This chapter introduces Apple events and the Apple Event Manager. Later chapters describe how your application can use the Apple Event Manager to respond to and send Apple events, locate Apple event objects, and record Apple events.

The interapplication communication (IAC) architecture for Macintosh computers consists of five parts: the Edition Manager, the Open Scripting Architecture (OSA), the Apple Event Manager, the Event Manager, and the Program-to-Program Communications (PPC) Toolbox. The chapter "Introduction to Interapplication Communication" in this book provides an overview of the relationships among these parts.

The *Apple Event Registry: Standard Suites* defines both the actions performed by the standard Apple events, or "verbs," and the standard Apple event object classes, which can be used to create "noun phrases" describing objects on which Apple events act. If your application uses the Apple Event Manager to respond to some of these standard Apple events, you can make it scriptable—that is, capable of responding to scripts written in a scripting language such as AppleScript. In addition, your application can use the Apple Event Manager to create and send Apple events and to allow user actions in your application to be recorded as Apple events.

Before you use this chapter or any of the other chapters about the Apple Event Manager, you should be familiar with the chapters "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* and "Process Manager" in *Inside Macintosh: Processes*.

This chapter begins by describing Apple events and some of the data structures they contain. The rest of the chapter introduces the use of the Apple Event Manager to

- respond to Apple events
- send Apple events to request services or information
- work with object specifier records
- classify Apple event objects
- locate Apple event objects

Finally, this chapter summarizes the tasks you can perform with the Apple Event Manager and explains where to locate information you need to perform those tasks.

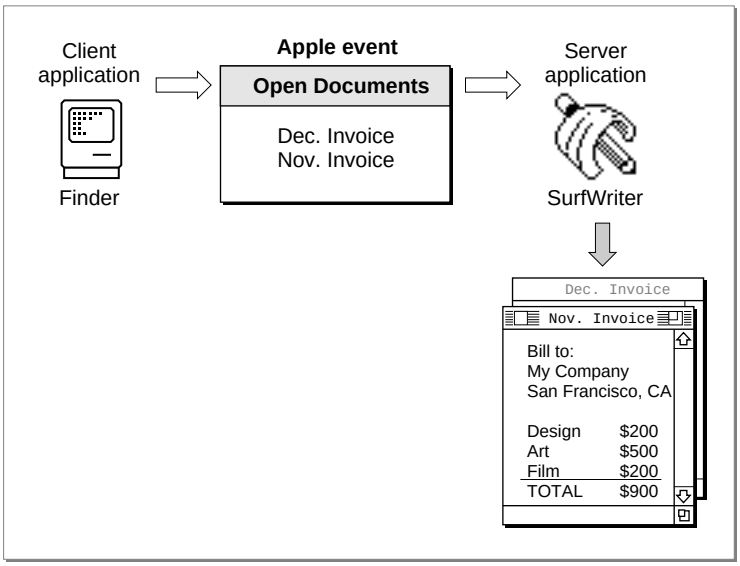
About Apple Events

An ***Apple event*** is a high-level event that conforms to the Apple Event Interprocess Messaging Protocol. The Apple Event Manager uses the services of the Event Manager to send Apple events between applications on the same computer, between applications on remote computers, or from an application to itself.

Applications typically use Apple events to request services and information from other applications or to provide services and information in response to such requests. Communication between two applications that support Apple events is initiated by a ***client application,*** which sends an Apple event to request a service or information. The application providing the service or the requested information is called a ***server application.*** The client and server applications can reside on the same local computer or on remote computers connected to a network. An application can also send Apple events to itself, thus acting as both client and server.

Figure 3-1 shows a common Apple event, the Open Documents event. The Finder is the client; it requests that the SurfWriter application open the documents named "Dec. Invoice" and "Nov. Invoice." The SurfWriter application responds to the Finder's request by opening windows for the specified documents.

Figure 3-1 An Open Documents event



The Finder is considered the client application for the Open Documents event shown in Figure 3-1 because the Finder initiates the request for a service. The Finder can also be considered the source application for the same Open Documents event. A ***source application*** for an Apple event is one that sends an Apple event to another application or to itself. Similarly, the SurfWriter application can be described as either the server application or the target application for the Open Documents event shown in Figure 3-1. A ***target application*** for an Apple event is the one addressed to receive the Apple event. The terms *client application* and *source application* are not always synonymous, nor are the terms *server application* and *target application.* Typically, an Apple event client application sends an Apple event requesting a service to an Apple event server application;

in this case, the server application is the target application for the Apple event. A server application may return information to the client in a reply Apple event—in which case, the client application is the target application for the reply.

To perform the requested service—that is, to open the specified documents—the SurfWriter application shown in Figure 3-1 first uses the Apple Event Manager to identify the event (the Open Documents event) and to dispatch the event to SurfWriter's handler for that Apple event. An ***Apple event handler*** is an application-defined function that extracts pertinent data from an Apple event, performs the requested action, and (usually) returns a result. In this case, SurfWriter's Open Documents event handler examines the Apple event to determine which documents to open (Dec. Invoice and Nov. Invoice), then opens them as requested.

To identify Apple events and respond appropriately, every application can rely on a vocabulary of standard Apple events that developers and Apple have established for all applications to use. These events are defined in the *Apple Event Registry: Standard Suites.* The standard ***suites,*** or groups of related Apple events that are usually implemented together, include the Required suite, the Core suite, and functional-area suites such as the Text suite and the Database suite. To function as a server application, your application should be able to respond to all the Apple events in the Required suite and any of those in the Core and functional-area suites that it is likely to receive. For example, most word-processing applications should be capable of responding to the Apple events in the Text suite, and most database applications should be capable of responding to those in the Database suite.

If necessary, you can extend the definitions of the standard Apple events to match specific capabilities of your application. You can also define your own custom Apple events; however, before defining custom events, you should check with the Apple Event Registrar to find out whether you can adapt existing Apple event definitions or definitions still under development to the needs of your application.

By supporting the standard Apple events in your application, you ensure that your application can communicate effectively with other applications that also support them. Instead of supporting many different custom events for a limited number of applications, you can support a relatively small number of standard Apple events that can be used by any number of applications.

You can begin supporting Apple events by making your application a reliable server application: first for the required Apple events, then for the core and functional-area Apple events as appropriate. Once your application can respond to the appropriate standard Apple events, you can make it ***scriptable,*** or capable of responding to instructions written in a system-wide scripting language such as AppleScript. If necessary, your application can also send Apple events to itself or to other applications.

"About the Apple Event Manager," which begins on page 3-48, provides more information about the steps you need to take to support Apple events in your application.

The next section, "Apple Events and Apple Event Objects," describes how Apple events can describe data and other items within an application or its documents. Subsequent sections describe the basic organization of Apple events and the data structures from which they are constructed.

Apple Events and Apple Event Objects

The Open Documents event shown in Figure 3-1, like the other three required events, specifies an action and the applications or documents to which that action applies. The *Apple Event Registry: Standard Suites* provides a vocabulary of actions for use by all applications. In addition to a vocabulary of actions, effective communication between applications requires a method of referring to windows, data (such as words or graphic elements), files, folders, volumes, zones, and other discrete items on which actions can be performed. The Apple Event Manager includes routines that allow any application to construct or interpret "noun phrases" that describe the objects on which Apple events act.

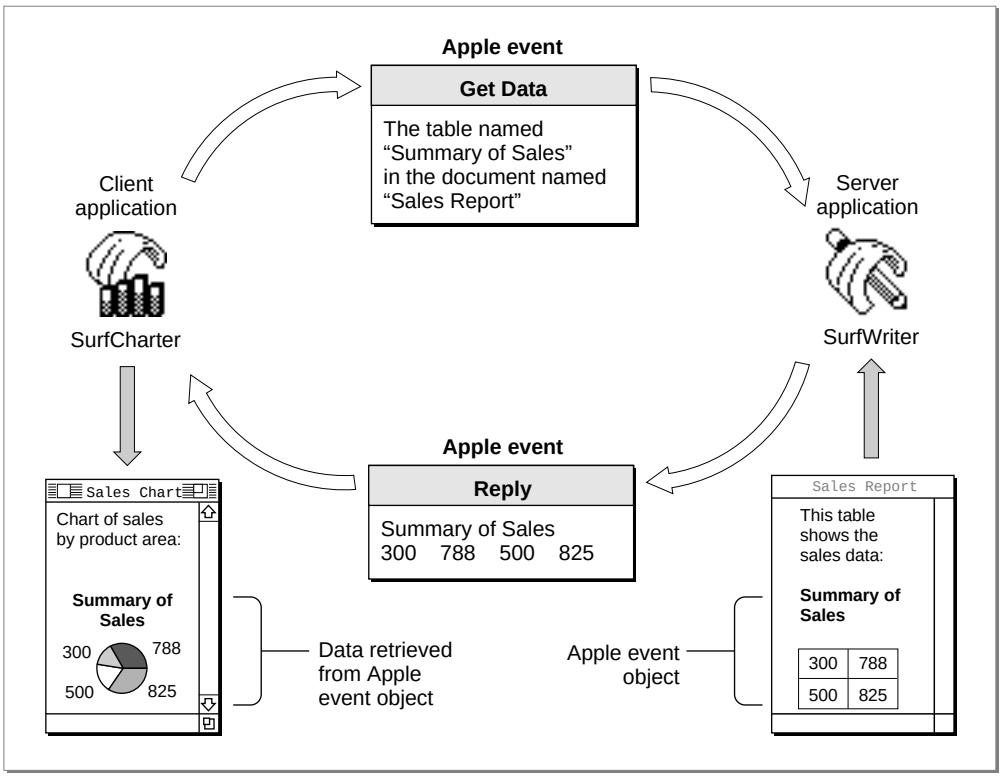
Most of the Apple event definitions in the *Apple Event Registry: Standard Suites* include definitions of ***Apple event object classes,*** which are simply names for objects that can be acted upon by each kind of Apple event. An ***Apple event object*** is any distinct item supported by an application that can be described within an Apple event. Apple event objects can be anything that an application can locate on the basis of such a description, including items that a user can differentiate and manipulate while using an application, such as words, paragraphs, shapes, windows, or style formats.

The definition for each Apple event object class in the *Apple Event Registry: Standard Suites* determines only how that kind of Apple event object should be described within an Apple event, not how it should be represented internally by an individual application. You do not have to write your application in an object-oriented programming language to support Apple event objects. Instead, you need to organize your application so that it can interpret a request for specific Apple event objects, locate the objects, and perform the requested action on them.

Figure 3-2 shows a common Apple event, the Get Data event from the Core suite. In this example, the SurfCharter application is the client application; it requests data contained in a specific table in a SurfWriter document. To obtain the data it wants, the SurfCharter application must include a description of the data in the Get Data event it sends to SurfWriter. This description identifies the requested data as an Apple event object called a table. The table is named "Summary of Sales" and is located in a document named "Sales Report."

The SurfWriter application's Get Data handler extracts information about the request, locates the specified table, and returns a result. The Apple Event Manager provides a reply Apple event to which the SurfWriter application adds the requested information in the form requested by the Get Data event. The Apple Event Manager sends the reply event back to the SurfCharter application, which can use the requested data in whatever way is appropriate—in this case, displaying it as a pie chart.

Figure 3-2 A Get Data event



Apple Event Attributes and Parameters

When an application creates and sends an Apple event, the Apple Event Manager uses arguments passed to Apple Event Manager routines to construct the data structures that make up the Apple event. An Apple event consists of attributes (which identify the Apple event and denote its task) and, often, parameters (which contain information to be used by the target application).

An ***Apple event attribute*** is a record that identifies the event class, event ID, target application, or some other characteristic of an Apple event. Taken together, the attributes of an Apple event denote the task to be performed on any data specified in the Apple event's parameters. A client application can use Apple Event Manager routines to add attributes to an Apple event. After receiving an Apple event, a server application can use Apple Event Manager routines to extract and examine its attributes.

An ***Apple event parameter*** is a record containing data that the target application uses. Unlike Apple event attributes (which contain information that can be used by both the Apple Event Manager and the target application), Apple event parameters contain data used only by the target application. For example, the Apple Event Manager uses the event class and event ID attributes to identify the server application's handler for a specific Apple event, and the server application must have a handler to process the event identified by those attributes. By comparison, the list of documents contained in a parameter to an Open Documents event is used only by the server application. As with attributes, a client application can use Apple Event Manager routines to add parameters to an Apple event, and a server application can use Apple Event Manager routines to extract and examine the parameters of an Apple event it has received.

Note that Apple event parameters are different from the parameters of Apple Event Manager functions. Apple event parameters are records used by the Apple Event Manager; function parameters are arguments you pass to the function or that the function returns to you. You can specify both Apple event parameters and Apple event attributes in parameters to Apple Event Manager functions. For example, the `AEGetParamPtr` function uses a buffer to return the data contained in an Apple event parameter. You can specify the Apple event parameter whose data you want in one of the parameters of the `AEGetParamPtr` function.

Apple Event Attributes

Apple events are identified by their event class and event ID attributes. The ***event class*** is the attribute that identifies a group of related Apple events. The event class appears in the `message` field of the event record for an Apple event. For example, the four required Apple events have the value `'aevt'` in the `message` fields of their event records. The value `'aevt'` can also be represented by the `kCoreEventClass` constant. Several event classes are shown here:

Event class	Value	Description
`kCoreEventClass`	`'aevt'`	A required Apple event
`kAECoreSuite`	`'core'`	A core Apple event
`kAEFinderEvents`	`'FNDR'`	An event that the Finder accepts
`kSectionEventMsgClass`	`'sect'`	An event sent by the Edition Manager

The ***event ID*** is the attribute that identifies the particular Apple event within its event class. In conjunction with the event class, the event ID uniquely identifies the Apple event and communicates what action the Apple event should perform. (The event IDs appear in the `where` field of the event record for an Apple event. For more information about event records, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.) For example, the event ID of an Open Documents event has the value `'odoc'` (which can also be represented by the `kAEOpenDocuments` constant). The `kCoreEventClass` constant in combination with the `kAEOpenDocuments` constant identifies the Open Documents event to the Apple Event Manager.

Here are the event IDs for the four required Apple events:

Event ID	Value	Description
`kAEOpenApplication`	`'oapp'`	Perform tasks required when a user opens your application without opening or printing any documents
`kAEOpenDocuments`	`'odoc'`	Open documents
`kAEPrintDocuments`	`'pdoc'`	Print documents
`kAEQuitApplication`	`'quit'`	Quit your application

In addition to the event class and event ID attributes, every Apple event must include an attribute that specifies the target application's address. Remember that the target application is the one addressed to receive the Apple event. Your application can send an Apple event to itself or to another application (on the same computer or on a remote computer connected to the network).

