Resolving and Creating Object Specifier Records

This chapter describes how your application can use the Apple Event Manager and application-defined functions to resolve object specifier records. Your application must be able to resolve object specifier records to respond to core and functional-area Apple events defined in the *Apple Event Registry: Standard Suites.*

For example, after receiving a Get Data event that requests a table in a document, your application can use the Apple Event Manager and application-defined functions to parse the object specifier record in the direct parameter, locate the requested table, and send a reply Apple event containing the table's data back to the application that requested it.

This chapter also describes how your application can use the Apple Event Manager to create object specifier records. If you want to factor your application for Apple event recording, or if you want to send Apple events directly to other applications, you need to know how to create object specifier records.

To use this chapter, you should be familiar with the chapters "Introduction to Apple Events" and "Responding to Apple Events" in this book. The section "Working With Object Specifier Records," which begins on page 3-32, provides a general introduction to the subject.

If you plan to create object specifier records, you should also be familiar with the chapter "Creating and Sending Apple Events." If you are factoring your application, you should read the chapter "Recording Apple Events" before you write code for resolving or creating object specifier records.

This chapter begins with an overview of the way your application works with the Apple Event Manager to resolve object specifier records. It then describes

- how the data in an object specifier record is organized
- how to install entries in the object accessor tables
- how to write object accessor and object callback functions
- how to create an object specifier record

IMPORTANT
Versions 1.0 and 1.01 of the Apple Event Manager do not include the routines for resolving and creating object specifier records described in this chapter. To use these routines with those versions of the Apple Event Manager, you must link the Object Support Library (OSL) with your application when you build it, and call the `AEObjectInit` function before calling any of the routines. ▲

Resolving Object Specifier Records

If an Apple event parameter consists of an object specifier record, your handler for the Apple event should ***resolve*** the object specifier record: that is, locate the Apple event objects it describes. The first step is to call the `AEResolve` function with the object specifier record as a parameter.

The `AEResolve` function performs tasks that are required to resolve any object specifier record, such as parsing its contents, keeping track of the results of tests, and handling memory management. When necessary, `AEResolve` 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 or counting the number of Apple event objects in a container.

Note
Object specifier records are only valid while the Apple event that contains them is being handled. For example, if an application receives an Apple event asking it to cut row 5 of a table, what was row 6 then becomes row 5, and the original object specifier record that referred to row 5 no longer refers to the same row. ◆

The `AEResolve` function can call two kinds of application-defined functions. ***Object accessor functions*** locate Apple event objects. ***Object callback functions*** perform other tasks that only an application can perform, such as counting, comparing, or marking Apple event objects. This section provides an overview of the way `AEResolve` calls object accessor and object callback functions when it resolves object specifier records.

Each time `AEResolve` calls one of your application's object accessor functions successfully, the object accessor function should return a special descriptor record created by your application, called a ***token,*** that identifies either an element in a specified container or a property of a specified Apple event object. The Apple Event Manager examines the token's descriptor type but does nothing with the token's data. When it needs to refer to the object the token identifies, the Apple Event Manager simply passes the token back to your application.

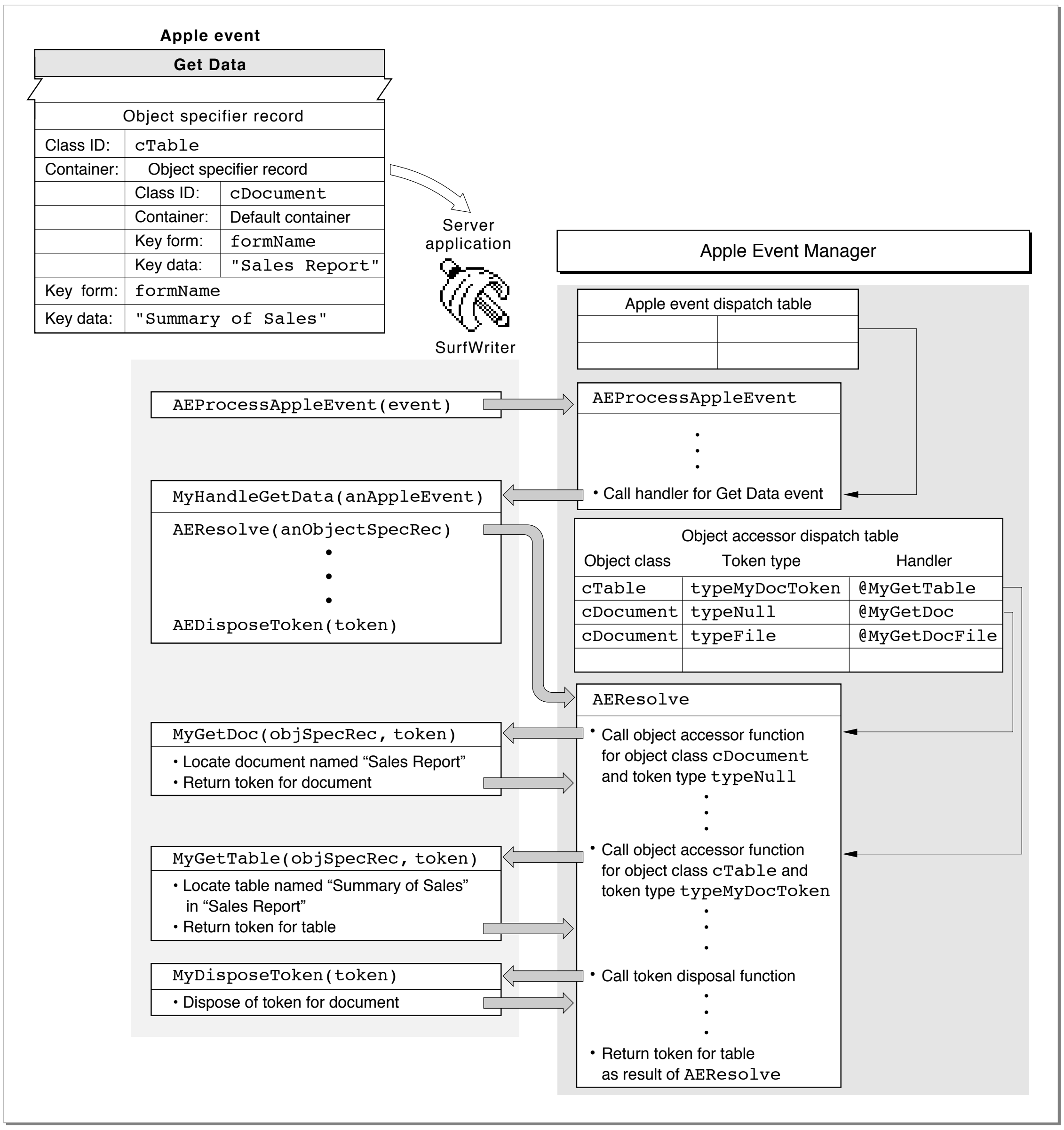
Each object accessor function provided by your application should either find elements of a given object class in a container identified by a token of a given descriptor type, or find properties of an Apple event object identified by a token of a specified descriptor type. The Apple Event Manager uses the object class ID and the descriptor type of the token that identifies the object's container to determine which object accessor function to call.

It is up to you to decide how many object accessor functions you need to write for your application. You can write one object accessor function that locates Apple event objects of several different object classes, or you can write separate object accessor functions for certain object classes. Similarly, you may want to use only one descriptor type for all the tokens returned by your object accessor functions, or you may want to use several descriptor types. The way you define your tokens depends on the needs of your application.

You can use the `AEInstallObjectAccessor` function to create an ***object accessor dispatch table*** that the Apple Event Manager uses to map requests for Apple event objects to the appropriate object accessor function in your application. The Apple Event Manager uses the object class of each requested object and the descriptor type of the token that identifies the object's container to determine which object accessor function to call. Depending on the container hierarchy for a given object specifier record and the way your application defines its object accessor functions, the Apple Event Manager may need to call a series of object accessor functions to resolve the nested object specifier records that describe an Apple event object's container. For information about creating and using the object accessor dispatch table, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

Figure 6-1 illustrates the major steps involved in resolving an object specifier record. The SurfWriter application shown in Figure 6-1 receives a Get Data event whose direct parameter is an object specifier record for a table named "Summary of Sales" in a document named "Sales Report." The SurfWriter application's handler for the Get Data event calls the `AEResolve` function with the object specifier record as a parameter. The `AEResolve` function begins to parse the object specifier record. The first object accessor function that `AEResolve` calls is usually the function that can identify the Apple event object in the application's ***default container***— the outermost container in the container hierarchy. In Figure 6-1, the object specifier record for the document "Sales Report" specifies the default container, so the Apple Event Manager calls the object accessor function in the SurfWriter application that can locate a document in a container identified by a descriptor record of descriptor type `typeNull`.

Figure 6-1 Resolving an object specifier record for a table in a document


Apple event
Get Data
Object specifier record
Class ID: cTable
Container: Object specifier record
Class ID: cDocument
Container: Default container
Key form: formName
Key data: "Sales Report"
Key form: formName
Key data: "Summary of Sales"
Server application
SurfWriter
Apple Event Manager
Apple event dispatch table
AEProcessAppleEvent(event)
AEProcessAppleEvent
• Call handler for Get Data event
MyHandleGetData(anAppleEvent)
AEResolve(anObjectSpecRec)
AEDisposeToken(token)
Object accessor dispatch table
Object class
Token type
Handler
cTable typeMyDocToken @MyGetTable
cDocument typeNull @MyGetDoc
cDocument typeFile @MyGetDocFile
AEResolve
MyGetDoc(objSpecRec, token)
• Locate document named "Sales Report"
• Return token for document
• Call object accessor function for object class cDocument and token type typeNull
MyGetTable(objSpecRec, token)
• Locate table named "Summary of Sales" in "Sales Report"
• Return token for table
• Call object accessor function for object class cTable and token type typeMyDocToken
MyDisposeToken(token)
• Dispose of token for document
• Call token disposal function
• Return token for table as result of AEResolve

After locating the document named "Sales Report," the SurfWriter application returns a token to the Apple Event Manager—that is, a descriptor record that SurfWriter uses to identify the document. The Apple Event Manager examines the descriptor type of the token but does not need to know anything about the token's data to continue parsing the object specifier record. Next, the Apple Event Manager calls the object accessor function that can identify a table in a container identified by a token of descriptor type `typeMyDocToken`. When the Apple Event Manager calls this object accessor function, it uses the token that describes the document to identify the table's container. After the SurfWriter application has located the table named "Summary of Sales" in the document named "Sales Report," it returns a token describing that table to the Apple Event Manager.

After your application has successfully located an Apple event object, the Apple Event Manager disposes of all previous tokens returned during resolution of the object specifier record for the object. The Apple Event Manager disposes of tokens by calling either the `AEDisposeDesc` function or your application's ***token disposal function,*** if you have provided one, which is an object callback function that disposes of a token. In Figure 6-1, the `AEResolve` function calls the SurfWriter application's token disposal function to dispose of the token for the document after `AEResolve` receives the token for the table. After the SurfWriter application has disposed of the token for the document, the `AEResolve` function returns the result of the resolution—that is, the token for the requested table—to the handler in the SurfWriter application that originally called `AEResolve`.

The Apple Event Manager can complete the cycle of parsing the object specifier record and calling the appropriate object accessor function to obtain a token as many times as necessary to identify every container in the container hierarchy and finish resolving an object specifier record, including disposing of the tokens for the containers. However, one token will always be left over—the token that identifies the requested Apple event object. After `AEResolve` returns this final token and your application performs the action requested by the Apple event, it is up to your application to dispose of the token. Your application can do so by calling the `AEDisposeToken` function, which in turn calls either `AEDisposeDesc` or your application's token disposal function.

You need to provide a token disposal function only if a call to `AEDisposeDesc` is not sufficient by itself to dispose of a token or if you provide ***marking callback functions,*** which are three object callback functions that allow your application to use its own marking scheme rather than tokens when identifying large groups of Apple event objects. Your application is not required to provide marking callback functions.

To handle object specifier records that specify a test, your application must provide two object callback functions: (a) an ***object-counting function,*** which counts the number of elements of a given object class in a given container so that the Apple Event Manager can determine how many elements it must test to find the element or elements that meet a specified condition, and (b) an ***object-comparison function,*** which compares one element to another element or to a descriptor record and returns `TRUE` or `FALSE`.

Your application may also provide an ***error callback function*** that can identify which descriptor record caused the resolution of an object specifier record to fail. Your application is not required to provide an error callback function.

If your application resolves object specifier records without the help of the Apple Event Manager, it must extract the equivalent descriptor records and coerce them as necessary to get access to their data. The Apple Event Manager includes coercion handlers for these coercions; for information about this default coercion handling, see Table 4-1 on page 4-43.

For more information about object accessor functions, see "Writing Object Accessor Functions," which begins on page 6-28. For more information about object callback functions, see "Writing Object Callback Functions," which begins on page 6-45.

The next section, "Descriptor Records Used in Object Specifier Records," describes how the data in an object specifier record is interpreted by the Apple Event Manager.

Descriptor Records Used in Object Specifier Records

An object specifier record is a coerced AE record of descriptor type `typeObjectSpecifier`. The data to which its data handle refers consists of four keyword-specified descriptor records:

Keyword	Value	Description of data
`keyAEDesiredClass`	`'want'`	A four-character code for the object class
`keyAEContainer`	`'from'`	An object specifier record (or in some cases a descriptor record with a handle whose value is `NIL`) that identifies the container for the requested objects
`keyAEKeyForm`	`'form'`	A four-character code for the key form
`keyAEKeyData`	`'seld'`	Data or nested descriptor records that specify a property, name, position, range, or test, depending on the key form

This section describes the descriptor types and data associated with each of these keywords. You need this information if your application resolves or creates object specifier records.

For a summary of the descriptor types and key forms discussed in this section, see Table 6-11 on page 6-76. For an overview of object specifier records, see "Working With Object Specifier Records," which begins on page 3-32.

Object Class

The object class of the requested objects is identified by an object class ID. The corresponding keyword-specified descriptor record takes this form:

Keyword	Descriptor type	Data
`keyAEDesiredClass`	`typeType`	Object class ID

The *Apple Event Registry: Standard Suites* defines constants for the standard object class IDs.

Container

The container for the requested objects is usually the object in which they are located. It can be identified in one of four ways:

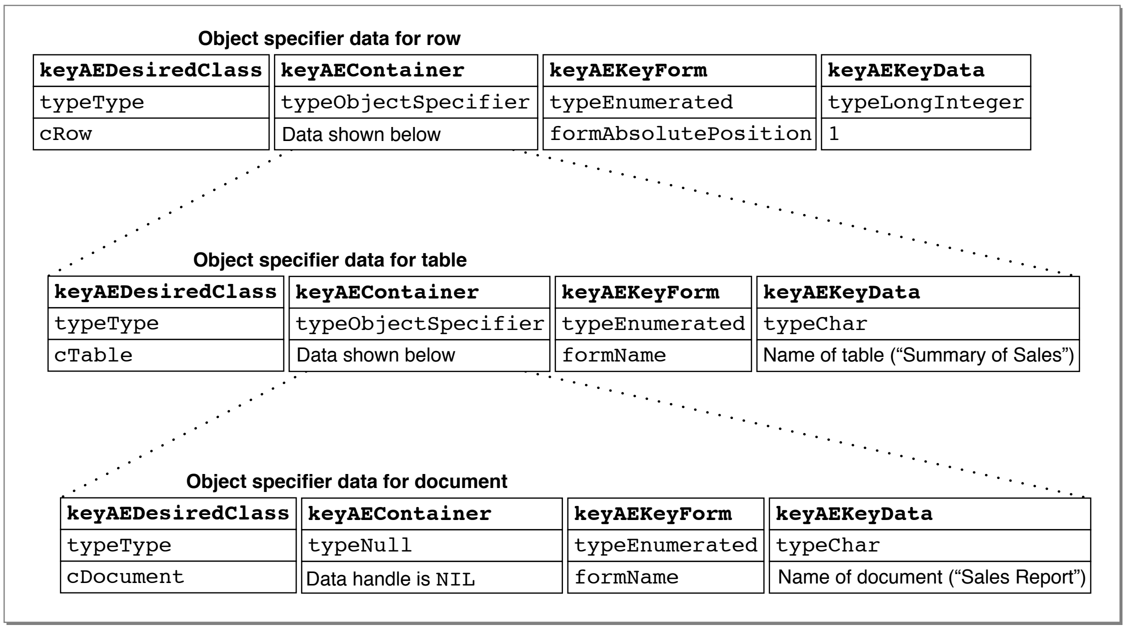
Keyword	Descriptor type	Data
`keyAEContainer`	`typeObjectSpecifier`	Object specifier record.
	`typeNull`	Value of data handle is `NIL`. Specifies the default container at the top of the container hierarchy.
	`typeObjectBeingExamined`	Value of data handle is `NIL`. Specifies the container for elements that are tested one at a time; used only within key data for key form `formTest`.
	`typeCurrentContainer`	Value of data handle is `NIL`. Specifies a container for an element that demarcates one boundary in a range. Used only within key data for key form `formRange`.

The data that describes a container usually consists of another object specifier record. The ability to nest one object specifier record within another in this way makes it possible to identify a chain of containers that fully describes the location of one or more Apple event objects.

For example, Figure 6-2 shows nested object specifier records that specify the first row of a table named “Summary of Sales” in a document named “Sales Report.” The container specified by the object specifier record at the bottom of the figure describes the outermost container in the container hierarchy—the container for the document “Sales Report.”

Because a container must be specified for each Apple event object in a container hierarchy, a ***null descriptor record*** (that is, a descriptor record whose descriptor type is `typeNull` and whose data handle has the value `NIL`) is used to specify the application’s default container—the outermost container for any container hierarchy in that application.

Figure 6-2 Nested object specifier records that specify a container hierarchy



Object specifier data for row

keyAEDesiredClass	**keyAEContainer**	**keyAEKeyForm**	**keyAEKeyData**
typeType	typeObjectSpecifier	typeEnumerated	typeLongInteger
cRow	Data shown below	formAbsolutePosition	1

Object specifier data for table

keyAEDesiredClass	**keyAEContainer**	**keyAEKeyForm**	**keyAEKeyData**
typeType	typeObjectSpecifier	typeEnumerated	typeChar
cTable	Data shown below	formName	Name of table (“Summary of Sales”)

Object specifier data for document

keyAEDesiredClass	**keyAEContainer**	**keyAEKeyForm**	**keyAEKeyData**
typeType	typeNull	typeEnumerated	typeChar
cDocument	Data handle is NIL	formName	Name of document (“Sales Report”)

Note

The format used in Figure 6-2 and similar figures throughout this chapter does not show the structure of the nested object specifier records as they exist within an Apple event. Instead, these figures show what you would obtain after calling `AEGetKeyDesc` repeatedly to extract the object specifier records from an Apple event record.

When you call `AEGetKeyDesc` to extract a null descriptor record, the function returns a descriptor record of type `AEDesc` with a descriptor type of `typeNull` and a data handle whose value is 0. ◆

The object specifier data at the bottom of Figure 6-2 uses a null descriptor record to specify the document's container—that is, the default container for the application. The object specifier record for the document identifies the document named "Sales Report"; the object specifier record for the table identifies the table named "Summary of Sales" in the document "Sales Report"; and the object specifier record for the row identifies the first row of the table named "Summary of Sales" in the document "Sales Report."

An object specifier record in an Apple event parameter almost always includes nested object specifier records that specify the container hierarchy for the requested Apple event object. For the nested object specifier records shown in Figure 6-2, the relationship between each Apple event object and its container is always simple containment: it is located inside its container.

In other cases, the specified container may not actually contain the requested Apple event object. Instead, the relationship between a "container" and a specified object can be defined differently, depending on the key form. For example, the key form `formRelativePosition` indicates that the requested object is before or after its container.

Object specifier records that specify the key form `formTest` or `formRange` require key data that consists of several nested descriptor records, including additional object specifier records that identify either a group of elements to be tested or the boundary elements that demarcate a range. These object specifier records use two special descriptor types to specify containers: `typeObjectBeingExamined` (see page 6-19), which specifies a container that changes as a group of elements are tested one at a time, and `typeCurrentContainer` (see page 6-20), which specifies the container for a boundary element in a range. Both of these descriptor types require a data handle whose value is `NIL`, since they act much like variables whose value is supplied by the Apple Event Manager according to other information provided in the container hierarchy.

Key Form

The key form indicates how the key data should be interpreted. It can be specified by one of eight constants:

Keyword	Descriptor type	Data

`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
		`formName`
		`formUniqueID`
		`formAbsolutePosition`
		`formRelativePosition`
		`formTest`
		`formWhose`
		`formRange`

The next section describes the key data that corresponds to each key form.

Key Data

The nature of the information provided by the key data depends both on the specified key form and on the descriptor type of the descriptor record for the key data. Table 6-1 summarizes these relationships for the standard key forms.

Table 6-1 Standard descriptor types used with `keyAEKeyData`

Key form	Descriptor type	Data
`formPropertyID`	`typeType`	Property ID for an element's property
`formName`	`typeChar` or other text type	Element's name
`formUniqueID`	Any appropriate type	A value that uniquely identifies an object within its container or across an application
`formAbsolutePosition`	`typeLongInteger`	Offset from beginning (positive) or end (negative) of container
	`typeAbsoluteOrdinal`	`kAEFirst` `kAEMiddle` `kAELast` `kAEAny` `kAEAll`
`formRelativePosition`	`typeEnumerated`	`kAENext` `kAEPrevious`
`formTest`	`typeCompDescriptor`	(see Table 6-2 on page 6-16)
	`typeLogicalDescriptor`	(see Table 6-3 on page 6-17)
`formRange`	`typeRangeDescriptor`	(see Table 6-4 on page 6-20)
`formWhose`	`typeWhoseDescriptor`	(see Table 6-5 on page 6-42)

Most applications that resolve object specifier records need to support only the key forms `formPropertyID`, `formName`, `formUniqueID`, `formAbsolutePosition`, `formRelativePosition`, and `formRange` explicitly. You do not need to support these key forms for all object classes; for example, words usually do not have names, so most applications should return `errAEEventNotHandled` if they receive a request for a word by name.

