turnItOn parameter is set to be true every time TrackFeedback is called, except the final time after the mouse button is released. (??? What can this be used for ???)
- The mouseDidMove parameter is true if the mouse has moved since the last time TrackFeedback is called. Normally,you won't have to worry about this parameter because MacApp won't even call TrackFeedback unless the mouse has moved.

  - *Note:* You can request that MacApp call TrackFeedback regardless of whether the mouse has moved by setting the fTrackNonMovement field of the command object to true. Typically you would do this in your command object's initialization method

Providing visual feedback

Where to give feedback varies from application to application . In IconEdit, the appropriate feedback is to draw "fatbits" in the icon as the mouse is being dragged. If the mouse was originally pressed over a white bit, then subsequent bits should be drawn in black as the mouse is dragged. If the mouse was originally pressed over a black fatbit, then subsequent bits should be drawn in white (erased) as the mouse is dragged. All of this processing can be done simply in your TrackMouse override, using the aTrackPhase parameter.

One important point to note is that as dragging is taking place, you want to give immediate visual feedback to the user. This means drawing in the view directly as the mouse is being dragged.

Remember for normal drawing, this is not the case. Normally, you would alter the data in the document, and then force the entire view to redraw itself. This approach to drawing is usually not fast enough for visual feedback while the mouse is being dragged. In each call to TrackMouse, you must immediately draw the appropriate "fatbit" in the view. Therefore, your view object will now have two drawing methods: Draw, which draws the entire view when MacApp tells it, and DrawBit, which immediately draws a specified "fatbit" in the view.

TrackMouse can call DrawBit, then, to give immediate visual feedback. Of course, the result of the drawing should do more than change the appearance of the view. It should also change the data in the document. Normally, this will be done in the DoIt, UndoIt, and RedoIt methods of the command object. However, since the drawing command is not undoable in this case, TrackMouse has to alter the document's data itself as the mouse is being dragged.

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.

Constrain the activity of the mouse, if desired

MacApp calls TrackConstrain before calling TrackFeedback and TrackMouse.

The TrackConstrain method allows you to constrain the activity of the mouse to a certain region. By default, MacApp constrains the mouse activity to the bounds of the view. On the Macintosh, you can't actually constrain the movement of the mouse pointer. Macintosh users are always free to move the pointer anywhere they like. However, you can constrain the feedback as if the pointer to were constrained to a particular area.

For example, in IconEdit the icon edit view has a border area (of five pixels) where no drawing is done. If the user moves the mouse into this border area while drawing, you'd like the application to respond as if the mouse hadn't left the drawable area of the view. You can do this with the TrackConstrain method.

The mechanism behind TrackConstrain is fairly simple. MacApp sends TrackConstrain the anchorPoint, previousPoint, and nextPoint of the mouse. If the actual nextPoint of the mouse isn't in your constraining area, then TrackConstrain can change the value of nextPoint.

This new value of nextPoint is the value that will actually be sent to TrackFeedback and TrackMouse. This way, TrackConstrain can ensure that TrackFeedback and TrackMouse never receive a nextPoint out of the constraining area.

Next statement is from Interim cookbook. Still true?

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.

```
PROCEDURE TCommand.TrackConstrain(anchorPoint, previousPoint: VPoint;
                                  VAR nextPoint : VPoint);
```

The parameters to TrackConstrain are as follows:

- anchorPoint
- previousPoint
- nextPoint. You can alter nextPoint so that your feedback acts as if the mouse was constrained to a certain area.

The purpose of TrackConstrain is to examine the nextPoint parameter. If it has gone out of the constraining region, then TrackConstrain should change nextPoint to bring it back in the proper area. This can be done by ensuring that the horizontal and vertical coordinates of nextPoint are always greater than the border size, but less than the size of the view less the border size:
nextPoint.h := Max(kBorder, Min(nextPoint.h, fIconView.fSize.h - kBorder - 1));
nextPoint.v := Max(kBorder, Min(nextPoint.v, fIconView.fSize.v - kBorder - 1));

Override TrackMouse

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. 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.

However, 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 TrackMouse method allows you to give more complicated feedback and do other processing while the mouse is being dragged. Often it is more convenient to do all feedback in the TrackMouse method, leaving the TrackFeedback method empty.

```
FUNCTION TCommand.TrackMouse(aTrackPhase : TrackPhase;
                             VAR anchorPoint, previousPoint, nextPoint: VPoint;
                             mouseDidMove : BOOLEAN): TCommand;
```

If your view creates a command object in response to a mouse click, then MacApp repeatedly calls the TrackMouse method of that command object while the mouse button is still down. TrackMouse is very similar to TrackFeedback, except that MacApp provides TrackMouse more information, and more control. The parameters for the TrackMouse method are as follows:

- The aTrackPhase parameter tells TrackMouse specifically what the current phase of mouse-tracking is. This parameter is always one of three predefined MacApp values, as follows:

trackPress	The mouse has just been pressed—this is the first call to TrackMouse.
trackMove	The mouse is currently being dragged.
trackRelease	The mouse has just been released—this is the last call to TrackMouse.

- The anchorPoint parameter represents the point where the mouse button was pressed.
- The previousPoint parameter represents the point where TrackMouse was called previously.
- The nextPoint parameter represents the current location of the mouse.
- The mouseDidMove parameter indicates whether the mouse has moved since the last call to TrackMouse (that is, if the previousPoint and nextPoint parameters have different values).

If the track phase is trackPress, the anchorPoint, previousPoint, and nextPoint parameters are all the same. If the track phase is trackRelease, then the nextPoint parameter gives the location of where the mouse was released.

Because the previous point is sent to TrackMouse as well as the nextPoint, that TrackMouse can do more complex feedback than TrackFeedback—by tracking the mouse point by point.

Because the points are VAR parameters, your TrackMouse override can alter these values. (??? When is this useful ???) If you alter anchorPoint, for example, the new value of anchorPoint will be sent to TrackMouse (as well as TrackFeedback) the next time through MacApp's loop. If you alter nextPoint, its new value will become the value of previousPoint the next time through the loop.

TrackMouse is a function returning a command object. Your command object should always return a reference to itself in its TrackMouse method, like this:
TrackMouse := SELF;

This tells MacApp to continue using this command object. (You could create and return a new command object here, and MacApp would continue as if that had been the command object all along. You probably won't need to use this functionality often.)
If the mouseDidMove parameter is false, then TrackMouse doesn't do anything.
If the mouse did move, then you must first figure out which bit of the icon
was clicked over.

The nextPoint
parameter of TrackMouse is given in View coordinates. However, PointToBit expects QuickDraw coordinates. For small views, you can use the TView utility method ViewToQDPoint to do the conversion.
The next thing that TrackMouse must do is check to see if the track phase is TrackPress. If
so, then it must determine if the bits should be turned on or off, depending on the state of the bit clicked over. Using the GetIconBit utility routine defined in this chapter, you can store this information in the fTurnBitsOn field:

```
IF aTrackPhase = TrackPress THEN
        fTurnBitsOn := NOT GetIconBit(fIconDocument.fIconBitMap, whichBit);
```

Of course, if you use GetIconBit, you must define it somewhere in your implementation file.
Finally, TrackMouse must actually invert the bit that the mouse is currently over. You do this
in two steps. First, you must change the actual data in the document, using the SetIconBit utility routine. Then, you must give immediate visual feedback using the DrawBit method of TIconView:

```
SetIconBit(fIconDocument.fIconBitMap, whichBit, fTurnBitsOn);
fIconView.DrawBit(whichBit, fTurnBitsOn);
```

As always, TrackMouse should return SELF as its result, so that MacApp will continue to send
mouse-tracking messages to this command object.

Tracking the mouse

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.

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.)

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.

Selecting

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.

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.

Step 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.

Step 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.

Step 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

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.

Step 1 Create a dragger object in DoMouseCommand

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.

This recipe 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.

Step 2 Implement the dragger object

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.

Step 3 Add a dragging field to your view

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.

