This chapter describes the general characteristics of a recordable application and provides some examples of how to factor your application for recording. It also provides guidelines to help you decide which user actions to record and how to record them.

Before you read this chapter, you should read the chapter "Introduction to Scripting" in this book. To factor your application, you must know how to respond to Apple events, create and send Apple events, and resolve and create object specifier records. For comprehensive information about implementing Apple events, see the chapters "Introduction to Apple Events," "Responding to Apple Events," "Creating and Sending Apple Events," and "Resolving and Creating Object Specifier Records" in this book.

The first three sections in this chapter provide

- a description of the basic requirements for recordable applications
- examples of how to begin factoring your application
- guidelines for what to record

The fourth section describes how Apple event recording works. You need to read it only if you are developing a script editor, an application that can initiate recording, or a scripting component.

About Recordable Applications

A ***recordable application*** is an application that uses Apple events to report user actions to the Apple Event Manager for recording purposes. One way to do this is to separate the code that implements your application's user interface from the code that actually performs work when the user manipulates the interface. This is called ***factoring*** your application. A factored application translates low-level events generated by the user into recordable Apple events that the application sends to itself to perform tasks.

A ***recordable event*** is any Apple event that any recordable application sends to itself while recording is turned on for the local computer, with the exception of events that are sent with the `kAEDontRecord` flag set in the `sendMode` parameter of `AESend`. A ***recording process*** is any process (for example, the Script Editor application) that can turn recording on and off and can receive and record recordable Apple events.

After Apple event recording has been turned on by a recording process, the Apple Event Manager sends that process copies of all recordable Apple events on the local computer. For example, when a user presses the Record button in the Script Editor application, it calls a scripting component routine to turn on recording for the AppleScript component (or any other scripting component). While recording is on, the Apple Event Manager sends Script Editor copies of all subsequent recordable Apple events, which Script Editor records (with the aid of the scripting component) in the form of a compiled script. After turning off recording from Script Editor, the user can edit or execute the recorded script.

Although factoring your application is the recommended method of making your application recordable, it is also possible to report user actions by means of Apple events only when Apple event recording is turned on, even though the application may respond to those actions by some means other than Apple events. In effect, the application uses Apple events to describe user actions without actually using the events to perform the action. To indicate that you want the Apple Event Manager to send a copy of a recordable event to the recording process without actually sending the event to your application, add the constant `kAEDontExecute` to the `sendMode` parameter of the `AESend` function.

Even in a factored application, it may not always be possible to send an Apple event that actually executes the task initiated by the user. For example, if the user types some text, it is more practical to use standard TextEdit or QuickDraw routines to draw the text than to send a separate Apple event each time the user presses a key. In this case, the application can draw the text as it is typed in the most convenient manner available; then, when the user finishes typing a sequence of characters—by clicking the mouse button while the cursor is somewhere else in the document or performing some other action—the application can create an Apple event that corresponds to the typing and add the constant `kAEDontExecute` to the `sendMode` parameter when it sends the Apple event.

If your application needs to know when Apple event recording is turned on and off, it should install handlers for the Recording On and Recording Off events.

Recording On—perform actions associated with beginning of recording session

Event class	`kCoreEventClass`
Event ID	`kAENotifyStartRecording`
Parameters	None
Description	Sent by the Apple Event Manager to all running processes on the local computer to inform them that recording has been turned on

Recording Off—perform actions associated with end of recording session

Event class	`kCoreEventClass`
Event ID	`kAENotifyStopRecording`
Parameters	None
Description	Sent by the Apple Event Manager to all running processes on the local computer to inform them that recording has been turned off

When a recording process turns on recording, the Apple Event Manager sends all running processes on the local computer a Recording On event. When a user turns off recording, the Apple Event Manager sends all running processes the Recording Off event with the `kAEWaitReply` flag set. If an application has stored some data (for instance, keystrokes) that needs to be recorded as an Apple event, this is the last chance to send an event for recording purposes. If your application needs to know which recording process has turned recording on or off, it can check the `keyOriginalAddressAttr` attribute of the Recording On or Recording Off event for the address of the recording process.

Factoring your application is the recommended method of making your application recordable because it guarantees that any action a user can perform via your application's user interface can also be accomplished via Apple events. Factoring also allows you to avoid duplicating code within your application. Instead of using one piece of code to respond to some user action within your application, and another piece of code to respond to the equivalent Apple event, you can use the same code to respond to the Apple event, whether it is sent by your application in response to a user action, by some other application, or by a scripting component in the course of executing a script.

The next section, "Factoring Your Application for Recording," provides some examples of how to go about factoring your application. Regardless of how you factor your application, making it recordable requires you to make decisions about the most useful methods of recording user actions that can be described in several different ways in scripts. If a user moves a window, for example, the window can be described in the corresponding recorded script as `window 1`, or `the window named Fred`, or `the first window`. Although the OSA permits recording at a high level and thus avoids many of the problems users encounter with applications that record low-level events such as keystrokes and mouse clicks, scripting components cannot predict what information a user cares about in a given situation. Therefore, a recordable application should send Apple events that correspond to the simplest possible statements in a scripting language. "What to Record," which begins on page 9-14, provides some general guidelines for making these kinds of decisions.

"***How Apple Event Recording Works,"*** which begins on page 9-35, describes the Apple Event Manager's recording mechanism in more detail, including the role of the Recording On and Recording Off events.

Factoring Your Application for Recording

The recommended way to make your application recordable, or capable of sending Apple events to itself whenever a user performs a significant action, is to factor the code that controls your application's user interface from the code that responds to the user's manipulation of the interface. A fully factored application translates user actions into Apple events that the application sends to itself to initiate tasks.