If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the Apple Event Manager can handle `formTest` automatically.

The Apple Event Manager uses the key form `formWhose` internally to optimize resolution of object specifier records that specify `formTest`. Applications that translate tests into their own query languages need to support `formWhose` explicitly. "Handling Whose Tests," which begins on page 6-41, describes `formWhose` in detail.

You can define custom key forms and the format for corresponding data for use by your own application if necessary. If you think you need to do this, check with the Apple Event Registrar first to find out whether existing key forms or others still under development can be adapted to the needs of your application.

One simple kind of key form involves identifying an object on the basis of a specified property. For example, the corresponding data for key form `formUniqueID` (defined in the *Apple Event Registry: Standard Suites*) always consists of a unique ID for the requested object. This ID is stored as a property identified by the constant `pID`. The four-character code that corresponds to both `formUniqueID` and `pID` is `'ID  '`.

If you discover that you do need to define a custom key form based on a property, use the same four-character code for both the key form and the associated property.

The rest of this section describes how the key data for the other key forms shown in Table 6-1 identifies Apple event objects.

Key Data for a Property ID

The key data for `formPropertyID` is specified by a descriptor record of descriptor type `typeType`. The *Apple Event Registry: Standard Suites* defines constants for the standard property IDs.

An object specifier record for a property specifies `cProperty` as the object class ID, an object specifier record for the object that contains the property as the container, `formPropertyID` as the key form, and a constant such as `pFont` as the key data. For example, if you were sending a Set Data event to change the font of a word to Palatino®, you could specify the data for the object specifier record in the direct parameter as follows:

Keyword	Descriptor type	Data
`keyAEDesiredClass`	`typeType`	`cProperty`
`keyAEContainer`	`typeObjectSpecifier`	Object specifier record for word to which property belongs
`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
`keyAEKeyData`	`typeType`	`pFont`

In this example, the Set Data Apple event parameter identified by the keyword `keyAETheData` would specify `Palatino` as the value to which to set the specified property. The reply Apple event for a subsequent Get Data event that included an object specifier record for the same property would return `Palatino` in the parameter identified by the keyword `keyAEResult`.

Key Data for an Object's Name

The key data for `formName` is specified by a descriptor record whose data consists of text, with a descriptor type such as `typeChar` or `typeIntlText`.

Figure 6-2 on page 6-10 includes two object specifier records that specify `formName`.

Key Data for a Unique ID

The key data for `formUniqueID` consists of a value that identifies an object. This ID must be unique either within the container, at a minimum, or unique across the application. A unique ID can be specified by a descriptor record of any appropriate type; for example, type `typeInteger`.

Key Data for Absolute Position

The key data for `formAbsolutePosition` consists of an integer that specifies either an offset or an ordinal position. For descriptor type `typeLongInteger`, the data is either a positive integer, indicating the offset of the requested element from the beginning of the container, or a negative integer, indicating its offset from the end of the container. The first object specifier record shown in Figure 6-2 on page 6-10 specifies `formAbsolutePosition` with key data that consists of the positive integer 1.

For descriptor type `typeAbsoluteOrdinal`, the data consists of one of these constants:

Constant	**Meaning**
`kAEFirst`	The first element in the specified container
`kAEMiddle`	The element in the middle of the specified container
`kAELast`	The last element in the specified container
`kAEAny`	A single element chosen at random from the specified container
`kAEAll`	All the elements in the specified container

If an object specifier record specifies `kAEMiddle` and the number of elements in the container is even, the Apple Event Manager rounds down; for example, the second word would be the "middle" word in a range of four words.

Key Data for Relative Position

The key data for `formRelativePosition` is specified by a descriptor record of type `typeEnumerated` whose data consists of one of these constants:

Constant	Meaning
`kAENext`	The Apple event object after the specified container
`kAEPrevious`	The Apple event object before the specified container

The "container" can be a single Apple event object or a group of Apple event objects; the requested elements are located immediately before or immediately after it, not inside it.

If your application can locate objects of the same class by absolute position, it can easily locate the same objects by relative position. For example, all applications that support `formAbsolutePosition` can easily locate the table immediately after a container specified as another table named "Summary of Sales."

Some applications may also be able to locate an object of one class before or after an object of another class. For example, a word processor might be able to locate the paragraph immediately after a container specified as a table named "Summary of Sales."

Key Data for a Test

The key data for `formTest` is specified by either a comparison descriptor record or a logical descriptor record. If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the Apple Event Manager can handle `formTest` for you. Some applications may perform tests more efficiently by translating them into the application's own query language. For information about handling tests yourself, see "Handling Whose Tests," which begins on page 6-41.

The container for objects that pass a test can be one or more Apple event objects. The objects specified are those in the container that pass the test specified by the key data. For example, an object specifier record can describe "the first row in which the First Name column equals 'John' and the Last Name column equals 'Chapman' in the table 'MyAddresses' of the database 'SurfDB.'" To resolve such an object specifier record, the Apple Event Manager must evaluate a logical expression that applies the logical operator `AND` to two separate comparisons for each row: a comparison of the First Name column to the word "John" and a comparison of the Last Name column to the word "Chapman."

The Apple Event Manager evaluates comparisons and logical expressions on the basis of the information in comparison descriptor records and logical descriptor records. A ***comparison descriptor record*** is a coerced AE record of type `typeCompDescriptor` that specifies an Apple event object and either another Apple event object or data for the Apple Event Manager to compare to the first object. The Apple Event Manager can also use the information in a comparison descriptor record to compare elements in a container, one at a time, either to an Apple event object or to data. The data for a comparison descriptor record consists of three keyword-specified descriptor records with the descriptor types and data shown in Table 6-2.

Table 6-2 Keyword-specified descriptor records for `typeCompDescriptor`

Keyword	Descriptor type	Data

`keyAECompOperator`	`typeType`	`kAEGreaterThan`
		`kAEGreaterThanEquals`
		`kAEEquals`
		`kAELessThan`
		`kAELessThanEquals`
		`kAEBeginsWith`
		`kAEEndsWith`
		`kAEContains`

`keyAEObject1`	`typeObjectSpecifier`	Object specifier data
	`typeObjectBeingExamined`	Value of data handle is `NIL`
`keyAEObject2`	`typeObjectSpecifier`	Object specifier data for object to be compared
	`typeObjectBeingExamined`	Value of data handle is `NIL`
	any other type (`AEDesc`)	Data to be compared

The keyword `keyAEObject1` identifies a descriptor record for the element that is currently being compared to the object or data specified by the descriptor record for the keyword `keyAEObject2`. Either object can be described by a descriptor record of type `typeObjectSpecifier` or `typeObjectBeingExamined`. A descriptor record of `typeObjectBeingExamined` acts as a placeholder for each of the successive elements in a container when the Apple Event Manager tests those elements one at a time. The keyword `keyAEObject2` can also be used with a descriptor record of any other descriptor type whose data is to be compared to each element in a container.

You don't have to support all the available comparison operators for all Apple event objects; for example, the "begins with" operator probably doesn't make sense for objects of type `cRectangle`. It is up to you to decide which comparison operators are appropriate for your application to support, and how to interpret them.

If necessary, you can define your own custom comparison operators. If you think you need to do this, check with the Apple Event Registrar first to find out whether existing definitions of comparison operators or definitions still under development can be adapted to the needs of your application.

A ***logical descriptor record*** is a coerced AE record of type `typeLogicalDescriptor` that specifies a logical expression—that is, an expression that the Apple Event Manager evaluates to either `TRUE` or `FALSE`. The logical expression is constructed from a logical operator (one of the Boolean operators `AND`, `OR`, or `NOT`) and a list of logical terms to which the operator is applied. Each logical term in the list can be either another logical descriptor record or a comparison descriptor record. The Apple Event Manager short-circuits its evaluation of a logical expression as soon as one part of the expression fails a test. For example, if while testing a logical expression such as A `AND` B `AND` C the Apple Event Manager discovers that A `AND` B is not true, it will evaluate the expression to `FALSE` without testing C.

The data for a logical descriptor record consists of two keyword-specified descriptor records with the descriptor types and data shown in Table 6-3.

Table 6-3 Keyword-specified descriptor records for `typeLogicalDescriptor`

Keyword	Descriptor type	Data

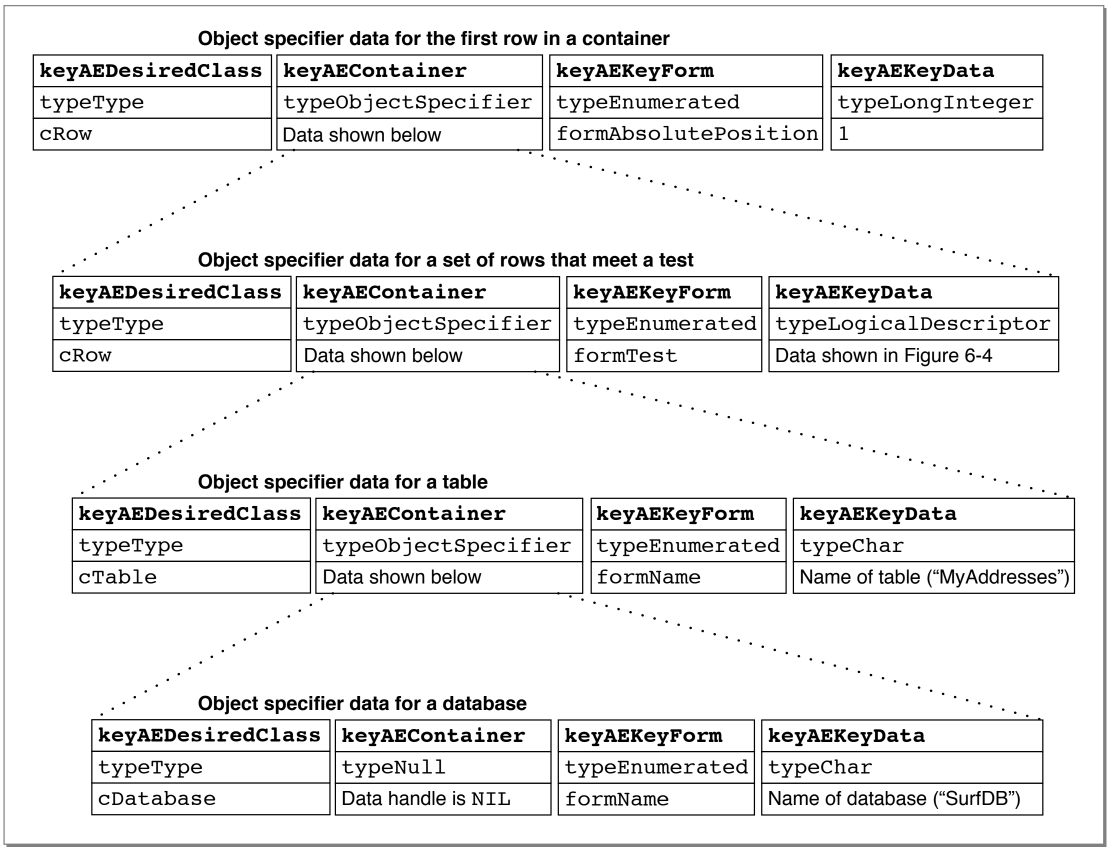
`keyAELogicalOperator`	`typeEnumerated`	`kAEAND`
		`kAEOR`
		`kAENOT`

`keyAELogicalTerms`	`typeAEList`	One or more comparison or logical descriptor records

If the logical operator is `AND` or `OR`, the list can contain any number of logical terms, and the logical operator is applied to all the terms in the list. For example, the logical descriptor data shown in Figure 6-4 on page 6-19 consists of the logical operator `AND` and a list of logical terms that contains two comparison descriptor records. The entire logical descriptor record corresponds to the logical expression "the First Name column equals 'John' `AND` the Last Name column equals 'Chapman.'" If the logical operator is `NOT`, the list must contain a single term.

Figure 6-3 shows four object specifier records that specify the container hierarchy for the first row in the table "MyAddresses" of the database "SurfDB" that meets a test. The object specifier record at the top of Figure 6-3 specifies the first row contained in the set of rows that form its container. The container for the first row is specified by an object specifier record for a set of rows that meet a test. The two object specifier records at the bottom of Figure 6-3 specify the table named "MyAddresses," which contains the rows to be tested, in the database named "SurfDB."

Figure 6-3 The container hierarchy for the first row in a table that meets a test



Object specifier data for the first row in a container

keyAEDesiredClass	keyAEContainer	keyAEKeyForm	keyAEKeyData
typeType	typeObjectSpecifier	typeEnumerated	typeLongInteger
cRow	Data shown below	formAbsolutePosition	1

Object specifier data for a set of rows that meet a test

keyAEDesiredClass	keyAEContainer	keyAEKeyForm	keyAEKeyData
typeType	typeObjectSpecifier	typeEnumerated	typeLogicalDescriptor
cRow	Data shown below	formTest	Data shown in Figure 6-4

Object specifier data for a table

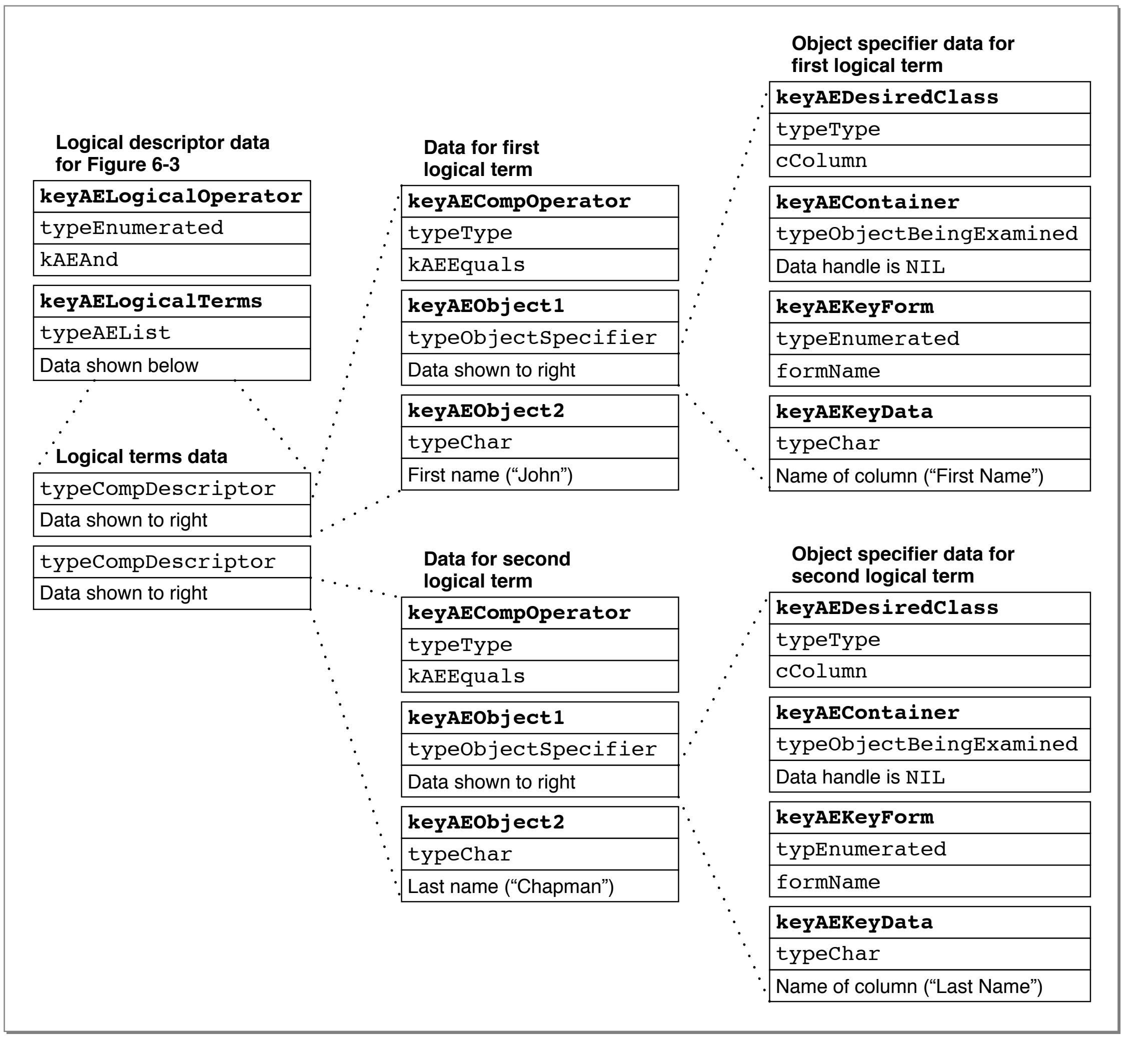
keyAEDesiredClass	keyAEContainer	keyAEKeyForm	keyAEKeyData
typeType	typeObjectSpecifier	typeEnumerated	typeChar
cTable	Data shown below	formName	Name of table (“MyAddresses”)

Object specifier data for a database

keyAEDesiredClass	keyAEContainer	keyAEKeyForm	keyAEKeyData
typeType	typeNull	typeEnumerated	typeChar
cDatabase	Data handle is `NIL`	formName	Name of database (“SurfDB”)

The object specifier record in Figure 6-3 for a set of rows that meet a test specifies `formTest`. The corresponding key data consists of the logical descriptor record shown in Figure 6-4, which applies the logical operator `AND` to two logical terms: a comparison descriptor record that specifies all the rows in the container (the table “MyAddresses”) in which the column named “First Name” equals “John,” and another comparison descriptor record that specifies all the rows in which the column named “Last Name” equals “Chapman.” A row in the table “MyAddresses” passes the test only if both comparison descriptor records evaluate as `TRUE`.

Figure 6-4 A logical descriptor record that specifies a test



The keyword-specified descriptor records with the keyword `keyAEObject1` in Figure 6-4 each consist of an object specifier record that identifies a column by name. The row for each column is specified by a descriptor record of `typeObjectBeingExamined`, which acts as a placeholder for each row as the Apple Event Manager tests successive rows in the table. The Apple event object specified by each of these object specifier records consists of a column in the row. The Apple Event Manager (with the help of an object-comparison function) compares the contents of the column in successive rows to the string identified by the keyword `keyAEObject2` using the comparison operator identified by the keyword `keyAECompOperator`.

Key Data for a Range

The key data for `formRange` is specified by a ***range descriptor record,*** which is a coerced AE record of type `typeRangeDescriptor` that identifies two Apple event objects marking the beginning and end of a range of elements. The data for a range descriptor record consists of two keyword-specified descriptor records with the descriptor types and data shown in Table 6-4.

Table 6-4 Keyword-specified descriptor records in a descriptor record of type `typeRangeDescriptor`

Keyword	Descriptor type	Data
`keyAERangeStart`	`typeObjectSpecifier`	An object specifier record for the first Apple event object in the desired range
`keyAERangeStop`	`typeObjectSpecifier`	An object specifier record for the last Apple event object in the desired range

The elements that identify the beginning and end of the range, which are known as ***boundary objects,*** do not have to belong to the same object class as the elements in the range itself. If the boundary objects belong to the same object class as the elements in the range, the boundary objects are included in the range. For example, the range of tables specified by boundary elements that are also tables would include the two boundary tables.

The container for boundary objects is usually the same as the container for the entire range, in which case the container for a boundary object can be specified by a placeholder—that is, a descriptor record of type `typeCurrentContainer` whose data handle has the value `NIL`.

When `AEResolve` calls an object accessor function to locate a range of objects, the Apple Event Manager replaces the descriptor record of type `typeCurrentContainer` with a token for the container of each boundary object. When using `AEResolve` to resolve the object specifier record, your application doesn't need to examine the contents of this token, because the Apple Event Manager keeps track of it. If your application attempts to resolve some or all of the object specifier record without calling `AEResolve`, the application may need to examine the token before it can locate the boundary objects. The token provided by the Apple Event Manager for a boundary object's container is a descriptor record of type `typeToken` whose data handle refers to a structure of type `ccntTokenRecord`.

```
TYPE ccntTokenRecord =
RECORD
   tokenClass:    DescType;      {class ID of container }
                                 { represented by token}
   token:         AEDesc;        {token for current container}
END;
```

This data type is of interest only if you attempt to resolve an object specifier record for a range without calling `AEResolve`. Otherwise, the Apple Event Manager keeps track of the container.

Installing Entries in the Object Accessor Dispatch Tables

If the direct parameter for an Apple event consists of an object specifier record, your handler for the event should call the `AEResolve` function to resolve the object specifier record: that is, to find the Apple event objects or properties it describes. The `AEResolve` function resolves the object specifier record with the help of object accessor functions provided by your application. Your application installs entries for its object accessor functions in an object accessor dispatch table, which is used by the Apple Event Manager to map requests for Apple event objects or their properties to the appropriate object accessor functions.

After being called by `AEResolve`, an object accessor function should return a token that identifies (in whatever manner is appropriate for your application) the specified Apple event object or property. An object accessor function also returns a result code that indicates whether it found the Apple event object or property. The token, which is a descriptor record of data type `AEDesc`, can be of any descriptor type, including descriptor types you define yourself. For an overview of the way `AEResolve` works with your application's object accessor functions to locate Apple event objects, see "Resolving Object Specifier Records," which begins on page 6-4.

Each object accessor function provided by your application should either find elements of a specified object class contained in an Apple event object identified by a token of a specified descriptor type, or find properties of an Apple event object identified by a token of a specified descriptor type. To determine which object accessor function to dispatch, the Apple Event Manager uses the object class ID specified in an object specifier record and the descriptor type of the token that identifies the requested object's container. For object accessor functions that find properties, you should specify the object class ID as the constant `cProperty`.

To install entries in your application's object accessor dispatch table, use the `AEInstallObjectAccessor` function. For each object class and property your application supports, you should install entries that specify

- the object class of the requested Apple event object or property
- the descriptor type of the token used to identify the container for the requested Apple event object or property
- the address of the object accessor function that finds objects or properties of the specified object class in containers described by tokens of the specified descriptor type
- a reference constant

You provide this information in the first four parameters to the `AEInstallObjectAccessor` function. The fifth parameter allows you to indicate whether the entry should be added to your application's object accessor dispatch table or the system object accessor dispatch table.