Step 4 Define a command constant for the dragging command

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. };
```

Step 5 Test whether the dragging field is TRUE and the item is currently selected

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.

Step 6 Implement IDragger

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.)

Step 7 Implement TrackFeedback

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.)

Step 8 Add a prepare-to-drag method to the view

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.

Step 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.

Step 10 Implement the DoIt, UndoIt, and RedoIt methods

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.

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.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.

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

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.

Step 1 Create a sketcher command object in DoMouseCommand

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.

```
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;
```

Step 2 Use the sketcher object to track the mouse

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;
```

Step 3 Override TrackFeedback, if desired

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.

```
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;
```

Handling several types of mouse events

The preceding recipes in this chapter 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.

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.

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.

```
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

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.

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;
```

Chapter 26 MPW and MacApp

*****No recipes yet*****

Creating a MakeFile for applications

Creating a MakeFile for libraries

Using MABuild

Building a separate utility library

Using creator types

(MacApp$, 5-3-88)

(MacApp$, 6-1-88)

Chapter 27 Multifinder and Background Operations

No recipes yet

Running in the background

Creating a background application

Andy Swartz's suggestion; should it be here or in the Application chapter?

Communicating with other processes

*****Is this still to far in the future, or do we want to publish any of the following information, which was found on MacAppTech$?*****
Apple is investigating the proper interface for Macintosh IPC. In the meantime there are several ways to accomplish the same effect, as follows:

- If one app launches another it can pass info in the AppParmHandle.
- The system heap is fair game for shared memory.
- You *can* use AppleTalk on the same node.
- You can write your own tiny IPC driver that basically does memory copying.

The only thing to definitely avoid is passing pointers to memory across application heaps.

Creating an idle time sorter

Possibly Joost Kemink's recipe from contest

Monitoring events during batch processing

*****MacAppTech$ questions; is it worth a recipe?*****

What method in gApplication should be called regularly during batch processing to ensure that other events (such as a buttonDown in the Cancel button) are attended to? Or is I achieve this my rearranging gTarget?

Joost Kemink's response, and possibly a recipe from contest

If your problem contains a repetition of smaller subproblems, you may want to create an event handler which you then install in the idle chain. Every time the DoIdle method of your event handler is called, you calculate one of the subproblems, finally resulting in the complete problem being solved. For example, a bubble sort can be arranged this way. I am working on an example, which I will send you when finished. Essential is, that you save the state of your computation from one invocation of DoIdle to another.

Depending on the fIdleFreq field in your event handler, more or less of the idle time is devoted to calculating the solution.

Using this technique, you don't have to modify gTarget, and -more important- you don't put the application in a modal state. It also allows you to do your calculations in the background under MultiFinder. Cancellation of the task is simply done by removing your event handler from the idle chain.

Finding out if MultiFinder is running

*****Big philosophical debate on MacAppTech$; which way do we jump? Always with the appropriate caveats, such as these gleaned from the debate*****

Appropriate questions are as follows:

- Is WaitNextEvent implemented?
- Are the temporary memory calls available?

These questions can be answered by examining *The Programmer's Guide to MultiFinder.*

Testing for command-period

*****MacAppTech$ contribution to test for command period; could be the start of a recipe*****

- gMultiFinder is set beforehand using one of several methods to know if waitNextEvent is implemented. You can always get rid of it and call only getNextEvent.

- A consequence of calling waitNextEvent is that your application knows about MultiFinder and viceversa, if the rest of the code is well-behaved and the Background bit is on in the SIZE resource, then your application will fully interact with MF.

- It's not really needed to call ObeyEvent. If you do, be very careful to filter command-keys and Menu-Clicks (as I do here, with over-kill), otherwise you may call your process from within itself, and I don't think anybody writes reentrant MacApp code!
- This piece of code was taken out of an actual application.

```
gMultiFinder:boolean; {TRUE if the system supports multifinder}
btemp:boolean;
aEvent:EventRecord
aCommand:TCommand;
ch:char;

.................
if gMultiFinder then btemp:= waitnextevent(everyevent,aEvent,1,NIL)
                    else btemp:= getnextevent(everyevent,aEvent);
if btemp then
begin
        if (aEvent.what=keyDown) or (aEvent.what=autoKey) then
       begin
          ch:=char(loWord(BitAnd(aEvent.message,charCodeMask)));
           if ch='.' then
           begin
                {Clean Up and Exit the loop}
           end;
       end
        else if (aEvent.what<>mouseDown) then
       begin
            gApplication.ObeyEvent(@aEvent,aCommand);
       end;
```

```
end;
```

Chapter 28 Performance Tips

No recipes yet

Optimizing compiling

Optimizing linking

Chapter 29 **Printing**

The printing unit, UPrinting, provides standard printing capability that is sufficient for most applications. MacApp handles printing through objects called *print handlers.*

Enabling printing

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);
```

Standard print handling

MacApp provides two standard print handler classes: TPrintHandler is a "null" print handler that isn't capable of printing, but simply defines the minimal print handler interface. TStdPrintHandler fully implements standard Macintosh printing, as well as handling printing-related issues such as page setup and screen feedback.

(Steve, should we change courier font to normal font in the following example?)

The print handlers cooperate with views to accomplish the printing. The print handler deals with the actual printing, while the view performs the actual page drawing. In most cases, you will want to use an instance of a `TStdPrintHandler` object, in conjunction with your view, to produce printed output.

> *Note:* Custom printing is usually performed by overriding `TStdPrintHandler` methods, although you could write your own print handler.

MacApp manipulates several rectangles to determine the printable area of a page. These rectangles are stored in the handler field `fPageAreas`, and are as follows:

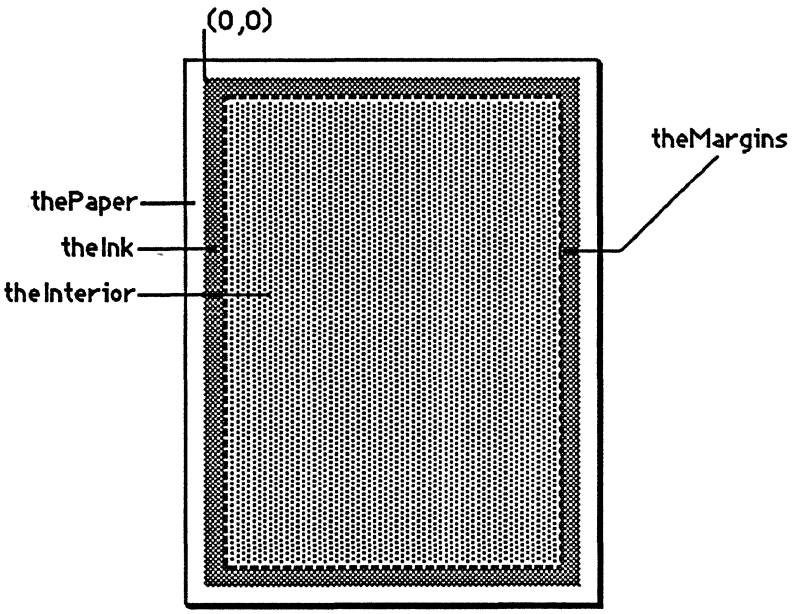
`theInk` A rectangle that defines the printable area of a page. The top-left coordinate is always (0,0).

`thePaper` A rectangle that defines the entire physical page, in the coordinates of `theInk`. The physical page is usually larger than the printable area, so the top-left coordinate of `thePage` is usually negative, and the bottom-right coordinate is greater than that of `theInk`.

`theMargins` An offset rectangle that is used to determine the printing area of the page. The coordinates of this rectangle specify how far `theInterior` is inset from `thePaper`.

`theInterior` A rectangle that defines the actual printing area; that is, the area where the view can actually be drawn. The theInterior rectangle is computed by *subtracting* `theMargins` from `thePaper`. `theInterior` can never be greater than `theInk`.

MacApp automatically initializes the `fPageAreas.theMargins` of the `TStdPrintHandler` object to 1" (72 Quickdraw pixels). This value is stored in the global variable `gStdPageMargins`. The relationship of these rectangles to the printed page is shown in the following figure.



Standard page rectangles.

Note that the value of the margin rectangle is not a real rectangle. This is because theMargins is an offset rectangle, so that theInterior := thePaper - theMargins. Also, remember that theInterior can never be larger than theInk, so theMargins should never be less than the difference between thePaper and theInk.
If you are changing the values of `fPageAreas`, you might wish to use the Inspector to check how the rectangles relate. If `TStdPrintHandler.fShowBreaks` is `TRUE`, then MacApp will draw a two-pixel gray line along the page breaks for your view (on-screen). Also, if `gDebugPrinting` is `TRUE`, then `TStdPrintHandler` will also draw rectangles specified by `theInk` and `theInterior` when printing.

Changing the margins

The margins for a printed page are by default in MacApp set to 1 inch or 72 pixels. In the Uprinting unit, MacApp provides an InstallMargins method in the TStdPrintHandler class. That method provides parameters you can use to change the printing area.

If you want to print a view that uses margins other than the default, take the following steps:

1. Call the `InstallMargins` method of the `TStdPrintHandler` object. `InstallMargins` takes two parameters: `newMargins` is a rectangle specifying the new margins, and `areMinimalMargins` is a boolean value stating whether custom or minimum margins should be used.

2. Either tell MacApp to install the minimal margin possible for the printing device, or specify the new margins, as follows:

   If you want to install the minimal margin possible for the printing device, set `areMinimalMargins` to TRUE. This technique allows your application to be as device-independent as possible. The `newMargins` parameter is ignored in this case.

   If you want to set the margins to a value other than the default or minimum settings, set `areMinimalMargins` to FALSE and set the `newMargins` parameter to the value of the desired rectangle. MacApp then sets the fPageAreas.theMargins field of the StdPrintHandler equal to that rectangle.

3. Call the DoPagination method of the view that is printing in order to recalculate the printable area on the page, and therefore recalculate the page breaks in the view.

The easiest place to change the margin settings is when the view and its print handler are created, usually in the `TDocument.DoMakeViews` method. The following code fragment is an example of how this might be implemented:

```
{-------------------------------------------------------------------------}
PROCEDURE TMyDocument.DoMakeViews(forPrinting: BOOLEAN); OVERRIDE;

VAR
  aWindow     : TWindow;
  myView      : TMyView;
  aHandler    : TStdPrintHandler;
  marginRect  : Rect;

BEGIN
  IF forPrinting THEN
    { Don't need window when printing from Finder }
    myView := TMyView(DoCreateViews(SELF, NIL, kMyViewID))
  ELSE
    BEGIN
      aWindow := NewTemplateWindow(kMyWindowID, SELF);
      myView := TMyView(aWindow.FindSubView('mine'));
    END;

  New(aHandler);    { create a StdPrintHandler       }
  FailNIL(aHandler);

  aHandler.IStdPrintHandler(SELF,       { its document      }
            myView,       { its view    }
            FALSE,        { does not have square dots      }
```

```
            TRUE,  { horizontal page size is fixed  }
            TRUE);       { vertical page size is fixed    }

  marginRect.bottom := -31;       { (31, 30)/(-31, -30) is the same }
  marginRect.right := -30;        { as setting minimal margins      }
  marginRect.top := 31;    { for printing on a LaserWriter  }
  marginRect.left := 30;   { or ImageWriter    }

  aHandler.InstallMargins(marginRect, TRUE); { set minimal margins       }

  aMyView.DoPagination;    { force recalc of page print area }
END;
{-------------------------------------------------------------------------}
```

In this example, the value of marginRect is arbitrary. Since areMinimalMargins was set to TRUE in the call to InstallMargins, the value of marginRect is ignored.

Chapter 30 Resources

*****No recipes yet*****

Short definition of resources and MacApp's relationship to Toolbox Resource Manager. X-Ref to menus, views by template, etc., and ViewEdit manual if the schedules fall into place.

Chapter 31 **Scrolling**

Definition of scrolling

MacApp handles most of the job of scrolling for you, including creating the scroll bars, responding to scrolling events, and ensuring that your view draws its contents in the right place. Your part of the job is to implement a new, invisible, view to the view hierarchy between the window and the view that you want to scroll .

To handle scrolling, you can:

- Create a scroller
- etc. **(***on through all the recipes provided by the chapter***)**

Scroller views

A scroller view is a subclass of TView with a few extra scrolling-related fields and methods. One of these fields is fTranslation. In this field the scroller stores a value specifying how much its subviews should be scrolled. Thus, by changing the coordinate translation of a scroller, MacApp in effect scrolls all of the views it contains.

When a window that contains a scrolling view needs to be redrawn, MacApp first focuses on the window, and then sends the window the Draw message. In turn, the window focuses on the scroller, and then sends the scroller the Draw message. Since scrollers are invisible, nothing is drawn.

The scroller then focuses on its subview. However, the scroller actually doesn't actually focus directly on its subview. Instead, it focuses on the location of its subview, offset by the value in its fTranslation field. Only then does the scroller tell its subview to Draw.

Drawing is still clipped to the superview, so even if the drawing would have been relocated outside the boundaries of the scroller, only that part of the drawing inside those boundaries would actually be drawn.

Creating a scroller

*****Material needs extensive work to fit into proposed format*****

The NewTemplateWindow routine, called in your document's DoMakeViews method, creates whatever view instance hierarchy is specified by the corresponding view template. Therefore, in order to add a scroller to the view instance hierarchy in your application, all you need to do is add a scroller to the view template in your rez file.

This recipe shows you how to implement scrolling in IconEdit. All you need to do is add a scroller view to the view hierarchy specified in your view templates. MacApp then uses the scroller view to provide scrolling to the user on command.

Step 1 Add a scroller view to your view template hierarchy.

Following material needs to be broken up

When you add a scroller view to your view hierarchy, the window view needs to be large enough to store the initial view and scrollbars. MacApp provides a constant kSBarSizeMinus1 to make this sizing easier.

Here is the standard form for a scroller template:

```
/* fields for all view objects                    */
'WIND',                    /* ID of the view's superview             */
'SCLR',                    /* ID of this particular view             */
{0, 0},                    /* location of this view in superview     */
{kheight, kwidth},         /* size of this view in pixels            */
sizeRelSuperView,          /* horizontal size determiner             */
sizeRelSuperView,          /* vertical size determiner               */
shown, enabled,            /* whether view is shown and enabled      */
Scroller                   /* what type of view this is--a scroller  */

/* fields specially for scrollers */
{ "TScroller",             /* class of this scroller--TScroller      */
vertScrollBar,             /* is there a vertical scroll bar         */
horzScrollBar,             /* is there a horizontal scroll bar       */
0,0,                       /* initial value of fTranslation          */
16,16,                     /* how big can translation movement be?   */
vertConstrain,             /* is scrolling vertically constrained?   */
horzConstrain,             /* is it horizontally constrained?        */
{0, 0, 0, 0} };            /* I forgot this one ???                  */
```

Notice that this view template has the same format as other view templates—the first nine fields apply to all view objects. The ninth field specifies what particular class of view object is being described further—in this case, a Scroller. The next fields are all contained in a pair of curly braces. These fields further describe the scroller.

In the above example, the scroller has the view ID 'SCLR'. It is a subview of the view with ID 'WIND'. It is located at point {0,0} in that window, and its initial size is determined by the constants kheight and kwidth.

The next two fields are the size determiners. These determine how MacApp maintains the fSize field of this view. If these are sizeRelSuperView, as they are in this case, then MacApp will automatically resize the scroller as its superview is resized. This is normally what you want for scrollers. (In the case of the TIconEditView from last chapter, the size determiners were sizeVariable. This meant MacApp always set the fSize field to be the result of the CalcMinSize method. This is usually what you want for view that can change size independently of their superview, like IconEditViews will be able to.)

The next two fields determine whether the view is initially shown and enabled. For all the views encountered so far, this has been the case. The next field, the ninth field, tells MacApp which type of view template is coming up. In this case, it is a Scroller. Therefore, MacApp knows that the next set of fields all deal specifically with a scroller view. In previous view templates, you've seen Window and View. Windows have even more fields than scrollers, while views have considerably fewer.

The next ten fields all appear within curly braces, and apply to scrollers in particular. The first is the class name, in this case "TScroller". You could put the name of any subclass of TScroller here, and MacApp would still fill in the inherited fields correctly. Of course, you would have to fill in any new fields in an IRes method. However, since the generic TScroller class does enough anyway, you can just put TScroller here.

The next two fields allow you to specify whether or not you want the scroller to have associated vertical and horizontal scrollbars. In the above example, both have been requested. When this scroller is created (by NewTemplateWindow, for example), it will automatically create two scrollbars, and they will be fully functional—you don't have to do anything more than specify that you want them in this view template.

The next two values specify how much the scroller should be scrolled initially. In this case the value is (0,0), which means that the scroller is initially set to not be scrolled at all—the subviews will all be in their initial position. The next two values specify the scroll unit—the minimum number of pixels that the subviews are scrolled by at once. In this case the value (for both the horizontal and vertical directions) is 16. In other words, whenever the user is scrolling the view in the window, it will actually make many little jumps of 16 pixels each time.

The last three fields I don't know.????? ???

Here are the new values for constants in your view template hierarchy, followed by the hierarchy itself.