The examples that follow demonstrate how to factor code that responds to relatively simple user actions such as creating a new document or moving a window. They are intended only to illustrate the general approach you should take; many of the decisions you will need to make while factoring will be unique to your application. "What to Record," which begins on page 9-14, provides guidelines for deciding which user actions to record and how to record them. For examples of factored applications, see the *AppleScript Software Developers' Kit.*

If you are factoring an existing application, it's usually a good idea to begin with the required Apple events and any other Apple events that you plan to send in order to execute commands in the File menu. You can then proceed to other menu commands and mouse actions. If you are designing a new application and want to make it recordable, you should build factoring into every aspect of your application design.

Factoring the Quit Command and the New Command

This section demonstrates how to factor two File menu commands: Quit and New.

When the user chooses a menu command, an application first determines which one was chosen and then performs the action associated with that command. For example, when a user chooses Quit from the File menu, an application that is not factored simply calls an application-defined `DoQuit` routine. Because Quit Application is one of the required Apple events, it is relatively easy for most applications that support Apple events to factor the code that responds to the Quit command.

After a factored application has determined that the user has chosen the Quit command, it sends the Quit Application event to itself by calling its `MyDoMenuQuit` routine.

```
PROCEDURE MyDoMenuQuit;
VAR
   myErr: OSErr;
BEGIN
   myErr := MySendAEQuit(kAEAskUser);
   {handle any errors}
END;
```

The `MyDoMenuQuit` routine in turn calls the `MySendAEQuit` routine shown in Listing 9-1, which creates the Quit Application event and sends it.

Listing 9-1 A function used by a factored application to send itself a Quit Application event

```
FUNCTION MySendAEQuit (saveOpt: DescType): OSErr;
VAR
   myAppleEvent, defReply: AppleEvent;
   myErr, ignoreErr:       OSErr;
BEGIN
   {create Quit event}
   myErr := AECreateAppleEvent(kCoreEventClass,
                                 kAEQuitApplication,
                                 gSelfAddrDesc,
                                 kAutoGenerateReturnID,
                                 kAnyTransactionID, myAppleEvent);
   IF myErr = noErr THEN
      {add optional parameter that specifies whether this app }
      { should prompt user if window is dirty}
      myErr := AEPutParamPtr(myAppleEvent, keyAESaveOptions,
                              typeEnumerated, @saveOpt,
                              SizeOf(saveOpt));
   IF myErr = noErr THEN
      {send event}
      myErr := AESend(myAppleEvent, defReply,
                        kAENoReply+kAEAlwaysInteract,
                        kAENormalPriority, kAEDefaultTimeOut,
                        NIL, NIL);
   MySendAEQuit := myErr;
   ignoreErr := AEDisposeDesc(myAppleEvent);
END;
```

The input to the `MySendAEQuit` routine is a constant that indicates whether to save dirty windows without asking the user (`kAEYes`), quit without saving dirty windows (`kAENo`), or ask the user whether each dirty window should be saved (`kAEAskUser`). In this example, the constant `kAEAskUser` passed to the `MySendAEQuit` routine indicates that the user will be asked whether each dirty window should be saved.

After the application receives the Quit Application event, the `MyHandleQuit` handler shown in Listing 9-2 performs all the actions associated with that event, such as saving any open documents. (Note that your application should call the `ExitToShell` procedure from the main event loop, not from your handler for the Quit Application event.)

Listing 9-2 A routine used by a factored application to handle a Quit Application event

```
FUNCTION MyHandleQuit (theAppleEvent, reply: AppleEvent;
                        handlerRefcon: LongInt): OSErr;
VAR
   userCanceled:           Boolean;
   saveOpt, returnedType:  DescType;
   actSize:                Size;
   myErr:                  OSErr;
BEGIN
   {check for missing required parameters}
   myErr := MyGotRequiredParams(theAppleEvent);
   IF myErr = noErr THEN
   BEGIN
      {pick up optional save parameter}
      saveOpt := kAEAskUser;  {the default}
      myErr := AEGetParamPtr(theAppleEvent, keyAESaveOptions,
                             typeEnumerated, returnedType,
                             @saveOpt, SizeOf(saveOpt), actSize);
      IF myErr = errAEDescNotFound THEN
         myErr := noErr;
      MyHandleQuit := myErr;
      IF myErr = noErr THEN
      BEGIN
         userCanceled := MyPrepareToTerminate(saveOpt);
         IF userCanceled THEN
            MyHandleQuit := kUserCanceled;
      END;
   END
   ELSE
      MyHandleQuit := myErr;
END;
```

The handler in Listing 9-2 calls another function supplied by the application, the `MyPrepareToTerminate` function. When the value of the optional parameter that specifies how to deal with dirty windows equals `kAEAskUser`, this function asks the user whether to save each dirty window and returns a Boolean value that indicates whether the user canceled the Quit request. It also responds appropriately to the other possible values of the optional parameter.

If recording has been turned on for a scripting component (for example, after a user clicks the Record button in the Script Editor application) and the user quits the application, the Apple Event Manager automatically sends the scripting component a copy of the Quit Application event sent by the `MySendAEQuit` routine. The scripting component records the event in a compiled script. When a user executes the recorded script, the scripting component sends the same Quit Application event to the application, which calls the `MyHandleQuit` function and responds to the event just as if the user had chosen Quit from the File menu.

After you have factored the commands associated with required Apple events for an existing application, you can move on to the other commands in the File menu, such as New. After a factored application has determined that the user has chosen New, it calls its `MyDoMenuNew` routine, which sends the Create Element event to the application.

```
PROCEDURE MyDoMenuNew;
VAR
   myErr := OSErr;
BEGIN
   myErr := MySendAECreateElement(gNullDesc, cDocument);
   {handle any errors}
END;
```

The container for the new element is the application's default container, specified by a null descriptor record, and the desired class is `cDocument`. The `MyDoMenuNew` routine in turn calls the `MySendAECreateElement` routine shown in Listing 9-3, which creates the Apple event and sends it.