The ***system object accessor dispatch table*** is a table in the system heap that contains object accessor functions available to all processes running on the same computer. The object accessor functions in your application's object accessor dispatch table are available only to your application. If `AEResolve` cannot find an object accessor function for the Apple event object class in your application's object accessor dispatch table, it looks in the system object accessor dispatch table. If it doesn't find an object accessor function there either, it returns the result code `errAEAccessorNotFound`.

If `AEResolve` successfully calls the appropriate object accessor function in either the application object accessor dispatch table or the system object accessor dispatch table, the object accessor function returns a token and result code. The `AEResolve` function uses the token and result code to continue resolving the object specifier record. If, however, the token identifies the final Apple event object or property in the container hierarchy, `AEResolve` returns the token for the final resolution in the `theToken` parameter.

If the `AEResolve` function calls an object accessor function in the system object accessor dispatch table, your Apple event handler may not recognize the descriptor type of the token returned by the function. If this happens, your handler should attempt to coerce the token to an appropriate descriptor type. If coercion fails, return the result code `errAEUnknownObjectType`. When your handler returns this result code, the Apple Event Manager attempts to locate a system Apple event handler that can recognize the token.

It is up to you to decide how many object accessor functions you need to write and install for your application. You can install one object accessor function that locates Apple event objects of several different object classes, or you can write separate object accessor functions for certain object classes. Similarly, you may want to use only one descriptor type for all the tokens returned by your object accessor functions, or you may want to use several descriptor types. The sections that follow provide examples of alternative approaches.

For more information about object accessor functions, see "Writing Object Accessor Functions," which begins on page 6-28.

Installing Object Accessor Functions That Find Apple Event Objects

Listing 6-1 demonstrates how to add entries to your application's object accessor dispatch table for the object class `cText` and three of its element classes: the object classes `cWord`, `cItem`, and `cChar`. In this example, the container for each of these object classes is identified by a token that consists of a descriptor record of descriptor type `typeMyText`.

Listing 6-1 Installing object accessor functions that find elements of different classes for container tokens of the same type

```
myErr := AEInstallObjectAccessor(cText, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cWord, typeMyText,
                                 @MyFindWordObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cItem, typeMyText,
                                 @MyFindItemObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cChar, typeMyText,
                                 @MyFindCharObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

The first call to `AEInstallObjectAccessor` in Listing 6-1 adds an entry to the application's object accessor dispatch table. This entry indicates that the `AEResolve` function should call the `MyFindTextObjectAccessor` function when resolving any Apple event object with the `cText` object class and a container identified by a token of descriptor type `typeMyText`. The other calls to `AEInstallObjectAccessor` in Listing 6-1 add entries for Apple event objects of object classes `cWord`, `cItem`, and `cChar` in a container identified by a token of descriptor type `typeMyText`. For example, because all the entries created by the code in Listing 6-1 specify the descriptor type `typeMyText` for the token that identifies the container, the `AEResolve` function calls the `MyFindWordObjectAccessor` function to locate a requested word regardless of whether the container for the word is a run of text, another word, a paragraph, or an item.

The fourth parameter for the `AEInstallObjectAccessor` function specifies a reference constant passed to your handler by the Apple Event Manager each time `AEResolve` calls your object accessor function. Your application can use this reference constant for any purpose. If your application doesn't use the reference constant, you can use 0 as the value, as shown in Listing 6-1.

The last parameter for the `AEInstallObjectAccessor` function is a Boolean value that determines whether the entry is added to the system object accessor dispatch table (`TRUE`) or to your application's object accessor dispatch table (`FALSE`).

If you add an object accessor function to the system object accessor dispatch table, the function that you specify must reside in the system heap. If there was already an entry in the system object accessor dispatch table for the same object class and container descriptor type, that entry is replaced unless you chain it to your system handler. You can do this the same way you chain a previously installed system Apple event handler to your own system handler. See the description of `AEInstallEventHandler` on page 4-62 for details.

▲ **WARNING**

Before an application calls a system object accessor function, system software has set up the A5 register for the calling application. For this reason, if you provide a system object accessor function, it should never use A5 global variables or anything that depends on a particular context; otherwise, the application that calls the system object accessor function may crash. ▲

The code shown in Listing 6-1 installs a separate object accessor function for each object class, even though the code specifies the same descriptor type for tokens that identify the containers for Apple event objects of each class. Most word-processing applications can specify the same object accessor function as well as the same token descriptor type for Apple event objects of these four classes, in which case the code shown in Listing 6-1 can be altered as shown in Listing 6-2.

Listing 6-2 Installing one object accessor function that finds elements of different classes for container tokens of one type

```
myErr := AEInstallObjectAccessor(cText, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cWord, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cItem, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cChar, typeMyText,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

In some situations you may want to write different object accessor functions to locate Apple event objects of the same object class in containers identified by tokens of different descriptor types. For example, the code in Listing 6-3 installs two different object accessor functions: one that finds a word in a container identified by a token of type `typeMyTextToken`, and one that finds a word in a container identified by a token of `typeMyGraphicTextToken`.

Listing 6-3 Installing object accessor functions that find elements of the same class for container tokens of different types

```
myErr := AEInstallObjectAccessor(cWord, typeMyTextToken,
                                 @MyFindTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cWord, typeMyGraphicTextToken,
                                 @MyFindGrphcTextObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

Every application must provide one or more object accessor functions that can find Apple event objects in the default container, which is always identified by a token of descriptor type `typeNull`. Listing 6-4 demonstrates how to add entries to your application's object accessor dispatch table for the object classes `cWindow` and `cDocument`. The container for each of these classes is identified by a token of descriptor type `typeNull`, which specifies an application's default container.

Listing 6-4 Installing object accessor functions that locate elements of different classes in the default container

```
myErr := AEInstallObjectAccessor(cWindow, typeNull,
                                 @MyFindWindowObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
myErr := AEInstallObjectAccessor(cDocument, typeNull,
                                 @MyFindDocumentObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

For any entry in your object accessor dispatch table, you can specify a wildcard value for the object class, for the descriptor type of the token used to identify the container, or for both. You specify a wildcard by supplying the `typeWildCard` constant when installing an entry into the object accessor dispatch table. A wildcard value matches all possible values.

If an object accessor dispatch table contains one entry for a specific object class and a specific token descriptor type, and another entry that is identical except that it specifies a wildcard value for either the object class or the token descriptor type, the Apple Event Manager dispatches the more specific entry. For example, if an object accessor dispatch table includes one entry that specifies the object class as `cWord` and the token descriptor type as `typeMyTextToken`, and another entry that specifies the object class as `cWord` and the token descriptor type as `typeWildCard`, the Apple Event Manager dispatches the object accessor function associated with the entry that specifies `typeMyTextToken`.

If you specify `typeWildCard` as the first parameter and `typeMyToken` as the second parameter for the `AEInstallObjectAccessor` function and no other entry in the dispatch table matches more exactly, the Apple Event Manager calls the object accessor function that you specify in the third parameter when resolving Apple event objects of any object class in containers identified by tokens of the `typeMyToken` descriptor type.

If you specify `cText` as the first parameter and `typeWildCard` as the second parameter for the `AEInstallObjectAccessor` function and no other entry in the dispatch table matches more exactly, the Apple Event Manager calls the object accessor function that you specify in the third parameter when resolving Apple event objects of the object class `cText` in containers identified by tokens of any descriptor type.

If you specify `typeWildCard` for both the first and second parameters of the `AEInstallObjectAccessor` function and no other entry in the dispatch table matches more exactly, the Apple Event Manager calls the object accessor function that you specify in the third parameter when resolving Apple event objects of any object class in containers identified by tokens of any descriptor type.

Once the Apple Event Manager finds a matching entry, whether exact or involving type `typeWildCard`, that is the only object accessor function it calls for that object class and token descriptor type. If that function fails, the Apple Event Manager won't look for another matching entry in the same table.

Installing Object Accessor Functions That Find Properties

The Apple event object to which a property belongs is that property's container. You should add entries to your application's object accessor dispatch table that specify object accessor functions for finding properties in containers identified by tokens of various descriptor types. Object specifier records do not specify a property's specific object class; instead, they specify the constant `cProperty` as the class ID for any property. Similarly, you should specify the constant `cProperty` as the object class for an object accessor function that can find any property of a container identified by a token of a given descriptor type. If you need to install different object accessor routines for finding properties of Apple event objects that belong to different object classes, you must use different descriptor types for the tokens that represent those Apple event objects.

For example, to specify an object accessor function that locates properties of Apple event objects identified by tokens of descriptor type `typeMyToken`, you can add a single entry to the object accessor dispatch table:

```
myErr := AEInstallObjectAccessor(cProperty, typeMyToken,
                                 @MyFindPropertyObjectAccessor,
                                 0, FALSE);
IF myErr <> noErr THEN DoError(myErr);
```

The code in this example adds an object accessor function to the application's object accessor dispatch table that can find any property of any container identified by a token of descriptor type `typeMyToken`. If the second parameter were specified as `typeWildCard`, the `MyFindPropertyObjectAccessor` function would have to be capable of finding any property of any Apple event object in your application except for those found by handlers with more specific entries in the object accessor dispatch table.

Writing Object Accessor Functions

If the direct parameter for an Apple event consists of an object specifier record, your handler for the event should call the `AEResolve` function to resolve the object specifier record: that is, to find the Apple event objects or properties it describe. The `AEResolve` function resolves object specifier records with the help of object accessor functions provided by your application. For an overview of the way `AEResolve` works with your application's object accessor functions to locate Apple event objects, see "Resolving Object Specifier Records," which begins on page 6-4.

This section describes how to write object accessor functions. You need to read this section if your application supports the Core suite or any of the functional-area suites in the *Apple Event Registry: Standard Suites.*

Your application should provide object accessor functions that can find Apple event objects and their properties for all object classes supported by your application, including their corresponding properties and element classes. Because the Apple Event Manager dispatches object accessor functions according to the class ID of the requested Apple event object and the descriptor type of the token that identifies its container, you have a great deal of flexibility in deciding what object accessor functions you need to write for your application. The installation and dispatching of object accessor functions are described in "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

For example, if your application is a word processor, one object accessor function will probably work equally well for Apple event objects of object classes `cParagraph`, `cItem`, and `cWord` located in containers identified by tokens of descriptor type `myTextToken`. If you use a single descriptor type for tokens that identify any containers in which objects of these three object classes can be found, you can dispatch requests for all such elements to the same object accessor function. However, the same word processor might use one descriptor type for tokens identifying containers of class `cCell` and another descriptor type for tokens identifying containers of class `cColumn`—in which case it would need an object accessor function for each descriptor type.

For each object class that your application supports, your application should also provide one object accessor function that can find all the properties of that object class, or one object accessor function that can find all the properties of several object classes.

Here's the declaration for a sample object accessor function:

```
FUNCTION MyObjectAccessor (desiredClass: DescType;
                           containerToken: AEDesc;
                           containerClass: DescType;
                           keyForm: DescType; keyData: AEDesc;
                           VAR theToken: AEDesc;
                           theRefCon: LongInt): OSErr;
```

The `AEResolve` function passes the following information to your object accessor function: the object class ID of the requested Apple event objects, the object class of their container, a token that identifies the specific container in which to look for them, the key form and key data that specify how to locate them, and the reference constant associated with the object accessor function. Your object accessor function uses this information to locate the requested objects.

Most applications that resolve object specifier records need to support only the key forms `formPropertyID`, `formName`, `formUniqueID`, `formAbsolutePosition`, `formRelativePosition`, and `formRange` explicitly. You do not need to support these key forms for all object classes; for example, words usually do not have names, so most applications should return `errAEEventNotHandled` if they receive a request for a word by name.

If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the Apple Event Manager can handle `formTest` automatically.

The Apple Event Manager uses the key form `formWhose` internally to optimize resolution of object specifier records that specify `formTest`. Only applications that translate tests into their own query languages need to support `formWhose` explicitly. "Handling Whose Tests," which begins on page 6-41, describes `formWhose` in detail.

If your object accessor function successfully locates the requested Apple event objects, your application should return the `noErr` result code and a token that identifies them. The token can be of any descriptor type, as long as it is a descriptor record. For example, to identify a file, your application might use a descriptor record of descriptor type `typeAlias` or `typeFSS`. To identify an open document, your application might define its own descriptor type, such as `typeMyDocToken`, for a descriptor record whose data handle refers to a pointer to a document record. For more information about tokens, see "Defining Tokens" on page 6-39.

IMPORTANT

Object accessor functions must not have side effects that change the number or order of elements in a container while an object specifier record is being resolved. If the number of elements in a container is changed during the resolution of an object specifier record, the Apple Event Manager may not be able to locate all the elements. ▲

Writing Object Accessor Functions That Find Apple Event Objects

The first three listings in this section demonstrate how to write three object accessor functions that might be called in the following situation: An application receives a Get Data event with a direct parameter that consists of an object specifier record for the first word in the third paragraph of a document. The application's handler for the Get Data event calls the `AEResolve` function to resolve the object specifier record. The `AEResolve` function first calls the application's object accessor function for objects of class `cDocument` in containers identified by a token of descriptor type `typeNull`.

The `AEResolve` function passes these values to the `MyFindDocumentObjectAccessor` function shown in Listing 6-5: in the `desiredClass` parameter, the constant `cDocument`; in the `containerToken` parameter, a descriptor record of descriptor type `typeNull` with a data handle whose value is `NIL`; in the `containerClass` parameter, the constant `typeNull`; in the `keyForm` parameter, the constant `formName`; in the `keyData` parameter, a descriptor record of descriptor type `typeText` whose data consists of the string `"MyDoc"`; and the reference constant specified in the application's object accessor dispatch table.

Listing 6-5 An object accessor function that locates Apple event objects of object class `cDocument`

```
FUNCTION MyFindDocumentObjectAccessor
                              (desiredClass: DescType;
                               containerToken: AEDesc;
                               containerClass: DescType;
                               keyForm: DescType; keyData: AEDesc;
                               VAR token: AEDesc;
                               theRefCon: LongInt): OSErr;
VAR
   docName:          Str255;
   actSize:          Size;
   foundDoc:         Boolean;
   foundDocRecPtr:   MyDocumentRecordPtr;
BEGIN
   IF keyform = formName THEN
   BEGIN
      {get the name of the document from the key data}
      MyGetStringFromDesc(keyData, docName, actSize);
      {look for a document with the given name by }
      { searching all document records}
      MySearchDocRecs(docName, foundDocRecPtr, foundDoc);
      IF NOT foundDoc THEN
         MyFindDocumentObjectAccessor := kObjectNotFound
      ELSE {create token that identifies the document}
         MyFindDocumentObjectAccessor :=
               AECreateDesc(typeMyDocToken, @foundDocRecPtr,
                              SizeOf(foundDocRecPtr), token);
   END
   {handle the other key forms you support}
   ELSE
      MyFindDocumentObjectAccessor := kKeyFormNotSupported;
END;
```

The `MyFindDocumentObjectAccessor` function uses the information in the `keyForm` and `keyData` parameters to find the specified document. If it finds the Apple event object, `MyFindDocumentObjectAccessor` returns a token of descriptor type `typeMyDocToken` to `AEResolve`. The data handle for this token refers to a pointer to a document record (see Figure 6-5 on page 6-39). The `MyFindDocumentObjectAccessor` function returns this token and the `noErr` result code to the `AEResolve` function.

In the Get Data example, the token returned to `AEResolve` by the `MyFindDocumentObjectAccessor` function identifies the document "MyDoc." The `AEResolve` function then calls the application's object accessor function for objects of class `cParagraph` in containers identified by a token of descriptor type `typeMyDocToken`.

In this case, `AEResolve` passes these values to the `MyFindParaObjectAccessor` function shown in Listing 6-6: in the `desiredClass` parameter, the constant `cParagraph`; in the `containerToken` parameter, the token returned by the `MyFindDocumentObjectAccessor` function; in the `containerClass` parameter, the constant `cDocument`; in the `keyForm` parameter, the constant `formAbsolutePosition`; in the `keyData` parameter, a descriptor record with the `typeLongInteger` descriptor type and data that consists of the value 3 (indicating the third paragraph); and the reference constant specified in the application's object accessor dispatch table.

Listing 6-6 An object accessor function that locates Apple event objects of object class `cParagraph`

```
FUNCTION MyFindParaObjectAccessor (desiredClass: DescType;
                                    containerToken: AEDesc;
                                    containerClass: DescType;
                                    keyForm: DescType;
                                    keyData: AEDesc;
                                    VAR token: AEDesc;
                                    theRefCon: LongInt): OSErr;
VAR
   index:            LongInt;
   {MyFoundTextRecord is an application-defined data type }
   { consisting of three fields: start, ending, and docPtr}
   foundParaRec:     MyFoundTextRecord;
   foundParaStart:   LongInt;
   foundParaEnd:     LongInt;
   foundDocRecPtr:   MyDocumentRecordPtr;
   success:          Boolean;
BEGIN
   IF keyForm = formAbsolutePosition THEN
   BEGIN
      {get the index of the paragraph from the key data}
      MyGetIndexFromDesc(keyData, index);
      {get the desired paragraph by index}
      success := MyGetPara(index, containerToken, foundParaStart,
                           foundParaEnd, foundDocRecPtr);
      IF NOT success THEN
         MyFindParaObjectAccessor := kObjectNotFound
      ELSE {create token that identifies the paragraph}
      BEGIN
         foundParaRec.start := foundParaStart;
         foundParaRec.ending := foundParaEnd;
         foundParaRec.docPtr := foundDocRecPtr;
         MyFindParaObjectAccessor :=
               AECreateDesc(typeMyTextToken, @foundParaRec,
                              SizeOf(foundParaRec), token);
      END;
   END
      {handle the other key forms you support}
   ELSE
      MyFindParaObjectAccessor := kKeyFormNotSupported;
END;
```

The MyFindParaObjectAccessor function uses another application-defined function, MyGetPara, to search the data structures associated with the document and find the desired paragraph. If it finds the paragraph, MyGetPara returns a value that identifies the beginning of the paragraph, a value that identifies the end of the paragraph, and a pointer to the document (which MyGetPara gets from the containerToken parameter). The MyFindParaObjectAccessor function returns an application-defined token that contains this information. This token is of descriptor type typeMyTextToken; it describes a range of characters that can be used to identify any range of text, including a paragraph or a word. The MyFindParaObjectAccessor function returns this token and the noErr result code to the AEResolve function.

In the Get Data example, the token returned to AEResolve by the MyFindParaObjectAccessor function identifies the third paragraph in the document "MyDoc." The AEResolve function then calls the application's object accessor function for objects of class cWord in containers identified by a token of descriptor type typeMyTextToken.

In this case, the AEResolve function passes these values to the MyFindWordObjectAccessor function shown in Listing 6-7: in the desiredClass parameter, the constant cWord; in the containerToken parameter, the token returned by the MyFindParaObjectAccessor function (a token of descriptor type typeMyTextToken that identifies a paragraph); in the containerClass parameter, the constant cParagraph; in the keyForm parameter, the constant formAbsolutePosition; in the keyData parameter, a descriptor record with the typeLongInteger descriptor type and data that consists of the value 1 (indicating the first word); and the reference constant specified in the application's object accessor dispatch table.

The MyFindWordObjectAccessor function uses another application-defined function, MyGetWord, to search the paragraph to find the desired word. If it finds the word, MyGetWord returns a value that identifies the beginning of the word, a value that identifies the end of the word, and a pointer to the document (which MyGetWord gets from the containerToken parameter). The MyFindWordObjectAccessor function returns a token that contains this information. This token is also of descriptor type typeMyTextToken; in this case, the token identifies a specific word. The MyFindWordObjectAccessor function returns this token and the noErr result code to the AEResolve function, which in turn returns the token to the Get Data event handler that originally called AEResolve.

Listing 6-7 An object accessor function that locates Apple event objects of object class `cWord`

```
FUNCTION MyFindWordObjectAccessor
                              (desiredClass: DescType;
                               containerToken: AEDesc;
                               containerClass: DescType;
                               keyForm: DescType; keyData: AEDesc;
                               VAR token: AEDesc;
                               theRefCon: LongInt): OSErr;
VAR
   index:             LongInt;
   foundWordRec:      MyFoundTextRecord;
   foundWordStart:    LongInt;
   foundWordEnd:      LongInt;
   foundDocRecPtr:    MyDocumentRecPtr;
   success:           Boolean;
BEGIN
   IF keyForm = formAbsolutePosition THEN
   BEGIN
      {get the index of the word from the key data}
      MyGetIndexFromDesc(keyData, index);
      {get the desired word by index}
      success := MyGetWord(index, containerToken, foundWordStart,
                           foundWordEnd, foundDocRecPtr);
      IF NOT success THEN
         MyFindWordObjectAccessor := kObjectNotFound
      ELSE {create token that identifies the paragraph}
      BEGIN
         foundWordRec.start := foundWordStart;
         foundWordRec.ending := foundWordEnd;
         foundWordRec.docPtr := foundDocRecPtr;
         MyFindWordObjectAccessor :=
               AECreateDesc(typeMyTextToken, @foundWordRec,
                              SizeOf(foundWordRec), token);
      END;
   END
   {handle the other key forms you support}
   ELSE
      MyFindWordObjectAccessor := kKeyFormNotSupported;
END;
```

Listing 6-5 on page 6-30 shows an object accessor function that locates a document in the default container. Every application must provide one or more object accessor functions that can find Apple event objects in the default container, which is always identified by a descriptor record of descriptor type `typeNull`. Listing 6-8 provides another example of an object accessor function that locates an Apple event object in the default container. If the `MyFindWindowObjectAccessor` function shown in Listing 6-8 were installed in an application's object accessor dispatch table, the `AEResolve` function would call it as necessary to locate an object of class `cWindow` in a container identified by a token of descriptor type `typeNull`.