Every Apple event must include event class, event ID, and target address attributes. Some Apple events can include other attributes; see "Keyword-Specified Descriptor Records," which begins on page 3-15, for a complete list.

Apple Event Parameters

As with attributes, there are various kinds of Apple event parameters. A ***direct parameter*** usually specifies the data to be acted upon by the target application. For example, the direct parameter of the Print Documents event contains a list of documents. Some Apple events also take ***additional parameters,*** which the target application uses in addition to the data specified in the direct parameter. Thus, an Apple event for arithmetic operations might include additional parameters that specify operands in an equation.

The *Apple Event Registry: Standard Suites* describes all parameters as either required or optional. A ***required parameter*** is one that must be present for the target application to carry out the task denoted by the Apple event. An ***optional parameter*** is a supplemental Apple Event parameter that also can be used to specify data to the target application. Optional parameters need not be included in an Apple event; default values for optional parameters are part of the event definition. The target application that handles the event must supply default values if the optional parameters are omitted.

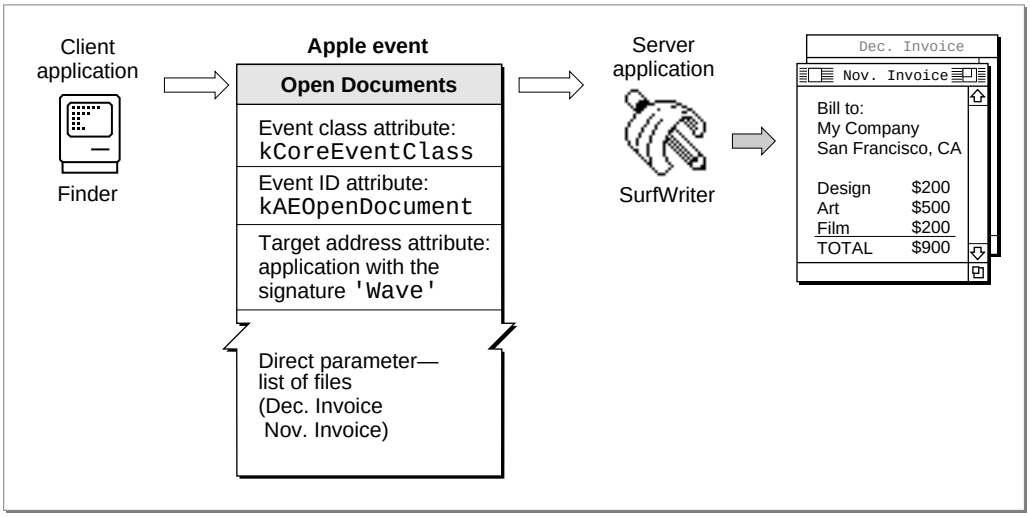
Direct parameters are usually defined as required parameters in the *Apple Event Registry: Standard Suites;* additional parameters may be defined as either required or optional. However, the Apple Event Manager does not enforce the definitions of required and optional events. Instead, the source application specifies, when it sends the event, which Apple event parameters the target can treat as if they were optional.

For more information about optional parameters, see "Specifying Optional Parameters for an Apple Event," which begins on page 5-7.

Interpreting Apple Event Attributes and Parameters

Figure 3-3 shows the major Apple event attributes and direct parameter for the Open Documents event introduced in Figure 3-1.

Figure 3-3 Major attributes and direct parameter of an Open Documents event



When the SurfWriter application receives any high-level event, it calls the `AEProcessAppleEvent` function to process the event. For an Apple event such as the Open Documents event shown in Figure 3-3, the `AEProcessAppleEvent` function uses the event class and event ID attributes to dispatch the event to the SurfWriter application's Open Documents handler. In response, the Open Documents handler opens the documents specified in the direct parameter.

The definition of a given Apple event in the *Apple Event Registry: Standard Suites* suggests how the source application can organize the data in the Apple event's parameters and how the target application interprets that data. The data in an Apple event parameter may use standard or private data types and may include a description of an Apple event object. Each Apple event handler provided by an application should be written with the format of the expected data in mind.

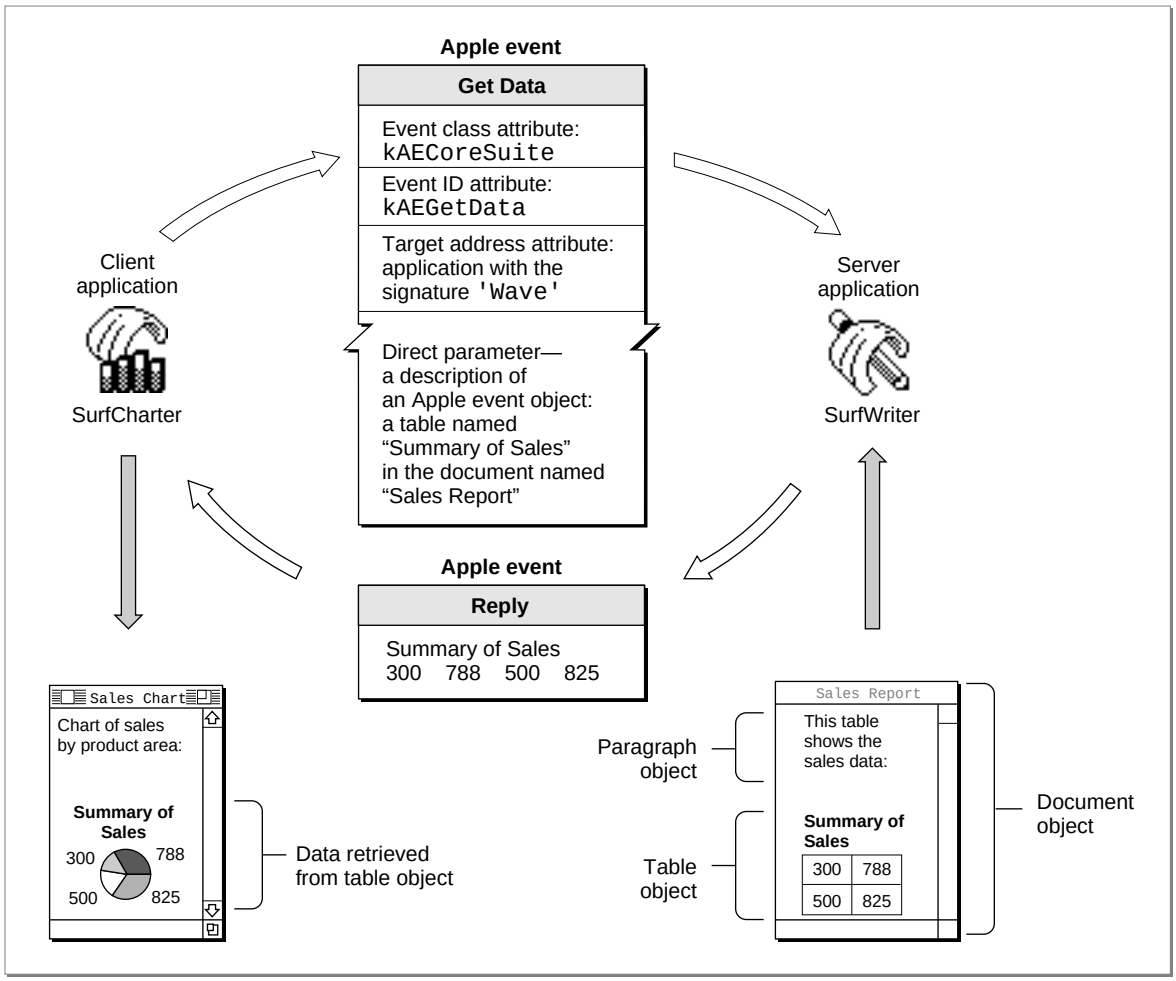
Apple events can use standard data types, such as strings of text, long integers, Boolean values, and alias records, for the corresponding data in Apple event parameters. For example, the Get Data event can contain an optional parameter specifying the form in which the requested data should be returned. This optional parameter always consists of a list of four-character codes denoting desired descriptor types in order of preference. Apple events can also use special data types defined by the Apple Event Manager.

Apple event parameters often contain descriptions of Apple event objects. These descriptions make use of a standard classification scheme summarized in "The Classification of Apple Event Objects," which begins on page 3-39.

For example, every Get Data event includes a required parameter that describes the Apple event object containing the data requested by the client application. Thus, one application can send a Get Data event to another application, requesting, for instance, one paragraph of a document, the first and last paragraphs of a document, all pictures in the document, all paragraphs containing the word "sales," or pages 10 through 12 of the document.

Figure 3-4 shows the Apple event attributes and direct parameter for the Get Data event introduced in Figure 3-2. The direct parameter for the Get Data event sent by the SurfCharter application describes the requested Apple event object as a table called "Summary of Sales" in the document "Sales Report." Both the table and the document are Apple event objects. The description of an Apple event object always includes a description of its location. In most cases, Apple event objects are located inside other Apple event objects.

Figure 3-4 Major attributes and direct parameter of a Get Data event



To process the information in the Get Data event, the SurfWriter application calls the `AEProcessAppleEvent` function. The `AEProcessAppleEvent` function uses the event class and event ID attributes to dispatch the event to the SurfWriter application's handler for the Get Data Apple event. The SurfWriter application responds to the Get Data event by ***resolving*** the description of the Apple event object—that is, by using the `AEResolve` function, other Apple Event Manager routines, and its own application-defined functions to locate the table named "Summary of Sales." After locating the table, SurfWriter adds a copy of the table's data to the reply event, which the Apple Event Manager then sends to the SurfCharter application. The SurfCharter application then displays the data in its active window.

The next section, "Data Structures Within Apple Events," describes the data structures the Apple Event Manager uses for Apple event attributes and parameters.

Data Structures Within Apple Events

The Apple Event Manager constructs its own internal data structures to contain the information in an Apple event. Neither the sender nor the receiver of an Apple event should manipulate data directly after it has been added to an Apple event; each should rely on Apple Event Manager functions to do so.

This section describes the most important data structures used by the Apple event Manager to construct Apple events. The first structure described is the descriptor record, a data structure of type `AEDesc`. Applications may access the data in an individual descriptor record directly if it is not part of another Apple Event Manager data structure.

In some cases it is convenient for the Apple Event Manager to describe descriptor records by data types that indicate their contents; thus, it also defines data structures such as type `AEAddressDesc`, `AEDescList`, and `AERecord`, which are descriptor records used to hold addresses, lists of other descriptor records, and Apple event parameters, respectively. These and most of the other data structures described in this section are formally defined as data structures of type `AEDesc`; they differ only in the purposes for which they are used.

Descriptor Records

Descriptor records are the building blocks used by the Apple Event Manager to construct Apple event attributes and parameters. A ***descriptor record*** is a data structure of type `AEDesc`; it consists of a handle to data and a descriptor type that identifies the type of the data to which the handle refers.

```
TYPE AEDesc =
RECORD                              {descriptor record}
   descriptorType:   DescType;      {type of data}
   dataHandle:       Handle;        {handle to data}
END;
```

If a descriptor record exists separately from other Apple Event Manager data structures, it is possible to retrieve the data associated with its handle by dereferencing the handle twice. After a descriptor record has been added to any other Apple Event Manager data structure, you must use Apple Event Manager routines to extract data from the descriptor record.

The ***descriptor type*** is a structure of type `DescType`, which in turn is of data type `ResType`—that is, a four-character code. Constants are usually used in place of these four-character codes when referring to descriptor types. Descriptor types represent various data types. Here are some of the major descriptor type constants, their values, and the kinds of data they identify.

Descriptor type	Value	Description of data
`typeBoolean`	`'bool'`	1-byte Boolean value
`typeChar`	`'TEXT'`	Unterminated string
`typeLongInteger`	`'long'`	32-bit integer
`typeShortInteger`	`'shor'`	16-bit integer
`typeMagnitude`	`'magn'`	Unsigned 32-bit integer
`typeAEList`	`'list'`	List of descriptor records
`typeAERecord`	`'reco'`	List of keyword-specified descriptor records
`typeAppleEvent`	`'aevt'`	Apple event record
`typeEnumerated`	`'enum'`	Enumerated data
`typeType`	`'type'`	Four-character code
`typeFSS`	`'fss '`	File system specification
`typeKeyword`	`'keyw'`	Apple event keyword
`typeNull`	`'null'`	Nonexistent data (handle whose value is `NIL`)

For a complete list of the basic descriptor types used by the Apple Event Manager, see Table 4-2 on page 4-57.

Figure 3-5 illustrates the logical arrangement of a descriptor record with a descriptor type of `typeChar`, which specifies that the data handle refers to an unterminated string (in this case, the text “Summary of Sales”).

Figure 3-5 A descriptor record whose data handle refers to an unterminated string

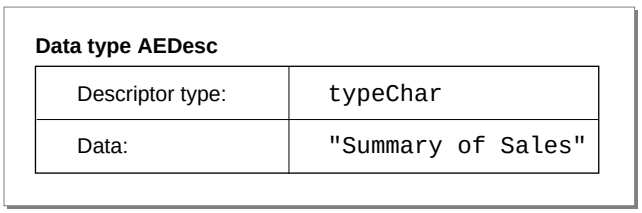
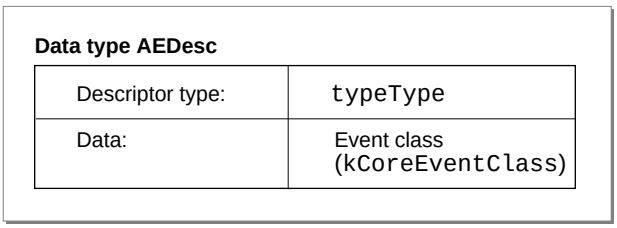


Figure 3-6 illustrates the logical arrangement of a descriptor record with a descriptor type of `typeType`, which specifies that the data handle refers to a four-character code (in this case the constant `kCoreEventClass`, whose value is `'aevt'`). This descriptor record can be used in an Apple event attribute that identifies the event class for any Apple event in the Core suite.