```
/* the view stays the same initial size */
#define kIconViewHeight          32 * 7 + 10    /* 7 times actual size, 5 pixel
borders */
```

```
#define kIconViewWidth            32 * 7 + 10    /* 7 times actual size, 5 pixel
borders */
/* the window must be large enough for the view and scrollbars */
#define kIconWindowHeight  kIconViewHeight + kSBarSizeMinus1
#define kIconWindowWidth   kIconViewWidth + kSBarSizeMinus1

/* the scroller is the same size as the window, less the scrollbars */
#define kScrollerHeight           kIconWindowHeight - kSBarSizeMinus1
#define kScrollerWidth            kIconWindowWidth - kSBarSizeMinus1

/* ============================    Icon Window    ================================ */

resource 'view' (kIconWindowId, purgeable) {
  {
    root, 'WIND', { 50, 20 }, { kIconWindowHeight, kIconWindowWidth },
    sizeVariable, sizeVariable, shown, enabled,
    Window          { "TWindow", zoomDocProc, goAwayBox, resizable, modeless,
                    ignoreFirstClick, freeOnClosing, disposeOnFree, closesDocument,
                    openWithDocument, dontAdaptToScreen, stagger, forceOnScreen,
                    dontCenter, 'ICON', "" };

    /* here is the new template--for the scroller. */
    'WIND', 'SCLR',        { 0, 0 },
    { kIconWindowHeight-kSBarSizeMinus1, kIconWindowWidth-kSBarSizeMinus1 },
    sizeRelSuperView, sizeRelSuperView, shown, enabled,
    Scroller { "TScroller", vertScrollBar, horzScrollBar, 0, 0, 16, 16,
                    vertConstrain, horzConstrain, { 0, 0, 0, 0 } };

    'SCLR', IncludeViews  { kIconEditViewId }
  }
};

resource 'view' (kIconEditViewId, purgeable) {
  {
    root, 'ICON', { 0, 0 }, { kIconViewHeight, kIconViewWidth },
    sizeVariable, sizeVariable, shown, enabled,
    View            { "TIconEditView"}
  }
};
```

Notice that the 'SCLR' scroller view is a subview of the 'WIND' view. Also, the 'ICON' view is now a subview of the 'SCLR' view.

```
/* the view stays the same initial size */
```

```
#define kIconViewHeight          32 * 7 + 10    /* 7 times actual size, 5 pixel
borders */
#define kIconViewWidth           32 * 7 + 10    /* 7 times actual size, 5 pixel
borders */

/* the window must be large enough for the view and scrollbars */
#define kIconWindowHeight  kIconViewHeight + kSBarSizeMinus1
#define kIconWindowWidth   kIconViewWidth + kSBarSizeMinus1

/* the scroller is the same size as the window, less the scrollbars */
#define kScrollerHeight          kIconWindowHeight - kSBarSizeMinus1
#define kScrollerWidth           kIconWindowWidth - kSBarSizeMinus1

/* ============================    Icon Window    ================================ */

resource 'view' (kIconWindowId, purgeable) {
  {
    root, 'WIND', { 50, 20 }, { kIconWindowHeight, kIconWindowWidth },
    sizeVariable, sizeVariable, shown, enabled,
    Window          { "TWindow", zoomDocProc, goAwayBox, resizable, modeless,
                    ignoreFirstClick, freeOnClosing, disposeOnFree, closesDocument,
                    openWithDocument, dontAdaptToScreen, stagger, forceOnScreen,
                    dontCenter, 'ICON', "" };

    /* here is the new template--for the scroller. */
    'WIND', 'SCLR',        { 0, 0 },
    { kIconWindowHeight-kSBarSizeMinus1, kIconWindowWidth-kSBarSizeMinus1 },
    sizeRelSuperView, sizeRelSuperView, shown, enabled,
    Scroller { "TScroller", vertScrollBar, horzScrollBar, 0, 0, 16, 16,
                    vertConstrain, horzConstrain, { 0, 0, 0, 0 } };

    'SCLR', IncludeViews  { kIconEditViewId }
  }
};

resource 'view' (kIconEditViewId, purgeable) {
  {
    root, 'ICON', { 0, 0 }, { kIconViewHeight, kIconViewWidth },
    sizeVariable, sizeVariable, shown, enabled,
    View            { "TIconView"}
  }
};
```

```
/* the icon resource */

/* the menu resources */
```

Notice that the view hierarchy is split into two resources. The first resource includes the window and the scroller, and also includes (through an IncludeViews directive) the icon view. The other resource includes only the icon view. The reason for dividing the hierarchy into two resources is explained in the printing chapter ???.

Handling scrolling in lists

Recipe needed?

(Deb Orton's response to a question on MacAppTech$)

It sounds like the sizeDeterminer for your TextListView is set incorrectly. This is a field of TView used to determine how much information needs to be passed between views and sub-views with regard to size changes. Your scroller view should probably be "sizeFixed" for both horizontal and vertical and your TTextListView (which should be a sub-view of the scroller view) should be "sizeSuperView" (or "sizeRelSuperView") so that when rows or columns are added to the view, the scroller will respond properly.

Chapter 32 Sound

*****No recipes yet*****

MacApp's relationship to Toolbox Sound Manager

Chapter 33 **Text Editing**

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.

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.

.c4.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;
```

Creating a TextEdit view by procedure

1. Obtain handle for the text
2. Fill in parameters for TextEdit records
3. Associate TextEdit records with documents

Creating a TextEdit view by template

1. Register the type by calling InitUTEView

Displaying and manipulating text

Assigning text attributes

(Fonts, text colors, etc. Also relationship with Macintosh Font Manager)

Chapter 34 Toolbox and MacApp

*****No recipes yet*****

MacApp's relationship to Toolbox; e.g. MacApp takes care of initialization

Chapter 35 **Undo**

To **undo**, in the Macintosh world, is to reverse the effects of a command that somehow changes a document. An important part of the Macintosh user interface, you should implement an Undo menu command 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.

To use MacApp to implement Undo, you will create a command object that contains at least DoIt, UndoIt, and RedoIt methods. MacApp then automatically enables the Undo menu command when all of the following conditions are true:

- A command object has had its DoIt method executed
- The fCanUndo field of that command object is set to TRUE
- The command object has not been superceded by another command object. (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.

Implementing undoable menu commands

The Undo command is handled by the current command object.

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.

who does the invalidation, MacApp or the application programmer?

If the command has results that are too complicated to undo directly, DoIt and RedoIt can 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. For more information on filtering, see "Creating Filtered Commands" later in this chapter.

Run-time summary of implementing Undo

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. The user can thus choose Undo, Redo, Undo, Redo indefinitely.

Figure 15-X provides a summary of MacApp's actions at runtime when the Undo capability is implemented.

- **Figure X-1** MacApp's actions in relationship to this recipe

Overview of your responsibilities

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 15-X. Each of the steps is explained in detail in the recipes that follow.