Listing 9-3 A routine used by a factored application to send itself a Create Element event

```
FUNCTION MySendAECreateElement (cont: AEDesc;
                                elemClass: DescType): OSErr;
VAR
   myAppleEvent, defReply: AppleEvent;
   myErr, ignoreErr:       OSErr;
BEGIN
   {create Create Element event}
   myErr := AECreateAppleEvent(kCoreEventClass, kAECreateElement,
                               gSelfAddrDesc,
                               kAutoGenerateReturnID,
                               kAnyTransactionID, myAppleEvent);
   IF myErr = noErr THEN
      {add parameter that specifies insertion location for the }
      { new element}
      myErr :=  AEPutParamDesc(myAppleEvent,keyAEInsertHere,cont);
   IF myErr = noErr THEN
      {add parameter that specifies new element's object class}
      myErr := AEPutParamPtr(myAppleEvent, keyAEObjectClass,
                             typeType, @elemClass,
                             SizeOf(elemClass));
   IF myErr = noErr THEN
      {send the event}
      myErr := AESend(myAppleEvent, defReply,
                      kAENoReply+kAECanInteract,
                      kAENormalPriority, kAEDefaultTimeOut, NIL,
                      NIL);
   MySendAECreateElement := myErr;
   ignoreErr := AEDisposeDesc(myAppleEvent); {must dispose of }
                                             { event}
END;
```

For the purposes of this example, the routine shown in Listing 9-3 sends only the required parameters and can only create a new active window with the default name. After the application receives the Create Element event, its `MyHandleCreateElement` handler performs the requested action, as shown in Listing 9-4. In this case, it creates a new active window with a default title.

Listing 9-4 The Create Element event handler for a factored application

```
FUNCTION MyHandleCreateElement (theAppleEvent: AppleEvent;
                                 reply: AppleEvent;
                                 handlerRefCon: LongInt): OSErr;
VAR
   myCont:                      AEDesc;
   returnedType, newElemClass: DescType;
   actSize:                     Size;
   contClass:                   DescType;
   window:                      WindowPtr;
   myErr:                       OSErr;
BEGIN
   {get the parameters out of the event}
   {first get the direct parameter, which specifies insertion }
   { location for new window--that is, frontmost window}
   myCont.dataHandle := NIL;
   myErr := AEGetParamDesc(theAppleEvent, keyAEInsertHere,
                           typeWildCard, myCont);
   IF myErr = noErr THEN
      {get the other required parameter, which specifies class }
      { cDocument when MyHandleCreateElement creates a new window}
      myErr := AEGetParamPtr(theAppleEvent, keyAEObjectClass,
                             typeType, returnedType,
                             @newElemClass,
                             SizeOf(newElemClass), actSize);
   IF myErr = noErr THEN
      myErr := MyGotRequiredParams(theAppleEvent);
   MyHandleCreateElement := myErr;
   IF myErr = noErr THEN
   BEGIN
      {check container and class, just to make sure}
      IF (myCont.descriptorType <> typeNull) OR (newElemClass <>
            cDocument) THEN
         MyHandleCreateElement := kWrongContainerOrElement
      ELSE
         {MyNewWindow creates a new window with a default name }
         { and returns a pointer to it in the window parameter}
         MyHandleCreateElement := MyNewWindow(window);
   END;
   myErr := AEDisposeDesc(myCont);
   {if your app sends a reply in response to the Create Element }
   { event, then set up the reply event as appropriate}
END;
```

If recording has been turned on for a scripting component (for example, after a user clicks the Record button in the Script Editor application), the Apple Event Manager automatically sends the scripting component a copy of the Create Element event sent by the `MySendAECreateElement` routine. The scripting component records the Apple event as a statement in a compiled script. When a user executes the recorded script, the scripting component sends the same Create Element event to the application, which calls its `MyHandleCreateElement` handler and responds to the event just as if the user had chosen New from the File menu.

Sending Apple Events Without Executing Them

If an application is fully factored, it carries out almost all the tasks a user can perform by sending itself Apple events in the manner illustrated by the listings in the preceding sections. However, in some cases it may not be practical to send an Apple event that actually executes a task performed by the user.

For example, if the user drags a window by its title bar from one position to another, it is inefficient to send a series of Apple events that move the window through a series of positions until the user releases the mouse button. Instead, your application can call the Window Manager routine `DragWindow` to allow the user to drag the window to a new position. Until the user releases the mouse button, it's not possible to send a single Apple event that drags the window to the new position, because the new position is not yet known. When `DragWindow` returns, the window has already been dragged to its new position, and its window record has been updated.

At this point your application can send itself the Set Data event that performs the same action; but to avoid repeating the action that was just performed with `DragWindow`, you should add the `kAEDontExecute` constant to the `sendMode` parameter of the `AESend` function when you send the event. The Apple Event Manager then sends the Set Data event to the recording process, if any, but does not send it to the application.

Listing 9-5 shows an application-defined routine, `MyDoDragWindow`, that illustrates this approach. The `MyDoDragWindow` routine calls `DragWindow` in the usual way, then uses another application-defined routine, `MyCreateAESetWindowPos`, and the `AESend` function to create and send a Set Data Apple event that sets the window position to the new location. However, because the window has already been moved, there is no need to execute the Set Data event. To send the event for recording purposes without actually executing it, the `MyDoDragWindow` routine adds the `kAEDontExecute` constant to the `sendMode` parameter of the `AESend` function when it sends the Set Data event.

Listing 9-5 A routine used by a factored application to handle window movement

```
PROCEDURE MyDoDragWindow (theWindow: WindowPtr; startPt: Point;
                          boundsRect: Rect);
VAR
   newPos:        Point;
   index:         Integer;
   theAppleEvent: AppleEvent;
   reply:         AppleEvent;
   myErr:         OSErr;
BEGIN
   DragWindow(theWindow, startPt, boundsRect);
   newPos := WindowPeek(theWindow)^.contRgn^^.rgnBBox.topLeft;
   index := MyIndexFromWndwPtr(theWindow);
   MyCreateAESetWindowPos(index, newPos, theAppleEvent);
   myErr := AESend(theAppleEvent, reply, kAENoReply +
                     kAECanInteract + kAEDontExecute,
                     kAENormalPriority, kAEDefaultTimeout, NIL,
                     NIL);
END;
```

If recording has been turned on and the user moves a window, the Apple Event Manager automatically sends the scripting component a copy of the Set Data event sent by the `MyDoDragWindow` routine but does not send the event to the application. The scripting component records the event as a statement in a compiled script. When a user executes the recorded script, the scripting component sends the same Set Data event to the application. The application's handler for the Set Data event then changes the position of the window.