Listing 6-8 An object accessor function that locates Apple event objects of object class `cWindow`

```
FUNCTION MyFindWindowObjectAccessor (desiredClass: DescType;
                                     containerToken: AEDesc;
                                     containerClass: DescType;
                                     keyForm: DescType;
                                     keyData: AEDesc;
                                     VAR token: AEDesc;
                                     theRefCon: LongInt): OSErr;
VAR
   windowName:    Str255;
   actSize:       Size;
   windTitle:     Str255;
   window:        WindowPtr;
   index, iLoop:  Integer;
   found:         Boolean;
BEGIN
   IF keyForm = formName THEN
   BEGIN
      {get the name of the window to find from the keyData }
      { parameter. MyGetStringFromDesc gets data out of an }
      { AEDesc and returns a string and the string's size}
      MyGetStringFromDesc(keyData, windowName, actSize);
      {look for a window with the given name}
      window := FrontWindow;
      found := FALSE;
      WHILE ((window <> NIL) AND (found = FALSE)) DO
      BEGIN
         GetWTitle(window, windTitle);
         found := EqualString(windTitle, windowName, FALSE, TRUE);
         IF NOT found THEN
            window := WindowPtr(WindowPeek(window)^.nextWindow);
      END;     {of while}
   END {of formName}
   ELSE
   IF keyForm = formAbsolutePosition THEN
      {find the window given an index in key data}
      BEGIN    {get the index from the key data}
         MyGetIndexFromDesc(keyData, index);
         found := FALSE;
         iLoop := 0;
         window := FrontWindow;
         WHILE (window <> NIL) AND (found <> TRUE) DO
         BEGIN
            iLoop := iLoop +1;
            IF iLoop = index THEN
               found := TRUE
            ELSE
            window := WindowPtr(WindowPeek(window)^.nextWindow);
         END; {of while}
      END {of formAbsolutePosition}
   {handle the other key forms you support}
   ELSE
   BEGIN
      MyFindWindowObjectAccessor := kKeyFormNotSupported;
      Exit(MyFindWindowObjectAccessor);
   END;
   IF window = NIL THEN
      MyFindWindowObjectAccessor := kObjectNotFound
   ELSE {create token that identifies the window}
      MyFindWindowObjectAccessor :=
               AECreateDesc(typeMyWindow, @window,
                               SizeOf(window), token);
END;
```

The `keyForm` parameter of the `MyFindWindowObjectAccessor` function describes how the function should interpret the `keyData` parameter. If the key form is `formName`, then the key data contains the name of the window to locate. If the key form is `formAbsolutePosition`, the key data contains the position of the window to locate in the window list; for example, a value of 1 identifies the frontmost window.

The `MyFindWindowObjectAccessor` function supports only the `formName` and `formAbsolutePosition` key forms. Your object accessor functions should support all key forms that make sense for the kinds of objects the functions can locate.

For the `formName` keyword, the `MyFindWindowObjectAccessor` function starts with the frontmost window and compares the window's title to the name specified by the `keyData` parameter. It continues this search until it reaches either the end of the window list or finds a match. If the `MyFindWindowObjectAccessor` function finds a match, it uses the `AECreateDesc` function to create a descriptor record for the token, specifying the application-defined `typeMyWindow` descriptor type and the data for this descriptor type as a window pointer.

The `MyFindWindowObjectAccessor` function then sets its function result appropriately, and the `AEResolve` function either returns this function result and token, or uses the returned token to request the next Apple event object in the container hierarchy, such as a document in the window.

Writing Object Accessor Functions That Find Properties

The Apple event object to which a property belongs is that property's container. Your application should provide an object accessor function for finding properties in containers identified by tokens of various descriptor types. Your application does not need to be given a property's specific object class in order to find that property; instead, you can specify the object class ID for any property with the constant `cProperty`. Thus, you can write a single object accessor function that can find any property of an object identified by a token of a given descriptor type.

To install such an object accessor function, you can add a single entry to the object accessor dispatch table that specifies the desired object class as `cProperty` for a given token descriptor type. For example, Listing 6-9 shows an object accessor function that identifies any property of a window.

Listing 6-9 An object accessor function that identifies any property of a window

```
FUNCTION MyFindPropertyOfWindowObjectAccessor
                              (desiredClass: DescType;
                              containerToken: AEDesc;
                              containerClass: DescType;
                              keyForm: DescType; keyData: AEDesc;
                              VAR token: AEDesc;
                              theRefCon: LongInt): OSErr;
VAR
   theProperty: DescType;
BEGIN
   MyFindPropertyOfWindowObjectAccessor := noErr;
   MyGetPropFromKeyData(keyData, theProperty);
   IF keyForm = formPropertyID THEN
   BEGIN
      IF theProperty = pName THEN
         {create token that identifies name property of the }
         { window}
         MyCreateToken(typeMyWindowProp, containerToken, pName,
                       token)
      ELSE
      IF theProperty = pBounds THEN
         {create token that identifies bounds property of the }
         { window}
         MyCreateToken(typeMyWindowProp, containerToken, pBounds,
                       token)

      {create tokens for other properties as appropriate}

      ELSE
         MyFindPropertyOfWindowObjectAccessor :=
                                          kErrorPropNotFound;
   END
   ELSE
      MyFindPropertyOfWindowObjectAccessor :=
                                          kKeyFormNotSupported;
END;
```

The `MyFindPropertyOfWindowObjectAccessor` function takes a token that identifies a window and creates a token that identifies the requested property of that window. See Figure 6-6 on page 6-40 for an illustration of the logical organization of a token of descriptor type `typeMyWindowProp`.

This simplified example merely translates information about the requested property and the window to which it belongs into the form of a token of type `typeMyWindowProp`. This token can then be used by Apple event handlers to identify the corresponding window and its property, so that a handler can either retrieve the value of the property (for example, a Get Data handler) or change the value of the property (for example, a Set Data handler). Like other tokens, a token that identifies a property should always contain a reference to the corresponding property and the object to which it belongs—not a copy of the data for that object's property.

Defining Tokens

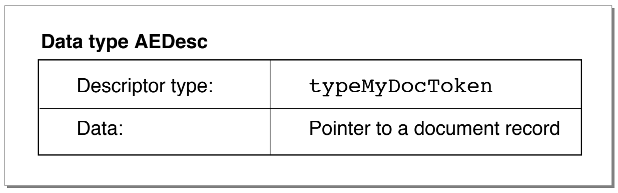
It is up to you to decide how many token descriptor types you need to define for your application. In many cases you may be able to define one token that can identify Apple event objects of several different object classes, such as a token of type `typeMyTextToken` that identifies Apple event objects of object classes `cText`, `cWord`, `cItem`, and `cChar`. In other cases you may need to define specific token descriptor types for specific object classes.

For example, the `MyFindDocumentObjectAccessor` routine shown in Listing 6-5 on page 6-30 returns a token of descriptor type `typeMyDocToken`, which identifies a document record.

```
CONST                                {application-defined token}
   typeMyDocToken        = 'docr'; {identifies a document record}
```

Figure 6-5 shows the logical arrangement of a descriptor record of descriptor type `typeMyDocToken` whose data is specified by a pointer to a document record.

Figure 6-5 Descriptor record for an application-defined token that identifies a document



Data type AEDesc

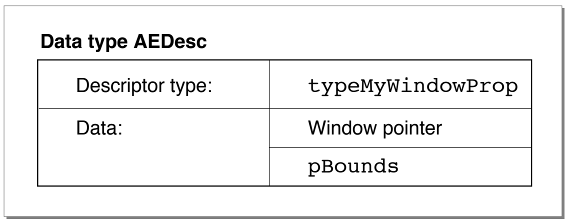
Descriptor type:	`typeMyDocToken`
Data:	Pointer to a document record

The `MyFindPropertyOfWindowObjectAccessor` routine shown in Listing 6-9 returns a token of descriptor type `typeMyWindowProp` for every property that it can locate.

```
CONST                                 {application-defined token}
   typeMyWindowProp     = 'wprp'; {a window pointer and a }
                                      { property ID}
```

Figure 6-6 shows the logical arrangement of a descriptor record of descriptor type `typeMyWindowProp` that identifies the bounds property of a window. Its data consists of a window pointer and the constant `pBounds`. The application can use this token either to return or to change the window's bounds setting, depending on the Apple event that specified the property. If the token specified `pName` instead, the application could use it either to return the window's name as a string or to change the window's name.

Figure 6-6 Descriptor record for an application-defined token that identifies the `pbounds` property of a window


Data type AEDesc

Descriptor type:	`typeMyWindowProp`
Data:	Window pointer
	`pBounds`

A token's data should always contain a reference to the corresponding Apple event objects—not a copy of the data for those objects. This allows the same token to be used for both reading and writing tokens.

It's often possible to use the same token type for objects of several object classes, or for both an object of a given class and one of its properties. A token's data is private to your application and can be organized in any way that is convenient.

When an object accessor function that supports key form `formRange` locates a range of Apple event objects, it should normally return a descriptor list (`AEDescList`) of tokens for the individual objects. A typical exception is an object accessor function that returns a range of objects of class `cText`, which should return a single token representing the entire range. For example, an object accessor function that finds "all the characters from char 1 to char 1024" should return a token that consists of a list of 1024 objects, each of class `cChar`, whereas an object specifier function that finds "all the text from char 1 to char 1024" should return a single token for a single item of class `cText` that is 1024 characters long.

A token is valid only until the Apple Event Manager has located the requested element in the container the token represents and returned another token for the element. The Apple Event Manager disposes of intermediate tokens after it finishes resolving an object specifier record, but one token is always left over—the token that identifies the specified Apple event object or objects. Your application should dispose of this final token by calling the `AEDisposeToken` function, which in turn calls your application's token disposal function (if one exists), an optional object callback function that disposes of a token. See page 6-99 for the declaration of a token disposal function.

If your application does not provide a token disposal function, the Apple Event Manager uses the `AEDisposeDesc` function to dispose of tokens. This function does the job as long as disposing of tokens involves nothing more than simply disposing of a descriptor record. Otherwise, you need to provide a custom token disposal function. For example, suppose the data field of a token descriptor record contains a handle to a block that in turn contains references to memory for the Apple event object referred to by the token. In this case, the application should provide a token disposal function that performs the tasks required to dispose of the token and any associated structures.

Handling Whose Tests

If your application provides an object-counting function and an object-comparison function in addition to the appropriate object accessor functions, the Apple Event Manager can resolve object specifier records that specify `formTest` without any other assistance from your application. The Apple Event Manager translates object specifier records of key form `formTest` into object specifier records of key form `formWhose`. This involves collapsing the key form and key data from two object specifier records in a container hierarchy into one object specifier record with the key form `formWhose`.

Some applications may find it more efficient to translate whose tests into their own query languages rather than letting the Apple Event Manager handle the tests. This is useful only for applications that can make use of a test combined with either an absolute position or a range to locate objects. If you want the Apple Event Manager to let your application handle whose tests, set the `kAEIDoWhose` flag in the `callbackFlags` parameter of the `AEResolve` function. If for any reason one of your application's object accessor functions chooses not to handle a particular whose descriptor record, it should return `errAEEventNotHandled` as the result code, and the Apple Event Manager will try again using the original object specifier records, just as if the `kAEIDoWhose` flag were not set.

The key data for `formWhose` is specified by a ***whose descriptor record,*** which is a coerced AE record of descriptor type `typeWhoseDescriptor`. The data for a whose descriptor record consists of the two keyword-specified descriptor records shown in Table 6-5.

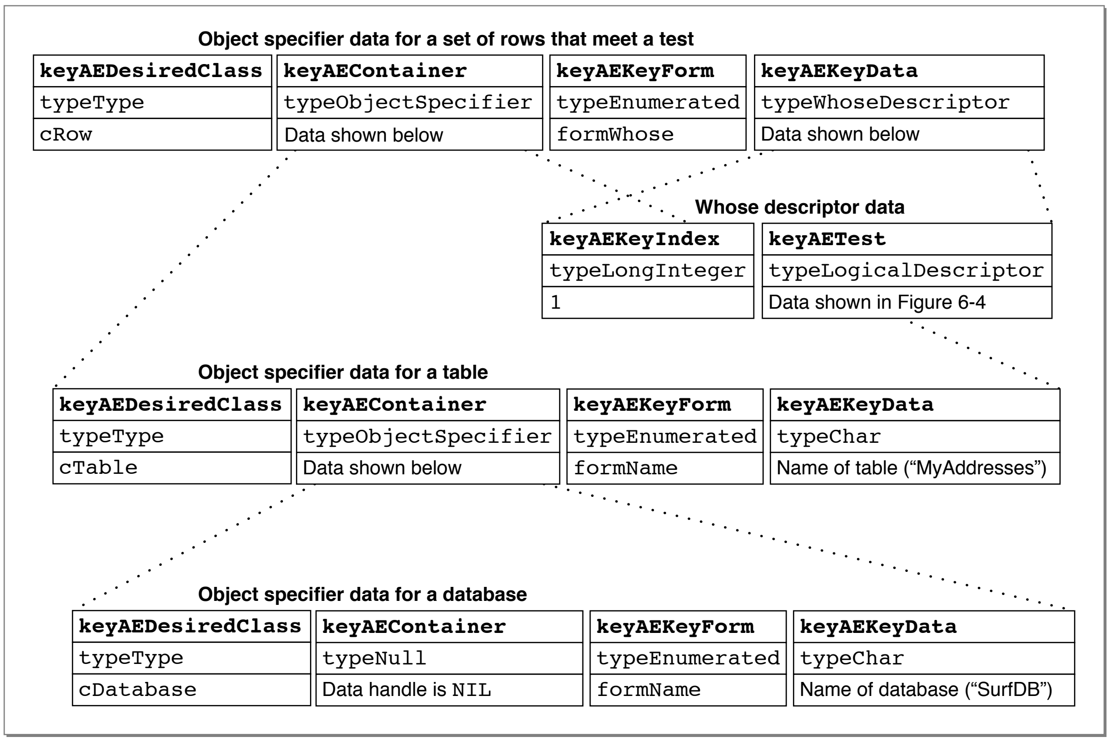
Table 6-5 Keyword-specified descriptor records for `typeWhoseDescriptor`

Keyword	Descriptor type	Data
`keyAEIndex`	`typeLongInteger`	Offset of requested element in group of elements that pass a test
	`typeAbsoluteOrdinal`	`kAEFirst` `kAEMiddle` `kAELast` `kAEAny` `kAEAll`
	`typeWhoseRange`	Whose range descriptor record
`keyAETest`	`typeCompDescriptor`	Comparison descriptor record
	`typeLogicalDescriptor`	Logical descriptor record

A whose descriptor record is never created directly by an application. The Apple Event Manager creates a whose descriptor record whenever an object specifier record of key form `formTest` is used to describe the container for elements described by an object specifier record of key form `formAbsolutePosition` or `formRange`, with some exceptions as noted in this section.

For example, Figure 6-3 on page 6-18 shows four object specifier records that show the container hierarchy for the first row that meets a test in the table "MyAddresses" of the database "SurfDB." The top two object specifier records in that figure use the key forms `formAbsolutePosition` and `formTest` to describe elements in a container. When it receives these two object specifier records, the Apple Event Manager collapses them into one, as shown in Figure 6-7. It then calls the application's object-counting function to find out how many objects of class `cRow` the table contains and the object-comparison function to test the rows in the table until it finds the first row that passes the test.

Figure 6-7 A container hierarchy created by the Apple Event Manager using a whose descriptor record



Object specifier data for a set of rows that meet a test

keyAEDesiredClass	**keyAEContainer**	**keyAEKeyForm**	**keyAEKeyData**
typeType	typeObjectSpecifier	typeEnumerated	typeWhoseDescriptor
cRow	Data shown below	formWhose	Data shown below

Whose descriptor data

keyAEKeyIndex	**keyAETest**
typeLongInteger	typeLogicalDescriptor
1	Data shown in Figure 6-4

Object specifier data for a table

keyAEDesiredClass	**keyAEContainer**	**keyAEKeyForm**	**keyAEKeyData**
typeType	typeObjectSpecifier	typeEnumerated	typeChar
cTable	Data shown below	formName	Name of table (“MyAddresses”)

Object specifier data for a database

keyAEDesiredClass	**keyAEContainer**	**keyAEKeyForm**	**keyAEKeyData**
typeType	typeNull	typeEnumerated	typeChar
cDatabase	Data handle is NIL	formName	Name of database (“SurfDB”)

If the elements to be tested are described by an object specifier record of key form `formAbsolutePosition` or `formRange` but are not of the same object class as their container, the Apple Event Manager cannot collapse the existing object specifier records into a whose descriptor record. Instead, the Apple Event Manager creates a whose descriptor record as if a third object specifier record of key form `formAbsolutePosition` and `kAEAll` were inserted between the object specifier record for the container and that for the tested elements. For example, the Apple Event Manager would interpret a request for “character 1 of word whose first letter = ‘a’” as “character 1 of every word whose first letter = ‘a’”.

When an object specifier record of key form `formTest` is used to describe the container for elements described by an object specifier record of key form `formRange`, the Apple Event Manager will, under certain conditions, coerce the corresponding range descriptor record to a ***whose range descriptor record,*** which is a coerced AE record of `typeWhoseRange`. The data for a whose range descriptor record consists of two keyword-specified descriptor records with the descriptor types and data shown in Table 6-6.

Table 6-6 Keyword-specified descriptor records for `typeWhoseRange`

Keyword	Descriptor type	Data
`keyAEWhoseRangeStart`	`typeLongInteger`	Offset of beginning of range

	`typeAbsoluteOrdinal`	`kAEFirst`
		`kAEMiddle`
		`kAELast`
		`kAEAny`
		`kAEAll`

`keyAEWhoseRangeStop`	`typeLongInteger`	Offset of end of range

	`typeAbsoluteOrdinal`	`kAEFirst`
		`kAEMiddle`
		`kAELast`
		`kAEAny`
		`kAEAll`

A whose range descriptor record describes the absolute position of the boundary elements, within the set of all elements that pass a test, that identify the beginning and end of the desired range.

The Apple Event Manager coerces a range descriptor record to a whose range descriptor record if the specified container and its elements are of the same class, if the container for the specified range of elements is a group of Apple event objects that pass a test, and if the boundary objects in the original range descriptor record meet these conditions:

- Both boundary objects are of the same object class as the Apple event objects in the range they specify.
- The object specifier record for each boundary object specifies its container with a descriptor record of descriptor type `typeCurrentContainer`.
- The object specifier record for each boundary object specifies a key form of `formAbsolutePosition`.

If these conditions are not met, the Apple Event Manager doesn't create a whose range descriptor record. Instead, as described earlier in this section, the Apple Event Manager creates a whose descriptor record as if the original request specified every element that passed the test.

If your application sets the `kAEIDoWhose` flag in the `callbackFlags` parameter of `AEResolve`, you should provide object accessor functions that can handle `formWhose`. These functions should coerce the whose descriptor record specified as key data for an object specifier record to an AE record and extract the data from the AE record by calling the `AEGetKeyPtr` and `AEGetKeyDesc` functions. If the keyword-specified descriptor record with the keyword `keyAEIndex` specifies descriptor type `typeWhoseRange`, your object accessor function must also coerce that descriptor record to an AE record and extract the data. Your object accessor function should then attempt to locate the requested objects and, if successful, return a token that identifies them.

If your application sets the `kAEIDoWhose` flag and attempts to resolve every whose descriptor record it receives, the Apple Event Manager does not attempt to resolve object specifier records of any key form. The object-counting and object-comparison functions are never called, and your application is solely responsible for determining the formats and types of all tokens.

Writing Object Callback Functions

If an Apple event parameter consists of an object specifier record, your handler for the Apple event typically calls `AEResolve` to begin the process of locating the requested Apple event object or objects. In turn, `AEResolve` calls object accessor functions and, if necessary, object callback functions provided by your application.

Every application that supports Apple event objects should provide object accessor functions that can locate Apple event objects belonging to any of the supported object classes. For an overview of the way `AEResolve` calls object accessor functions to locate Apple event objects described by object specifier records, see "Resolving Object Specifier Records," which begins on page 6-4.

In addition to object accessor functions, your application can provide up to seven object callback functions:

- An *object-counting function* counts the number of elements of a specified class in a specified container, so that the Apple Event Manager can determine how many elements it must examine to find the element or elements that pass a test. Your application must provide one object-counting function to handle object specifier records that specify tests. (See "Writing an Object-Counting Function," which begins on page 6-48.)

- An *object-comparison function* compares one element either to another element or to a descriptor record and returns either `TRUE` or `FALSE`. Your application must provide one object-comparison function to handle object specifier records that specify tests. (See "Writing an Object-Comparison Function" on page 6-50.)
- A *token disposal function* disposes of a token after your application calls the `AEDisposeToken` function. If your application doesn't provide a token disposal function, the Apple Event Manager uses the `AEDisposeDesc` function instead. Your application must provide a token disposal function if it requires more than a call to `AEDisposeDesc` to dispose of one of its tokens. This is true, for example, if your application supports marking by modifying its own data structures. (See page 6-99 for the declaration of a token disposal function.)
- An *error callback function* gives the Apple Event Manager an address to which to write the descriptor record it is currently working with if an error occurs while `AEResolve` is attempting to resolve an object specifier record. Your application is not required to provide an error callback function. (See page 6-100 for the declaration of an error callback function.)
- Three *marking callback functions* are used by the Apple Event Manager to get a mark token from your application, to mark specific Apple event objects, and to pare down a group of marked Apple event objects. Your application must provide all three marking functions if it supports marking. (See "Writing Marking Callback Functions" on page 6-53.)

To make your object callback functions available to the Apple Event Manager, use the `AESetObjectCallbacks` function:

```
myErr := AESetObjectCallbacks (@MyCompareObjects,
                               @MyCountObjects, @MyDisposeToken,
                               @MyGetMarkToken, @MyMark,
                               @MyAdjustMarks, @MyGetErrDesc);
```

Each parameter to the `AESetObjectCallbacks` function consists of either a pointer to the corresponding application-defined function or `NIL` if no function is provided. The `AESetObjectCallbacks` function sets object callback functions that are available only to your application. To set system object callback functions, which are available to all applications and processes running on the same computer, use the `AEInstallSpecialHandler` function as described on page 4-100.