This section also assumes that you already know the techniques for implementing simple menu commands. For more information, see Chapter X, "Menus."

■ **Table X-1** Overview: Undoing

Step	Your action:	Because:
1.	Define a command object as a subclass of TCommand.	The TCommand class provides ***what relevant abilities?***.
2.	Return that command object in response to a menu command.	MacApp calls this method whenever the application creates a new object, and the default TApplication.DoMakeDocuments method is empty.
3.	Override the DoIt method for the command object.	MacApp calls this method whenever the appropriate command object is returned, and the default TCommand.DoIt method is empty.
4.	Override the UndoIt method for the command object.	MacApp calls this method whenever the user chooses the Undo menu item, and the default TCommand.UndoIt method is empty.
5.	Override the RedoIt method for the command object.	MacApp calls this method whenever the user chooses the Redo menu item, and the default TCommand.RedoIt method is empty.

Step 1 Define and initialize a command object as a subclass of TCommand

The TCommand class is used by MacApp to embody undoable commands, as well as commands that require mouse-tracking. Among others, TCommand defines three methods, DoIt, UndoIt, and RedoIt, all of which must be overridden to implement an undoable command.

Try not to allocate any memory in your IYourCommand method. There is nothing actually wrong with this, but since the previous command has not yet been committed and freed, and the Undo Clipboard might still be around, it's more likely to fail. If you allocate your memory in your DoIt method, more space will be available.

Step 2 Return that command object in response to a menu command

How is this done?

Step 3 Override the DoIt method

When you return a command object when executing a command, MacApp calls command.DoIt using the command object you return. You then override TCommand.DoIt to execute your command.

What else can be said that is general enough and not tied to a specific example?

Step 4 Override the UndoIt method

MacApp calls the UndoIt method for the command object if the user chooses the Undo menu command.

What else can be said that is general enough and not tied to a specific example?

Step 5 Override the RedoIt method

RedoIt is called if the user chooses the Redo menu command.

What else can be said that is general enough and not tied to a specific example?

Creating filtered commands

.i.commands: undoable;.i.commands: filtered;.i.commands: hard-to-undo;

(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.)

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.

Step 1 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.

need example

Step 2 Mark the changed items and invalidate the images of the items

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.

Step 3 Check the changed items and alter the way the data is displayed

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.

need example

Step 4 Make the actual changes in the Commit method

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.

If your command object overrides the Commit method, it is vital that it not cause a Failure. MacApp must call the Commit method of the most recent command (if it has not been undone) in order to save the document or quit the application; if Commit fails, your user will be really stuck, unable even to exit the application. The MacApp command architecture assumes that by the time Commit is called, the command was successful. There are three ways of dealing with this problem, as follows:

- The first and best way is to set up your data structures in such a way that your Commit method doesn't need to allocate any memory (or increase net memory use).
- Preallocate the memory which your command needs to Commit in your DoIt method. Then, when Commit is called, free this memory and proceed.
- Allocate the memory you need to Commit from the temporary memory pool instead of the permanent memory pool. Do this only if your Commit method's memory use has an upper bound, and if the memory will be disposed of by the end of the Commit method. If you use this technique, you must make sure that your temporary memory reserve is big enough to accommodate this memory use (you can use a mem! resource to do this).

For an example of how subtle these considerations can be, look at the Paste command in the DrawShapes sample application, whose Commit method can actually fail under certain circumstances. The Commit method moves the shapes which have been pasted from the list of "virtual" shapes associated with the command onto the actual list of shapes for the document. It does this by repeatedly deleting a shape from the virtual list, then inserting it into the actual list. Although this seems safe because it does not cause net memory usage to increase, it can still fail, because it may not be possible to grow the actual list's handle by four bytes even though another handle has just shrunk by four bytes and the heap may be completely unfragmented. This is just a consequence of the way the Macintosh Memory Manager works.

Two possible solutions to this problem: grow the actual list in the DoIt method so that the memory is already allocated, or perform the transfer of shapes with the permanent allocation flag off (currently this is not possible without overriding TList). Note that this latter option is safe because net memory utilization is not increasing; by allowing the Commit method to briefly eat into temporary space, we are just giving the Memory Manager a little more "breathing room" in which to rearrange the heap.

You should also make sure that your UndoIt and RedoIt methods can't fail either. A less desirable alternative is to detect in your IMyCommand or DoIt method that Undo won't work and put up an alert that the command will not be undoable. Do this before your command makes any changes, and give the user a chance to cancel the command.

If the user says OK, remember to set fCanUndo to FALSE in your command so Undo will be disabled. If the user says Cancel, you can cause your command to fail without putting up an error message by calling Failure(0, 0). MacApp uses this technique itself to handle Cancel choices in dialogs.

Chapter 36 **Views**

Views are a MacApp concept, and do not have a counterpart in the Macintosh environment. Everything displayed in a MacApp program is displayed in a view object. Among other things, views have a visual representation (that is, they can draw themselves), they have a size in pixels and a location relative to its background view (superview), and they can respond to events such as mouse clicks, keystrokes, and menu commands. and are part of a view object hierarchy.

The ancestor class of all view objects is called TView. MacApp provides predefined subclasses of TView for many view objects, among them the following:

- **TWindow.** This class represents Macintosh windows. and inherits fields and methods from TView 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.** This class calculates coordinate translations to create the illusion of scrolling through a document.
- **TDialogView.** This 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.** This class displays and manipulates text.
- **TControl.** This class provides subclasses that contain methods to deal with traditional Macintosh controls, such as scroll bars, push buttons, radio buttons, and check boxes, and other types of controls, such as pop-up menus, pictures, and icons
- **TGridView.** This class provides methods to deal with rows and columns.
- **TDeskScrapView.** TDeskScrapView provides methods to allow your application to show the Clipboard.

This chapter provides a general overview on how you can create a view using templates or procedures. Details on each of the predefined classes are available in appropriate chapters in this manual. If the predefined view classes do not provide the objects you want, you can create your own subclass of TView, instantiate it, and add the instance to your view hierarchy.

In any case, you are responsible for defining the methods that draw and that calculate the content area of your views.

To handle your application's views, you can:

- Create a view using a template
- etc. **(***on through all the recipes provided by the section***)**

This chapter describes in detail the steps you need to take to accomplish these tasks.

TView and view hierarchies

TView is the ancestor class of all view objects, including windows, dialog boxes, dialog controls, and even your specialized view classes. The class TView is defined as follows:

```
TView = OBJECT (TEvtHandler)

  fSuperView:       TView;
  fSubViews: TList;
  fDocument: TDocument;
  fLocation: VPoint;
  fSize:            VPoint;
  { a few more fields... }

  PROCEDURE TView.IRes (itsDocument: TDocument; itsSuperView: TView; VAR itsParams:
Ptr);

  PROCEDURE TView.CalcMinSize (VAR minSize: VPoint);

  PROCEDURE TView.Draw (area: Rect);

  { many more methods... }

END;
```

The fSuperView and fSubviews fields are used to create view instance hierarchies. For window objects the fSuperview field is always empty, while the fSubviews field is usually filled with a list of one or more views.

The fDocument field is a reference to the view's related document. When you call NewTemplateWindow you supply a reference to the correct document. This reference is stored in the fDocument field of the window the view created.

The fLocation field represents the offset of the upper-left corner of a view from the upper-left corner of its superview.

The fSize field is the current size of the view, in pixels.

The three methods, IRes, CalcMinSize, and Draw, are explained later in the chapter.

View hierarchies have the following important characteristics:

All of the view objects that comprise a single window are organized in a view instance hierarchy. The view object representing the window frame is the top of this hierarchy.

- The window view (the view responsible for the window frame) is at the top of the view instance hierarchy and has no superview.
- Each view may have one superview and a list of subviews.
- Each view displayed within a window is a subview of the window view.
- A view is drawn on top of its superview. Thus, for a superview with a specialized view, the window will first draw the blank, white content area, and then your specialized view on top of that.
- A view is drawn clipped to its superview. This way all subviews are clipped to their window, so that no drawing is done outside the region of the window. For example, even if a main content view was larger than the window frame view, only that portion of the main content that fits inside the window frame would be displayed.
- Each view has its own coordinate system; that is, each view has a point (0,0) in its upper left hand corner. Even if the view is offset from its superview, it has its own coordinate system. Thus, when a view draws inside its own boundaries, it draws assuming that it is drawing into its own local coordinate system. For example, if your content view draws a line from the point (5,5) to (20,30), you know that that line is drawn from the fifth pixel down and over from the upper left corner of this particular view. Even if the view is offset from its window by many pixels, the drawing always occurs in the view's local coordinate system.