Figure 3-6 A descriptor record whose data handle refers to event class data



Every Apple event includes an attribute specifying the address of the target application. A descriptor record that contains an application's address is called an ***address descriptor record.***

```
TYPE  AEAddressDesc = AEDesc;       {address descriptor record}
```

The address in an address descriptor record can be specified as one of these four basic types (or as any other descriptor type you define that can be coerced to one of these types):

Descriptor type	Value	Description
`typeApplSignature`	`'sign'`	Application signature
`typeSessionID`	`'ssid'`	Session reference number
`typeTargetID`	`'targ'`	Target ID record
`typeProcessSerialNumber`	`'psn '`	Process serial number

Like several of the other data structures defined by the Apple Event Manager for use in Apple event attributes and Apple event parameters, an address descriptor record is identical to a descriptor record of data type `AEDesc`; the only difference is that the data for an address descriptor record must always consist of an application's address.

Keyword-Specified Descriptor Records

After the Apple Event Manager has assembled the necessary descriptor records as the attributes and parameters of an Apple event, your application cannot examine the contents of the Apple event directly. Instead, your application must use Apple Event Manager routines to request each attribute and parameter by keyword. ***Keywords*** are arbitrary names used by the Apple Event Manager to keep track of various descriptor records. The `AEKeyword` data type is defined as a four-character code.

```
TYPE  AEKeyword = PACKED ARRAY[1..4] OF Char; {keyword for a }
                                              { descriptor record}
```

Constants are typically used for keywords. Here is a list of the keyword constants for Apple event attributes:

Attribute keyword	Value	Description
`keyAddressAttr`	`'addr'`	Address of target or client application
`keyEventClassAttr`	`'evcl'`	Event class of Apple event
`keyEventIDAttr`	`'evid'`	Event ID of Apple event
`keyEventSourceAttr`	`'esrc'`	Nature of the source application
`keyInteractLevelAttr`	`'inte'`	Settings for allowing the Apple Event Manager to bring a server application to the foreground, if necessary, to interact with the user
`keyMissedKeywordAttr`	`'miss'`	Keyword for first required parameter remaining in an Apple event
`keyOptionalKeywordAttr`	`'optk'`	List of keywords for parameters of the Apple event that should be treated as optional by the target application
`keyOriginalAddressAttr`	`'from'`	Address of original source of Apple event if the event has been forwarded (available only in version 1.01 or later versions of the Apple Event Manager)
`keyReturnIDAttr`	`'rtid'`	Return ID for reply Apple event
`keyTimeoutAttr`	`'timo'`	Length of time, in ticks, that the client will wait for a reply or a result from the server
`keyTransactionIDAttr`	`'tran'`	Transaction ID identifying a series of Apple events

Here is a list of the keyword constants for commonly used Apple event parameters:

Parameter keyword	Value	Description
`keyDirectObject`	`'----'`	Direct parameter
`keyErrorNumber`	`'errn'`	Error number parameter
`keyErrorString`	`'errs'`	Error string parameter

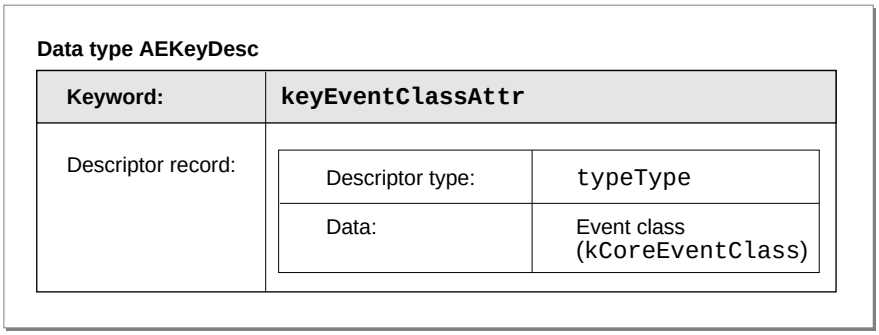
The *Apple Event Registry: Standard Suites* defines additional keyword constants for Apple event parameters that can be used with specific Apple events.

The Apple Event Manager associates keywords with specific descriptor records by means of a ***keyword-specified descriptor record,*** a data structure of type `AEKeyDesc` that consists of a keyword and a descriptor record.

```
TYPE  AEKeyDesc = {keyword-specified descriptor record}
RECORD
   descKey:       AEKeyword;     {keyword}
   descContent:   AEDesc;        {descriptor record}
END;
```

Figure 3-7 illustrates a keyword-specified descriptor record with the keyword `keyEventClassAttr`—the keyword that identifies an event class attribute. The figure shows the logical arrangement of the event class attribute for the Open Documents event shown in Figure 3-3 on page 3-10. The descriptor record in Figure 3-7 is identical to the one in Figure 3-6; its descriptor type is `typeType`, and the data to which its handle refers identifies the event class as `kCoreEventClass`.

Figure 3-7 A keyword-specified descriptor record for the event class attribute of an Open Documents event



Descriptor Lists

When extracting data from an Apple event, you use Apple Event Manager functions to copy data to a buffer specified by a pointer, or to return a descriptor record whose data handle refers to a copy of the data, or to return lists of descriptor records (called descriptor lists).

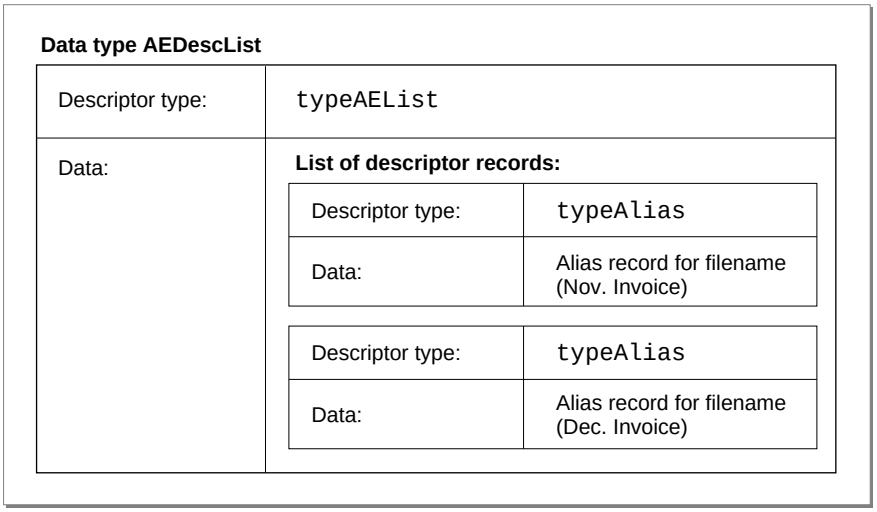
As previously noted, the descriptor record (of data type `AEDesc`) is the fundamental structure in Apple events, and it consists of a descriptor type and a handle to data. A ***descriptor list*** is a data structure of type `AEDescList` defined by the data type `AEDesc`—that is, a descriptor list is a descriptor record whose data handle refers to a list of other descriptor records (unless it is an empty list).

```
TYPE  AEDescList = AEDesc;       {list of descriptor records}
```

Like several other Apple Event Manager data structures, a descriptor list is identical to a descriptor record of data type `AEDesc`; the only difference is that the data in a descriptor list must always consist of a list of other descriptor records.

Figure 3-8 illustrates the logical arrangement of the descriptor list that specifies the direct parameter of the Open Documents event shown in Figure 3-3 on page 3-10. This descriptor list consists of a list of descriptor records that contain alias records to filenames. (See the chapter "Alias Manager" in *Inside Macintosh: Files* for a detailed description of alias records.)

Figure 3-8 A descriptor list for a list of aliases



The descriptor list in Figure 3-8 provides the data for a keyword-specified descriptor record. Keyword-specified descriptor records for Apple event parameters can in turn be combined in an ***AE record,*** which is a descriptor list of data type `AERecord`.

```
TYPE  AERecord = AEDescList;        {list of keyword-specified }
                                    { descriptor records}
```

The handle for a descriptor list of data type `AERecord` refers to a list of keyword-specified descriptor records that can be used to construct Apple event parameters. The Apple Event Manager provides routines that allow your application to create AE records and extract data from them when creating or responding to Apple events.

An AE record has the descriptor type `typeAERecord` and can be coerced to several other descriptor types. An ***Apple event record*** , which is different from an AE record, is another special descriptor list of data type `AppleEvent` and descriptor type `typeAppleEvent`.

```
TYPE  AppleEvent = AERecord;        {list of attributes and }
                                    { parameters necessary for }
                                    { an Apple event}
```

An Apple event record describes a full-fledged Apple event. Like the data for an AE record, the data for an Apple event record consists of a list of keyword-specified descriptor records. Unlike an AE record, the data for an Apple event record is divided into two parts, one for attributes and one for parameters. This division within the Apple event record allows the Apple Event Manager to distinguish between an Apple event's attributes and its parameters.

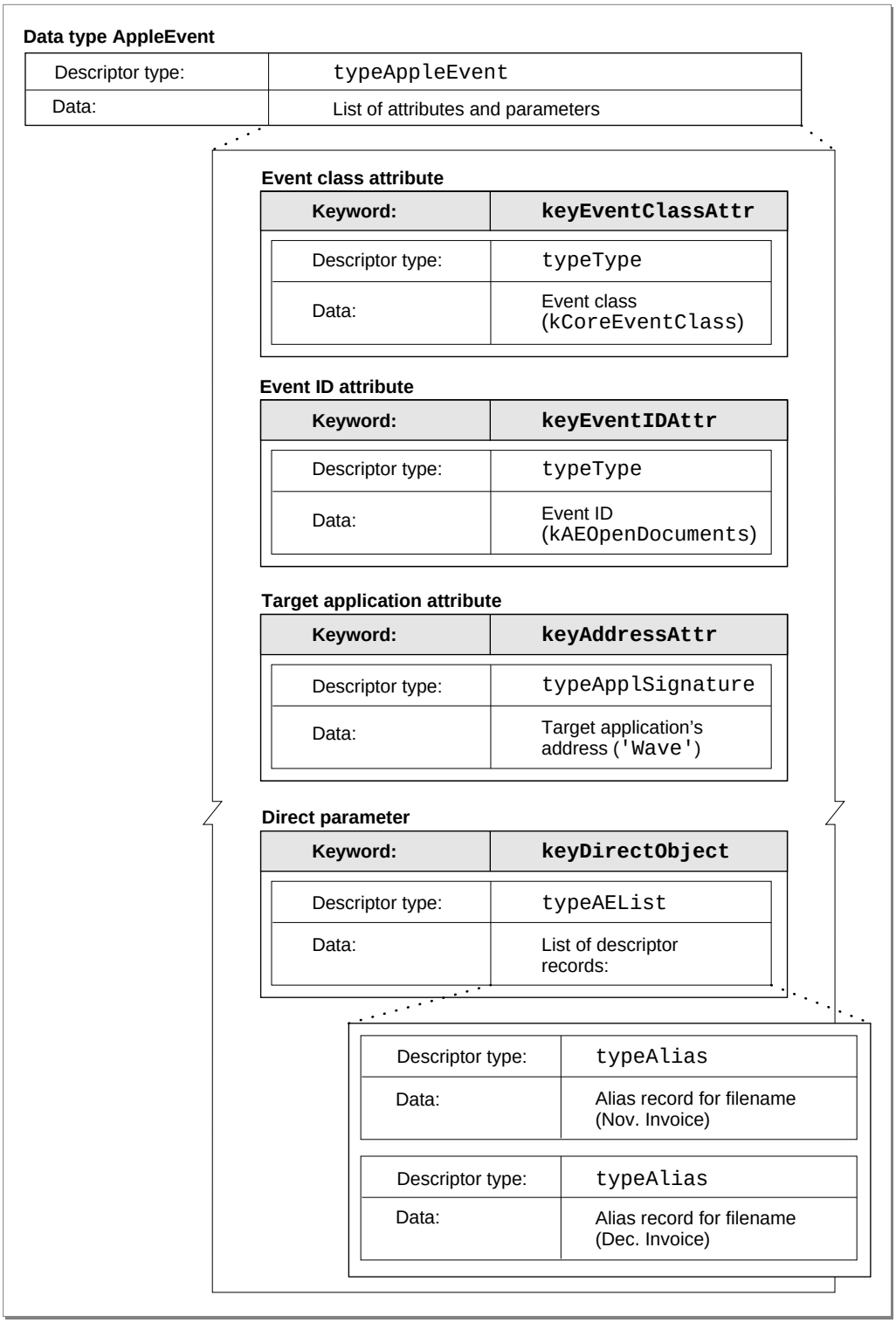
Descriptor lists, AE records, and Apple event records are all descriptor records whose handles refer to a nested list of other descriptor records. The data associated with each data type may be organized differently and is used by the Apple Event Manager for different purposes. In each case, however, the data is identified by a handle in a descriptor record. This means that you can pass an Apple event record to any Apple Event Manager function that expects an AE record. Similarly, you can pass Apple event records and AE records, as well as descriptor lists and descriptor records, to any Apple Event Manager functions that expect records of data type `AEDesc`.

When you use the `AECreateAppleEvent` function, the Apple Event Manager creates an Apple event record containing the attributes for an Apple event's event class, event ID, target address, return ID, and transaction ID. You then use Apple Event Manager functions such as `AEPutParamDesc` and `AEPutAttributeDesc` to add or modify attributes and to add any necessary parameters to the Apple event.

Figure 3-9 shows an example of a complete Apple event—a data structure of type `AppleEvent` containing a list of keyword-specified descriptor records that name the attributes and parameters of an Open Documents event. The figure includes the event class attribute shown in Figure 3-7 and the descriptor list shown in Figure 3-8, which forms the direct parameter—the keyword-specified descriptor record with the keyword `keyDirectObject`. The entire figure corresponds to the Open Documents event shown in Figure 3-3 on page 3-10.

The next two sections, "Responding to Apple Events" and "Creating and Sending Apple Events," provide a quick overview of the steps your application must take to respond to and send Apple events.

Figure 3-9 Data structures within an Open Documents event



Responding to Apple Events

A client application typically uses the Apple Event Manager to create and send an Apple event requesting a service or information. A server application responds by using the Apple Event Manager to process the Apple event, extract data from the attributes and parameters of the Apple event, and if necessary add requested data to the reply event returned by the Apple Event Manager to the client application. The server usually provides its own Apple event handler for performing the action requested by the client's Apple event.

As its first step in supporting Apple events, your application should support the required Apple events sent by the Finder. If you plan to implement publish and subscribe capabilities, your application must also respond to the Apple events sent by the Edition Manager. Your application should also be able to respond to the standard Apple events that other applications are likely to send to it or that it can send to itself. This section provides a quick overview of the tasks your application must perform in responding to Apple events.