What to Record

Factoring an application involves making decisions about which user actions generate Apple events, about the content of those events, and about when to send events for recording purposes. For example, the preceding section, "Sending Apple Events Without Executing Them," describes how an application should generate an Apple event that corresponds to a change in the position of a window. Other actions can be more complicated to define in terms of Apple events. This section provides general guidelines for deciding which user actions should generate Apple events and how those events should be defined.

When the user records a series of actions as a script, playing the recorded script back later in exactly the same circumstances must produce exactly the same result. If the circumstances at execution time are similar but not exactly the same as when the script was recorded, the script should also work correctly. However, certain differences will always lead to unexpected results or cause execution to fail.

The goal of these guidelines is to help you create scripts that will work correctly in the largest number of circumstances with the fewest post-recording changes by the user. To accomplish this goal, a recordable application should send itself Apple events that describe as specifically as possible the user's actions in the application's domain without making guesses about the user's intentions.

The way your application uses Apple events to record a user's actions depends in part on the kind of script being recorded. From the user's perspective, there are at least three kinds of scripts:

- A script application. The icons for these files appear in the Finder, for example, in the Apple Menu Items folder or the Startup Items folder.
- A script that functions like a menu command, usually acting on the current selection in the current application, and stored either as a compiled script file that appears in the Finder or as a script stored within an application or one of its documents.
- A script that is "embedded" in an application—that is, explicitly associated with something in a document, such as a field in a form, a cell or row of a spreadsheet, or a button.

The recording guidelines in the sections that follow apply to the recording of scripts that function like menu commands and scripts that are embedded in an application. Because such scripts are executed under a user’s direct control, the user expects their execution to cause something to happen, possibly changing the current selection, the Clipboard, or the active window.

The execution of a script application, however, may cause a scripting component to send events to one or more applications intermittently without the user’s knowledge. If the script in a script application refers to the current selection, the Clipboard, or the active window, its execution may interfere with other tasks being performed by the user or tasks performed during the execution of other scripts. To create a script application and ensure that it works correctly when executed, a scripter may need to modify the script after it has been recorded.

For example, to eliminate references to the Clipboard, a scripter can use a script variable as a user-defined Clipboard and convert Cut, Copy, and Paste statements to appropriate combinations of Move, Copy, New, and Delete statements, while supplying the previously defined selection as the argument. It may also be necessary to convert a description such as “the front document” to a specific filename or a variable.

Recording User Actions

Two general guidelines apply to the recording of all user actions:

- Send Apple events that correspond to simple statements in a script rather than compound statements.
- Don’t record superfluous actions.

In most cases, if the user performs several related actions, your application should send Apple events for each action rather than saving the actions and creating an event that combines them.

For example, if the user selects some text, cuts it, and then pastes it somewhere else, your application should send itself four events that correspond to these actions:

1. Select the text
2. Cut
3. Set the insertion point
4. Paste

Thus, if the user selects characters 5 through 20 of the frontmost document, chooses the Cut command from the Edit menu, places the insertion point after character 72, and chooses the Paste command, your application should send the following events.

- A Select event (event class `kAEMiscStandards`, event ID `kAESelect`) with this direct parameter:

Keyword	**Descriptor type**	**Data**
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cChar`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cDocument`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`1`
`keyAEKeyForm`	`typeEnumerated`	`formRange`
`keyAEKeyData`	`typeRangeDescriptor`	(see indented record)
`keyAERangeStart`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cChar`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`5`
`keyAERangeStop`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cChar`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`20`

- A Cut event (event class `kAEMiscStandards`, event ID `kAECut`)

- A Select event with this direct parameter:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cInsertionLoc`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cChar`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cDocument`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`1`
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`72`
`keyAEKeyForm`	`typeEnumerated`	`formRelativePosition`
`keyAEKeyData`	`typeEnumerated`	`kAEAfter`

- A Paste event

Note

The format used for the direct parameters in this example and throughout this chapter does not show the structure of the direct parameters as they exist within the Apple events. Instead, this format shows what you would obtain after calling `AEGetKeyDesc` repeatedly to extract the nested descriptor records from the Apple events.

When you call `AEGetKeyDesc` to extract the descriptor record that specifies an application's default container, `AEGetKeyDesc` returns a descriptor record of type `AEDesc` with a descriptor type of `typeNull` and a data handle whose value is 0. ◆

The first Select event in this example sets the application's `pSelection` property (that is, the current selection) to the objects identified by the object specifier record in the direct parameter—characters 5 through 20. The second Select event places the insertion point after the object identified by the object specifier in the direct parameter—after character 72.

You could also interpret these four actions as a single Move event that simply moves characters 5 through 20 to after character 72. A user could write such a statement in a script, but for recording purposes four separate events correspond more precisely to the user's actions. For example, if the user performed another paste operation after the first four actions, a Move event would not produce the correct results.

It is equally important for a recordable application not to send superfluous events. For example, your application should not send an event every time the user makes a selection. Instead, it should keep track of the most recent selection made. When the user performs some action on the selection, the application should send an event that sets the selection followed by the event that corresponds to the action taken by the user. However, if the user doesn't perform an action on the selection, the application should not send an event.

IMPORTANT

If something is already selected when recording begins, your application should not record that selection. Subsequent user actions should be recorded assuming that there is a selection. By not recording the current selection, you allow the user to record scripts that work, without further modification, much like menu commands that operate on the current selection. ▲

The example just discussed assumes that the application has multiple documents. In such an application, document 1 is always the document in the frontmost window. The examples that follow are simplified, as if they were generated by an application like TeachText that can have only one document open at a time and can therefore locate objects such as characters in the default container. For more complex applications that locate text in cells, documents, and other containers, you must specify additional containers as appropriate.