The view instance hierarchy is important for several reasons—drawing, for example. When a window object receives a Draw message, it first draws itself (the window frame and the empty window content), and then sends the Draw message along to each of its subviews. Each subview, in turn, draws its part of the window's contents over the blank window view.

Is the following statement still true?

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.

Creating a view by using view templates

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.

View resources can be used to create a single window object or can be used to create an entire view hierarchy.

Run-time summary of creating a view using templates

MacApp asks your application to create a new view by ***doing what?**** As default behavior, this happens when the ***what?***

- *Note:* To change MacApp's default behavior in this instance, you need to override ***what****. See Chapter X, "????" for more information.

Figure 36-X provides a summary of MacApp's actions at runtime when a new view is to be created from a template

- **Figure 36-1** MacApp's actions in relationship to this recipe

Figure TBD; for example see Chapter 14, "Documents"

Overview of your responsibilities

The actions you must take, the reasons you must take those actions, and what MacApp can provide to help you take those actions are summarized in Table 36-1. Each of the steps is explained in detail in the recipes that follow.

This section also assumes that you have already taken preliminary actions. For more information, see Chapter X, "XXXXX."

*****Please note: all of the following material has to be combined and significantly reworked*****

■ **Table 36-1** Overview: creating a view using templates

Step	Your action:	Because:
1.	Define a specialized view class.	MacApp provides a general TView class.
2.	**etc.**	**etc.**

Table TBD; for example see Chapter 14, "Documents"

Step 1 Define a specialized view class

In your interface file, define the interface for your new view class. This class should be a descendant of the class TView, should declare two new fields, and should override four methods of TView:

When you define a specialized view subclass, you will typically add many fields. For example, you generally want to add a field to your view class that reference the related document object.

Where is the relationship between views, windows, and documents explained?

All TView objects come with an fDocument field, which NewTemplateWindow initializes for you. However, this field is declared to be a reference to the generic class TDocument. Since your document objects are usually subclasses of TDocument (TIconDocument, for example), when you want to access a field or method of the TIconDocument object, you will almost always have to typecast the fDocument field:

```
TIconDocument(fDocument).DoSomething;
```

Since this continual casting can be tedious, you usually add another field to your view objects—another field that references the same document object, but doesn't require casting. For instance, if you added the following field:

```
fIconDocument : TIconDocument;
```

to your view class, you could use the fIconDocument field without casting to access the fields and methods of the document. In addition to this field, which you generally add to all of your view classes that display a document's data, you need to add any fields containing specific data about that view.

Although frequently you will add many new methods and override many others for your view objects, you must override at least three methods to create a functional view class: IRes, Draw, and CalcMinSize. These three methods, and how to override them, are explained in the next three sections.

The following sample code from UIconEdit.incl.p illustrates these steps.

```
{-------------------------------------------------------------------}
TIconView = OBJECT(TView)
  fIconDocument : TIconDocument;   {reference to related document    }
  fMagnification : integer;         { current magnification of the icon }

  PROCEDURE TIconView.IRes (itsDocument: TDocument;
                            itsSuperView: TView;
                                  itsParams: Ptr); OVERRIDE;
    { initializes the view from a resource }

  PROCEDURE TIconView.CalcMinSize (VAR minSize: VPoint); OVERRIDE;
    { Returns the view's current "minimum" size. }

  PROCEDURE TIconView.Draw (area: Rect); OVERRIDE;
    { Draws the document's icon in this view }

{$IFC qDebug}
  PROCEDURE TIconView.Fields (PROCEDURE DoToField (fieldName: Str255;
                                                   fieldAddr: Ptr;
```

```
                                                        fieldType: INTEGER));
OVERRIDE;
{$ENDC}
END;
{-------------------------------------------------------------------------}
```

Step 1 Define the new view class

In your interface file, define the interface for your new view class. This class should be a descendant of the class TView, should declare two new fields, and should override four methods of TView:

```
TIconView = OBJECT(TView)
        fIconDocument : TIconDocument;  { a reference to the related document  }
        fMagnification : integer;       { the current magnification of the icon }

        PROCEDURE TIconView.IRes (itsDocument: TDocument;
                                  itsSuperView: TView;
                                  itsParams: Ptr); OVERRIDE;
                { initializes the view from a resource }

        PROCEDURE TIconView.CalcMinSize (VAR minSize: VPoint); OVERRIDE;
                { Returns the view's current "minimum" size. }

        PROCEDURE TIconView.Draw (area: Rect); OVERRIDE;
                { Draws the document's icon in this view }

{$IFC qDebug}
        PROCEDURE TIconView.Fields (PROCEDURE DoToField (fieldName: Str255;
                                                         fieldAddr:
Ptr;
                                                         fieldType:
INTEGER)); OVERRIDE;
{$ENDC}
END;
```

Step 2 Implement the IRes method

The purpose of the IRes method is to call the inherited version of IRes to initialize the inherited fields of TView from a resource, and then toinitialize the new fields unique to TIconView.

To initialize the fIconDocument field of TIconView, you can use the itsDocument parameter of IRes. Unfortunately, this parameter is a reference to class TDocument, so you must cast it before you make the assignment:

```
fIconDocument := TIconDocument(itsDocument);
```

The other unique field of TIconView is the fMagnification field. This field should initially be set to the initial magnification of the icon. In the previous chapter you made the default window large enough to hold an icon magnified 7 times. It's a good idea to define this as a constant in the constant declaration part of the implementation file:

```
CONST kDefaultMagnification = 7;
```

and then use this constant to initialize the fMagnification field:

```
fMagnification := kDefaultMagnification;
```

Step 2 Implement the IRes method

When creating a specialized view instance, MacApp does as much initializiation as possible from the view template. However, since your view class is specialized, this means you've probably added fields to your class that MacApp doesn't already know about and thus can't initialize. Instead, when you create a view from a template, MacApp calls the IRes method of that view.

Therefore, it is your job to create an IRes method for your new view class that initializes your view class's unique fields.

```
PROCEDURE TView.IRes (itsDocument: TDocument; itsSuperView: TView;
                      VAR itsParams: Ptr);
```

In your override of IRes, you want to first call the inherited version of IRes. This allows MacApp to initialize its parts of your view from the view template. Then you want to initialize any new fields you declared for your view class.

MacApp calls your view's IRes method when creating a view from a resource, typically in response to NewTemplateWindow. In general, you can ignore the parameters to IRes. The one you might find useful is the itsDocument parameter. For example, you can use this to initialize the fIconDocument field of your icon-editting view.

Step 3 Implement your Draw method

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).

The Draw method that comes with TView is empty. If you want your view objects to draw anything, you must override the Draw method to draw the contents of the view. MacApp calls the Draw method whenever the view needs to be redrawn. This includes when the view is first shown and when the view has been covered and is uncovered by some user action. MacApp specifies which portion of the view in the area parameter.

When MacApp calls your view's Draw method, drawing has already been focused on your view. In other words, you call drawing routines using your view's local coordinate system, and the drawing will take place in the correct location. Also, all drawing that you do will be clipped to the view's fSize, and to the superview of the view.

You will often do the actual drawing by using QuickDraw commands. A complete list of these drawing routimes is available in the QuickDraw chapter of Inside Macintosh.

- Note: MacApp programmers can ignore the discussion of GrafPorts and GrafPort routines. MacApp handles those for you. You can concentrate on the drawing routines, a complete list of which is given in Appendix ???. (in which book???)

If your data consists of objects organized into a list of instances that draw themselves(generally stored in an object of class TList), and each object type has a TItem.Draw method. TItem.Draw actually draws the object.***What message is sent??*****

Any other options to drawing

If your application cannot be organized like that, have TYourView.Draw do the drawing itself.