To handle object specifier records that specify tests, your application must provide an object-counting function and an object-comparison function. The Apple Event Manager calls your application's object-counting function to determine the number of Apple event objects in a specified container that need to be tested. The Apple Event Manager calls your application's object-comparison function when it needs to compare one Apple event object to either another Apple event object or to a value in a descriptor record.

If your application does not provide a token disposal function, the Apple Event Manager uses the `AEDisposeDesc` function to dispose of tokens. This function does the job as long as disposing of tokens involves nothing more than simply disposing of a descriptor record. Otherwise, you need to provide custom token disposal function. For example, suppose the data field of a token descriptor record contains a handle to a block that in turn contains references to storage for the Apple event object referred to by the token. In this case, the application can provide a token disposal function that performs the tasks required to dispose of the token and any associated structures.

Whenever more than one Apple event object passes a test, `AEResolve` can either return a list of tokens or make use of a target application's ability to mark its own objects. Sometimes a list of tokens can become unmanageably large. For example, if a Get Data event asks for the names and addresses of all customers with a specified zip code who have purchased a specified product, the object accessor function that locates all the customers with the specified zip code might return a list of many thousands of tokens; the elements identified by those tokens would then have to be tested for the specified product. However, if your application uses some method of marking objects, you can choose simply to mark the requested objects rather than returning a list of tokens. "Writing Marking Callback Functions" on page 6-53 describes how to do this. If your application supports marking by modifying its own data structures, you must provide a token disposal function.

When one of your application's Apple event handlers calls the `AEResolve` function, the handler should pass a value in the `callbackFlags` parameter that specifies whether your application supports whose descriptor records or provides marking callback functions. You can add the following constants, as appropriate, to provide a value for the `callbackFlags` parameter:

```
CONST kAEIDoMinimum  =  $0000; {does not handle whose tests or }
                                 { provide marking callbacks}
      kAEIDoWhose    =  $0001; {supports key form formWhose}
      kAEIDoMarking  =  $0004; {provides marking functions}
```

For example, this code instructs the Apple Event Manager to call any marking functions previously set with the `AESetObjectCallbacks` function while resolving the object specifier record in the `objectSpecifier` parameter:

```
VAR
   objectSpecifier:    AEDesc;
   resultToken:        AEDesc;
   myErr:              OSErr;

myErr := AEResolve(objectSpecifier, kAEIDoMarking, resultToken);
```

If any of the marking callback functions are not installed, `AEResolve` returns the error `errAEHandlerNotFound`.

IMPORTANT

If your application doesn't specify `kAEIDoWhose`, the Apple Event Manager attempts to resolve all object specifier records of key form `formTest`. To do so, the Apple Event Manager uses your application's object-counting and object-comparison functions, and returns a token of type `typeAEList`.

If your application does specify `kAEIDoWhose`, the Apple Event Manager does not attempt to resolve object specifier records of any key form. In this case, the object-counting and object-comparison functions are never called; your application determines the formats and types of all tokens; and your application must interpret whose descriptor records created by the Apple Event Manager during the resolution of object specifier records. For more information, see "Handling Whose Tests," which begins on page 6-41. ▲

Writing an Object-Counting Function

To handle object specifier records that specify tests, your application should provide an object-counting function (unless it specifies `kAEIDoWhose` as just described). Your object-counting function should be able to count the number of elements of a given object class in a given container. For example, if your application supports Apple event objects that belong to the object class `cText` in the Text suite, your application should provide an object-counting function that can count Apple event objects of each element class listed in the definition of `cText` in the *Apple Event Registry: Standard Suites.* In this case, your application should provide an object-counting function that can count the number of words, items, or characters in a text object.

You specify your object-counting function with the `AESetObjectCallbacks` function. Whenever it is resolving an object specifier record and it requires a count of the number of elements in a given container, the Apple Event Manager calls your object-counting function.

Here's the declaration for a sample object-counting function:

```
FUNCTION MyCountObjects (desiredClass: DescType;
                         containerClass: DescType;
                         containerToken: AEDesc;
                         VAR result: LongInt): OSErr;
```

The Apple Event Manager passes the following information to your object-counting function: the object class ID of the Apple event objects to count, the object class of their container, and a token identifying their container. (The container class can be useful if you want to use one token type for several object classes.) Your object-counting function uses this information to count the number of Apple event objects of the specified object class in the specified container. After counting the Apple event objects, your application should return the noErr result code and, in the result parameter, the number of Apple event objects counted.

Listing 6-10 shows an application-defined function, MyCountObjects, that counts the number of objects for any object class supported by the application.

Listing 6-10 An object-counting function

```
FUNCTION MyCountObjects (desiredClass: DescType; containerClass: DescType;
                         containerToken: AEDesc; VAR result: LongInt): OSErr;
VAR
   window: WindowPtr;
BEGIN
   result := 0;
   IF desiredClass = cWindow THEN
   BEGIN
      IF containerClass = typeNull THEN
      BEGIN
         {count the number of windows}
         window := FrontWindow;
         WHILE window <> NIL DO
            BEGIN
               result := result + 1;
               window := WindowPtr(WindowPeek(window)^.nextWindow);
            END; {of while}
      END;
      MyCountObjects := noErr;
   END {of cWindow}
   ELSE
      IF desiredClass = cWord THEN
         {count the number of words in the container}
         MyCountObjects := MyCountWords(containerClass, containerToken,
                                        result)
   ELSE
      IF desiredClass = cParagraph THEN
         {count the number of paragraphs in the container}
         MyCountObjects := MyCountParas(containerClass, containerToken,
                                        result)
   ELSE
      {this app does not support any other object classes}
      MyCountObjects := kObjectClassNotFound;
END;
```

Writing an Object-Comparison Function

To handle object specifier records that specify tests, your application should provide an object-comparison function (unless it specifies `kAEIDoWhose` as described on page 6-48). Your object-comparison function should be able to compare one Apple event object to another Apple event object or to another descriptor record.

You specify your object-comparison function with the `AESetObjectCallbacks` function. Whenever it is resolving object specifier records and needs to compare the value of an Apple event object with another object or with data, the Apple Event Manager calls your object-comparison function.

Here's the declaration for a sample object-comparison function:

```
FUNCTION MyCompareObjects (comparisonOperator: DescType;
                            object: AEDesc;
                            objectOrDescToCompare: AEDesc;
                            VAR result: Boolean): OSErr;
```

The Apple Event Manager passes the following information to your object-comparison function: a comparison operator that specifies how the two objects should be compared, a token for the first Apple event object, and either a token that describes the Apple event object to compare or a descriptor record.

It is up to your application to interpret the comparison operators it receives. The meaning of comparison operators differs according to the Apple event objects being compared, and not all comparison operators apply to all object classes. After successfully comparing the Apple event objects, your object-comparison function should return the `noErr` result code and, in the `result` parameter, a Boolean value specifying `TRUE` if the result of the comparison is true and `FALSE` otherwise. If for any reason your comparison function is unable to compare the specified Apple event objects, it should return the result code `errAEEventNotHandled`; then the Apple Event Manager will try an alternative method of comparing the Apple event objects, such as calling the equivalent system object-comparison function, if one exists.

Your object-comparison function should be able to compare an Apple event object belonging to any object class with another Apple event object. Your function should also be able to compare two Apple event objects with different object classes, if appropriate. For example, an object-comparison function for a word-processing application might be asked to compare the First Name column of a specified row in a table with the first word on a specified page—that is, to compare an Apple event object of object class `cColumn` with an Apple event object of object class `cWord`. You must decide what kinds of comparisons make sense for your application.

The *Apple Event Registry: Standard Suites* defines standard comparison operators. Here is a list of the constants that correspond to these comparison operators:

```
CONST
   kAEGreaterThan       = '>   ';
   kAEGreaterThanEquals = '>=  ';
   kAEEquals            = '=   ';
   kAELessThan          = '<   ';
   kAELessThanEquals    = '<=  ';
   kAEBeginsWith        = 'bgwt';
   kAEEndsWith          = 'ends';
   kAEContains          = 'cont';
```

The comparison operators always relate the first operand to the second. For example, the constant `kAEGreaterThan` means that the object-comparison function should determine whether or not the value of the first operand is greater than the value of the second operand. For more information, see page 6-90.

Listing 6-11 shows an application-defined function, `MyCompareObjects`, that compares two Apple event objects of any object class supported by the application.

Listing 6-11 Object-comparison function that compares two Apple event objects

```
FUNCTION MyCompareObjects (comparisonOperator: DescType;
                           theObject: AEDesc;
                           objectOrDescToCompare: AEDesc;
                           VAR result: Boolean): OSErr;
BEGIN
   result := FALSE;
   {compare two objects for equivalence}
   IF comparisonOperator = kAEEquals THEN
      MyCompareObjects := MyCompEquals(theObject,
                                       objectOrDescToCompare,
                                       result)
   ELSE
      {compare two objects for greater than}
      IF comparisonOperator = kAEGreaterThan THEN
         MyCompareObjects := MyCompGreaterThan(theObject,
                                          objectOrDescToCompare,
                                          result)
   ELSE
      {compare two objects for less than}
      IF comparisonOperator = kAELessThan THEN
         MyCompareObjects := MyCompLessThan(theObject,
                                          objectOrDescToCompare,
                                          result)
   ELSE
   {this app does not support any other comparison operators}
      MyCompareObjects := errAEEventNotHandled;
END;
```

The `MyCompareObjects` function calls a separate application-defined routine for each comparison operator. In each case, the application-defined routine that actually performs the comparison can compare an Apple event object with either another Apple event object or with a descriptor record's data. If for any reason the comparison cannot be performed, the `MyCompareObjects` function returns the result code `errAEEventNotHandled`.

Writing Marking Callback Functions

Marking callback functions allow applications such as databases that can mark their own objects to take advantage of that capability when resolving object specifier records. Instead of returning a list of tokens for a group of Apple event objects that pass a test, your application can simply mark the Apple event objects and return a token that identifies how they have been marked. In this way, you can speed the resolution of complex object specifier records and reduce the amount of memory you need to allocate for tokens.

The use of marking callback functions is optional and usually makes sense if (a) you can reasonably expect that the tokens created in the process of resolving some object specifier records might not all fit in memory at once or (b) your application already uses a marking mechanism. If you want the Apple Event Manager to use marking callback functions provided by your application, you must add the `kAEIDoMarking` constant to the value of the `callbackFlags` parameter for the `AEResolve` function. If for any reason your application cannot mark a requested set of Apple event objects, it should return `errAEEventNotHandled` as the result code, and the Apple Event Manager will attempt to continue resolving the object specifier record by some other method, such as using a system marking function, if one exists.

If your application supports marking callback functions, it must provide three functions with declarations that match these examples:

```
FUNCTION MyGetMarkToken (containerToken: AEDesc;
                         containerClass: DescType;
                         VAR Result: AEDesc): OSErr;

FUNCTION MyMark (theToken: AEDesc; markToken: AEDesc;
                 markCount: LongInt): OSErr;

FUNCTION MyAdjustMarks (newStart, newStop: LongInt;
                        markToken: AEDesc): OSErr;
```

For more detailed information about these sample declarations, see “Object Callback Functions,” which begins on page 6-96.

To resolve a given object specifier record with the aid of the marking callback functions provided by your application, the Apple Event Manager first calls your application’s ***mark token function*** (`MyGetMarkToken`), passing a token that identifies the container of the elements to be marked in the `containerToken` parameter and the container’s object class in the `containerClass` parameter. The mark token function returns a mark token. A ***mark token,*** like other tokens, can be a descriptor record of any type; however, unlike other tokens, it identifies the way your application marks Apple event objects during the current session while resolving a single test. A mark token does not identify a specific Apple event object; rather, it allows your application to associate a group of objects with a marked set.

After it receives the mark token, the Apple Event Manager can call your application's ***object-marking function*** (`MyMark`) repeatedly to mark specific Apple event objects. The Apple Event Manager passes the following information to your marking function: in the `theToken` parameter, a token for the object to be marked (obtained from the appropriate object accessor function); in the `markToken` parameter, the current mark token; and in the `markCount` parameter, the mark count. The ***mark count*** indicates the number of times the Apple Event Manager has called the marking function for the current mark token. Your application should associate the mark count with each Apple event object it marks.

When the Apple Event Manager needs to identify either a range of elements or the absolute position of an element in a group of Apple event objects that pass a test, it can use your application's ***mark-adjusting function*** (`MyAdjustMarks`) to unmark objects that it has previously marked. For example, suppose an object specifier record specifies "any row in the table 'MyCustomers' for which the City column is 'San Francisco.'" The Apple Event Manager first uses the appropriate object accessor routine to locate all the rows in the table for which the City column is "San Francisco" and calls the application's marking function repeatedly to mark them. It then generates a random number between 1 and the number of rows it found that passed the test and calls the application's mark-adjusting function to unmark all the rows whose mark count does not match the randomly generated number. If the randomly chosen row has a mark count value of 5, the Apple Event Manager passes the mark-adjusting function 5 in both the `newStart` parameter and the `newStop` parameter, and the current mark token in the `markToken` parameter. The `newStart` and `newStop` parameters identify the beginning and end of the new set of marked objects that the mark-adjusting function will create by unmarking those previously marked objects not included in the new set.

When the Apple Event Manager calls your mark-adjusting function, your application must dispose of any data structures that it may have created to mark the previously marked objects. The Apple Event Manager calls your mark-adjusting function only once for a given mark token.

A mark token is valid until the Apple Event Manager either disposes of it (by calling `AEDisposeToken`) or returns it as the result of the `AEResolve` function. If the final result of a call to the `AEResolve` function is a mark token, the Apple event objects currently marked for that mark token are those specified by the object specifier record passed to `AEResolve`, and your application can proceed to do whatever the Apple event has requested. Note that your application is responsible for disposing of a final mark token with a call to `AEDisposeToken`, just as for any other final token.

If your application supports marking, it should also provide a token disposal function. When the Apple Event Manager calls `AEDisposeToken` to dispose of a mark token that is not the final result of a call to `AEResolve`, the subsequent call to your token disposal function lets you know that you can unmark the Apple event objects marked with that mark token. A call to `AEDisposeDesc` to dispose of a mark token (which would occur if you did not provide a token disposal function) would leave the objects marked.

Creating Object Specifier Records

If your application creates and sends Apple events that require the target application to locate Apple event objects, your application must create object specifier records for those events. This section describes how to use the four keyword-specified descriptor records described in "Descriptor Records Used in Object Specifier Records," which begins on page 6-8, to specify the object class ID, container, key form, and key data for an object specifier record.

Because the internal structure of an object specifier record is nearly identical to the internal structure of an AE record, it is possible to use `AECreateList`, `AEPutPtr`, and `AEPutKeyDesc` to add the four keyword-specified descriptor records to an AE record, then use `AECoerceDesc` to coerce the AE record to a descriptor record of type `typeObjectSpecifier`. However, it is usually preferable to use the `CreateObjSpecifier` function to accomplish the same goal. The `CreateObjSpecifier` function adds the keyword-specified descriptor records directly to an object specifier record, thus eliminating several steps that are required if you create an AE record first. The instructions that follow make use of `CreateObjSpecifier`.

To specify the class ID for an object specifier record, your application can specify the appropriate class ID value as the `desiredClass` parameter for the `CreateObjSpecifier` function, which uses it to create a keyword-specified descriptor record with the keyword `keyAEDesiredClass` as part of an object specifier record.

To specify the container for an object specifier record, your application must create a keyword-specified descriptor record with the keyword `keyAEContainer` that fully describes the container of the Apple event object. Because this container is usually another Apple event object, the container is usually specified by another object specifier record.

To specify the complete container hierarchy of an Apple event object, your application must create a series of nested object specifier records, starting with the object specifier record for the Apple event object whose container is outermost. With the exception of this first object specifier record, each object specifier record specifies another object specifier record in the chain as a container.

For example, Figure 6-2 on page 6-10 shows a series of nested object specifier records that specify the first row of a table named "Summary of Sales" in a document named "Sales Report." The logical organization of the same object specifier records is summarized in Table 6-7.

Table 6-7 Nested object specifier records that describe a container hierarchy

Keyword	Descriptor type	Data
`keyAEDesiredClass`	`typeType`	`cRow`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cTable`
`keyAEContainer`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cDocument`
`keyAEContainer`	`typeNull`	Data handle is `NIL`
`keyAEKeyForm`	`typeEnumerated`	`formName`
`keyAEKeyData`	`typeChar`	`"Sales Report"`
`keyAEKeyForm`	`typeEnumerated`	`formName`
`keyAEKeyData`	`typeChar`	`"Summary of Sales"`
`keyAEKeyForm`	`typeEnumerated`	`formAbsolutePosition`
`keyAEKeyData`	`typeLongInteger`	`1`

Note
The format used in Table 6-7 and similar tables throughout this chapter does not show the structure of nested object specifier records as they exist within an Apple event. Instead, this format shows what you would obtain after calling `AEGetKeyDesc` repeatedly to extract the object specifier records from an Apple event record.

When you call `AEGetKeyDesc` to extract a null descriptor record, `AEGetKeyDesc` returns a descriptor record of type `AEDesc` with a descriptor type of `typeNull` and a data handle whose value is 0. ◆

To specify the default container for an object specifier record (such as the container for the document in Table 6-7), you can use `AECreateDesc` to create a null descriptor record, which you can then pass in the `theContainer` parameter of the `CreateObjSpecifier` function. The `CreateObjSpecifier` function uses the null descriptor record to create a keyword-specified descriptor record with the keyword `keyAEContainer` as part of an object specifier record.

The object specifier record that specifies the default container is always the first record you create in a series of nested object specifier records that specifies the complete container hierarchy for an Apple event object. Each one in the series uses the previously created object specifier record to specify its container. As with the null descriptor record, you can pass an object specifier record as the second parameter to the `CreateObjSpecifier` function, which uses it to create a keyword-specified descriptor record with the keyword `keyAEContainer`.

To specify the key form for an object specifier record, your application can specify a key form constant as the third parameter to the `CreateObjSpecifier` function, which uses it to create a keyword-specified descriptor record with the keyword `keyAEKeyForm` as part of an object specifier record. The standard key forms for object specifier records are summarized in Table 6-1 on page 6-12.

For example, the key form for the object specifier records in Table 6-7 that specify the document and the table is `formName`. In other words, the key data identifies the document and the table by their names. Similarly, the key form for the object specifier record in Table 6-7 that specifies the first row in the table is `formAbsolutePosition`. In other words, the key data identifies the position of the row compared to other rows in the same container.

To specify the key data for an object specifier record, your application must create a keyword-specified descriptor record with the keyword `keyAEKeyData` whose data handle refers to the appropriate data for the specified key form. You can use `AECreateDesc`, `CreateCompDescriptor`, `CreateLogicalDescriptor`, and related functions to create the descriptor record, which you can then pass in the fourth parameter of the `CreateObjSpecifier` function. The `CreateObjSpecifier` function uses this descriptor record to create a keyword-specified descriptor record with the keyword `keyAEKeyData` as part of an object specifier record.

Creating a Simple Object Specifier Record

This section shows how to use the `CreateObjSpecifier` function to create the object specifier record shown in Table 6-7. The `CreateObjSpecifier` function creates the necessary keyword-specified descriptor records for the class ID, container, key form, and key data and returns the resulting object specifier record as a descriptor record of type `typeObjectSpecifier`.

Listing 6-12 shows how the `CreateObjSpecifier` function creates an object specifier record from parameters that an application specifies.

Listing 6-12 Creating an object specifier record using `CreateObjSpecifier`

```
VAR
   desiredClass:        DescType;
   myObjectContainer:   AEDesc;
   myKeyForm:           DescType;
   myKeyDataDesc:       AEDesc;
   disposeInputs:       Boolean;
   myObjSpecRec:        AEDesc;
   myErr:               OSErr;

desiredClass := cRow;
myObjectContainer := MyGetContainer;
myKeyForm := formAbsolutePosition;
myKeyDataDesc := MyGetKeyData;
disposeInputs := TRUE;
{create an object specifier record}
myErr := CreateObjSpecifier(desiredClass, myObjectContainer,
                              myKeyForm, myKeyDataDesc,
                              disposeInputs, myObjSpecRec);
```

The code shown in Listing 6-12 demonstrates how an application might use the `CreateObjSpecifier` function to create four keyword-specified descriptor records as part of a descriptor record of type `typeObjectSpecifier`. The `CreateObjSpecifier` function returns a result code of `noErr` if the object specifier record was successfully created. The object specifier record returned in the `myObjSpecRec` parameter describes an Apple event object of the class specified by the `desiredClass` parameter, located in the container specified by the `myObjectContainer` parameter, with the key form specified by the `myKeyForm` parameter and key data specified by the `myKeyDataDesc` parameter.

You can specify `TRUE` in the `disposeInputs` parameter if you want the `CreateObjSpecifier` function to dispose of the descriptor records you created for the `myObjectContainer` and `myKeyDataDesc` parameters. If you specify `FALSE`, then your application is responsible for disposing of these leftover descriptor records.

Listing 6-13 shows an application-defined function that uses `CreateObjSpecifier` to create an object specifier record for the first row in the table named “Summary of Sales” in the document “Sales Report,” then uses the object specifier record returned in the `myObjSpecRec` parameter as the direct parameter for a Get Data event.