To respond to Apple events, your application must

- set the appropriate flags in its `'SIZE'` resource
- test for high-level events in its event loop
- provide Apple event handlers for the Apple events it supports
- use the `AEInstallEventHandler` function to install its Apple event handlers
- use the `AEProcessAppleEvent` function to process Apple events

Before your application can respond to Apple events sent from remote computers, the user of your application must allow network users to link to your application. To do this, the user selects your application in the Finder, chooses Sharing from the File menu, and then clicks the Allow Remote Program Linking checkbox. If the user has not yet started program linking, the Sharing command offers to display the Sharing Setup control panel so that the user can start program linking. The user must also authorize remote users for program linking by using the Users & Groups control panel. Program linking and setting up authenticated sessions are described in the chapter "Program-to-Program Communications Toolbox" in this book.

Accepting and Processing Apple Events

To accept or send Apple events (or any other high-level events), you must set the appropriate flags in your application's `'SIZE'` resource and include code to handle high-level events in your application's main event loop.

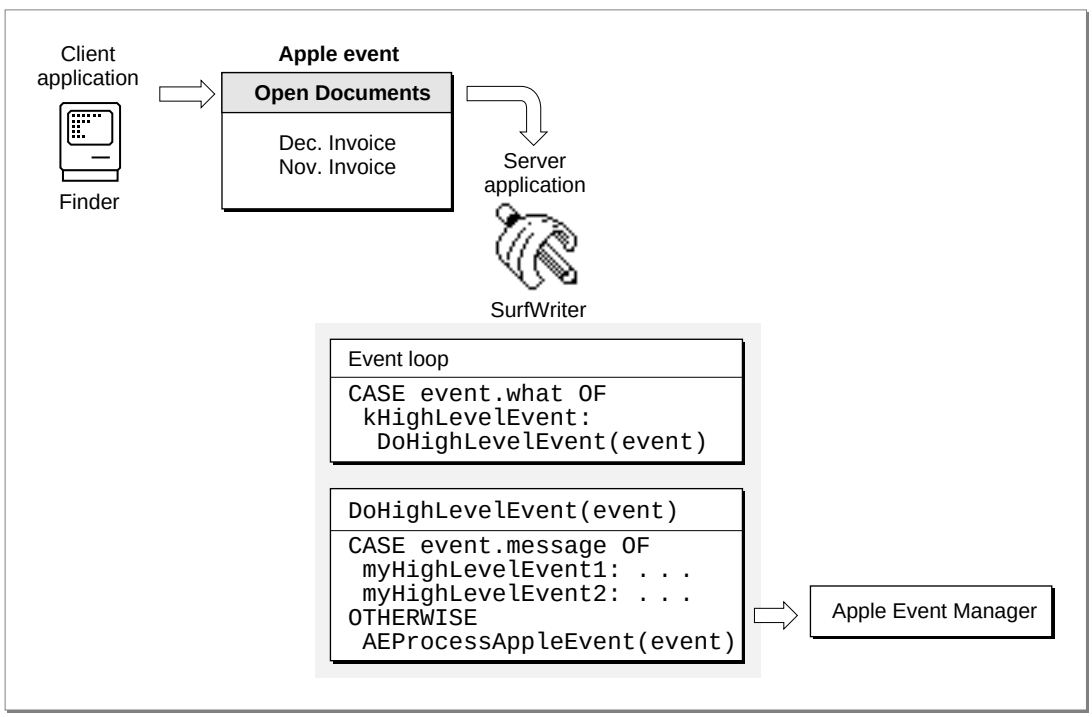
Two flags in the 'SIZE' resource determine whether an application receives high-level events:

- The isHighLevelEventAware flag must be set for your application to receive any high-level events.
- The localAndRemoteHLEvents flag must be set for your application to receive high-level events sent from another computer on the network.

An Apple event (like all high-level events) is identified by a message class of kHighLevelEvent in the what field of the event record. You test the what field of the event record to determine whether it contains the value represented by the kHighLevelEvent constant; if your application defines any high-level events other than Apple events, you should also test the message field of the event record to determine whether the high-level event is something other than an Apple event. If the high-level event is not one that you've defined for your application, assume that it is an Apple event. (You are encouraged to use Apple events instead of defining your own high-level events whenever possible.)

After determining that an event is an Apple event, use the AEProcessAppleEvent function to let the Apple Event Manager identify the event. Figure 3-10 shows how the SurfWriter application accepts and begins to process an Apple event sent by the Finder.

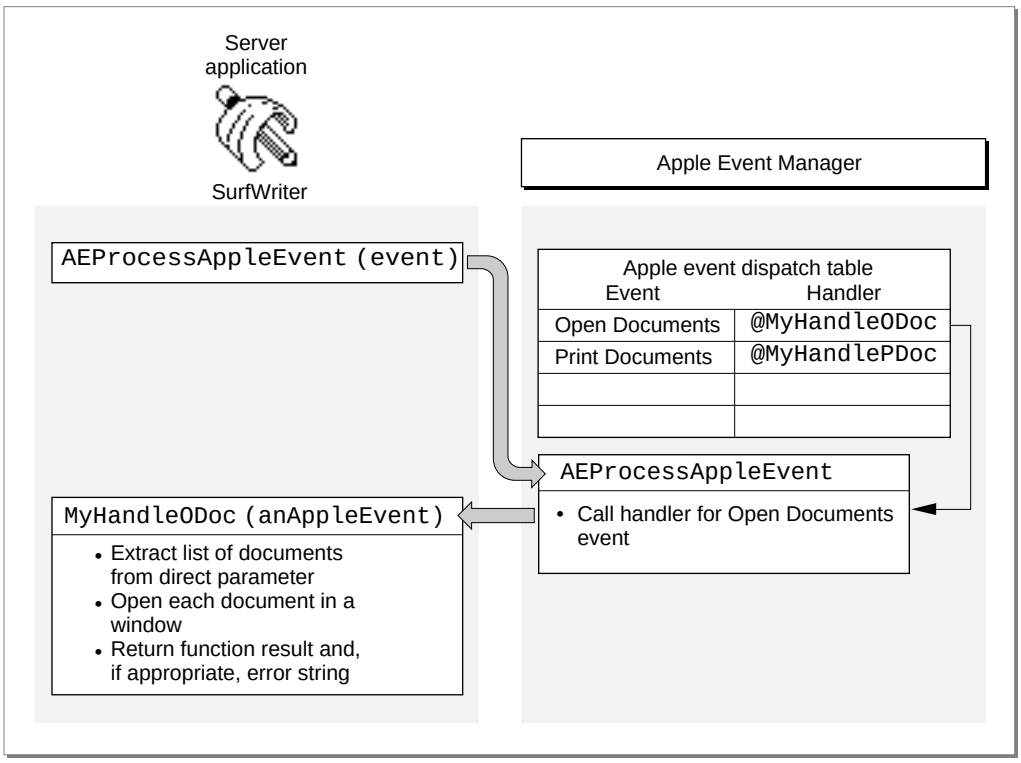
Figure 3-10 Accepting and processing an Open Documents event



The `AEProcessAppleEvent` function first identifies the Apple event by examining the data in the event class and event ID attributes. The function then uses that data to call the Apple event handler that your application provides for that event. The Apple event handler extracts the pertinent data from the Apple event, performs the requested action, and returns a result. For example, if the event has an event class of `kCoreEventClass` and an event ID of `kAEOpenDocuments`, the `AEProcessAppleEvent` function calls your application's handler for the Open Documents event.

Before your application attempts to accept or process any Apple events, it must use the `AEInstallEventHandler` function to install Apple event handlers. This function installs handlers in an ***Apple event dispatch table*** for your application; the Apple Event Manager uses this table to map Apple events to handlers in your application. When your application calls the `AEProcessAppleEvent` function to process an Apple event, the Apple Event Manager checks the Apple event dispatch table and, if your application has installed a handler for that Apple event, calls that handler. Figure 3-11 shows how the flow of control passes from your application to the Apple Event Manager and back to your application.

Figure 3-11 The Apple Event Manager calling the handler for an Open Documents event



About Apple Event Handlers

Your Apple event handlers must generally perform the following tasks:

- extract the parameters and attributes from the Apple event
- check that all the required parameters have been extracted
- locate any Apple event objects specified by object specifier records in the Apple event parameters
- if your application needs to interact with the user, use the `AEInteractWithUser` function to bring it to the foreground
- perform the action requested by the Apple event
- dispose of any copies of descriptor records that have been created
- add information to the reply Apple event if requested

This section describes how your application's Apple event handlers can use the Apple Event Manager to accomplish some of these tasks. The chapter "Responding to Apple Events" in this book provides detailed information about handling Apple events and interacting with the user.

Extracting and Checking Data

You must use Apple Event Manager functions to extract the data from Apple events. You can also use Apple Event Manager functions to extract data from descriptor records, descriptor lists, and AE records. Most of these routines are available in two forms: they either return a copy of the data in a buffer or return a copy of the descriptor record for the data, including a copy of the data.

The following list shows the main functions you can use to access the data of an Apple event:

Function	Description
`AEGetParamPtr`	Uses a buffer to return a copy of the data contained in an Apple event parameter. Usually used to extract data of fixed length or known maximum length; for example, to extract the result code from the `keyErrorNumber` parameter of a reply Apple event.
`AEGetParamDesc`	Returns a copy of the descriptor record or descriptor list for an Apple event parameter. Usually used to extract data of variable length; for example, to extract the descriptor list for a list of alias records specified in the direct parameter of the Open Documents event.
`AEGetAttributePtr`	Uses a buffer to return a copy of the data contained in an Apple event attribute. Used to extract data of fixed length or known maximum length; for example, to determine the source of an Apple event by extracting the data from the `keyEventSourceAttr` attribute.

continued

Function	Description (continued)
`AEGetAttributeDesc`	Returns a copy of the descriptor record for an attribute. Used to extract data of variable length; for example, to make a copy of a descriptor record containing the address of an application.
`AECountItems`	Returns the number of descriptor records in a descriptor list. Used, for example, to determine the number of alias records for documents specified in the direct parameter of the Open Documents event.
`AEGetNthPtr`	Uses a buffer to return a copy of the data for a descriptor record contained in a descriptor list. Used to extract data of fixed length or known maximum length; for example, to extract the name and location of a document from the descriptor list specified in the direct parameter of the Open Documents event.
`AEGetNthDesc`	Returns a copy of a descriptor record from a descriptor list. Used to extract data of variable length; for example, to get the descriptor record containing an alias record from the list specified in the direct parameter of the Open Documents event.

You can specify the descriptor type of the resulting data for these functions; if this type is different from the descriptor type of the attribute or parameter, the Apple Event Manager attempts to coerce it to the specified type. In the direct parameter of the Open Documents event, for example, each descriptor record in the descriptor list is an alias record; each alias record specifies a document to be opened. As explained in the chapter "Introduction to File Management" of *Inside Macintosh: Files*, all your application usually needs is the file system specification (`FSSpec`) record of the document. When you extract the descriptor record from the descriptor list, you can request that the Apple Event Manager return the data to your application as a file system specification record instead of an alias record.

After extracting all known Apple event parameters, your handler should check that it retrieved all the parameters that the source application considered to be required. To do so, determine whether the `keyMissedKeywordAttr` attribute exists. If so, your handler has not retrieved all the required parameters, and it should return an error.

Although the *Apple Event Registry: Standard Suites* defines Apple event parameters as either required or optional, the Apple Event Manager does not enforce the definitions of required and optional events. Instead, the source application specifies, when it sends the event, which Apple event parameters the target can treat as if they were optional. For more information about optional parameters, see "Specifying Optional Parameters for an Apple Event," which begins on page 5-7.

If any of the Apple event parameters include object specifier records, your handler should use the `AEResolve` function, other Apple Event Manager routines, and your own application-defined functions to locate the corresponding Apple event objects. For more information about locating Apple event objects, see "Working With Object Specifier Records," which begins on page 3-32.

Interacting With the User

In some cases, the server may need to interact with the user when it handles an Apple event. For example, your handler for the Print Documents event may need to display a print options dialog box and get settings from the user before printing. By specifying flags to the `AESetInteractionAllowed` function, you can set preferences to allow user interaction with your application (a) only when your application is sending the Apple event to itself, (b) only when the client application is on the same computer as your application, or (c) for any event sent by any client application on any computer. In addition, your handler should always use the `AEInteractWithUser` function before displaying a dialog box or alert box or otherwise interacting with the user. The `AEInteractWithUser` function determines whether user interaction can occur and takes appropriate action depending on the circumstances.

Both the client and server specify their preferences for user interaction: the client specifies whether the server should be allowed to interact with the user, and the server specifies when it allows user interaction while processing an Apple event. The Apple Event Manager does not allow a server application to interact with the user in response to a client application's Apple event unless at least two conditions are met: First, the client application must set flags in the `sendMode` parameter of the `AESend` function indicating that user interaction is allowed. Second, the server application must either set no user interaction preferences, in which case `AEInteractWithUser` assumes that only interaction with a client on the local computer is allowed; or it must set flags to the `AESetInteractionAllowed` function indicating that user interaction is allowed.

If these two conditions are met and if `AEInteractWithUser` determines that both the client and server applications allow user interaction under the current circumstances, `AEInteractWithUser` brings your application to the foreground if it isn't already in the foreground. Your application can then display its dialog box or alert box or otherwise interact with the user. The `AEInteractWithUser` function brings your server application to the front either directly or after the user responds to a notification request.

For detailed information about how to specify flags to the `AESetInteractionAllowed` function and how the Apple Event Manager determines whether user interaction is allowed, see the section "Interacting With the User," which begins on page 4-45.

Performing the Requested Action and Returning a Result

When your application responds to an Apple event, it should perform the standard action requested by that event. For example, your application should respond to the Open Documents event by opening the specified documents in titled windows just as if the user had selected each document from the Finder and then chosen Open from the File menu.

Many Apple events can ask your application to return data. For instance, if your application is a spelling checker, the client application might expect your application to return data in the form of a list of misspelled words. Figure 3-14 on page 3-38 shows a similar example: a Get Data event that asks the server application to locate a specific Apple event object and return the data associated with it.

If the client application requests a reply, the Apple Event Manager prepares a reply Apple event by passing a default reply Apple event to your handler. If the client application does not request a reply, the Apple Event Manager passes a ***null descriptor record***—that is, a descriptor record of type `typeNull` whose data handle has the value `NIL`—to your handler instead of a default reply Apple event. The default reply Apple event has no parameters when it is passed to your handler, but your handler can add parameters to it. If your application is a spelling checker, for example, you can return a list of misspelled words in a parameter. However, your handler should check whether the reply Apple event exists before attempting to add any attributes or parameters to it. Any attempt to add an Apple event attribute or parameter to a null descriptor record generates an error.