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.

2. Implement TYourView.Draw. The interface of that method is
PROCEDURE TYourView.Draw(area: Rect);

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.

Where would optimizing drawing go?

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 in Chapter X, "Undoing" for more information.

The following sample code from UIconEdit.incl.p illustrates these steps.

```
{-------------------------------------------------------------------}
CONST kBorder = 5;  { Constant to set border of 5 pixels around icon }
...
SetRect(destRect, kBorder, kBorder,      { Location where icon should appear }
  kBorder + (32 * fMagnification),       { Current magnification of the icon }
  kBorder + (32 * fMagnification));
...
plotIcon(destRect,  fIconDocument.fIconBitMap); { Draw icon at the set location}
{-------------------------------------------------------------------}
```

Notice that this code tells the Macintosh ROM to draw the icon reference by the fIconBitMap field of this view's document object in the rectangle specified by destRect. This works only if you initialized the fIconDocument field correctly in your IRes method.

Step 4 Implement the CalcMinSize method

The purpose of CalcMinSize is to calculate the current size of the view. MacApp uses this value when drawing and printing the view, and eventually when calculating the position of scrollbars.

For example, for an icon view, you want the view to be large enough to hold the entire icon, at its current level of magnification, plus borders on both size. You return the value of this calculation in the VAR parameter minSize:

The following sample code from UIconEdit.inc1.p illustrates these steps.

```
{------------------------------------------------------------------------}
PROCEDURE TIconView.CalcMinSize (VAR minSize: VPoint); OVERRIDE;

BEGIN
  minSize.h := (32 * fMagnification) + (2 * kBorder);
  minSize.v := (32 * fMagnification) + (2 * kBorder);
END;
{------------------------------------------------------------------------}
```

The VAR parameter minSize is a VPoint record—it has an h vield for the horizontal size and a v field for the vertical size. For the displayed icons, these values are the same.

Sometimes MacApp needs to know the current size of your view. For example, when printing, or when your view has changed size and scrollbars need to be reset. In these cases, MacApp calls your view's CalcMinSize method. MacApp then takes the result of this method, uses it, and stores that value in your view's fSize field. Therefore, you never need to change the fSize fields of your view yourself. MacApp does it for you, as long as your CalcMinSize method calculates the correct current size of the view.

The TView version of CalcMinSize always returns (in the VAR parameter) the current fSize of the view. Your override version should calculate the current size of the view, and return that in the minSize parameter. MacApp will call CalcMinSize when it needs to know the size of the view, and will automatically update the fSize field for you (you never need to set the fSize field yourself???)

MacApp calls your view's Draw methods whenever it needs to know the size of a view. It uses this information for drawing, scrolling a printing. The reason that this method is called CalcMinSize, instead of simply CalcSize, is that for special types of printable views, MacApp will round the result of this method up to the nearest page size—therefore you've calculated the minimum acceptable size, and MacApp calculates the actual size. For more information, see Chapter X, "Printing."

That section does not yet exist!

Typically in your CalcMinSize procedure you analyze the fields of the view instance, and calculate its current size based on them.

Step 5 Add the view to your view template hierarchy

Now that you've defined a view for your class, you need to create an instance of that class for every window.

For the icon example, you add a TIconView template to your view template hierarchy. The NewTemplateWindow routine (that you called in DoMakeViews) then creates the view instance for you. The view template you can use should look like the template presented in this chapter.

The following sample code from UIconEdit.inc1.p illustrates these steps.

```
{------------------------------------------------------------------------}
#define kIconWindowID       1000
#define kIconWindowHeight  32 * 7 + 10
#define kIconWindowWidth   32 * 7 + 10

#define kIconViewID       1001
#define kIconViewHeight  32 * 7 + 10
#define kIconViewWidth   32 * 7 + 10

resource 'view' (kWindowRsrcID, purgeable) {
  {
  root, 'WIND', {50, 20}, {70, 70},
  sizeVariable, sizeVariable, shown, enabled,
  Window      { "TWindow", zoomDocProc, goAwayBox, resizable, modeless,
          ignoreFirstClick, freeOnClosing, disposeOnFree, closesDocument,
           openWithDocument, dontAdaptToScreen, stagger, forceonScreen,
          dontCenter, noId, ""};

  'WIND', IncludeViews   { kYourViewID }
  }
};

resource 'view' (kIconViewID, purgeable) {
  {
  root, 'SUBV', {0, 0}, {kIconViewHeight, kIconViewWidth},
  sizeVariable, sizeVariable, shown, enabled,
  View        { "TIconView" };
  }
};
{------------------------------------------------------------------------}
```

Notice that the TWindow view and the TIconView view have different view ID numbers, but at this point, they have the same size.

Initializing views from templates

For example, the following view template defines a view hierarchy of two views: a window view with one subview.

```
resource 'view' (kWindowRsrcID, purgeable) {
  {
  root, 'WIND', {50, 20}, {70, 70},
  sizeVariable, sizeVariable, shown, enabled,
  Window     { "TWindow", zoomDocProc, goAwayBox, resizable, modeless,
          ignoreFirstClick, freeOnClosing, disposeOnFree, closesDocument,
          openWithDocument, dontAdaptToScreen, stagger, forceonScreen,
          dontCenter, noId, ""};

  /* Here is the new view */
  'WIND',                       /* ID of its superview            */
  'SUBV',                       /* ID of this view                */
  {0, 0},                            /* location in its superview      */
  {kHeight, kWidth},                 /* size of this view              */
  sizeVariable, sizeVariable,   /* size determiners of this view */
  shown, enabled,               /* other attributes of this view */
  View                                        /* the type of the view resource */
  { "TYourView" };              /* the class of this view         */
  }
};
```

This definition specifies a view instance hierarchy. The window view (with view ID 'WIND') is the root of the hierarchy. The view (with view ID 'SUBV') is a subview of the window view. The first field of the template for the new view is 'WIND'. This specifies that the superview (of this new view) is to be the 'WIND' view, or the window view defined above. The second field is 'SUBV'. This is the view ID of the new view.

The next field is the location of the new view in its superview. In this case, the upper-left corner of this view is located exactly at the upper-left corner of its superview (the upper-left corner of the content region of the window.) The value of {0, 0} specifies this. The next field is the initial height and width of the view. Typically you will make these constants in your rez file.

The next four fields are other (attributes of the view—the view's size determiners, and whether the view is shown and enabled. ***Need to describe these here***

Specify the type of the view resource. In the case of the window object, you put the word Window, and then followed it by a set of parameters specific to windows. Since this new view is a special class defined by you (TYourView), MacApp doesn't know anything about it. Therefore, you put the keyword View in this field. This specifies to MacApp that this is a specialized view class, not a predefined one.

Give the specific class name of the view. This allows MacApp to create the correct type of view object in response to NewTemplateWindow.

Notice that the window template had many more fields than the new view template does. This is because MacApp knows about windows, and can do a lot of initialization for you. Unfortunately, MacApp doesn't know about your specially-defined views. So, instead of completely initializing them from templates, you have to initialize your view object's special fields in its IRes method.

Reading a specialized view

Sometimes you will want to read a specialized view in separately from its window superview (normally for printing reasons, explained in chapter ???)