Recording the Selection of Text Objects

When your application needs to record a selection that the user has made by dragging through a range of text, it should send itself a Select event that selects a range of characters. For example, a Select event with this direct parameter selects characters 80 through 764:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cChar`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formRange`
`keyAEKeyData`	`typeRangeDescriptor`	(see indented record)
`keyAERangeStart`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cChar`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`80`
`keyAERangeStop`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cChar`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`764`

It is sufficient to record such a text selection as a range of characters. However, recording selections in other units can make the corresponding scripts easier to read. If you decide to record text selections in other units, keep these guidelines in mind:

- Use the largest whole unit that completely describes the selection.
- Do not mix units.
- Use units appropriate to the method of selection.
- Use logical units rather than units that vary with reformatting.
- Don’t try to guess the user’s intentions.

The rest of this section provides examples of how to apply these guidelines.

If you do record text selections in units other than characters, record each selection in terms of the largest whole unit that completely describes the selection. For example, suppose the user selects characters 115 through 170 by dragging. Further, suppose the selected characters are exactly the same as words 33 through 50 and also the same as paragraph 2. In this case your application should send itself a Select event with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cParagraph
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	2

However, if the selected characters don't match a larger unit exactly—for example, if paragraph 2 is larger than the selection or the selection is a portion of two paragraphs—use the largest unit available, in this case words.

For example, a Select event with this direct parameter selects word 33 through word 45:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cText
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formRange
keyAEKeyData	typeRangeDescriptor	(see indented record)
keyAERangeStart	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	33
keyAERangeStop	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cWord
keyAEContainer	typeCurrentContainer	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	45

Do not mix units. You should not send Apple events that define selections like character 2 of word 3 of line 5 of paragraph 2 in document "MyDocument." Instead, define selections as simply as possible; for example, character 45 in the document "MyDocument."

When the user selects text by double-clicking it, your application should send a Select event that specifies words. For example, your application should send a Select event with this direct parameter when the user double-clicks word 5:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cWord`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	5

If the user double-clicks word 5 and then extends the selection through word 9, your application should send a Select event with this direct parameter:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cText`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formRange`
`keyAEKeyData`	`typeRangeDescriptor`	(see indented record)
`keyAERangeStart`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cWord`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	5
`keyAERangeStop`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cWord`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	9

If your application supports selection of a paragraph, for example by clicking the left margin, triple-clicking, or some other action, your application should send a Select event that selects the paragraph. For example, a Select event with this direct parameter selects paragraph 2:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cParagraph`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	2

If your application supports the selection of other units (for instance, cells, rows, and columns in a spreadsheet; embedded graphics in a word processor; or buttons) and if users can select a range of such units, your application should record using those units when appropriate. For example, a Select event with this direct parameter selects row 5 through row 23:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cRow`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formRange`
`keyAEKeyData`	`typeRangeDescriptor`	(see indented record)
`keyAERangeStart`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cRow`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	5
`keyAERangeStop`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cRow`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	23

A Select event with this direct parameter selects the second `'PICT'` image:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cPICT`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`2`

When the user chooses a Select All command, your application should send a Select event with this direct parameter to select the contents of the document:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cProperty`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
`keyAEKeyData`	`typeLongInteger`	`pContents`

Units that vary with reformatting, such as lines and pages in a text document, are not as useful as logical units that describe the data more precisely. Whenever possible, use logical units such as character, word, paragraph, section, and so on.

Don’t try to guess the user’s intentions. For example, if a selection can be described as either “word 14” or as “the third bold word in paragraph 3,” use the simpler description. If you guess the user’s intentions, you will be wrong often enough to cause the user to distrust the recording process.

Recording Insertion Points

The insertion point and a selection are synonymous in the Macintosh Operating System. However, scripting languages need a way of specifying a zero-width selection. Sometimes the best way to specify an insertion location is in relation to another object; for example, “after word 5.” This section describes recommended methods of specifying an insertion point in a recordable event.

The insertion point can be specified in Apple events by either an insertion location descriptor record (`typeInsertionLocation`) or an object specifier record (`typeObjectSpecifier`) that specifies the class `cInsertionLoc` and the key form `formRelativePosition`. The Move, Clone, and Create events accept an insertion location descriptor record; other events, including Select and Set Data, require an object specifier record.

Five constants can be used to describe an insertion point in relation to an object or container:

Constant	Corresponding insertion point
`kAEReplace`	The specified object will be replaced if not qualified by one of the other phrases
`kAEBefore`	Just before the specified object (either type `typeObjectSpecifier` or type `typeInsertionLocation`)
`kAEAfter`	Just after the specified object (either type `typeObjectSpecifier` or type `typeInsertionLocation`)
`kAEBeginning`	In the specified container and before all other elements of the same class in that container (type `typeInsertionLocation` only)
`kAEEnd`	In the specified container and after all other elements of the same class in that container

For more information about the way AppleScript uses insertion location descriptor records, see "Defining Terminology for Use by the AppleScript Component," which begins on page 8-3, and the *Apple Event Registry: Standard Suites*. The rest of this section provides examples of object specifier records used to specify insertion points.

Users usually insert objects after some other object. So, unless the insertion point is clearly at the beginning or end of a container or identifies an object to be replaced, use the constant `kAEAfter` to record the location.

For example, if the user places the insertion point after character 2, your application should send a Select event with this direct parameter:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`  keyAEDesiredClass`	`  typeType`	`  cInsertionLoc`
`  keyAEContainer`	`  typeObjectSpecifier`	(see indented record)
`    keyAEDesiredClass`	`    typeType`	`    cChar`
`    keyAEContainer`	`    typeNull`	No data
`    keyAEKeyForm`	`    typeEnumerated`	`    formAbsolutePosition`
`    keyAEKeyData`	`    typeLongInteger`	2
`  keyAEKeyForm`	`  typeEnumerated`	`  formRelativePosition`
`  keyAEKeyData`	`  typeEnumerated`	`  kAEAfter`

If the selection is not 0 characters wide, the user is replacing the selection with another object, so you can specify the location simply as the object specifier record for the object to be replaced.