Listing 6-13 Using `CreateObjSpecifier` in an application-defined function

```
FUNCTION MyRequestRowFromTarget (targetAddress: AEAddressDesc;
                                  VAR reply: AppleEvent): OSErr;
VAR
   desiredClass:           DescType;
   myKeyForm:              DescType;
   myObjectContainer:      AEDesc;
   myObjSpecRec:           AEDesc;
   myKeyDataDesc:          AEDesc;
   keyData:                LongInt;
   theAppleEvent:          AppleEvent;
   myErr:                  OSErr;
   ignoreErr:              OSErr;

BEGIN
   {initialize (set to null descriptor records) the two descriptor records }
   { that must eventually be disposed of}
   MyInit2DescRecs(myObjSpecRec, theAppleEvent);

   desiredClass := cRow;                   {specify the class}
                                           {specify container for the row}
   myErr := MyCreateTableContainer(myObjectContainer,
                                     'Summary of Sales', 'Sales Report');
   IF myErr = noErr THEN
   BEGIN
      myKeyForm := formAbsolutePosition;   {specify the key form}
      keyData := 1;                        {specify the key data for row}
      myErr := AECreateDesc(typeLongInteger, @keyData, Sizeof(keyData),
                              myKeyDataDesc);
      IF myErr = noErr THEN
         {create the object specifier record}
         myErr := CreateObjSpecifier(desiredClass, myObjectContainer,
                                       myKeyForm, myKeyDataDesc,
                                       TRUE, myObjSpecRec);
      IF myErr = noErr THEN
         {myObjSpecRec now describes an Apple event object, and will become }
         { direct parameter of a Get Data event; first create Get Data event}
         myErr := AECreateAppleEvent(kAECoreSuite, kAEGetData, targetAddress,
                                       kAutoGenerateReturnID,
                                       kAnyTransactionID, theAppleEvent);
      IF myErr = noErr THEN
         {add myObjSpecRec as the direct parameter of the Get Data event}
         myErr := AEPutParamDesc(theAppleEvent, keyDirectObject,
                                 myObjSpecRec);
      IF myErr = noErr THEN
         myErr := AESend(theAppleEvent, reply, kAEWaitReply +
                         kAENeverInteract, kAENormalPriority, 120,
                         @MyIdleFunction, NIL);
   END;
   ignoreErr := AEDisposeDesc(myObjSpecRec);
   ignoreErr := AEDisposeDesc(theAppleEvent);
   MyRequestRowFromTarget := myErr;
END;
```

The `MyRequestRowFromTarget` function shown in Listing 6-13 specifies the class ID as `cRow`, indicating that the desired Apple event object is a row in a table. It uses the application-defined function `MyCreateTableContainer` to create an object specifier record for the table that contains the row, passing "Summary of Sales" and "Sales Report" as the second and third parameters to identify the name of the table and the name of the document that contains the table. (The next section, "Specifying the Container Hierarchy," explains how to construct the `MyCreateTableContainer` function.) It then specifies the key form as the constant `formAbsolutePosition`, which indicates that the key data specifies the position of the row within its container; sets the `keyData` variable to 1, indicating the first row, and uses `AECreateDesc` to create a descriptor record for the key data; and uses `CreateObjSpecifier` to create the object specifier record that describes the desired word.

The desired row is now fully described by the `myObjSpecRec` variable, which contains a descriptor record of type `typeObjectSpecifier` that contains the three nested object specifier records shown in Table 6-7 on page 6-56. After using `AECreateAppleEvent` to create a Get Data event, the `MyRequestRowFromTarget` function uses the `AEPutParamDesc` function to add the `myObjSpecRec` variable to the Get Data event as a direct parameter, then uses `AESend` to send the Get Data event.

Note that the `MyRequestRowFromTarget` function begins by using the application-defined function `MyInit2DescRecs` to set `myObjSpecRec` and `theAppleEvent` to null descriptor records. These two functions must be disposed of whether the function is successful or not. By setting them to null descriptor records, the function can dispose of them at the end regardless of where an error may have occurred.

Specifying the Container Hierarchy

Because the container for an object specifier record usually consists of a chain of other object specifier records that specify the container hierarchy, your application must create all the object specifier records in the chain, starting with the record for the outermost container. Listing 6-14 and Listing 6-15 demonstrate how to use the `CreateObjSpecifier` function to create the first two object specifier records in such a chain: the records for a document and a table.

Listing 6-14 Specifying a document container

```
FUNCTION MyCreateDocContainer (VAR myDocContainer: AEDesc;
                               docName: Str255): OSErr;
VAR
   myDocDescRec:  AEDesc;
   nullDescRec:   AEDesc;
   myErr:         OSErr;
BEGIN
   {create a descriptor record for the name of the document}
   myErr := AECreateDesc(typeChar, @docName[1],
                        Length(docName), myDocDescRec);
   IF myErr = noErr THEN
      {create a null descriptor record}
      myErr := AECreateDesc(typeNull, NIL, 0, nullDescRec);
   IF myErr = noErr THEN
      {create an object specifier record to specify the }
      { document object}
      myErr := CreateObjSpecifier(cDocument, nullDescRec,
                                  formName, myDocDescRec, TRUE,
                                  myDocContainer);
   MyCreateDocContainer := myErr;
END;
```

The function `MyCreateDocContainer` in Listing 6-14 creates an object specifier record that identifies a document by name. It starts by using the `AECreateDesc` function to create two descriptor records: one of type `typeChar` for the name of the document, and one of type `typeNull` for the null descriptor record that specifies the default container (because the document is not contained in any other Apple event object). These two descriptor records can then be used as parameters for the `CreateObjSpecifier` function, which returns an object specifier record (that is, a descriptor record of type `typeObjectSpecifier`) in the `myDocContainer` variable. The object specifier record specifies an Apple event object of the object class `cDocument` in the container specified by the `nullDescRec` variable with a key form of `formName` and the key data specified by the `myDocDescRec` variable. This object specifier can be used by itself to specify a document, or it can be used to specify the container for another Apple event object.

Listing 6-15 shows an application-defined function, `MyCreateTableContainer`, that creates an object specifier record describing a table contained in a document.

Listing 6-15 Specifying a table container

```
FUNCTION MyCreateTableContainer (VAR myTableContainer: AEDesc;
                                 tableName: Str255;
                                 docName: Str255): OSErr;
VAR
   myDocDescRec:        AEDesc;
   myTableDescRec:      AEDesc;
   myErr:               OSErr;
BEGIN
   {create a container for the document}
   myErr := MyCreateDocContainer(myDocDescRec, docName);
   IF myErr = noErr THEN
   BEGIN
      {create the table container, }
      { first specify the descriptor record for the key data}
      myErr := AECreateDesc(typeChar, @tableName[1],
                           Length(tableName), myTableDescRec);
      IF myErr = noErr THEN
         myErr := CreateObjSpecifier(cTable, myDocDescRec,
                                    formName, myTableDescRec,
                                    TRUE, myTableContainer);
   END;
   MyCreateTableContainer := myErr;
END;
```

The function `MyCreateTableContainer` in Listing 6-15 starts by using the function `MyCreateDocContainer` from Listing 6-14 to create an object specifier record that identifies the table's container—the document in which the table is located. Then it uses the `AECreateDesc` function to create a descriptor record for the key data—a name that, when combined with the key form `formName`, will identify the table in the document. The object specifier record for the document and the descriptor record specifying the table's name are passed to the function `CreateObjSpecifier`. It returns an object specifier record in the `myTableContainer` parameter that specifies an Apple event object of the object class `cTable` in the container specified by the `MyDocDescRec` variable with a key form of `formName` and the key data specified by the `myTableDescRec` variable. This object specifier record can be used by itself to specify a table, or it can be used to specify the container for another Apple event object.

Listing 6-13 uses the `MyCreateTableContainer` function shown in Listing 6-15 to specify the container hierarchy illustrated in Table 6-7 on page 6-56. The nested object specifier records shown in Table 6-7 use the key forms `formName` and `formRelativePosition`. You can create key data for the key forms `formPropertyID`, `formUniqueID`, and `formRelativePosition` using similar techniques.

Specifying a Property

The key form `formPropertyID` allows your application to specify key data identifying a property of the object specified as a container. For example, an object specifier record that identifies the font property of a word specifies `cProperty` as the class ID, an object specifier record for the word as the property's container, `formPropertyID` as the key form, and the constant `pFont` as the key data.

Note that an object specifier record that identifies a property does not include a value for the property, such as `Palatino`. The value of a property is returned or set as a parameter of an Apple event. For example, an application that sends a Get Data event to get the `pFont` property of a word receives a value such as `Palatino` in the `keyAEResult` parameter of the reply event, and an application that sends a Set Data event to change the `pFont` property of a word specifies a font in the `keyAEData` parameter of the Set Data event.

To specify the key data for a key form of `formPropertyID`, your application must create a descriptor record of `typeType` whose data consists of a constant specifying a property. You can use `AECreateDesc` to create a descriptor record that specifies the constant for a property, then use `CreateObjSpecifier` to add the descriptor record to an object specifier record as a keyword-specified descriptor record with the keyword `keyAEKeyData`.

For more information about object specifier records that specify a property, see "Key Data for a Property ID" on page 6-13.

Specifying a Relative Position

The key form `formRelativePosition` allows your application to specify key data identifying an element or a set of elements that are immediately before or after the specified container. For example, if the container is a table, you could use a key form of `formRelativePosition` to specify the paragraph before or after the table.

To specify the key data for a key form of `formRelativePosition`, your application must create a descriptor record of `typeEnumerated` whose data consists of a constant specifying either the element after (`kAENext`) or the element before (`kAEPrevious`) the specified container.

You can use `AECreateDesc` to create a descriptor record that specifies one of these constants, then use `CreateObjectSpecifier` to add it to an object specifier record as a keyword-specified descriptor record with the keyword `keyAEKeyData`.

For more information about object specifier records that specify a relative position, see "Key Data for Relative Position" on page 6-15.

Creating a Complex Object Specifier Record

This section describes how to create object specifier records that specify a test or a range. You can specify the object class ID for these object specifier records the same way you would for any other object specifier record. When you create the other three keyword-specified descriptor records, however, you can use additional Apple Event Manager routines and descriptor types to specify any combination of Apple event objects.

Specifying a Test

The key form `formTest` allows your application to specify key data that identifies one or more elements in the specified container that pass a test. To do so, your application must construct several interconnected descriptor records that specify comparisons and, if necessary, logical expressions.

For example, to specify "the first row in which the First Name column equals 'John' and the Last Name column equals 'Chapman' in the table 'MyAddresses' of the database 'SurfDB,'" your application must construct an object specifier record whose key data describes a logical expression that applies the logical operator `AND` to two separate comparisons for each row: a comparison of the First Name column to the word "John" and a comparison of the Last Name column to the word "Chapman."

The logical organization of the data for the object specifier record that specifies this test is summarized in Table 6-8 and Table 6-9. (It is also illustrated in Figure 6-3 and Figure 6-4, beginning on page 6-18.) The listings in the remainder of this section demonstrate how to create this object specifier record. For general information about the organization of key data for a test, see "Key Data for a Test," which begins on page 6-15.

Table 6-8 Object specifier record for the first row that meets a test in the table named "MyAddresses"

Keyword	Descriptor type	Data
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cRow
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cTable
keyAEContainer	typeObjectSpecifier	(see indented record)
keyAEDesiredClass	typeType	cDatabase
keyAEContainer	typeNull	Data handle is NIL
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"SurfDB"
keyAEKeyForm	typeEnumerated	formName
keyAEKeyData	typeChar	"MyAddresses"
keyAEKeyForm	typeEnumerated	formTest
keyAEKeyData	typeLogicalDescriptor	(see Table 6-9)
keyAEKeyForm	typeEnumerated	formAbsolutePosition
keyAEKeyData	typeLongInteger	1

Table 6-9 Logical descriptor record that specifies a test

Keyword	Descriptor type	Data
`keyAELogicalOperator`	`typeEnumerated`	`kAEAnd`
`keyAELogicalTerms`	`typeAEList`	(see indented records)
	`typeCompDescriptor`	(see indented record)
`keyAECompOperator`	`typeType`	`kAEEquals`
`keyAEObject1`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cColumn`
`keyAEContainer`	`typeObjectBeingExamined`	Data handle is `NIL`
`keyAEKeyForm`	`typeEnumerated`	`formName`
`keyAEKeyData`	`typeChar`	`"First Name"`
`keyAEObject2`	`typeChar`	`"John"`
	`typeCompDescriptor`	(see indented record)
`keyAECompOperator`	`typeType`	`kAEEquals`
`keyAEObject1`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cColumn`
`keyAEContainer`	`typeObjectBeingExamined`	Data handle is `NIL`
`keyAEKeyForm`	`typeEnumerated`	`formName`
`keyAEKeyData`	`typeChar`	`"Last Name"`
`keyAEObject2`	`typeChar`	`"Chapman"`

Because both the database and the table shown in Table 6-8 are specified by name, it would be convenient to have an application-defined routine that creates an object specifier record that uses the key form `formName`. The `MyCreateFormNameObjSpecifier` function shown in Listing 6-16 can be used for this purpose.

Listing 6-16 Creating an object specifier record with the key form `formName`

```
FUNCTION MyCreateFormNameObjSpecifier
                        (class: DescType; container: AEDesc;
                         keyDataName: str255;
                         VAR resultObjSpecRec: AEDesc): OSErr;
VAR
   keyDataDescRec: AEDesc;
   myErr:          OSErr;
BEGIN
   myErr := AECreateDesc(typeChar, @keyDataName[1],
                           Length(keyDataName), keyDataDescRec);
   IF myErr = noErr THEN
      myErr := CreateObjSpecifier(class, container, formName,
                                  keyDataDescRec, TRUE,
                                  resultObjSpecRec);
   MyCreateFormNameObjSpecifier := myErr;
END;
```

The `MyCreateFormNameObjSpecifier` function shown in Listing 6-16 returns, in the `resultObjSpecRec` parameter, an object specifier record that describes an Apple event object of the class specified by the `class` parameter, located in the container specified by the `container` parameter, with the key form `formName` and key data specified by the `keyDataName` parameter. This function is used in Listing 6-19 on page 6-70 to create object specifier records that use the key form `formName` for the database and the table.

The nested object specifier records shown in Table 6-9 specify "the rows in which the First Name column equals 'John' and the Last Name column equals 'Chapman.'" To identify the rows that pass this test, the Apple Event Manager needs to evaluate two comparisons: the comparison of each row of the First Name column to the word "John," and the comparison of each row of the Last Name column to the word "Chapman."

The Apple Event Manager uses the information in comparison descriptor records to compare the specified elements in a container, one at a time, either to another Apple event object or to the data associated with a descriptor record. The two comparison descriptor records you need to create for this example are summarized in Table 6-9 on page 6-66.

You can use the `CreateCompDescriptor` function to create a comparison descriptor record, or you can create an AE record and use `AECoerceDesc` to coerce it to a comparison descriptor record. Listing 6-17 shows an example of an application-defined routine that creates an object specifier record and a descriptor record of `typeChar`, then uses the `CreateCompDescriptor` function to add them to a comparison descriptor record.

Listing 6-17 Creating a comparison descriptor record

```
FUNCTION MyCreateComparisonDescRec (VAR compDesc: AEDesc;
                                    colName: str255;
                                    name: str255): OSErr;
VAR
   logicalContainer, colNameDesc, nameDesc:  AEDesc;
   myObjectExaminedContainer:                AEDesc;
   myErr:                                    OSErr;
BEGIN
   {create the object specifier record for keyAEObject1; }
   { first create container}
   myErr := AECreateDesc(typeObjectBeingExamined, NIL, 0,
                         myObjectExaminedContainer);
   {create key data}
   IF myErr = noErr THEN
      myErr := AECreateDesc(typeChar, @colName[1],
                         Length(colName), colNameDesc);
   {now create the object specifier record}
   IF myErr = noErr THEN
      myErr := CreateObjSpecifier(cColumn,
                                  myObjectExaminedContainer,
                                  formName, colNameDesc, TRUE,
                                  logicalContainer);
   {create the descriptor record for keyAEObject2}
   IF myErr = noErr THEN
      myErr := AECreateDesc(typeChar, @name[1], Length(name),
                         nameDesc);
   {create the first logical term (comp descriptor record)}
   IF myErr = noErr THEN
      myErr := CreateCompDescriptor(kAEEquals, logicalContainer,
                                    nameDesc, TRUE, compDesc);
   MyCreateComparisonDescRec := myErr;
END;
```

The MyCreateComparisonDescRec function takes two strings and uses them to create a comparison descriptor record. The string passed in the second parameter specifies the name of the column whose contents should be compared to the string passed in the third parameter. First, the MyCreateComparisonDescRec function uses AECreateDesc to create a descriptor record of typeObjectBeingExamined, which is returned in the variable myObjectExaminedContainer. Next, AECreateDesc creates a descriptor record of descriptor type typeChar, whose data consists of the string in the variable colName, and which is returned in the variable colNameDesc. The code then passes the variables myObjectExaminedContainer and colNameDesc to the CreateObjSpecifier function, which uses them to create an object specifier record, returned in the logicalContainer variable, that becomes the keyword-specified descriptor record with the keyword keyAEObject1.

Next, the MyCreateComparisonDescRec function uses AECreateDesc and the name parameter to create the descriptor record for keyAEObject2, which AECreateDesc returns in the nameDesc variable. Finally, the code passes the constant kAEEquals, the variable logicalContainer, and the variable nameDesc to the CreateCompDescriptor function, which creates a comparison descriptor record that allows the Apple Event Manager (with the help of object-comparison functions provided by the server application) to determine whether the specified column in the row currently being checked equals the specified string.

You can use the MyCreateComparisonDescRec function to create both the descriptor records of type typeCompDescriptor shown in Table 6-9 on page 6-66. These descriptor records provide two logical terms for a logical descriptor record. The entire logical descriptor record corresponds to the logical expression "the First Name column equals 'John' AND the Last Name column equals 'Chapman.'"

You can use the CreateLogicalDescriptor function to create a logical descriptor record, or you can create an AE record and use the AECoerceDesc function to coerce it to a comparison descriptor record. Listing 6-18 shows an application-defined function that adds two comparison descriptor records to a descriptor list, then uses the CreateLogicalDescriptor function to create a logical descriptor record whose logical terms are the two comparison descriptor records.

Listing 6-18 Creating a logical descriptor record

```
FUNCTION MyCreateLogicalDescRec (VAR compDesc1, compDesc2: AEDesc;
                                 logicalOperator: DescType;
                                 VAR logicalDesc: AEDesc): OSErr;
VAR
   logicalTermsList: AEDescList;
   myErr:            OSErr;
BEGIN
   {create a logical descriptor record that contains two }
   { comparison descriptor records}
   {first create a list}
   myErr := AECreateList(NIL, 0, FALSE, logicalTermsList);
   IF myErr = noErr THEN
      myErr := AEPutDesc(logicalTermsList, 1, compDesc1);
   IF myErr = noErr THEN
      myErr := AEPutDesc(logicalTermsList, 2, compDesc2);
   IF myErr = noErr THEN
      myErr := AEDisposeDesc(compDesc1);
   IF myErr = noErr THEN
      myErr := AEDisposeDesc(compDesc2);
   IF myErr = noErr THEN
      myErr := CreateLogicalDescriptor(logicalTermsList,
                                       logicalOperator, TRUE,
                                       logicalDesc);
   MyCreateLogicalDescRec := myErr;
END;
```

Listing 6-19 uses the application-defined functions shown in Listing 6-16, Listing 6-17, and Listing 6-18 to build the object specifier record illustrated in Table 6-8 and Table 6-9.

Listing 6-19 Creating a complex object specifier record

```
FUNCTION MyCreateObjSpecRec (VAR theResultObj: AEDesc): OSErr;
VAR
   nullContainer, databaseContainer, tableContainer:  AEDesc;
   compDesc1, compDesc2:                              AEDesc;
   logicalTestDesc, rowTestContainer, rowOffset:      AEDesc;
   myErr:                                             OSErr;
```

```
BEGIN
   {create a null container}
   myErr := AECreateDesc(typeNull, NIL, 0, nullContainer);
   {create a container for the database}
   IF myErr = noErr THEN
      myErr := MyCreateFormNameObjSpecifier(cDatabase, nullContainer,
                                              'SurfDB', databaseContainer);
   {create a container for the table}
   IF myErr = noErr THEN
      myErr := MyCreateFormNameObjSpecifier(cTable, databaseContainer,
                                              'MyAddresses', tableContainer);
   {create a container for the row--an object specifier record that }
   { specifies a test (the row whose First Name column = 'John' and }
   { Last Name column = 'Chapman')}

   {create the first comparison descriptor record}
   IF myErr = noErr THEN
      myErr := MyCreateComparisonDescRec(compDesc1, 'First Name', 'John');
   {create the second comparison descriptor record}
   IF myErr = noErr THEN
      myErr := MyCreateComparisonDescRec(compDesc2, 'Last Name', 'Chapman');

   {create the logical descriptor record}
   IF myErr = noErr THEN
      myErr := MyCreateLogicalDescRec(compDesc1, compDesc2, kAEAND,
                                         logicalTestDesc);

   {now create the object specifier record that specifies the test}
   IF myErr = noErr THEN
      myErr := CreateObjSpecifier(cRow, tableContainer, formTest,
                                    logicalTestDesc, TRUE, rowTestContainer);

   {create the object specifier record for the Apple event object}
   {first, create the descriptor record for the key data}
   IF myErr = noErr THEN
      myErr := CreateOffsetDescriptor (1, rowOffset);
   {now create the object specifier record}
   IF myErr = noErr THEN
      myErr := CreateObjSpecifier (cRow, rowTestContainer,
                                     formAbsolutePosition, rowOffset,
                                     TRUE, theResultObj);
   MyCreateObjSpecRec := myErr;
END;
```

The `MyCreateObjSpecRec` function shown in Listing 6-19 begins by using `AECreateDesc` to create a null descriptor record, then uses the `MyCreateFormNameObjSpecifier` function (shown in Listing 6-16) to specify the default container for the database named "SurfDB." The code then calls the `MyCreateFormNameObjSpecifier` function again, this time passing the object specifier record for SurfDB to specify the container for the table "MyAddresses." The next two calls are both to the `MyCreateComparisonDescRec` function (shown in Listing 6-17), which creates the comparison descriptor records that allow the Apple Event Manager to compare the First Name column and Last Name column to the names "John" and "Chapman," respectively. The next call passes these two comparison descriptor records to the `MyCreateLogicalDescRec` function (shown in Listing 6-18) in the `compDesc1` and `compDesc2` variables.