When you extract a descriptor record using the `AEGetParamDesc`, `AEGetAttributeDesc`, `AEGetNthDesc`, or `AEGetKeyDesc` function, the Apple Event Manager creates a copy of the descriptor record for you to use. When your handler is finished using a copy of a descriptor record, you should dispose of it—and thereby deallocate the memory used by its data—by calling the `AEDisposeDesc` function.

Note
Outputs from functions such as `AEGetKeyPtr` and other routines whose names end in `-Ptr` use a buffer rather than a descriptor record to return data. Because these functions don't require the use of `AEDisposeDesc`, it is preferable to use them for any data that is not identified by a handle. ◆

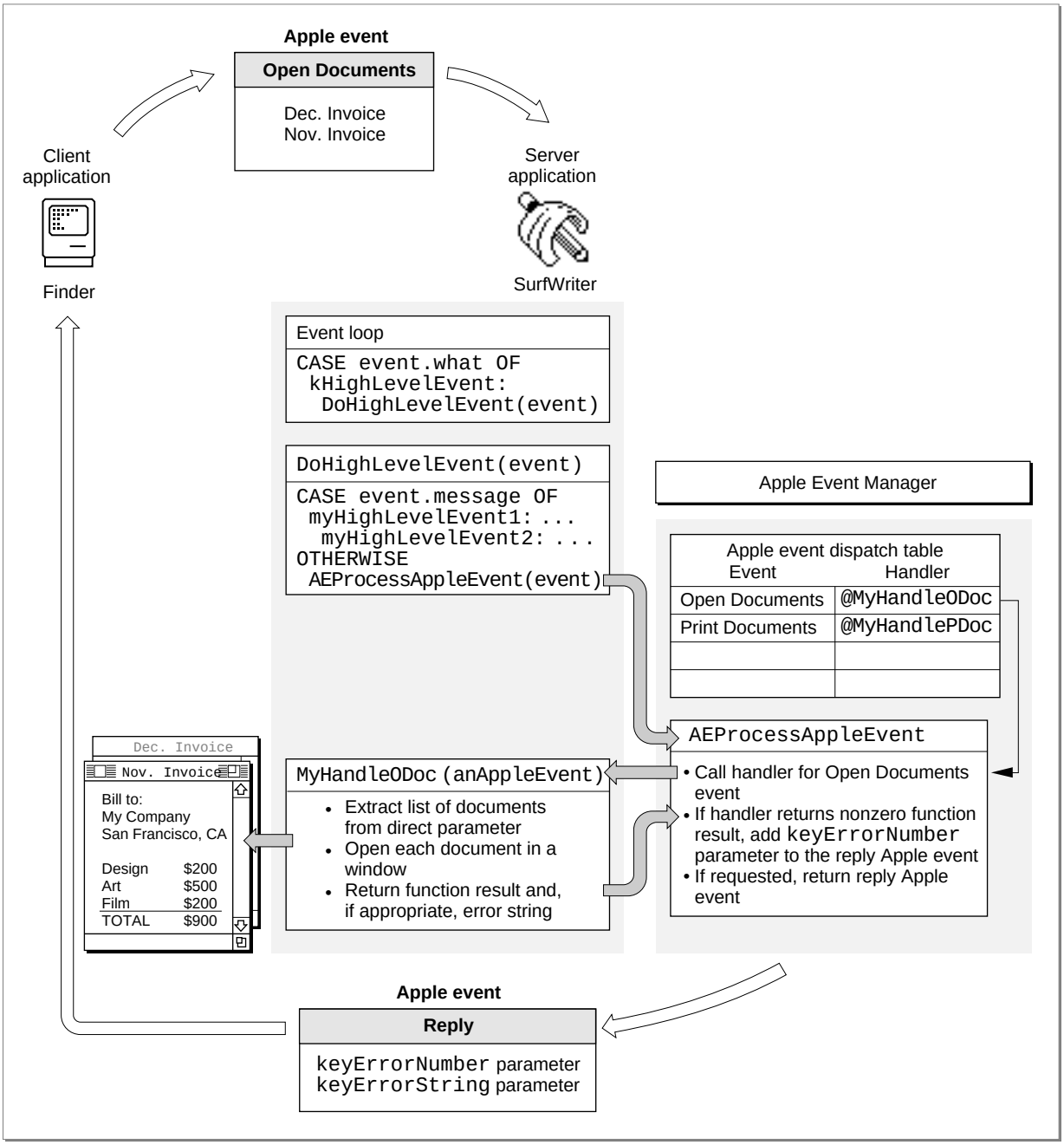
Your Apple event handler should always set its function result either to `noErr` if it successfully handles the Apple event or to a nonzero result code if an error occurs. If your handler returns a nonzero result code, the Apple Event Manager adds a `keyErrorNumber` parameter to the reply Apple event (unless you have already added a `keyErrorNumber` parameter). This parameter contains the result code that your handler returns. The client should check whether the `keyErrorNumber` parameter exists to determine whether your handler performed the requested action. In addition to returning a result code, your handler can also return an error string in the `keyErrorString` parameter of the reply Apple event. The client can use this string in an error message to the user.

If the client application requested a reply, the Apple Event Manager returns the reply Apple event, which is identified by the event class `kCoreEventClass` and by the event ID `kAEAnswer`. When the client has finished using the reply Apple event, it should dispose of both the reply event and the original event—and thereby deallocate the memory they use—by calling the `AEDisposeDesc` function. The Apple Event Manager takes care of disposing both the Apple event and the reply Apple event after a server

application's handler returns to `AEProcessAppleEvent`, but a server application is responsible for disposing of any Apple event data structures it creates while extracting data from the Apple event.

Figure 3-12 shows the entire process of responding to an Apple event.

Figure 3-12 Responding to an Open Documents event



When your handler returns a result code to the Apple Event Manager, you have finished your response to the client application's Apple event.

Creating and Sending Apple Events

Your application can use Apple events to request services or information from other applications, send information to other applications, or trigger actions within your application. For example, you can use the core Apple event Get Data to request specific data from another application's documents. Similarly, you can use other Apple events to request services—for example, asking a spell-checking application to check the text in a document created by your application. Consult the *Apple Event Registry: Standard Suites* for the format and function of the standard Apple events that you want your application to send.

To communicate with another application by sending an Apple event, your application must

- set the appropriate flags in its `'SIZE'` resource
- create an Apple event record by calling the `AECreateAppleEvent` function
- use Apple Event Manager functions to add parameters and any additional attributes to the Apple event
- call the `AESend` function to send the Apple event
- dispose of any copies of descriptor records that you have created
- handle the reply Apple event (if necessary)

The sections that follow describe how your application can use the Apple Event Manager to accomplish these tasks. The chapter "Creating and Sending Apple Events" in this book provides detailed information about creating and sending Apple events.

To act as a server for your application, the target application must support high-level events and must be running. The server can be your own application, another application running on the user's computer, or an application running on another user's computer connected to the network.

Your application should also allow the user to choose among the various applications available as servers. The `PPCBrowser` function allows users to select target applications on their own computers or on computers connected to the network. The `PPCBrowser` function presents a standard user interface for choosing a target application, much as the Standard File Package provides a standard user interface for opening and saving files. See the chapter "Program-to-Program Communications Toolbox" in this book for details on using the `PPCBrowser` function.

If the server application is on a remote computer on a network, the user of that computer must allow program linking to the server application. The user of the server application does this by selecting the application icon in the Finder, choosing Sharing from the File menu, then clicking the Allow Remote Program Linking checkbox. If the user has not yet started program linking, the Sharing command offers to display the Sharing Setup control panel so that the user can start program linking. The user must also authorize remote users for program linking by using the Users & Groups control panel. Program linking and setting up authenticated sessions are described in the chapter "Program-to-Program Communications Toolbox" in this book.

Creating an Apple Event Record

Use the `AECreateAppleEvent` function to create an Apple event record. Using the arguments you pass to the `AECreateAppleEvent` function, the Apple Event Manager constructs the data structures describing the event class, the event ID, and the target address attributes of an Apple event. The event class and event ID, of course, identify the particular event you wish to send. The target address identifies the intended recipient of the Apple event.

You can specify two other attributes with the `AECreateAppleEvent` function: the reply ID and the transaction ID. For the reply ID attribute, you usually specify the `kAutoGenerateReturnID` constant to the `AECreateAppleEvent` function. This constant ensures that the Apple Event Manager generates a unique return ID for the reply Apple event returned from the server. For the transaction ID attribute, you usually specify the `kAnyTransactionID` constant, which indicates that this Apple event is not one of a series of interdependent Apple events.

Adding Apple Event Attributes and Parameters

The Apple event record created with the `AECreateAppleEvent` function serves as a foundation for the Apple event you want to send. Descriptor records and descriptor lists are the building blocks from which the complete Apple event record is constructed. To create descriptor records and descriptor lists and add items to a descriptor list, use the following functions:

Function	Description
`AECreateDesc`	Takes a descriptor type and a pointer to data and converts them into a descriptor record
`AECreateList`	Creates an empty descriptor list or AE record.
`AEPutPtr`	Takes a descriptor type and a pointer to data and adds the data to a descriptor list as a descriptor record; used, for example, to add to a descriptor list a number used as the parameter of an Apple event requesting a calculation.
`AEPutDesc`	Adds a descriptor record to a descriptor list; used, for example, to add to a descriptor list an alias record used as the direct parameter of an Apple event requesting file manipulation.

To add the remaining attributes and parameters necessary for your Apple event to the Apple event record, you can use these additional Apple Event Manager functions:

Function	Description
`AEPutParamPtr`	Takes a keyword, descriptor type, and pointer to data and adds the data to an Apple event record as a parameter with the specified keyword (replacing any existing parameter with the same keyword); used, for example, to put numbers into the parameters of an Apple event that asks the server to perform a calculation.
`AEPutParamDesc`	Takes a keyword and a descriptor record and adds the descriptor record to an Apple event record as a parameter with the specified keyword (replacing any existing parameter with the same keyword); used, for example, to place a descriptor list containing alias records into the direct parameter of an Apple event that requests a server to manipulate files.
`AEPutAttributePtr`	Takes a keyword, descriptor type, and pointer to data and adds the descriptor record to an Apple event record as an attribute with the specified keyword (replacing any existing attribute with the same keyword); used, for example, to change the transaction ID of an Apple event record that is waiting to be sent.
`AEPutAttributeDesc`	Takes a keyword and a descriptor record and adds the descriptor record to an Apple event record as an attribute with the specified keyword (replacing any existing attribute with the same keyword); used, for example, to replace the descriptor record used for the target address attribute in an Apple event record waiting to be sent.

Apple event parameters for core events and functional-area events can include descriptions of Apple event objects in special descriptor records called object specifier records. For an overview of object specifier records, see "Working With Object Specifier Records," which begins on page 3-32.

Sending an Apple Event and Handling the Reply

After you add all the attributes and parameters required for the Apple event, use the `AESend` function to send the Apple event. The Apple Event Manager uses the Event Manager to transmit the Apple event to the server application.

The `AESend` function requires that you specify whether your application should wait for a reply from the server. If you specify that you want a reply, the Apple Event Manager prepares a reply Apple event for your application by passing a default reply Apple event to the server. The Apple Event Manager returns any nonzero result code from the server's handler in the `keyErrorNumber` parameter of the reply Apple event. The server can return an error string in the `keyErrorString` parameter of the reply Apple event. The server can also use the reply Apple event to return any data you requested—for example, the results of a numeric calculation or a list of misspelled words.

You specify how your application should wait for a reply by using one of these flags in the `sendMode` parameter of the `AESend` function:

Flag	Description
`kAENoReply`	Your application does not want a reply Apple event.
`kAEQueueReply`	Your application wants a reply Apple event; the reply appears in your event queue as soon as the server has the opportunity to process and respond to your Apple event.
`kAEWaitReply`	Your application wants a reply Apple event and is willing to give up the processor while waiting for the reply; for example, if the server application is on the same computer as your application, your application yields the processor to allow the server to respond to your Apple event.

If you specify the `kAEWaitReply` flag, you should provide an idle function. This function should process any non-high-level events that occur while your application is waiting for a reply. You supply a pointer to your idle function as a parameter to the `AESend` function. So that your application can process other Apple events while it is waiting for a reply, you can also specify an optional filter function to the `AESend` function.

If you specify the `kAENoReply` flag, the reply Apple event prepared by the Apple Event Manager for the server application consists of a null descriptor record.

If your Apple event may require the user to interact with the server application (for example, to specify print or file options), you can communicate your user interaction preferences to the server by specifying additional flags in the `sendMode` parameter of the `AESend` function. These flags specify the conditions, if any, under which the server application can interact with the user and, if interaction is allowed, whether the server should come directly to the foreground or post a notification request.

The server application specifies its own preferences for user interaction by specifying flags to the `AESetInteractionAllowed` function, as described in the previous section. The interaction of the client and server applications' preferences is explained in detail in "Interacting With the User," which begins on page 4-45.

After you send an Apple event, your application is responsible for disposing of the Apple event record—and thereby deallocating the memory its data uses—by calling the `AEDisposeDesc` function. If you create one descriptor record and add it to another, the Apple Event Manager adds a copy of the newly created one to the existing one and also makes a copy of the associated data. For example, you might use the `AECreateDesc` function to create a descriptor record that you wish to add to an Apple event. When you use the `AEPutParamDesc` function, it adds a copy of your newly created descriptor record, including its data, as a parameter to an existing Apple event. When you no longer need the original descriptor record, you should call `AEDisposeDesc` to dispose of it.

Your application should dispose of all the descriptor records that are created for the purposes of adding parameters and attributes to an Apple event. You normally dispose of your Apple event and its reply after you receive a result from the `AESend` function. You should dispose of these even if `AESend` returns an error result.

If you specify the `kAEWaitReply` flag, the reply Apple event is returned in a parameter you pass to the `AESend` function. If you specify the `kAEQueueReply` flag to the `AESend` function, the reply Apple event is returned in the event queue. In this case, the reply is identified by the event class `kCoreEventClass` and the event ID `kAEAnswer`. Your application processes reply events in its event queue in the same way that server applications process Apple events.