If the user clicks the white space after a paragraph somewhere in the middle of the document, defining the insertion point becomes more complex because different applications deal with this situation in different ways. Some place the insertion point at the end of the current paragraph, while others place the insertion point at the beginning of the next paragraph. Depending on the way your application handles this situation, you should use an object specifier record that specifies either `kAEBeginning` or `kAEEnd`.

Remember that the Select event requires an object specifier record. Thus, if you want to place the insertion point at the beginning of a paragraph, use an object specifier record that specifies a location just before the first item of the paragraph, rather than an insertion location descriptor record.

For example, a Select event with this direct parameter places the insertion point just before the first item of paragraph 3:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`  keyAEDesiredClass`	`  typeType`	`  cInsertionLoc`
`  keyAEContainer`	`  typeObjectSpecifier`	(see indented record)
`    keyAEDesiredClass`	`    typeType`	`    cItem`
`    keyAEContainer`	`    typeObjectSpecifier`	(see indented record)
`      keyAEDesiredClass`	`      typeType`	`      cParagraph`
`      keyAEContainer`	`      typeNull`	No data
`      keyAEKeyForm`	`      typeEnumerated`	`      formAbsolutePosition`
`      keyAEKeyData`	`      typeLongInteger`	3
`    keyAEKeyForm`	`    typeType`	`    formAbsolutePosition`
`    keyAEKeyData`	`    typeLongInteger`	1
`  keyAEKeyForm`	`  typeEnumerated`	`  formRelativePosition`
`  keyAEKeyData`	`  typeEnumerated`	`  kAEPrevious`

If the user clicks the left edge of the first line in a paragraph, thus setting the insertion point before the beginning of the paragraph, you should use a similar strategy. However, this is the only situation in which you should use `kAEPrevious`.

When the insertion point is at the end of a document record, use an object specifier record that specifies the location after the last item in the document.

For example, a Select event with this direct parameter places the insertion point just after the last item in a document:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cInsertionLoc`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cItem`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`-1`
`keyAEKeyForm`	`typeEnumerated`	`formRelativePosition`
`keyAEKeyData`	`typeEnumerated`	`kAENext`

A Select event with this direct parameter places the insertion point just after the last item in paragraph 3:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cInsertionLoc`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cParagraph`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`3`
`keyAEKeyForm`	`typeEnumerated`	`formRelativePosition`
`keyAEKeyData`	`typeEnumerated`	`kAENext`

Recording Typing

In general, to record typing your application should send itself a Set Data event that sets the contents of the selection. The data should be unstyled text. When your application handles the Set Data event, it should apply the styles that prevail at the insertion point. If your application supports styled text, you need to decide how to apply styles to new text and how to record style changes to selected text. Follow these general guidelines for recording typing:

- When the user sets an insertion point and types new text, use the styles defined for the text just before the insertion location.
- When a user selects text and changes its style, apply the changes to the selection.
- If a user types or pastes new text into a selection, place the insertion point after the new text.

The rest of this section provides examples of how to apply these guidelines.

Suppose the user sets an insertion point and then types something. Your application should use the style, font, size, and other characteristics of the text just before the insertion point for the new text, and it should record only the new characters inserted. For example, to place the insertion point after word 30 and insert the text "This is the new text," your application can send a Select event followed by a Set Data event:

- A Select event with this direct parameter places the insertion point after word 30:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cInsertionLoc`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cWord`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`30`
`keyAEKeyForm`	`typeEnumerated`	`formRelativePosition`
`keyAEKeyData`	`typeEnumerated`	`kAENext`

- A Set Data event (event class `kAECoreSuite`, event ID `kAESetData`) with these parameters (`keyDirectObject` and `keyAEData`) sets the selection to the new text:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cProperty`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cProperty`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
`keyAEKeyData`	`typeLongInteger`	`pSelection`
`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
`keyAEKeyData`	`typeType`	`pContents`
`keyAEData`	`typeChar`	`"This is the new text"`

Notice that the Select event in this example causes your application to set its `pSelection` property (the current selection) to the location specified by the object specifier record in the direct parameter—that is, after word 30. The Set Data event then sets the contents of the selection to a text string. The `pContents` property specified by the object specifier record in the direct parameter of the Set Data event represents the contents of the selection, and the text string in the `keyAEData` parameter is the text to which the selection's contents is to be set.

At this stage, the insertion point is after word 35—the last word added by typing. If the user now selects one of the new words, say word 34, and changes the style to boldface and the font to Helvetica®, send a Select event and two Set Data events to record the action:

- A Select event with this direct parameter selects word 34:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cWord`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`34`

■ A Set Data event with these parameters sets the style of the selection to boldface:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cProperty`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cProperty`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
`keyAEKeyData`	`typeLongInteger`	`pSelection`
`keyAEKeyForm`	`typeType`	`formPropertyID`
`keyAEKeyData`	`typeType`	`pTextStyles`
`keyAEData`	`typeEnumerated`	`kAEBold`

■ A Set Data event with these parameters sets the font of the selection to Helvetica:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cProperty`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cProperty`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
`keyAEKeyData`	`typeLongInteger`	`pSelection`
`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
`keyAEKeyData`	`typeType`	`pFont`
`keyAEData`	`typeChar`	`"Helvetica"`

After these three events are sent, word 34 remains selected. Thus, subsequent user actions upon the same selection do not require your application to send an additional event to set the selection. Your application should maintain the selection as long as the selected text is not replaced. If the user types or pastes new text into the selection, your application should place the insertion point after the new text.

Such a strategy might result in a series of events like these:

- A Set Data event with these parameters sets the contents of a selection to "More new text":

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cProperty
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeLongInteger	pSelection
keyAEKeyForm	typeEnumerated	formPropertyID
keyAEKeyData	typeType	pContents
keyAEData	typeChar	"More new text"

- Two Paste events paste the contents of the Clipboard twice after the new text.