You can split the view resource out of the view hierarchy resource using the IncludeViews directive, like this:

```
resource 'view' (kWindowRsrcID, purgeable) {
  {
  root, 'WIND', {50, 20}, {70, 70},
  sizeVariable, sizeVariable, shown, enabled,
  Window      { "TWindow", zoomDocProc, goAwayBox, resizable, modeless,
          ignoreFirstClick, freeOnClosing, disposeOnFree, closesDocument,
          openWithDocument, dontAdaptToScreen, stagger, forceonScreen,
          dontCenter, noId, ""};

  'WIND', IncludeViews   { kYourViewID }
  }
};

resource 'view' (kYourViewID, purgeable) {
  {
  root, 'SUBV', {0, 0}, {kYourViewHeight, kYourViewWidth},
```

```
  sizeVariable, sizeVariable, shown, enabled,
  View        { "TYourView" };
  }
};
```

Because the 'SUBV' subview has its own resource number—kYourViewID—the subview can be created separately from the window. However, since the window resource has an IncludeViews directive, whenever the window is created, the subview will automatically be created too.
Creating your view instance

Once your view class is defined, you need to actually create your view instance. This is done in the DoMakeViews method of your document object, just as your window object was. Remember in your document's DoMakeViews method that you made the call

```
aWindow := NewTemplateWindow(kIconWindowID, SELF);
```

Since NewTemplateWindow can read in and create an entire view hierarchy, all you need to do is add the new view to your view template. NewTemplateWindow will do the rest. Therefore, you don't have to change the implementation of DoMakeViews at all to create the entire view hierarchy—only the view template is changed.

Creating a view

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 "Supporting The Clipboard" Chapter X.

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

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;


        { See "Constants" in Chapter 9, for the other possible values. For
          the significance of the other parameters, see the description of IView under
"TView" in Chapter 10.}
```

Creating view templates

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.

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. */
        }
};
```

Each view entry in the resource file has a format like this:

```
ParentViewsID, ThisViewsID, LocationInParentView, SizeOfView,
VerticalSizeDeterminer, HorizontalSizeDeterminer, ShownDeterminer, EnabledDeterminer,
TypeOfView { "class name", <more fields depending on type of view> }
```

These fields are as follows:

- **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. 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. 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.

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 using templates

Once you've created your view resources, you must still call the correct routines to create the actual view instances.

How to do it

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;
```

Focusing a view

new B7 stuff about asserting the focus for a view

Showing a reduced view

*****Suggestion on MacAppTech$*****

There must be a simple way to show reduced view like MacDraw's "Reduce to Fit". However, we don't know how to do it. It would be great to be able to draw and handle mouse commands in the reduced view without having to define another object type or create an instance of it.

Changing the size of a view

You should call the AdjustSize method of your view whenever you change its size. AdjustSize figures out the new size of the view (by calling your CalcMinSize method), and does the necessary calculations to ensure that the view continues to be refreshed correctly, and also that the scroll bars are in synch with the new view size.

AdjustSize takes no parameters; simply make the call, which informs MacApp that the size of your view needs to be adjusted, and MacApp takes care of the rest.

The following sample code from UIconEdit.incl.p illustrates this step.

```
{-------------------------------------------------------------------}
PROCEDURE TIconEditView.SetMagnification (magnification: INTEGER);

BEGIN
  fMagnification := Max(1, magnification);   { Set the new magnification.   }
  AdjustSize;                          { Magnification affects the view's size.}
  ForceRedraw;                               { Force the view to be entirely redrawn.}
END;
{-------------------------------------------------------------------}
```

Forcing a view to redraw

MacApp normally redraws your view whenever necessary (for example, when the view's window has been covered but then is brought to the front). However, when you change the appearance of a view you should call the ForceRedraw method of your view. ForceRedraw takes no parameters; simply make the call, which informs MacApp that your view needs to be redrawn, and MacApp takes care of the rest.

The following sample code from UIconEdit.incl.p illustrates this step.

```
{-------------------------------------------------------------------}
PROCEDURE TIconEditView.SetMagnification (magnification: INTEGER);

BEGIN
  fMagnification := Max(1, magnification);   { Set the new magnification.   }
  AdjustSize;                          { Magnification affects the view's size.}
  ForceRedraw;                               { Force the view to be entirely redrawn.}
```

```
END;
{-------------------------------------------------------------------}
```

Freeing reuseable views

*****Suggestion on MacAppTech$. Will it be implemented, or is there a workaround? Please note that the "I" is the original author of the suggestion, and not the manual writer*****

I am writing a very large application that will have lots of views. If I free each one after closing, then there is a performance penalty if the user wants to open it again. In tight memory situations or under multi-finder, there won't be enough room to keep them all open.

I have a general idea for a solution that would become part of MacApp.

When running low on memory (when printing, attempting to open a new view, etc.), the application would look at the list of windows for the document and would free windows according to several criteria. Windows would have an fLowMemoryCanFree, fLastOpened, and an fFreePriority. The routine would use these (an maybe some others) in a intelligent fashion when deciding which to free.

MacApp might also call a user method to free any non essential stuff determined by the user.

I don't know how MacApp is used by most users, but in an accounting application such as mine, each window roughly corresponds to an accounting module that the user may wish to use again shortly after closing. It sometimes takes many windows to complete one module.

I create all of the windows for a module at once when requested. Some windows are reused throughout the program and would have a very high priority. Other windows are only useful when the "Main window" for the module exists so it would not be a good idea to free some of a "Main window's" subsidiary windows and not others. Have all of them exist or none of them.

This would be accomplished with by setting the fLowMemoryCanFree to false for the subsidiary windows and having the main window free them if called (or use some sort of a fIsSubsidary variable?). If a two "Main" windows had the same priority, it would be useful to have an fLastOpened variable so as to free the oldest.

Freeing windows this way would also take care of the problem with freeing a Modeless dialog window from a button or such. (BTW, the oct 31 suggestion of using CloseByUser to do solve this problem will crash the program).

Because the "Main" window may not be visible or a target (one of its sub windows may be in front obscuring it), there should probably be a variable to prevent this "main" window from being freed while it (the module) is in use or being opened.

Chapter 37 **Windows**

Should this chapter be combined with the View chapter?

Macintosh applications display the data for a document in windows. Each window consists of many different parts. For example, a window can have a window frame (which may include a title bar, grow box, zoom box, and close box), scroll bars, and the main content of the window. In MacApp, each part of the window is represented by a view object.

In other words, one view object represents the window frame, another represents each scroll bar, and yet another represents the main content of the window. Each view object can draw the part of the window it represents, and each view object can respond to events relating specifically to its part.

For example, the view object representing the window frame is responsible for drawing the window frame and the blank white background of the window, as well as responding to mouse clicks in the window frame, which may signify dragging, resizing, zooming, or closing the window. The view object representing a scrollbar is responsible for drawing the scrollbar, as well as responding to mouse clicks in the scrollbar. Typically these mouse clicks cause the scrollbar to be redrawn, and a message to be sent to the main content view of the window telling it to scroll itself.

The view object representing the main content of the window varies widely from application to application. Each window on the Macintosh screen is actually represented by a number of view objects, each representing some part of the window.

To handle your application's documents, you can:

- Create a new window procedurally
- etc. **(***on through all the recipes provided by the chapter***)**

This chapter describes in detail the steps you need to take to accomplish these tasks.

The view instance hierarchy

All of the view objects that comprise a single window are organized in a **view instance hierarchy**. The view object representing the window frame is the top of this hierarchy. Each view displayed within the window is a subview of the window view.

Subviews are drawn on top of their superviews. In other words, the main content view and the scrollbar views are all drawn on top of the blank window view. However, subviews are also clipped to the boundary of their superview. In other words, even though the main content view is larger than the window frame view, only that portion of the main content that fits inside the window frame is displayed.

This hierarchy is important for several reasons—drawing, for example. When a window object receives a Draw message, it first draws itself (the window frame and the empty window content), and then sends the Draw message along to each of its subviews. Each subview, in turn, draws its part of the window's contents over the blank window view.

First, there aren't any scroll bars. In MacApp, scroll bars are not a property of a window. Second, we got the zoom box, grow box and close box because the window's 'view' resource indicates that they be included. This behavior is built into the TWindow class. The windows are also staggered when created. Staggering is also built into the TWindow class, and is controlled in the window's resource.

explain difference between procedures and templates?

Creating a window procedurally

Please note: the rest of the stuff in this chapter is grabbed from the Interim cookbook and hardly edited at all. It needs extensive rework to fit into the proposed format

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 Chapter X, "Views".

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.

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.

```
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 scrolling window

cross-reference to scrolling chapter

Creating a palette window

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.

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.

```
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

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.

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.

```
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

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.)

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 in Chapter X, "Views".
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.

```
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 a window with multiple scrollable views that resize with the window

*****Possible submission from the recipe contest*****