Your application should check for the existence of the `keyErrorNumber` parameter of the reply Apple event to ensure that the server performed the requested action. The server can also return, in the `keyErrorString` parameter, any error messages you need to display to the user.

Whenever a server application provides an error string, it should also provide an error number. However, you can't count on all server applications to do so. The absence of the `keyErrorNumber` parameter doesn't necessarily mean that there won't an error string provided in the `keyErrorString` parameter. A client application should therefore check for both the `keyErrorNumber` and `keyErrorString` parameters before assuming that no error has occurred. If a string has been provided without an error number, an error has occurred.

After extracting the information it needs from the reply event, your handler should dispose of the reply by calling the `AEDisposeDesc` function. Similarly, when your handler no longer needs descriptor records it has extracted from the reply, it should call `AEDisposeDesc` to dispose of them.

The next section provides an overview of the way a source application identifies Apple event objects supported by a target application. If you are starting by supporting only the Required suite and the Apple events sent by the Edition Manager, you can skip the next section and go directly to "About the Apple Event Manager," which begins on page 3-48.

Working With Object Specifier Records

Most of the standard Apple events allow the source application to refer, in an Apple event parameter, to Apple event objects within the target application or its documents. The Apple Event Manager allows applications to construct and interpret such references by means of a standard classification system for Apple event objects. This system, described in detail in the *Apple Event Registry: Standard Suites,* is summarized in "The Classification of Apple Event Objects," which begins on page 3-39. A description in an Apple event parameter that uses this classification system takes the form of an object specifier record.

An ***object specifier record*** is a descriptor record of descriptor type `typeObjectSpecifier` that describes the location of one or more Apple event objects: for example, the table "Summary of Sales" in the document "Sales Report," or the third row in that table, or the last row of the column "Totals." With the aid of application-defined functions, the Apple Event Manager can conduct a step-by-step search according to such instructions in an object specifier record, locating first the document, then the table, then other objects, and so on until the requested object has been identified. Object specifier records can specify many combinations of identifying characteristics that cannot be specified using one of the simple data types.

This section introduces object specifier records and the organization of their data. You need to read this section (a) if you plan to support the Core suite or any of the standard functional-area suites and (b) if you want to make your application scriptable—that is, capable of responding to scripts written in a scripting language.

IMPORTANT

An object specifier record identifies one or more Apple event objects among many; it contains a description of each object, not the object itself. An Apple event object described by an object specifier record exists only in the server application's document or in the server application itself. ▲

A client application cannot retrieve an Apple event object from a server application unless the server application can accurately locate it. Thus, to locate characters of a specific color, a server application must be able to identify a single character's color; to locate a character in a cell, a server application must be able to locate both the table and the cell.

A client application can create object specifier records for use as Apple event parameters. Scripting components can also create object specifier records as Apple event parameters for the Apple events they generate in the course of executing a script. A server application that receives an Apple event containing an object specifier record should resolve the object specifier record—that is, locate the requested Apple event objects.

To respond to core and functional-area Apple events received by your application, you must first define a hierarchy of Apple event objects for your application that you want other applications or scripting languages to be able to describe. The Apple event objects for your application should be based as closely as possible on the standard object classes described by the *Apple Event Registry: Standard Suites*. After you have decided which of the standard Apple event objects make sense for your application, you can write functions that locate objects on the basis of information in an object specifier record. If you want your application to send specific Apple events to other applications, you must also write functions that can create object specifier records and add them to Apple events. Your application does not need to create object specifier records in order to be scriptable. However, to write functions that can help the Apple Event Manager resolve object specifier records, you need to know how they are constructed.

"Finding Apple Event Objects," which begins on page 3-46, provides an overview of the way the Apple Event Manager works with your application-defined functions to locate the Apple event objects described in an object specifier record. The chapter "Resolving and Creating Object Specifier Records" in this book describes in detail how to support object specifier records as a server or client application.

Data Structures Within an Object Specifier Record

The organization of the data for an object specifier record is nearly identical to that of the data for an AE record. An object specifier record is a structure of data type `AEDesc` whose data handle usually refers to four keyword-specified descriptor records describing one or more Apple event objects. An AE record is a structure of data type `AERecord` whose data handle refers to one or more Apple event parameters.

The four keyword-specified descriptor records for an object specifier record provide information about the requested Apple event object or objects.

Keyword	Description of data
`keyAEDesiredClass`	Four-character code indicating the object class ID
`keyAEContainer`	A description of the container for the requested object, usually in the form of another object specifier record
`keyAEKeyForm`	Four-character code for the key form, which indicates how to interpret the key data
`keyAEKeyData`	Key data, used to distinguish the desired Apple event object from other objects of the same object class in the same container

For example, the data for an object specifier record identifying a table named "Summary of Sales" in a document named "Sales Report" consists of four keyword-specified descriptor records that provide the following information:

- the object class ID for a table
- another object specifier record identifying the document "Sales Report" as the container for the table
- a key form constant indicating that the key data contains a name
- key data that consists of the string "Summary of Sales"

The ***object class ID*** specifies the Apple event object class to which the object belongs. An Apple event object class is a category for Apple event objects that share specific characteristics (see "Apple Events and Apple Event Objects" on page 3-6). The characteristics of each object class are listed in the *Apple Event Registry: Standard Suites*. For example, the Core suite defines object classes for documents, paragraphs, words, windows, and floating windows. The first keyword-specified descriptor record in an object specifier record uses a four-character code or a constant to specify the object class ID. The object class for words, for example, can be identified by either the object class ID `'cwor'` or the constant `cWord`.

Note
The object class ID identifies the object class of an Apple event object described in an object specifier record, whereas the event class and event ID identify an Apple event. ◆

The ***container*** for an Apple event object is usually another Apple event object. For example, the container for a document might be a window, and the container for characters, delimited items, or a word might be another word, a paragraph, or a document. The container is identified by the second keyword-specified descriptor record in an object specifier record; usually this is another object specifier record. The container can also be specified by a null descriptor record, which indicates a default container or a container already known to the Apple Event Manager.

The descriptor record in an object specifier record that identifies an Apple event object's container can in turn use another object specifier record to identify the container's container, and so on until the Apple event object is fully specified. For example, an object specifier record identifying a paragraph might specify the paragraph's container with another object specifier record that identifies a page. That object specifier record might in turn specify the page's container with another object specifier record identifying a document. The ability to nest one object specifier record within another in this way makes it possible to identify elements such as "the first row in the table named 'Summary of Sales' in the document named 'Sales Report.'"

The ***key form*** and ***key data*** distinguish the desired Apple event object from other Apple event objects of the same object class. The key form describes the form the key data takes. The third keyword-specified descriptor record in an object specifier record usually specifies the key form with one of seven standard constants:

Key form	Value	Corresponding key data
`formPropertyID`	`'prop'`	Property ID for an element's property
`formName`	`'name'`	Element's name
`formUniqueID`	`'ID  '`	A value that uniquely identifies an object within its container or across an application
`formAbsolutePosition`	`'indx'`	An integer or other constant indicating the position of one or more elements in relation to the beginning or end of their container
`formRelativePosition`	`'rele'`	A constant that specifies the element just before or after the container
`formTest`	`'test'`	Descriptor records that specify a test
`formRange`	`'rang'`	Descriptor records that specify a group of elements between two other elements

A key form of `formPropertyID` indicates key data that specifies a property. A ***property*** of an Apple event object is a specific characteristic of that object that can be identified by a constant. The properties associated with the object class for documents include the name of the document and a flag indicating whether the document has been modified since the last save. The properties associated with the object class for words include color, font, point size, and style.

Figure 3-13 shows the structure of a typical object specifier record: four keyword-specified descriptor records that specify the class ID, the container, the key form, and the key data. These four keyword-specified descriptor records are the data for a descriptor record (`AEDesc`) of descriptor type `typeObjectSpecifier`. Note the similarities between the object specifier record shown in Figure 3-13 and the Apple event record shown in Figure 3-9 on page 3-19. Like an Apple event record or an AE record, an object specifier record consists of a list of keyword-specified descriptor records.

Figure 3-13 shows the structure of a simple object specifier record that specifies the key form `formPropertyID`, `formName`, or `formAbsolutePosition`. For detailed information about the structure of object specifier records that specify the other key forms, see the chapter "Resolving and Creating Object Specifier Records" in this book.

Figure 3-13 Data structures within a simple object specifier record

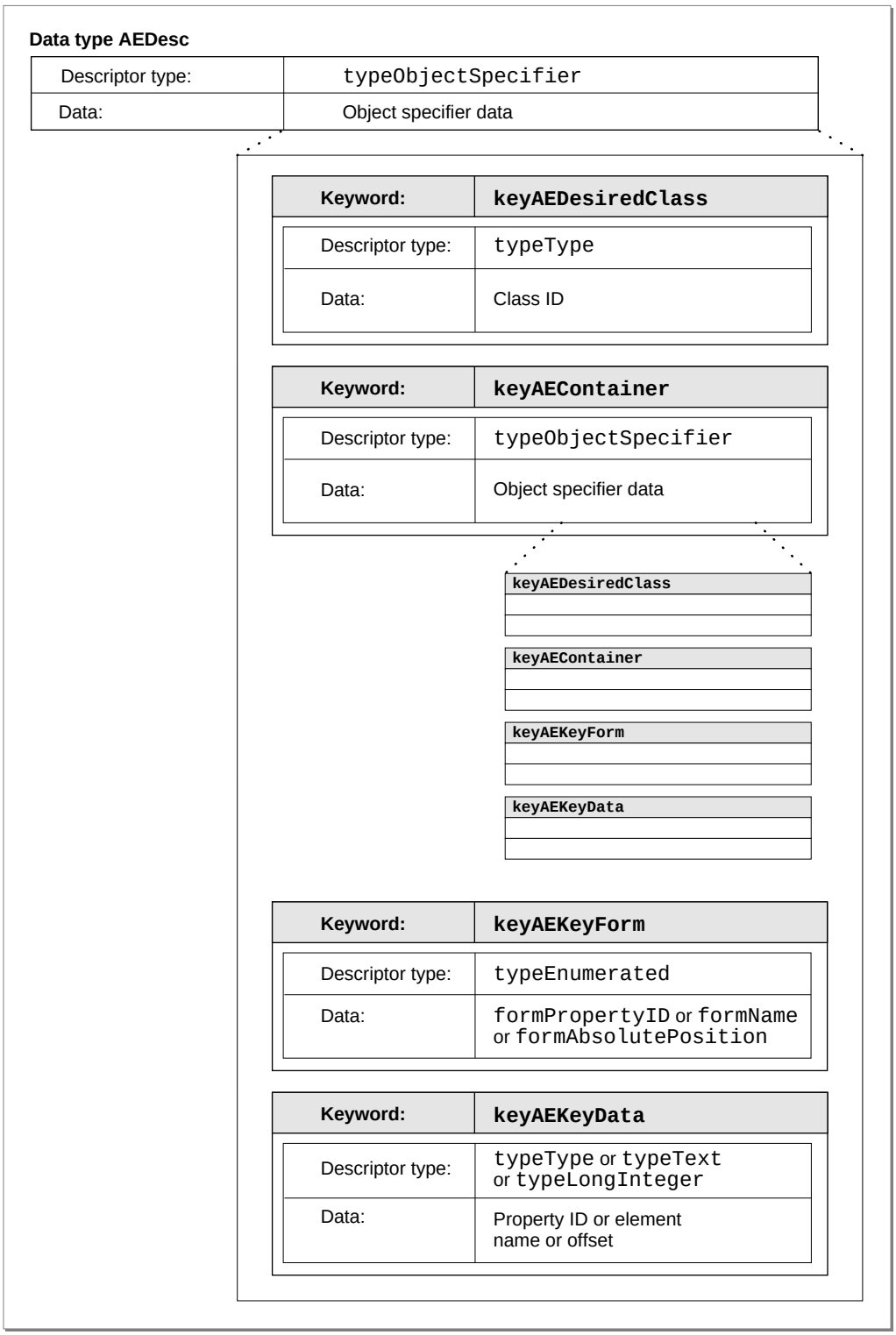
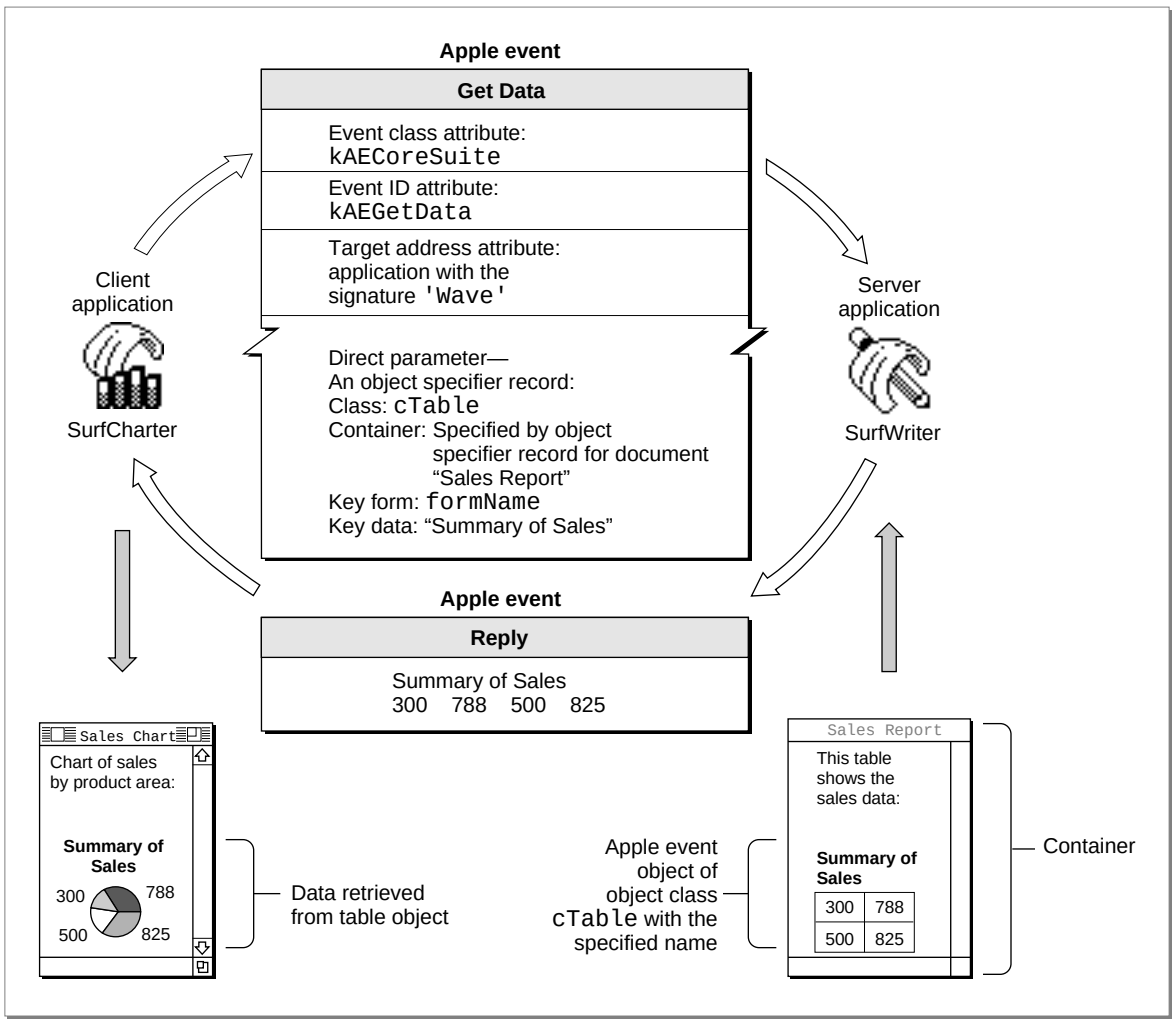


Figure 3-14 shows the object specifier record for the Get Data event previously illustrated in Figure 3-4 on page 3-11. The object class ID tells the SurfWriter application that the requested data is an element of class `cTable`. The container for the table is the document "Sales Report." The key form is `formName`, which tells the server application that the key data identifies the Apple event object by name. The key data is the name of the table.