Recording the Selection of Nontext Objects

The selection of nontext objects differs from the selection of text objects mainly in the way a recordable application specifies the objects. For example, if the user is working in a table or spreadsheet and selects row 5, column 3, your application can send a Select event with this direct parameter:

Keyword	Descriptor type	Data
keyDirectObject	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cColumn
keyAEContainer	typeNull	No data
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	3
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	5

When recording a range of cells, use a range of rows through a range of columns.

For example, if the user selects row 5 column 3 through row 6 column 4, specify columns 3 through 4 of rows 5 through 6 by sending a Select event with this direct parameter:

Keyword	**Descriptor type**	**Data**
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cRow`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cColumn`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formRange`
`keyAEKeyData`	`typeRangeDescriptor`	(see indented record)
`keyAERangeStart`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cColumn`
`keyAEContainer`	`typeCurrentContainer`	`No data`
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`3`
`keyAERangeStop`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cColumn`
`keyAEContainer`	`typeCurrentContainer`	`No data`
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`4`
`keyAEKeyForm`	`typeEnumerated`	`formRange`
`keyAEKeyData`	`typeRangeDescriptor`	(see indented record)
`keyAERangeStart`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cRow`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`5`
`keyAERangeStop`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cRow`
`keyAEContainer`	`typeCurrentContainer`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`6`

In some drawing and layout applications, users are used to dealing with insertion points at specific locations rather than relative to other objects. For example, setting an insertion point in a recordable drawing application might cause the application to send itself a Select event that places the insertion location at (235, 330)—that is, the location defined by a vertical coordinate of 235 and a horizontal coordinate of 330. A Select event that does this could have this direct parameter:

Keyword	**Descriptor type**	**Data**
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cInsertionLocation`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeAEList`	(see indented record)
	`typeLongInteger`	`235`
	`typeLongInteger`	`330`

Notice that the key data corresponds to an application's extension of the standard interpretation of key form `formAbsolutePosition`.

To set a selection that consists of noncontiguous objects, an application should send events that correspond to statements like these:

```
select {¬
   row 5 thru 6 of column 3 thru 4, ¬
   row 22 of column 6}

select {circle 2, rectangle 12, text frame 2}

select {file "Guidelines", file "Test Results"}
```

Identifying Objects

The way a recordable application identifies objects can involve assumptions about the user's criteria for selecting those objects. In general, such assumptions should be avoided. Follow these guidelines for identifying objects:

- If you aren't absolutely certain of the user's criteria for selecting an object, identify the object by name.
- If the object doesn't have a name, identify it by index.
- Determine the index based on the order in which a user would see the objects when reading a document.
- Identify windows and open documents on which actions are taken as the frontmost window or document.

The rest of this section provides examples of how to apply these guidelines.

Suppose a user is working with an electronic mail application that permits a variety of sorting methods for messages received. If the user is currently looking at messages sorted by date and then deletes the second message in the list, that message should be identified by name rather than by date. Use an object's name in any situation where it is not completely clear which identifying criteria the user had in mind.

Suppose a user has used the application's Find command to locate all messages created on a certain date. In this case it might be appropriate to identify "every message whose creation date . . ." in the corresponding Apple event. However, if the user did not ask for all messages created on that date, you can't be sure whether the user really wanted every message or only a particular one. For instance, perhaps the user couldn't remember the name, but only an approximate date. In this case a recordable application should identify the message by name.

Just as names are more specific and usually more desirable than whose tests, names are usually more specific and more readable than identifiers or indices. However, some objects may not have a name, only some other identifier or an index. Even though an identifier is more specific than an index, a logically defined index of position is more readable and is therefore recommended. For example, if a document contains unnamed illustrations, the user is more likely to identify a figure by index (order from the beginning of the file) than identifier (such as order created).

Suppose a document contains two figures that appear at first glance to be side by side, except that the right one is slightly taller and therefore begins higher on the page than the left one. In cases such as this, your application should determine the index based on the order in which the user would see the objects when reading a document. For Roman script systems, this means reading from left to right and from top to bottom. In the example just described, the leftmost, shorter figure would have a lower figure number than the rightmost, taller one.

When your application needs to refer to a window or a document, it should identify the object with an object specifier record that corresponds to the first, or front, window:

Keyword	Descriptor type	Data
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cWindow`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`1`

This strategy allows users to record scripts that will work on any window, regardless of its name. Similarly, events that act on an open document should identify it as "document 1."

It is usually possible to describe objects in several different ways. If an object has a unique name, use that. For example, instead of an object specifier record that describes "column number 7," use one that describes "the column named 'March'":

Keyword	**Descriptor type**	**Data**
`keyDirectObject`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cColumn`
`keyAEContainer`	`typeNull`	No data
`keyAEKeyForm`	`typeEnumerated`	`formName`
`keyAEKeyData`	`typeLongInteger`	`"March"`

It may be that such an object could also be described in a more complex manner, such as picture 1 of paragraph 302 of chapter 2. But complex descriptions like this should be used only as a last resort if no simpler name is available.

In general, be as specific as possible when you identify a selection in a recordable event. The user can generalize as necessary by editing the recorded script.

Moving the Selection During Recording

If recording is turned on and the user makes a selection, performs some action, and then makes a different selection, your application must make a decision: should it record the second selection in absolute terms or relative to the first selection? That is, should the corresponding AppleScript statement be

```
select insertion location before paragraph 5
```

or

```
select insertion location before paragraph after selection
```

Both statements may be appropriate under different conditions. But suppose that the user had selected paragraph 3 and now selects paragraph 12 or picture 3. Relative addressing doesn't make sense in these situations because the distance involved is too great or the unit is different. When you can't be sure of the user's intent, you should use absolute addressing. You can safely use relative addressing only when the user moves the selection or insertion point by only one unit, as with the arrow keys.