Now all the components of the logical descriptor record are ready to assemble. The next call, to `CreateObjSpecifier`, specifies the logical descriptor record in the `logicalTestDesc` variable as the key data for the object specifier record that specifies the test. A call to the Apple Event Manager routine `CreateOffsetDescriptor` then creates an offset descriptor record that contains the integer 1. Finally, the code passes the offset descriptor record to the `CreateObjSpecifier` function in the `rowOffset` variable to create the final object specifier record, which describes the requested row as the first row that passes the test.

The `CreateOffsetDescriptor` function creates a descriptor record of type `typeLongInteger` that can be used as the key data with a key form of `formAbsolutePosition` to indicate an element's offset within its container. A positive integer indicates an offset from the beginning of the container (the first element has an offset of 1), and a negative integer indicates an offset from the end of the container (the last element has an offset of –1). Using `CreateOffsetDescriptor` accomplishes the same thing as setting a variable to an integer and passing the variable to `AECreateDesc` to create a descriptor record of type `typeLongInteger`.

Specifying a Range

The key form `formRange` allows your application to specify key data that identifies a range of elements in the specified container. To do so, your application must first create a range descriptor record. The Apple Event Manager uses a range descriptor record to identify the two Apple event objects that specify the beginning and end of a range of elements.

For example, an object specifier record for a range of text in a document could specify the table named "Summary of Sales" as the first boundary object and the figure named "Best-Selling Widgets for 1991" as the second boundary object for a range that consists of all the text between the table and the figure. Any word processor that keeps track of the relative positions of text, tables, and figures should be capable of supporting such a request.

Table 6-10 summarizes the logical organization of the data for the object specifier record that specifies this range. For general information about the organization of data within a range descriptor record, see "Key Data for a Range" on page 6-20.

Table 6-10 A range descriptor record

Keyword	**Descriptor type**	**Data**
`keyAERangeStart`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cTable`
`keyAEContainer`	`typeCurrentContainer`	Data handle is `NIL`
`keyAEKeyForm`	`typeEnumerated`	`formName`
`keyAEKeyData`	`typeChar`	`"Summary of Sales"`
`keyAERangeStop`	`typeObjectSpecifier`	(see indented record)
`keyAEDesiredClass`	`typeType`	`cFigure`
`keyAEContainer`	`typeCurrentContainer`	Data handle is `NIL`
`keyAEKeyForm`	`typeEnumerated`	`formName`
`keyAEKeyData`	`typeChar`	`"Best-Selling Widgets for 1991"`

You can use the `CreateRangeDescriptor` function to create a range descriptor record, or you can create an AE record and use `AECoerceDesc` to coerce it to a range descriptor record. Listing 6-20 provides an example of an application-defined routine that creates two object specifier records, then uses the `CreateRangeDescriptor` function to add them to a range descriptor record.

The container for the boundary objects in the range descriptor record created by Listing 6-20 is the same as the container for the range itself. The object specifier record for the range's container is added to an object specifier record of key form `formRange` at the same time that the range descriptor record is added as key data. The container for the two boundary objects can therefore be specified in the range descriptor record by a descriptor record of type `typeCurrentContainer` whose data handle has the value `NIL`. The Apple Event Manager interprets this as a placeholder for the range's container when it is resolving the range descriptor record.

Listing 6-20 Creating a range descriptor record

```
FUNCTION MyCreateRangeDescriptor (VAR rangeDescRec: AEDesc): OSErr;
VAR
   rangeStart:          AEDesc;
   rangeEnd:            AEDesc;
   currentContainer:    AEDesc;
   tableNameDescRec:    AEDesc;
   figureNameDescRec:   AEDesc;
   myErr:               OSErr;
BEGIN
{create the object specifier record for the start of the range }
{ (the table named 'Summary of Sales' in 'MyDoc' document)}

   {create a descriptor record of type typeCurrentContainer}
   myErr := AECreateDesc(typeCurrentContainer, NIL, 0, currentContainer);

   {create the object specifier record}
   IF myErr = noErr THEN
      myErr := MyCreateNameDescRec(tableNameDescRec,
                                    'Summary of Sales');
   IF myErr = noErr THEN
   myErr := CreateObjSpecifier(cTable, currentContainer, formName,
                                 tableNameDescRec, FALSE, rangeStart);

   myErr := AEDisposeDesc(tableNameDescRec);
   {create the object specifier record for the end of the range }
   { (the figure named 'Best-Selling Widgets...' in 'MyDoc') }
   IF myErr = noErr THEN
      myErr := MyCreateNameDescRec(figureNameDescRec,
                                    'Best-Selling Widgets for 1991');

   IF myErr = noErr THEN
      myErr := CreateObjSpecifier(cFigure, currentContainer, formName,
                                 figureNameDescRec, TRUE, rangeEnd);

   {now create the range descriptor record}
   IF myErr = noErr THEN
      myErr := CreateRangeDescriptor(rangeStart, rangeEnd, TRUE,
                                       rangeDescRec);
   MyCreateRangeDescriptor := myErr;
END;
```

After creating a descriptor record of type `typeCurrentContainer` and a descriptor record for the first table's name, Listing 6-20 uses the `CreateObjSpecifier` function to create an object specifier record that identifies the beginning of the range. The parameters to `CreateObjSpecifier` specify that the beginning of the range is an Apple event object of the object class `cTable` in the current container, with a key form of `formName` and key data that identifies the table by name. A second call to `CreateObjSpecifier` creates the object specifier record that identifies the end of the range—an Apple event object of the `cFigure` object class in the current container, with a key form of `formName` and key data that identifies the figure by name. Finally, the code in Listing 6-20 uses the `CreateRangeDescriptor` function to create the range descriptor record, using the two previously created object specifier records to specify the beginning and end of the range.

Reference to Resolving and Creating Object Specifier Records

This section describes the Apple Event Manager routines your application can use to resolve and create object specifier records. It also describes application-defined object accessor functions and object callback functions that your application can provide for use by the Apple Event Manager in resolving object specifier records.

The first section, "Data Structures Used in Object Specifier Records," summarizes the descriptor types and associated data that can be used in an object specifier record. "Routines for Resolving and Creating Object Specifier Records," which begins on page 6-77, describes the Apple Event Manager routines you use to initialize the Object Support Library, resolve object specifier records, set and manipulate object accessor functions, deallocate memory for tokens, and create object specifier records. "Application-Defined Routines," which begins on page 6-94, describes the object accessor functions and object callback functions that a server application can provide.

Data Structures Used in Object Specifier Records

The data for object specifier records can be specified using a variety of descriptor records and descriptor types. These are described in detail in "Descriptor Records Used in Object Specifier Records," which begins on page 6-8, and summarized in Table 6-11.

Table 6-11 Keyword-specified descriptor records for `typeObjectSpecifier`

Keyword	Descriptor type	Data
`keyAEDesiredClass`	`typeType`	Object class ID
`keyAEContainer`	`typeObjectSpecifier`	Object specifier record
	`typeNull`	Data handle is `NIL`. Specifies the default container at the top of the container hierarchy.
	`typeObjectBeingExamined`	Data handle is `NIL`. Specifies the container for elements that are tested one at a time; used only with `formTest`.
	`typeCurrentContainer`	Data handle is `NIL`. Specifies a container for an element that demarcates one boundary in a range. Used only with `formRange`.

`keyAEKeyForm`	`typeEnumerated`	`formPropertyID`
		`formName`
		`formUniqueID`
		`formAbsolutePosition`
		`formRelativePosition`
		`formTest`
		`formRange`
		`formWhose`

`keyAEKeyData`		(See indented key forms)
for `formPropertyID`	`typeType`	Property ID for an element's property
for `formName`	`typeChar` or other text type	Element's name
for `formUniqueID`	Any appropriate type	Element's unique ID
for `formAbsolutePosition`	`typeLongInteger`	Offset from beginning (positive) or end (negative) of container

	`typeAbsoluteOrdinal`	`kAEFirst`
		`kAEMiddle`
		`kAELast`
		`kAEAny`
		`kAEAll`

for `formRelativePosition`	`typeEnumerated`	`kAENext` `kAEPrevious`
for `formTest`	`typeCompDescriptor`	(See Table 6-2 on page 6-16)
	`typeLogicalDescriptor`	(See Table 6-3 on page 6-17)
for `formRange`	`typeRangeDescriptor`	(See Table 6-4 on page 6-20)
for `formWhose`	`typeWhoseDescriptor`	(See Table 6-5 on page 6-42)

Routines for Resolving and Creating Object Specifier Records

This section describes routines for initializing the Object Support Library, resolving object specifier records, setting and manipulating object accessor functions, deallocating memory for tokens, and creating object specifier records.

Initializing the Object Support Library

You should call the `AEObjectInit` function to initialize the Apple Event Manager routines that handle object specifier records and Apple event objects. To make these routines available to your application with version 1.01 and earlier versions of the Apple Event Manager, you must also link the Apple Event Object Support Library with your application when you build it.

AEObjectInit

You use the `AEObjectInit` function to initialize the Object Support Library.

```
FUNCTION AEObjectInit: OSErr;
```

DESCRIPTION

You must call this function before calling any of the Apple Event Manager routines that describe or manipulate Apple event objects.

RESULT CODES

`noErr`	0	No error occurred
`memFullErr`	–108	Not enough room in heap zone
`errAENewerVersion`	–1706	Need a newer version of the Apple Event Manager

Setting Object Accessor Functions and Object Callback Functions

The Apple Event Manager provides two routines that allow you to specify the object accessor functions and object callback functions provided by your application. The `AEInstallObjectAccessor` function adds an entry for an object accessor function to either the application's object accessor dispatch table or the system object accessor dispatch table. The `AESetObjectCallbacks` function allows you to specify the object callback functions to be called for your application.

AEInstallObjectAccessor

You can use the `AEInstallObjectAccessor` function to add an entry for an object accessor function to either the application's object accessor dispatch table or the system object accessor dispatch table.

```
FUNCTION AEInstallObjectAccessor (desiredClass: DescType;
                                  containerType: DescType;
                                  theAccessor: AccessorProcPtr;
                                  accessorRefcon: LongInt;
                                  isSysHandler: Boolean): OSErr;
```

`desiredClass`
: The object class of the Apple event objects to be located by the object accessor function for this table entry.

`containerType`
: The descriptor type of the token used to specify the container for the desired objects. The object accessor function finds objects in containers specified by tokens of this type.

`theAccessor`
: A pointer to the object accessor function for this table entry. Note that an object accessor function listed in the system dispatch table must reside in the system heap; thus, if the value of the `isSysHandler` parameter is `TRUE`, the `theAccessor` parameter should point to a location in the system heap. Otherwise, if you put your system object accessor function in your application heap, you must call `AERemoveObjectAccessor` to remove the function before your application terminates.

`accessorRefcon`
: A reference constant passed by the Apple Event Manager to the object accessor function whenever the function is called. If your object accessor function doesn't use a reference constant, use 0 as the value of this parameter. To change the value of the reference constant, you must call `AEInstallObjectAccessor` again.

`isSysHandler`
: A value that specifies the object accessor dispatch table to which the entry is added. If the value of `isSysHandler` is `TRUE`, the Apple Event Manager adds the entry to the system object accessor dispatch table. Entries in the system object accessor dispatch table are available to all applications running on the same computer. If the value is `FALSE`, the Apple Event Manager adds the entry to your application's object accessor table. When searching for object accessor functions, the Apple Event Manager searches the application's object accessor dispatch table first; it searches the system object accessor dispatch table only if the necessary function is not found in your application's object accessor dispatch table.

DESCRIPTION

The `AEInstallObjectAccessor` function adds an entry to either the application or system object accessor dispatch table. You must supply parameters that specify the object class of the Apple event objects that the object accessor function can locate, the descriptor type of tokens for containers in which the object accessor function can locate objects, the address of the object accessor function for which you are adding an entry, and whether the entry is to be added to the system object accessor dispatch table or your application's object accessor dispatch table. You can also specify a reference constant that the Apple Event Manager passes to your object accessor function each time the Apple Event Manager calls the function.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	The handler pointer is `NIL` or odd, or `AEObjectInit` was not called before this function

SEE ALSO

For more information about installing object accessor functions, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

For a description of the `AERemoveObjectAccessor` function, see page 6-84.

AESetObjectCallbacks

You can use the `AESetObjectCallbacks` function to specify the object callback functions to be called for your application.

```
FUNCTION AESetObjectCallbacks (myCompareProc, myCountProc,
                               myDisposeTokenProc,
                               myGetMarkTokenProc, myMarkProc,
                               myAdjustMarksProc,
                               myGetErrDescProc: ProcPtr): OSErr;
```

`myCompareProc`
Either a pointer to the object-comparison function provided by your application or `NIL` if no function is provided.

`myCountProc`
Either a pointer to the object-counting function provided by your application or `NIL` if no function is provided.

`myDisposeTokenProc`
Either a pointer to the token disposal function provided by your application or `NIL` if no function is provided.

`myGetMarkTokenProc`
Either a pointer to the function for returning a mark token provided by your application or `NIL` if no function is provided.

`myMarkProc` Either a pointer to the object-marking function provided by your application or `NIL` if no function is provided.

`myAdjustMarksProc`
Either a pointer to the mark-adjusting function provided by your application or `NIL` if no function is provided.

`myGetErrDescProc`
Either a pointer to the error callback function provided by your application or `NIL` if no function is provided.

DESCRIPTION

Your application can provide only one each of the object callback functions specified by `AESetObjectCallbacks`: one object-comparison function, one object-counting function, and so on. As a result, each of these callback functions must perform the requested task (comparing, counting, and so on) for all the object classes that your application supports. In contrast, your application may provide many different object accessor functions if necessary, depending on the object classes and token types your application supports.

To replace object callback routines that have been previously installed, you can make another call to `AESetObjectCallbacks`. Each additional call to `AESetObjectCallbacks` replaces any object callback functions installed by previous calls to `AESetObjectCallbacks`. You cannot use `AESetObjectCallbacks` to replace system object callback routines or object accessor functions. Only those routines you specify are replaced; to avoid replacing existing callback functions, specify a value of `NIL` for the functions you don't want to replace.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	The handler pointer is `NIL` or odd, or `AEObjectInit` was not called before this function
`memFullErr`	–108	There is not enough room in heap zone
`errAENotASpecialFunction`	–1714	The keyword is not valid for a special function

SEE ALSO

For information about writing object callback functions, see "Application-Defined Routines," which begins on page 6-94.

To install system object callback functions, use the `AEInstallSpecialHandler` function described on page 4-100.

Getting, Calling, and Removing Object Accessor Functions

The Apple Event Manager provides three functions that allow you to get, call, and remove object accessor functions that you have installed in either the system or application object accessor dispatch table with the `AEInstallObjectAccessor` function. The `AEGetObjectAccessor` and `AECallObjectAccessor` functions get and call object accessor functions installed in the dispatch table you specify, and `AERemoveObjectAccessor` removes an installed function.

AEGetObjectAccessor

You can use the `AEGetObjectAccessor` function to get a pointer to an object accessor function and the value of its reference constant.

```
FUNCTION AEGetObjectAccessor (desiredClass: DescType;
                              containerType: DescType;
                              VAR theAccessor: AccessorProcPtr;
                              VAR accessorRefcon: LongInt;
                              isSysHandler: Boolean): OSErr;
```

`desiredClass`
: The object class of the Apple event objects located by the requested object accessor function. This parameter can also contain the constant `typeWildCard` or the constant `cProperty`.

`containerType`
: The descriptor type of the token that identifies the container for the objects located by the requested object accessor function. This parameter can also contain the constant `typeWildCard`.

`theAccessor`
: The `AEGetObjectAccessor` function returns a pointer to the requested object accessor function in this parameter.

`accessorRefcon`
: The `AEGetObjectAccessor` function returns the reference constant from the object accessor dispatch table entry for the specified object accessor function in this parameter.

`isSysHandler`
: A value that specifies the object accessor table from which to get the object accessor function and its reference constant. If the value of `isSysHandler` is `TRUE`, `AEGetObjectAccessor` gets the function from the system object accessor dispatch table. If the value of `isSysHandler` is `FALSE`, `AEGetObjectAccessor` gets the function from the application's object accessor dispatch table.

DESCRIPTION

The `AEGetObjectAccessor` function returns a pointer to the object accessor function installed for the object class specified in the `desiredClass` parameter and the descriptor type specified in the `containerType` parameter. It also returns the reference constant associated with the specified function. You must supply a value in the `isSysHandler` parameter that specifies which object accessor dispatch table you want to get the function from.

Calling `AEGetObjectAccessor` does not remove the object accessor function from an object accessor dispatch table.

To get an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the object class, you must specify `typeWildCard` as the value of the `desiredClass` parameter. Similarly, to get an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the descriptor type of the token used to specify the container, you must specify `typeWildCard` as the value of the `containerType` parameter.

To get an object accessor function whose entry in an object accessor dispatch table specifies `cProperty` (a constant used to specify a property of any object class), you must specify `cProperty` as the `desiredClass` parameter.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	`AEObjectInit` was not called before this function was called
`errAEAccessorNotFound`	–1723	There is no object accessor function for the specified object class and container type

AECallObjectAccessor

You can use the `AECallObjectAccessor` function to invoke one of your application's object accessor functions.

```
FUNCTION AECallObjectAccessor (desiredClass: DescType;
                               containerToken: AEDesc;
                               containerClass: DescType;
                               keyForm: DescType;
                               keyData: AEDesc;
                               VAR theToken: AEDesc): OSErr;
```

`desiredClass`
The object class of the desired Apple event objects.

`containerToken`	The token that identifies the container for the desired objects.
`containerClass`	The object class of the container for the desired objects.
`keyForm`	The key form specified by the object specifier record for the object or objects to be located.
`keyData`	The key data specified by the object specifier record for the object or objects to be located.
`theToken`	The object accessor function that is invoked returns a token specifying the desired object or objects in this parameter.

DESCRIPTION

If you want your application to do some of the Apple event object resolution normally performed by the `AEResolve` function, you can use `AECallObjectAccessor` to invoke an object accessor function. This might be useful, for example, if you have installed an object accessor function using `typeWildCard` for the `AEInstallObjectAccessor` function's `desiredClass` parameter and `typeAEList` for the `containerType` parameter. To return a list of tokens for a request like "every line that ends in a period," the object accessor function can create an empty list, then call `AECallObjectAccessor` for each requested element, adding tokens for each element to the list one at a time.

The parameters of `AECallObjectAccessor` are identical to the parameters of an object accessor function, with one exception: the parameter that specifies the reference constant passed to the object accessor function whenever it is called is added by the Apple Event Manager when it calls the object accessor function.

To call an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the object class, you must specify `typeWildCard` as the value of the `desiredClass` parameter.

To call an object accessor function whose entry in an object accessor dispatch table specifies `cProperty`, you must specify `cProperty` as the `desiredClass` parameter.

RESULT CODES

In addition to the following result codes, `AECallObjectAccessor` returns any other result codes returned by the object accessor function that is called.

`noErr`	0	No error occurred
`paramErr`	–50	`AEObjectInit` was not called before this function was called
`errAEAccessorNotFound`	–1723	No object accessor was found

AERemoveObjectAccessor

You can use the `AERemoveObjectAccessor` function to remove an object accessor function from an object accessor dispatch table.

```
FUNCTION AERemoveObjectAccessor (desiredClass: DescType;
                                 containerType: DescType;
                                 theAccessor: AccessorProcPtr;
                                 isSysHandler: Boolean): OSErr;
```

`desiredClass`
: The object class of the Apple event objects located by the object accessor function. The `desiredClass` parameter can also contain the constant `typeWildCard` or the constant `cProperty`.

`containerType`
: The descriptor type of the token that identifies the container for the objects located by the object accessor function. The `containerType` parameter can also contain the constant `typeWildCard`.

`theAccessor`
: A pointer to the object accessor function you want to remove. Although the parameters `desiredClass` and `containerType` would be sufficient to identify the function to be removed, providing the parameter `theAccessor` guarantees that you remove the correct function. If this parameter does not contain a pointer to the object accessor function you want to remove, its value should be `NIL`.

`isSysHandler`
: A value that specifies the object accessor dispatch table from which to remove the object accessor function. If the value of `isSysHandler` is `TRUE`, `AEGetObjectAccessor` removes the routine from the system object accessor dispatch table. If the value is `FALSE`, `AEGetObjectAccessor` removes the routine from the application object accessor dispatch table.

DESCRIPTION

The `AERemoveObjectAccessor` function removes the object accessor function you have installed for the object class specified in the `desiredClass` parameter and the descriptor type specified in the `containerType` parameter.

To remove an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the object class, you must specify `typeWildCard` as the value of the `desiredClass` parameter. Similarly, to remove an object accessor function whose entry in an object accessor dispatch table specifies `typeWildCard` as the descriptor type of the token used to specify the container for the desired objects, you must specify `typeWildCard` as the value of the `containerType` parameter.

To remove an object accessor function whose entry in an object accessor dispatch table specifies `cProperty` (a constant used to specify a property of any object class), you must specify `cProperty` as the `desiredClass` parameter.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	`AEObjectInit` was not called before this function was called
`errAEAccessorNotFound`	–1723	There is no object accessor function for the specified object class and container type

Resolving Object Specifier Records

If an Apple event parameter consists of an object specifier record, your handler for the event typically calls the `AEResolve` function to begin the process of resolving the object specifier record.

AEResolve

You can use the `AEResolve` function to resolve an object specifier record in an Apple event parameter.

```
FUNCTION AEResolve (objectSpecifier: AEDesc;
                    callbackFlags: Integer;
                    VAR theToken: AEDesc): OSErr;
```

`objectSpecifier`
The object specifier record to be resolved.

`callbackFlags`
A value that determines what additional assistance, if any, your application can give the Apple Event Manager when it parses the object specifier record. The value is specified by adding the following constants, as appropriate:

```
CONST kAEIDoMinimum  = $0000;    {supports minimum }
                                 { callbacks only}
      kAEIDoWhose    = $0001;    {supports formWhose}
      kAEIDoMarking  = $0004;    {provides marking }
                                 { functions}
```

`theToken`	The `AEResolve` function returns, in this parameter, a token that identifies the Apple event objects specified by the `objectSpecifier` parameter. Your object accessor functions may need to create many tokens to resolve a single object specifier record; this parameter contains only the final token that identifies the requested Apple event object. If an error occurs, `AEResolve` returns a null descriptor record.