Figure 3-14 An object specifier record in a Get Data event



To add an object specifier record to an Apple event as one of its parameters, your application must first create the object specifier record. "Creating Object Specifier Records," which begins on page 6-55, describes the Apple Event Manager routines for creating object specifier records.

To respond to Apple events that include object specifier records, your application should use the standard classification system for Apple event objects and provide functions that can locate those objects within your application or its documents. The next section summarizes the classification of Apple event objects as defined in the *Apple Event Registry: Standard Suites.*

The Classification of Apple Event Objects

To create or resolve object specifier records, your application should use the classification of Apple event objects defined by the *Apple Event Registry: Standard Suites.* This section summarizes the concepts that underlie that classification system. You should have a copy of the *Apple Event Registry: Standard Suites* available for reference purposes while you read this section.

You do not need to write your application in an object-oriented programming language in order to support Apple event objects in your application. However, you must understand the classification system described in this section in order to classify Apple event objects in your application and to write routines that can locate them on the basis of information contained in object specifier records.

Object Classes

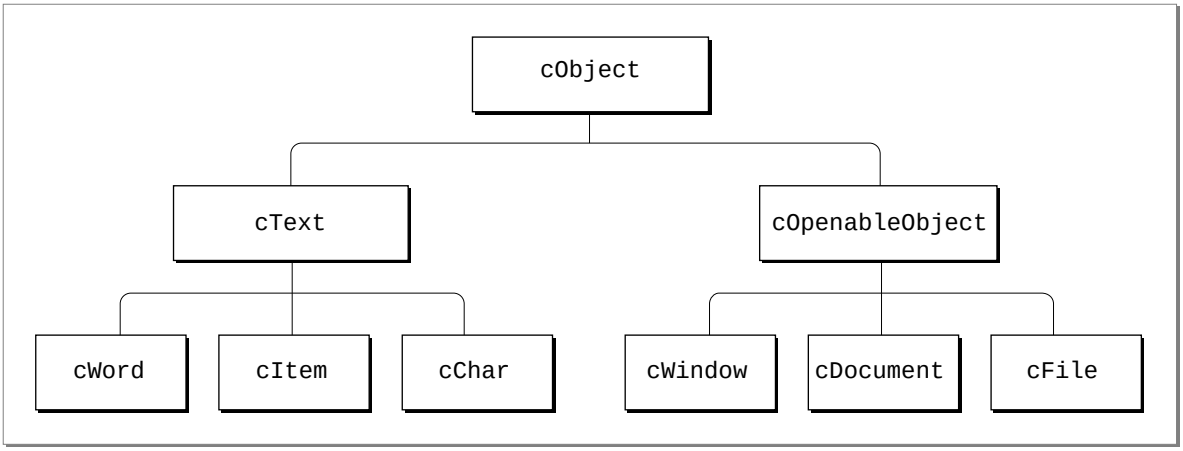
Except for the concept of inheritance, Apple event objects are different from the objects used in object-oriented programming languages. Apple event objects are distinct items in a server application or any of its documents that can be specified by an object specifier record in an Apple event sent by a client application. Apple event objects are often, but not always, items that a user can differentiate and manipulate within an application, such as words, paragraphs, shapes, windows, or style formats. Every Apple event object can be classified according to its object class, which defines both its characteristics and its behavior. The object classes listed in the *Apple Event Registry: Standard Suites* provide a method of describing Apple event objects that all applications can understand. Object classes permit more flexibility than simple descriptor types; for example, a word can be defined as a simple string, or it can be defined as an Apple event object with specific characteristics such as font or style.

Note
The definition of an object class only specifies conventions that determine how applications should handle Apple event objects that belong to that class. Your application must make sure that it uses the conventions correctly; they are not enforced by the Apple Event Manager. ◆

Each object class is identified by a four-character object class ID, which can also be represented by a constant. Constants for object classes always begin with the letter `c`.

The definition of an object class specifies its ***superclass***, which is the object class from which a ***subclass*** (the class being defined) inherits some of its characteristics. Characteristics can also be inherited from special object classes, called ***abstract superclasses,*** that are used only in definitions of object classes and do not refer to real Apple event objects. The pattern of inheritance among object classes is called the ***object class inheritance hierarchy.*** Figure 3-15 shows a portion of this hierarchy. The abstract superclass `cObject` is at the top of the hierarchy and is therefore the only object class that has no superclass. At the next level are `cText`, which is a regular object class, and `cOpenableObject`, which is an abstract superclass. Both are subclasses of `cObject` and superclasses for their own subclasses. The object classes `cWord`, `cItem`, and `cChar` are all subclasses of `cText`. Similarly, `cWindow`, `cDocument`, and `cFile` are subclasses of `cOpenableObject`. Every object class inherits all the characteristics of its superclass and can also add characteristics of its own.

Figure 3-15 Superclasses and subclasses



Here are some of the object classes defined for the Core suite:

Class	Class ID	Description
`cChar`	`'cha '`	Text characters
`cDocument`	`'docu'`	Macintosh documents
`cFile`	`'cfil'`	Macintosh files
`cSelection`	`'csel'`	User or application selections
`cText`	`'ctxt'`	Series of characters
`cWindow`	`'cwin'`	Standard Macintosh windows

Here are some of the object classes defined for the Text suite:

Class	Class ID	Description
`cChar`	`'cha '`	Text characters
`cLine`	`'clin'`	Lines of text
`cParagraph`	`'cpar'`	Paragraphs
`cText`	`'ctxt'`	Series of characters
`cTextFlow`	`'cflo'`	Text flows
`cWord`	`'cwor'`	Words

As you can see, some object classes, such as `cChar` and `cText`, are defined in more than one suite. For example, the definition of the `cText` object class in the Text suite is an ***extension*** of the `cText` object class defined in the Core suite; it duplicates all the characteristics of the Core suite object class and adds some of its own. Like a word in a dictionary, one object class ID can have several related definitions. You can choose to support the definition that best suits your application; or, if necessary, you can create extensions of your own. The extension of an object class is different from inheritance between object classes. An extension of a standard object class provides additional ways of describing an Apple event object of that class, whereas the object class inheritance hierarchy determines the pattern of characteristics shared by different object classes.

The definition of an object class always specifies a default descriptor type. Suppose, for example, that a client application sends a Get Data, Cut, or Copy event that specifies an Apple event object but does not specify a descriptor type for the returned data. In this case, the server application returns a descriptor record of the default descriptor type for the object class of the specified Apple event object. For example, the default descriptor type for Apple event objects of class `cWord` is `typeIntlText`, a descriptor type that specifies an undelimited string of characters in a specific language and script system. The client application can also request that the data be returned in a descriptor record of some other data type.

The definition of an object class includes three lists of characteristics: properties, element classes, and Apple events that support the object class. (The next section describes properties and element classes.) Any or all of these characteristics may be inherited from a superclass. An Apple event is listed for an object class if its parameters can specify objects of that class. The definition for `cWindow`, for example, lists 12 Apple events, including the Open, Close, and Move events, whose parameters can include object specifier records that specify windows. The `cWindow` class inherits all of these Apple events from its abstract superclass, `cOpenableObject`.

The *Apple Event Registry: Standard Suites* also defines ***primitive object classes,*** which describe Apple event objects that contain a single value. For example, the `cBoolean`, `cLongInteger`, and `cAlias` object classes are all primitive object classes. The object class ID for a primitive object class is the same as the four-character value of its descriptor type. Primitive object classes contain no properties; they contain only the value of the data.

Properties and Elements

The properties listed for an object class can be used to identify characteristics of Apple event objects that belong to that class. Each property is identified by a four-character property ID, which can also be represented by a constant. Constants for properties always begin with the letter `p`.

Here are constants and property IDs for some properties:

Property	Property ID	Description
`pName`	`'pnam'`	Name of an Apple event object
`pBounds`	`'pbnd'`	Coordinates of a window
`pVisible`	`'pvis'`	Indicates whether a window is visible
`pIsModal`	`'pmod'`	Indicates whether a window is modal
`pClass`	`'pcls'`	Class ID of an Apple event object
`pFont`	`'font'`	Font
`pTextStyle`	`'txst'`	Text style
`pColor`	`'colr'`	Text color
`pTextPointSize`	`'ptps'`	Point size
`pScriptTag`	`'psct'`	Script system identifier
`pFillColor`	`'flcl'`	Fill color

The property of an Apple event object is itself defined as a single Apple event object whose container is the object to which the property belongs. For example, the `pFont` property of a word is defined by the name of a font, such as New York; the string that identifies the font is an Apple event object of class `cText`.

The constant `cProperty` specifies the object class for any object specifier record that identifies a property.

```
CONST cProperty = 'prop';
```

An object specifier record for a property specifies `cProperty` as the object class ID, the Apple event object to which the property belongs as the container, `formPropertyID` as the key form, and a constant such as `pFont` as the key data.

The ***elements*** of a specific Apple event object are the other Apple event objects it contains, excluding those that define its properties. An object specifier record for an element specifies the Apple event object in which the element is located as the container and can specify any key form except `formPropertyID`. Each object class definition in the *Apple Event Registry: Standard Suites* includes a list of ***element classes,*** which are the object classes of the elements that an Apple event object can contain.

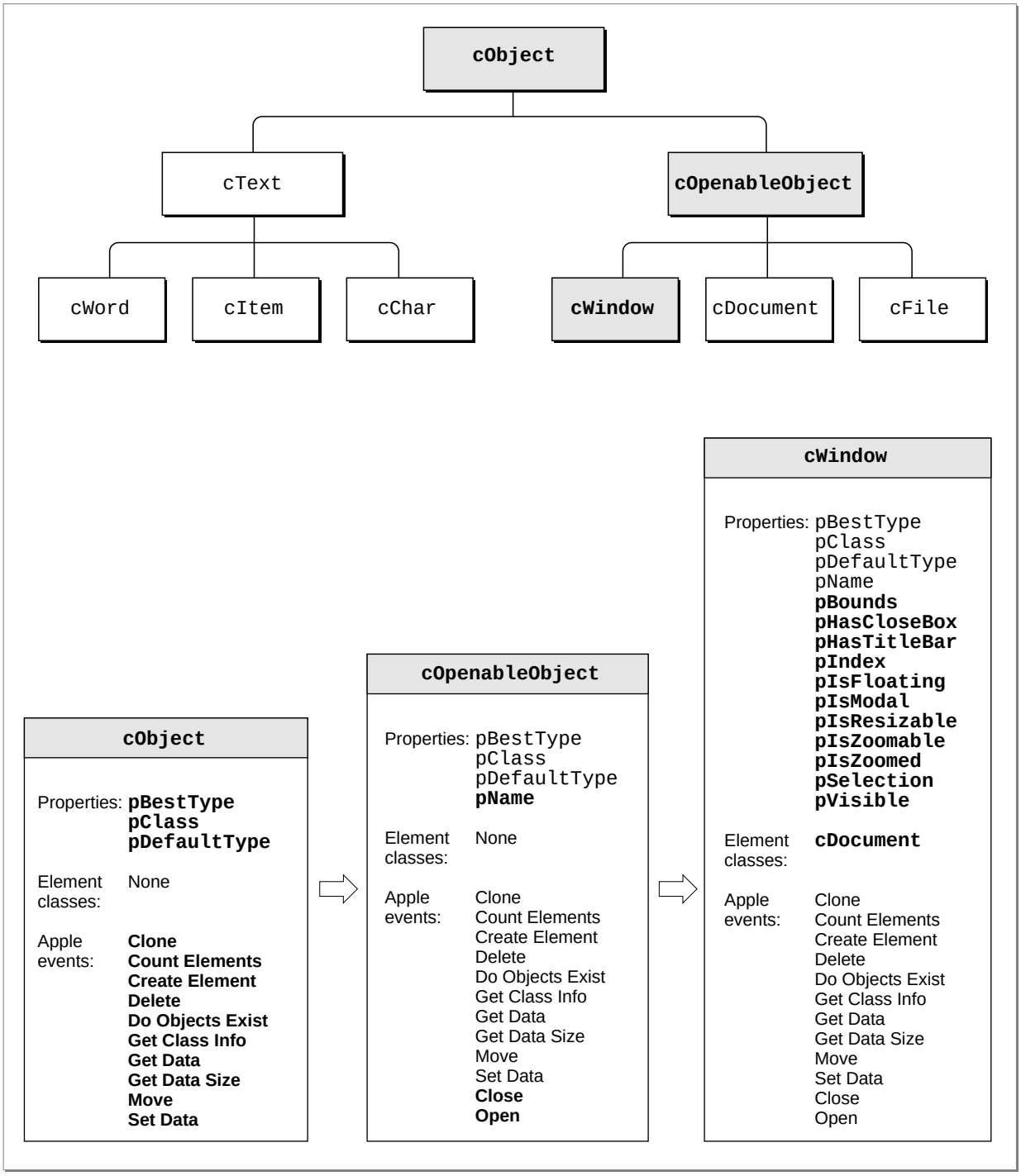
An Apple event object contains exactly one of each of its properties, whereas it can contain no elements or many elements of the same element class. In general, a property of an object describes something about that object; a property can be examined or changed but never deleted. An element can be one or more discrete objects contained in another object and can usually be deleted.