Even the use of the arrow keys does not guarantee that you can use relative addressing. For example, suppose that the user has selected cell 5 of row 2 in a spreadsheet and then presses the Left Arrow key three times. In this case, it is best to send Apple events equivalent to the statement

```
select cell 3 of row 2
```

rather than the statements

```
select the cell before selection
select the cell before selection
select the cell before selection
```

Using relative addressing in certain circumstances may minimize the amount of editing that the user must do after recording a script. However, recordable applications are not required to use relative addressing.

Recording Interactions With Dialog Boxes

When executing scripts, users normally do not want to see dialog boxes. Therefore, your application should record information specified by the user in dialog boxes rather than sending events that would cause the dialog boxes to appear during script execution.

For example, suppose a user chooses the Close command and the standard save changes dialog box appears. If the user then clicks Save, your application should send a Close event that corresponds to a statement like this:

```
close document "MyDoc" saving Yes
```

Any settings in a dialog box that the user does not change (such as the range of pages to print in a Print dialog box) should not be recorded.

How Apple Event Recording Works

Scripting components use the Apple Event Manager's recording mechanism to allow a recording process such as the Script Editor application to control recording into scripts. Script editors and applications that provide their own recording capabilities can take advantage of the recording mechanism via standard scripting component routines.

This section describes how scripting components use Apple event recording. You need to read this section if you are developing a scripting component or a script-editing application, or if you want your application to initiate and control Apple event recording. For information about using the standard scripting component routines to turn recording off and on, see "Recording Scripts" on page 10-26.

When a user turns on recording for a recording process (for example, by clicking the Record button in Script Editor), the recording process calls a scripting component routine (`OSAStartRecording`) to turn recording on. The scripting component responds by sending a Start Recording event to the recording process (or any running process on the local computer).

Start Recording—begin sending copies of recordable events to recording process

Event class	`kCoreEventClass`
Event ID	`kAEStartRecording`
Parameters	None
Description	Sent by a scripting component to the recording process (or to any running process on the local computer), but handled by the Apple Event Manager. The Apple Event Manager responds by turning on recording and sending a Recording On event to all running processes on the local computer. This event must be addressed using a process serial number (PSN); it should never be sent to an address specified as `kCurrentProcess`.

The recording process should not handle the Start Recording event. Instead, the Apple Event Manager handles it by sending a Recording On event to all running processes on the local computer and sending copies of all subsequent recordable events to the recording process. (The Recording On event is described on page 9-4.)

If an application that supports Apple events is launched on a computer for which recording is turned on, the Apple Event Manager will also send it a Recording On event for each recording process that is currently recording.

The recording process receives recordable events by means of a Receive Recordable Event handler—that is, a handler installed in the Apple event dispatch table for event class `kCoreEventClass` and event ID `kAENotifyRecording`. Scripting components install this handler on behalf of a recording process when recording is first turned on and remove the handler when recording is turned off. Much like a handler for event class `typeWildCard` and event ID `typeWildCard`, the Receive Recordable Event handler handles all recordable events sent to the recording process by the Apple Event Manager. Any other Apple events received by the recording process are dispatched in the usual manner. The Receive Recordable Event handler handles recordable events by recording them in the script specified by the recording process's call to `OSAStartRecording`.

Receive Recordable Event—receive and record a copy of a recordable event

Event class	`kCoreEventClass`
Event ID	`kAENotifyRecording`
Parameters	Same as Apple event being recorded
Description	Wildcard event class and event ID handled by a recording process in order to receive and record copies of recordable events sent to it by the Apple Event Manager. Scripting components install a handler for this event on behalf of a recording process when recording is turned on and remove the handler when recording is turned off.

Whenever the Receive Recordable Event handler receives a recordable event, the scripting component sends your application a Recorded Text event. The Recorded Text event contains the decompiled source data for the recorded event in the form of styled text. For a description of the Recorded Text event, see “Recording Scripts” on page 10-26.

When a user turns off recording (for example, by clicking Script Editor’s Stop button), the recording process calls a scripting component routine (`OSAStopRecording`) to turn recording off. The scripting component responds by sending a Stop Recording event to the recording process (or any running process on the local computer).

Stop Recording—stop sending copies of recordable events to recording process

Event class	`kCoreEventClass`
Event ID	`kAEStopRecording`
Parameters	None
Description	Sent by a scripting component to the recording process (or to any running process on the local computer), but handled by the Apple Event Manager. The Apple Event Manager responds by sending a Recording Off event to all running processes on the local computer. This event must be addressed using a process serial number (PSN); it should never be sent to an address specified as `kCurrentProcess`.

Like the Start Recording event, the Stop Recording event is handled by the Apple Event Manager. The Apple Event Manager responds by sending a Recording Off event to all running processes on the local computer. (The Recording Off event is described on page 9-4.)

Recording continues, and the recording process may continue to receive recordable events, until the Apple Event Manager has notified all running processes that recording has been turned off for that recording process. The Apple Event Manager sends all running processes the Recording Off event with the `kAEWaitReply` flag set. If an application has stored some data (for instance, keystrokes) that needs to be recorded as an Apple event, this is the last chance for the application to send the event for recording purposes. Recording stops only after the Apple Event Manager returns a reply for the Stop Recording event.

The Apple Event Manager supports multiple simultaneous recording processes. A Stop Recording event sent for one of them does not affect the others. If your application needs to know which of several recording processes has turned recording on or off, it can check the `keyOriginalAddressAttr` attribute of the Recording On or Recording Off event for the address of the recording process.

If the Apple Event Manager does not receive a Stop Recording event for a recording process that quits unexpectedly, the applications being recorded don’t find out immediately. When it attempts to send a copy of a recordable event to a recording process that is no longer active, the Apple Event Manager sends a Recording Off event to all running processes on behalf of that recording process and specifies the address for that process in the `keyOriginalAddressAttr` attribute. If a recording process that quits is the only actively recording process, recording stops completely after the Apple Event Manager has informed all running processes that recording has been turned off.