For example, because a paragraph can contain one or more words, one of the element classes listed for the object class `cParagraph` is the object class `cWord`. Individual words can be deleted from a paragraph. However, even though a word in a paragraph can be in a different font from the words around it, a paragraph can have only one `pFont` property. This property is defined as the font of the first character in the paragraph and consists of the name of a font. The paragraph's `pFont` property can be changed but not removed.

The properties and element classes listed for each object class definition in the *Apple Event Registry: Standard Suites* can be inherited from a superclass, or they can originate with a subclass. Figure 3-16 illustrates the object class inheritance hierarchy for the object class `cWindow` in the Core suite. Boldface terms in the figure represent those properties, element classes, or Apple events that are not inherited. The object class `cWindow` includes all the properties and Apple events of its superclass, `cOpenableObject`, which in turn includes all the properties and Apple events of its superclass, `cObject`. The object class `cWindow` also includes 11 properties and one element class that originate with `cWindow` and are not inherited.

The `pClass` property—the property that specifies the four-character class ID—originates with `cObject`. Because the definitions of all object classes are ultimately derived from `cObject`, `pClass` is inherited by all object classes. The definition for `cObject` also lists ten Apple events, which include common events such as Get Data, Move, and Delete Element. Because `cObject` is at the top of the object class inheritance hierarchy, these ten Apple events can use object specifier records that describe Apple event objects of any object class as a direct parameter. Like all abstract superclasses, `cObject` does not correspond to a real Apple event object, so its definition does not list any element classes. Unlike any other object class, `cObject` is at the top of the object class inheritance hierarchy and therefore does not have a superclass.

Figure 3-16 The object class inheritance hierarchy for the object class `cWindow`



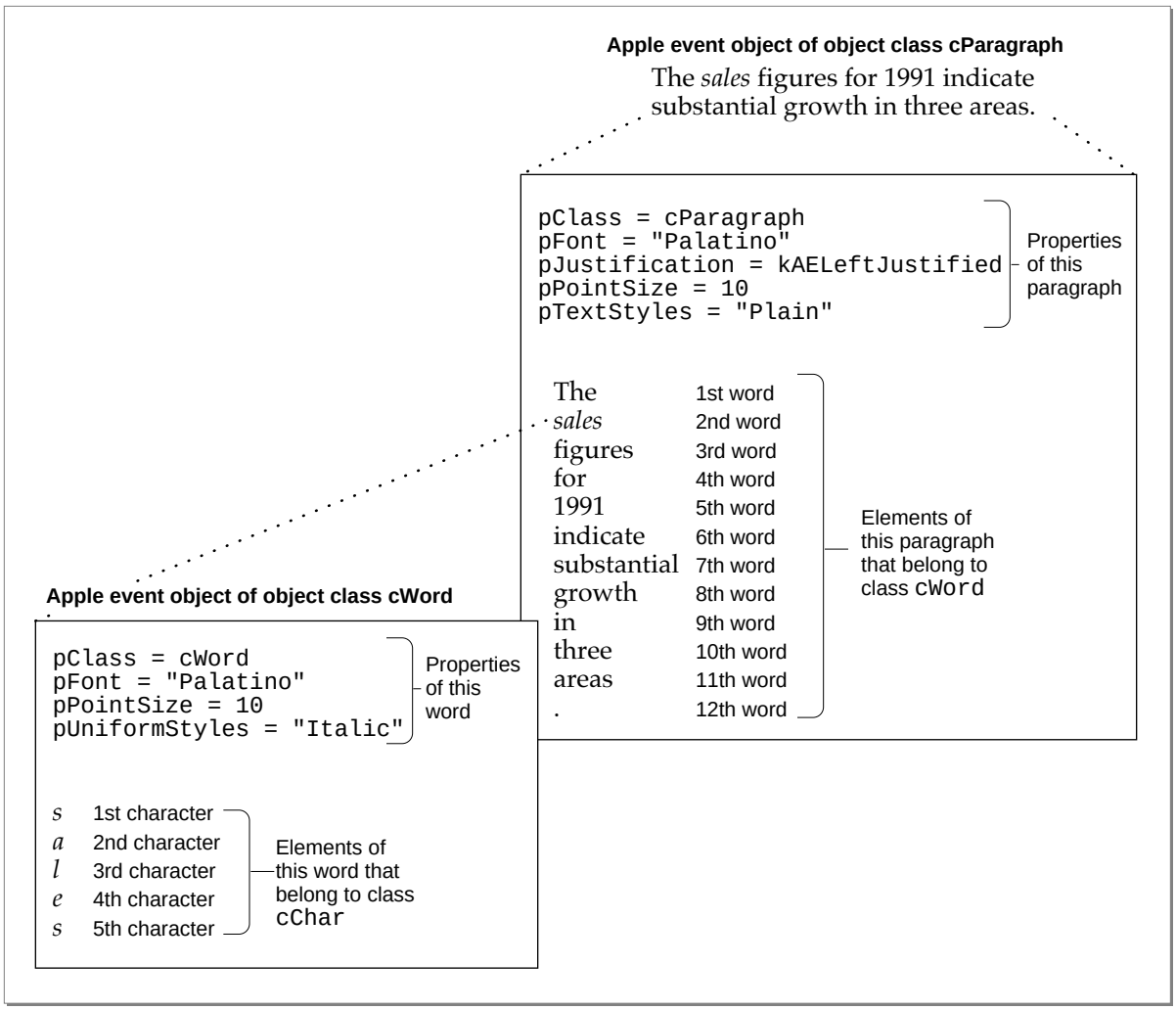
The chain of containers that determine the location of one or more Apple event objects is called the ***container hierarchy.*** The container hierarchy, which specifies the location of real Apple event objects, is different from the object class inheritance hierarchy, which is an abstract concept that determines which properties, element classes, and Apple events an object class inherits from its superclass. For example, the container hierarchy for an Apple event object of class `cWord` can vary from one word to another, because various combinations of other Apple event objects, such as a document, a paragraph, a delimited string, or another word, can contain a word.

Applications that support Apple event objects must be able to identify the order of several elements of the same class that are contained within another Apple event object. For example, each word in a paragraph should have an identifiable order, such as the 5th word or the 12th word. This allows other applications to identify Apple event objects by describing their absolute position within a container.

Figure 3-17 shows an Apple event object of object class `cWord`—the word "Sales"—contained in another Apple event object of object class `cParagraph`. (Both these object classes are defined in the Text suite.) The figure shows only a portion of the container hierarchy for the word, since a complete description of the word would also include the containers that specify the location of the paragraph.

Your application must take account of the definitions in the *Apple Event Registry: Standard Suites* for any object classes you want to support. For example, the definition for the object class `cText` lists paragraphs, lines, words, and characters as Apple event objects that can be contained in Apple event objects of class `cText`. To support Apple events that refer to elements of object class `cText`, your application should associate the `cText` object class with paragraphs, lines, words, and characters in its documents. The list of properties defined for class `cText` includes the properties `pColor`, `pFont`, `pPointSize`, `pScriptTag`, and `pTextStyles`. If you want to support Apple events that distinguish a boldface 12-point word of object class `cText` from an italic 14-point word, for example, your application must associate the point size and style of text in its documents with the properties `pPointSize` and `pTextStyles` defined for class `cText`.

Figure 3-17 An Apple event object of class `cWord` contained in an Apple event object of class `cParagraph`



Finding Apple Event Objects

Most of the Apple events in the Core suite and the functional-area suites defined in the *Apple Event Registry: Standard Suites* can include parameters that consist of object specifier records. Your application's handlers for these events can use the `AEResolve` function to resolve object specifier records: that is, to locate the Apple event objects they describe.

The `AEResolve` function parses an object specifier record and performs related tasks that are the same for all object specifier records. When necessary, the `AEResolve` function calls application-defined functions to perform tasks that are unique to the application, such as locating a specific Apple event object in the application's data structures.

Your application can provide two kinds of application-defined functions for use by `AEResolve`. ***Object accessor functions*** locate Apple event objects. Every application that supports simple object specifier records must provide one or more object accessor functions. ***Object callback functions*** perform other tasks that only an application can perform, such as counting, comparing, or marking Apple event objects.

Each time `AEResolve` calls one of your application's object accessor functions successfully, the object accessor function returns a special descriptor record, called a ***token,*** that identifies either an element in a specified container or a property of a specified Apple event object. The token can be of any descriptor type, including descriptor types you define yourself.

You install object accessor functions by using the `AEInstallObjectAccessor` function. Much like the `AEInstallEventHandler` function, `AEInstallObjectAccessor` uses an ***object accessor dispatch table*** to map requests for Apple event objects to the appropriate object accessor functions in your application. These requests refer to objects of a specified object class in containers identified by a token of a specified descriptor type.

Responding to an Apple event that includes an object specifier record typically involves these steps:

1. After determining that the event is an Apple event, your application calls `AEProcessAppleEvent`.
2. The `AEProcessAppleEvent` function uses the Apple event dispatch table to dispatch the event to the your application's handler for that event.
3. The Apple event handler extracts the Apple event parameters, and passes the object specifier records they contain to `AEResolve`.
4. The `AEResolve` function uses the object accessor dispatch table to call one or more object accessor functions, one at a time, that can identify the nested Apple event objects described by each object specifier record. Each object accessor function returns a token for the object it finds, which in turn helps to determine which object accessor function the `AEResolve` function will use to locate the next Apple event object.
5. The `AEResolve` function returns the final token for the requested object to the application's handler.

The resolution of an object specifier record always begins with the outermost container it specifies. For example, to locate a table named "Summary of Sales" in the document named "Sales Report," the `AEResolve` function first calls an object accessor function that can locate the document, then uses the token returned by that function to identify an object accessor function that can locate the table. It then returns the token for the table to the Apple event handler that called `AEResolve`.

The chapter "Resolving and Creating Object Specifier Records" in this book describes in detail how to resolve object specifier records and how to write and install object accessor and object callback functions.

About the Apple Event Manager

You can use the Apple Event Manager to

- respond to Apple events as a server application
- create and send Apple events as a client application
- resolve and create object specifier records
- support Apple event recording

This section briefly summarizes the steps involved in providing each kind of support and tells where to find the relevant information in this book.

Supporting Apple Events as a Server Application

You do not need to implement all Apple events at once. You can begin by supporting just the required events and, if necessary, the events sent by the Edition Manager. The beginning of the section "Handling Apple Events" on page 4-4 describes how to provide this minimal level of support.

It is relatively easy to respond to the required events and the events sent by the Edition Manager. If, however, your application cannot respond to any other Apple events, other applications will not be able to request services that involve locating specific Apple event objects within your application or its documents, and users will not be able to control your application by executing scripts. To respond to Apple events it is likely to receive from other applications or from scripting components, your application must be able to respond to the appropriate core and functional-area Apple events.

Once you have provided the basic level of support for the Required suite and for events sent by the Edition Manager, you should

- decide which other Apple event suites you want to support
- define the hierarchy of Apple event objects within your application that you want scripting components or other applications to be able to identify—that is, which Apple event objects can be contained by other Apple event objects in your application
- write handlers for the Apple events you support, and install corresponding entries in your application's Apple event dispatch table

To decide which Apple event suites you want to support and how to define the hierarchy of Apple event objects in your application, consult the *Apple Event Registry: Standard Suites* and evaluate which Apple events and Apple event object classes make sense for your application. If necessary, you can extend the definitions of the standard Apple events and Apple events objects to cover special requirements of your application. It is better to extend the standard definitions rather than to define your own custom Apple events, because only those applications that choose to support your custom Apple events explicitly will be able to make use of them.

The chapter "Responding to Apple Events" in this book describes how to write Apple event handlers and related routines. The chapter "Resolving and Creating Object Specifier Records" describes how to resolve object specifiers in an Apple event that describes Apple event objects in your application or its documents.

If your application can respond to Apple events, you can make it scriptable simply by adding an `'aete'` resource. Scripting components use your application's `'aete'` resource to obtain information about the Apple events and corresponding human-language terminology that your application supports. The chapter "Apple Event Terminology Resources" in this book describes how to optimize your implementation of Apple events for use by scripting components and how to create an `'aete'` resource.

Supporting Apple Events as a Client Application

Because users can send Apple events to a variety of applications simply by executing a script, many applications have no need to send Apple events. However, if you want to factor your application for recording, or if you want your application to send Apple events directly to other applications, you can use Apple Event Manager routines to create and send Apple events.

To send an Apple event, you must

- create the Apple event
- add parameters and attributes
- use the `AESend` function to send the event

The chapter "Creating and Sending Apple Events" in this book describes how to perform these tasks.

Supporting Apple Event Objects

If your application responds to core and functional-area Apple events, it must also be able to resolve object specifier records that describe the objects on which those Apple events can act. In addition to the tasks described in the chapter "Responding to Apple Events," you must perform the following tasks to handle Apple events that contain object specifier records:

- Write object accessor functions that can locate the Apple event objects you support, and install corresponding entries in your application's object accessor dispatch table.
- Write any object callback functions that you decide to provide. To handle object specifier records that specify a test, your application must provide at least two object callback functions: one that counts objects and one that compares them.
- Call `AEResolve` from your Apple event handlers whenever an Apple event parameter includes an object specifier record.

The chapter "Resolving and Creating Object Specifier Records" describes how to perform these tasks. It also describes how applications that send Apple events to themselves or directly to other applications can create object specifier records.

Supporting Apple Event Recording

If you make your application scriptable, you may also want to make it recordable. Users of recordable applications can record their actions in the form of Apple events that a scripting component translates into a script. When a user executes a recorded script, the scripting component sends the same Apple events to the application in which they were recorded.

To make your application recordable, you should use Apple events to report user actions to the Apple Event Manager in terms of Apple events. 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 acts as both the client and server application for Apple events it sends to itself in response to user actions. When recording is turned on, the Apple Event Manager sends a copy of every event than an application sends to itself to the scripting component or other process that turned recording on.

The chapter "Introduction to Scripting" in this book provides an overview of how to make your application both scriptable and recordable. The chapter "Recording Apple Events" describes how to factor your application for recording and explains the Apple Event Manager's recording mechanism.