DESCRIPTION

The `AEResolve` function resolves the object specifier record passed in the `objectSpecifier` parameter with the help of the object accessor functions and object callback functions provided by your application.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	`AEObjectInit` was not called before this function was called
`errAEHandlerNotFound`	–1717	The necessary object callback function was not found (this result is returned only for object callback functions; `errAEAccessorNotFound` [–1723] is returned when an object accessor function is not found)
`errAEImpossibleRange`	–1720	The range is not valid because it is impossible for a range to include the first and last objects that were specified; an example is a range in which the offset of the first object is greater than the offset of the last object
`errAEWrongNumberArgs`	–1721	The number of operands provided for the `kAENOT` logical operator is not 1
`errAEAccessorNotFound`	–1723	There is no object accessor function for the specified object class and token descriptor type
`errAENoSuchLogical`	–1725	The logical operator in a logical descriptor record is not `kAEAND`, `kAEOR`, or `kAENOT`
`errAEBadTestKey`	–1726	The descriptor record in a test key is neither a comparison descriptor record nor a logical descriptor record
`errAENotAnObjectSpec`	–1727	The `objSpecifier` parameter of `AEResolve` is not an object specifier record
`errAENegativeCount`	–1729	An object-counting function returned a negative result
`errAEEmptyListContainer`	–1730	The container for an Apple event object is specified by an empty list

In addition to the result codes listed here, `AEResolve` also returns any result code returned by one of your application's object accessor functions or object callback functions. For example, an object accessor function can return `errAENoSuchObject` (–1728) when it can't find an Apple event object, or it can return more specific result codes.

If any object accessor function or object callback function returns a result code other than `noErr` or `errAEEventNotHandled`, `AEResolve` immediately disposes of any existing tokens and returns. The result code it returns in this case is the result code returned by the object accessor function or the object callback function.

SEE ALSO

For an overview of the way `AEResolve` works with object accessor functions, see "Resolving Object Specifier Records," which begins on page 6-4.

Deallocating Memory for Tokens

Whenever the `AEResolve` function returns a final token to your event handler as the result of the resolution of an object specifier record passed to `AEResolve`, your application can call the `AEDisposeToken` function to deallocate the memory used by the token.

AEDisposeToken

You can use the `AEDisposeToken` function to deallocate the memory used by a token.

```
FUNCTION AEDisposeToken (VAR theToken: AEDesc): OSErr;
```

`theToken`	The token to be disposed of.

DESCRIPTION

When your application calls the `AEDisposeToken` function, the Apple Event Manager first calls your application's token disposal function, if you have provided one. If you haven't provided a token disposal function, or if your application's token disposal function returns `errAEEventNotHandled` as the function result, the Apple Event Manager calls the system token disposal function if one is available. If there is no system token disposal function or the function returns `errAEEventNotHandled` as the function result, the Apple Event Manager calls the `AEDisposeDesc` function to dispose of the token.

RESULT CODES

In addition to the following result codes, `AEDisposeToken` also returns result codes returned by the token disposal function that disposed of the token.

`noErr`	0	No error occurred
`paramErr`	–50	`AEObjectInit` was not called before this function was called
`notASpecialFunction`	–1714	The keyword is not valid for a special function

SEE ALSO

For information about writing a token disposal function, see page 6-99.

Creating Object Specifier Records

The Apple Event Manager provides five functions that you can use to create some of the components of an object specifier record or to assemble an object specifier record:

- The `CreateOffsetDescriptor` function creates an offset descriptor record, which specifies the position of an element in relation to the beginning or end of its container.
- The `CreateCompDescriptor` function creates a comparison descriptor record, which specifies how to compare one or more Apple event objects with either another Apple event object or a descriptor record.
- The `CreateLogicalDescriptor` function creates a logical descriptor record, which specifies a logical operator and one or more logical terms for the Apple Event Manager to evaluate.
- The `CreateRangeDescriptor` function creates a range descriptor record, which specifies a series of consecutive elements in the same container.
- The `CreateObjSpecifier` function assembles an object specifier record, which identifies one or more Apple event objects, from other descriptor records.

Instead of using these functions, you can create the corresponding descriptor records yourself using the `AECreateDesc` function, add them to an AE record using other Apple Event Manager routines, and coerce the AE record to a descriptor record of type `typeObjectSpecifier`. However, in most cases it is easier to use the functions listed in this section.

All of these functions except for `CreateOffsetDescriptor` include a `disposeInputs` parameter. If the value of this parameter is `TRUE`, the function automatically disposes of any descriptor records you have provided as parameters to the function. If the value is `FALSE`, the application must dispose of the records itself. A value of `FALSE` may be more efficient for some applications because it allows them to reuse descriptor records.

For more information about these functions and examples of their use, see “Creating Object Specifier Records,” which begins on page 6-55.

CreateOffsetDescriptor

You can use the `CreateOffsetDescriptor` function to create an offset descriptor record.

```
FUNCTION CreateOffsetDescriptor (theOffset: LongInt;
                                 VAR theDescriptor: AEDesc)
                                 : OSErr;
```

`theOffset`	A positive integer that specifies the offset from the beginning of the container (the first element has an offset of 1), or a negative integer that specifies the offset from the end (the last element has an offset of –1).
`theDescriptor`	The offset descriptor record created by `CreateOffsetDescriptor`.

DESCRIPTION

The `CreateOffsetDescriptor` function creates an offset descriptor record that specifies the position of an element in relation to the beginning or end of its container.

RESULT CODES

`noErr`	0	No error occurred
`memFullErr`	–108	Not enough room in heap zone

CreateCompDescriptor

You can use the `CreateCompDescriptor` function to create a comparison descriptor record.

```
FUNCTION CreateCompDescriptor (comparisonOperator: DescType;
                               VAR operand1: AEDesc;
                               VAR operand2: AEDesc;
                               disposeInputs: Boolean;
                               VAR theDescriptor: AEDesc)
                               : OSErr;
```

`comparisonOperator`	The comparison operator for comparing the descriptor records in the `operand1` and `operand2` parameters. The operator is specified by the constants listed in the description that follows.
`operand1`	An object specifier record.
`operand2`	A descriptor record (which can be an object specifier record or any other descriptor record) whose value is to be compared to the value of `operand1`.
`disposeInputs`	A Boolean value indicating whether the function (`TRUE`) or your application (`FALSE`) should dispose of the descriptor records for the two operands.
`theDescriptor`	The comparison descriptor record created by `CreateCompDescriptor`.

DESCRIPTION

The `CreateCompDescriptor` function creates a comparison descriptor record, which specifies how to compare one or more Apple event objects with either another Apple event object or a descriptor record.

The actual comparison of the two operands is performed by the object-comparison function provided by the client application. The way a comparison operator is interpreted is up to each application.

These are the currently defined standard comparison operators:

Constant	**Meaning**
`kAEGreaterThan`	The value of `operand1` is greater than the value of `operand2`.
`kAEGreaterThanEquals`	The value of `operand1` is greater than or equal to the value of `operand2`.
`kAEEquals`	The value of `operand1` is equal to the value of `operand2`.
`kAELessThan`	The value of `operand1` is less than the value of `operand2`.
`kAELessThanEquals`	The value of `operand1` is less than or equal to the value of `operand2`.
`kAEBeginsWith`	The value of `operand1` begins with the value of `operand2` (for example, the string `"operand"` begins with the string `"opera"`).
`kAEEndsWith`	The value of `operand1` ends with the value of `operand2` (for example, the string `"operand"` ends with the string `"and"`).
`kAEContains`	The value of `operand1` contains the value of `operand2` (for example, the string `"operand"` contains the string `"era"`).

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested Apple event data type
`errAEWrongDataType`	–1703	Wrong Apple event data type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

SEE ALSO

For an example of how to use the `CreateCompDescriptor` function to create a comparison descriptor record, see “Specifying a Test,” which begins on page 6-64.

CreateLogicalDescriptor

You can use the `CreateLogicalDescriptor` function to create a logical descriptor record.

```
FUNCTION CreateLogicalDescriptor
                              (VAR theLogicalTerms: AEDescList;
                               theLogicOperator: DescType;
                               disposeInputs: Boolean;
                               VAR theDescriptor: AEDesc): OSErr;
```

`theLogicalTerms`
: A list containing comparison descriptor records, logical descriptor records, or both. If the value of the parameter `theLogicOperator` is `kAEAND` or `kAEOR`, the list can contain any number of descriptors. If the value of the parameter `theLogicOperator` is `kAENOT`, logically this list should contain a single descriptor record. However, the function will not return an error if the list contains more than one descriptor record for a logical operator of `kAENOT`.

`theLogicOperator`
: A logical operator represented by one of the following constants:

```
CONST  kAEAND = 'AND ';
       kAEOR  = 'OR  ';
       kAENOT = 'NOT ';
```

`disposeInputs`
: A Boolean value indicating whether the function (`TRUE`) or your application (`FALSE`) should dispose of the descriptor records in the other parameters.

`theDescriptor`
: The logical descriptor record created by `CreateLogicalDescriptor`.

DESCRIPTION

The `CreateLogicalDescriptor` function creates a logical descriptor record, which specifies a logical operator and one or more logical terms for the Apple Event Manager to evaluate.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to requested Apple event data type
`errAEWrongDataType`	–1703	Wrong Apple event data type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

SEE ALSO

For an example of how to use the `CreateLogicalDescriptor` function to create a logical descriptor record, see "Specifying a Test," which begins on page 6-64.

CreateRangeDescriptor

You can use the `CreateRangeDescriptor` function to create a range descriptor record.

```
FUNCTION CreateRangeDescriptor (VAR rangeStart: AEDesc;
                                VAR rangeStop: AEDesc;
                                disposeInputs: Boolean;
                                VAR theDescriptor: AEDesc): OSErr;
```

`rangeStart`
: An object specifier record that identifies the first Apple event object in the range.

`rangeStop` An object specifier record that identifies the last Apple event object in the range.

`disposeInputs`
: A Boolean value indicating whether the function (`TRUE`) or your application (`FALSE`) should dispose of the descriptor records for the `rangeStart` and `rangeStop` parameters.

`theDescriptor`
: The range descriptor record created by `CreateRangeDescriptor`.

DESCRIPTION

The `CreateRangeDescriptor` function creates a range descriptor record, which specifies a series of consecutive elements in the same container. Although the `rangeStart` and `rangeStop` parameters can be any object specifier records—including object specifier records that specify more than one Apple event object—most applications expect these parameters to specify single Apple event objects.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested Apple event data type
`errAEWrongDataType`	–1703	Wrong Apple event data type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

SEE ALSO

For an example of how to use the `CreateRangeDescriptor` function to create a range descriptor record, see "Specifying a Range" on page 6-72.

CreateObjSpecifier

You can use the `CreateObjSpecifier` function to create an object specifier record.

```
FUNCTION CreateObjSpecifier (desiredClass: DescType;
                             VAR theContainer: AEDesc;
                             keyForm: DescType;
                             VAR keyData: AEDesc;
                             disposeInputs: Boolean;
                             VAR objSpecifier: AEDesc): OSErr;
```

`desiredClass`
The object class of the desired Apple event objects.

`theContainer`
A description of the container for the requested object, usually in the form of another object specifier record.

`keyForm` The key form for the object specifier record.

`keyData` The key data for the object specifier record.

`disposeInputs`
A Boolean value indicating whether the function (`TRUE`) or your application (`FALSE`) should dispose of the descriptor records for the other parameters.

`objSpecifier`
The object specifier record created by the `CreateObjSpecifier` function.

DESCRIPTION

The `CreateObjSpecifier` function assembles an object specifier record from the specified constants and other descriptor records.

RESULT CODES

`noErr`	0	No error occurred
`paramErr`	–50	Error in parameter list
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested Apple event data type
`errAEWrongDataType`	–1703	Wrong Apple event data type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

SEE ALSO

For information about how to assemble the components of an object specifier record with the `CreateObjSpecifier` function, see "Creating Object Specifier Records," which begins on page 6-55.

Application-Defined Routines

The `AEResolve` function performs tasks that are required to resolve any object specifier record, such as parsing its contents, keeping track of the results of tests, and handling memory management. When necessary, `AEResolve` 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 or counting the number of Apple event objects in a container.

`AEResolve` can call two kinds of application-defined functions:

- *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. You can provide up to seven object callback functions, depending on the needs of your application.

This section provides model declarations for the object accessor functions and object callback functions that your application can provide.

Object Accessor Functions

You must provide one or more object accessor functions that can locate all the element classes and properties listed in the *Apple Event Registry: Standard Suites* for the object classes supported by your application. This section provides the routine declaration for an object accessor function.

MyObjectAccessor

Object accessor functions locate Apple event objects of a specified object class in a container identified by a token of a specified descriptor type.

```
FUNCTION MyObjectAccessor (desiredClass: DescType;
                           containerToken: AEDesc;
                           containerClass: DescType;
                           keyForm: DescType; keyData: AEDesc;
                           VAR theToken: AEDesc;
                           theRefcon: LongInt): OSErr;
```

`desiredClass`	The object class of the desired Apple event objects.
`containerToken`	A token that specifies the container of the desired Apple event objects.
`containerClass`	The object class of the container.
`keyForm`	The key form specified by the object specifier record being resolved.
`keyData`	The key data specified by the object specifier record being resolved.
`theToken`	The token returned by the `MyObjectAccessor` function.
`theRefcon`	A reference constant that the Apple Event Manager passes to the object accessor function each time it is called.

DESCRIPTION

Each object accessor function provided by your application should either find elements of a specified object class or find properties of an Apple event object. The `AEResolve` function uses the object class ID of the specified Apple event object and the descriptor type of the token that identifies the object's container to determine which object accessor function to call. To install an object accessor function either in your application's object accessor dispatch table or in the system object accessor dispatch table, use the `AEInstallObjectAccessor` function, which is described on page 6-78.

SPECIAL CONSIDERATIONS

If the Apple Event Manager receives the result code `errAEEventNotHandled` after calling an object accessor function, it attempts to use other methods of locating the requested objects, such as calling an equivalent system object accessor function. Thus, an object accessor function that can't locate a requested object should return `errAEEventNotHandled`. This allows the Apple Event Manager to try other object accessor functions that may be available.

RESULT CODES

`noErr`	0	No error occurred
`errAEEventNotHandled`	–1708	The object accessor function is unable to locate the requested Apple event object or objects

SEE ALSO

For information about installing object accessor functions, see "Installing Entries in the Object Accessor Dispatch Tables," which begins on page 6-21.

For information about writing object accessor functions, see "Writing Object Accessor Functions," which begins on page 6-28.

Object Callback Functions

If an Apple event parameter consists of an object specifier record, your handler for the Apple event typically calls `AEResolve` to begin the process of locating the requested Apple event objects. The `AEResolve` function in turn calls object accessor functions and, if necessary, object callback functions provided by your application when it needs the information they can provide.

This section provides declarations for the seven object callback functions that your application can provide: the object-counting function (`MyCountObjects`), object-comparison function (`MyCompareObjects`), token disposal function (`MyDisposeToken`), error callback function (`MyGetErrorDesc`), mark token function (`MyGetMarkToken`), object-marking function (`MyMark`), and mark-adjusting function (`MyAdjustMarks`).

For information about writing and installing object callback functions, see "Writing Object Callback Functions," which begins on page 6-45.

MyCountObjects

If you want the Apple Event Manager to help your application resolve object specifier records of key form `formTest` (and if your application doesn't specify `kAEIDoWhose` as described on page 6-48), you should provide an object-counting function and an object-comparison function. An object-counting function counts the number of Apple event objects of a specified class in a specified container.

```
FUNCTION MyCountObjects (desiredClass: DescType;
                         containerClass: DescType;
                         theContainer: AEDesc;
                         VAR result: LongInt): OSErr;
```

`desiredClass`
The object class of the Apple event objects to be counted.

`containerClass`
The object class of the container for the Apple event objects to be counted.

`theContainer`
A token that identifies the container for the Apple event objects to be counted.

`result` Your object-counting function should return in this parameter the number of Apple objects of the specified class in the specified container.

DESCRIPTION

The Apple Event Manager calls your object-counting function when, in the course of resolving an object specifier record, the manager requires a count of the number of Apple event objects of a given class in a given container.

SPECIAL CONSIDERATIONS

If the Apple Event Manager receives the result code `errAEEventNotHandled` after calling an object-counting function, it attempts to use other methods of counting the specified objects, such as calling an equivalent system object-counting function. Thus, an object-counting function that can't count the specified objects should return `errAEEventNotHandled`. This allows the Apple Event Manager to try other object-counting functions that may be available.

RESULT CODES

`noErr`	0	No error occurred
`errAEEventNotHandled`	–1708	The object-counting function is unable to count the specified Apple event objects

SEE ALSO

For more information, see "Writing an Object-Counting Function" on page 6-48.

MyCompareObjects

If you want the Apple Event Manager to help your application resolve object specifier records of key form `formTest` (and if your application doesn't specify `kAEIDoWhose` as described on page 6-48), you should provide an object-counting function and an object-comparison function. After comparing one Apple event object to another or to the data for a descriptor record, an object-comparison function should return `TRUE` or `FALSE` in the `result` parameter.

```
FUNCTION MyCompareObjects (comparisonOperator: DescType;
                           object: AEDesc;
                           objectOrDescToCompare: AEDesc;
                           VAR result: Boolean): OSErr;
```

`comparisonOperator`	The comparison operator. See the description of `CreateCompDescriptor` on page 6-89 for standard comparison operators at the time of publication of this book. The current version of the *Apple Event Registry: Standard Suites* lists all the constants for comparison operators.
`object`	A token.
`objectOrDescToCompare`	A token or some other descriptor record that specifies either an Apple event object or a value to compare to the Apple event object specified by the `object` parameter.
`result`	Your object-comparison function should return, in this parameter, a Boolean value that indicates whether the values of the `object` and `objectOrDescToCompare` parameters have the relationship specified by the `comparisonOperator` parameter (`TRUE`) or not (`FALSE`).

DESCRIPTION

The Apple Event Manager calls your object-comparison function when, in the course of resolving an object specifier record, the manager needs to compare an Apple event object with another or with a value.

It is up to your application to interpret the comparison operators it receives. The meaning of comparison operators differs according to the Apple event objects being compared, and not all comparison operators apply to all object classes.

SPECIAL CONSIDERATIONS

If the Apple Event Manager receives the result code `errAEEventNotHandled` after calling an object-comparison function, it attempts to use other methods of comparison, such as calling an equivalent system object-comparison function. Thus, an object-comparison function that can't perform a requested comparison should return `errAEEventNotHandled`. This allows the Apple Event Manager to try other object-comparison functions that may be available.

RESULT CODES

`noErr`	0	No error occurred
`errAEEventNotHandled`	–1708	The object-comparison function is unable to compare the specified Apple event objects

SEE ALSO

For more information, see "Writing an Object-Comparison Function" on page 6-50.

MyDisposeToken

If your application requires more than a call to the `AEDisposeDesc` function to dispose of a token, or if it supports marking callback functions, you must provide one token disposal function. A token disposal function disposes of a specified token.

```
FUNCTION MyDisposeToken (VAR unneededToken: AEDesc): OSErr;
```

`unneededToken`
The token to dispose of.

DESCRIPTION

The Apple Event Manager calls your token disposal function whenever it needs to dispose of a token. It also calls your disposal function when your application calls the `AEDisposeToken` function. If your application does not provide a token disposal function, the Apple Event Manager calls `AEDisposeDesc` instead.

Your token disposal function must be able to dispose of all of the token types used by your application.

If your application supports marking, a call to `MyDisposeToken` to dispose of a mark token lets your application know that it can unmark the objects marked with that mark token.

SPECIAL CONSIDERATIONS

If the Apple Event Manager receives the result code `errAEEventNotHandled` after calling a token disposal function, it attempts to dispose of the token by some other method, such as calling an equivalent system token disposal function if one is available or, if that fails, by calling `AEDisposeDesc`. Thus, a token disposal function that can't dispose of a token should return `errAEEventNotHandled`. This allows the Apple Event Manager to try other token disposal functions that may be available.

RESULT CODES

`noErr`	0	No error occurred
`errAEEventNotHandled`	–1708	The token disposal function is unable to dispose of the token

MyGetErrorDesc

If you want to find out which descriptor record is responsible for an error that occurs during a call to the `AEResolve` function, you can provide an error callback function. An error callback function returns a pointer to an address. The Apple Event Manager uses this address to store the descriptor record it is currently working with if an error occurs during a call to `AEResolve`.

```
FUNCTION MyGetErrorDesc (VAR errDescPtr: DescPtr): OSErr;
```

`errDescPtr`
A pointer to an address.

DESCRIPTION

Your error callback function simply returns an address. Shortly after your application calls `AEResolve`, the Apple Event Manager calls your error callback function and writes a null descriptor record to the address returned, overwriting whatever was there previously. If an error occurs during the resolution of the object specifier record, the Apple Event Manager calls your error callback function again and writes the descriptor record—often an object specifier record—to the address returned. If `AEResolve` returns an error during the resolution of an object specifier record, this address contains the descriptor record responsible for the error.

Normally you should maintain a single global variable of type `AEDesc` whose address your error callback function returns no matter how many times it is called. Be careful if you use any other method. When recovering from an error, the Apple Event Manager never writes to the address you provide unless it already contains a null descriptor record. Thus, if you don't maintain a single global variable as just described, you should write null descriptor records to any addresses passed by your error callback function that are different from the addresses returned the first time your function is called after a given call to `AEResolve`.

If the result code returned by the `MyGetErrorDesc` function has a nonzero value, the Apple Event Manager continues to resolve the object specifier record as if it had never called the error callback function.

RESULT CODE

`noErr`	0	No error occurred

MyGetMarkToken

If your application supports marking, you must provide one mark token function. A mark token function returns a mark token.

```
FUNCTION MyGetMarkToken (containerToken: AEDesc;
                          containerClass: DescType;
                          VAR result: AEDesc): OSErr;
```

`containerToken`
The Apple event object that contains the elements to be marked with the mark token.

`containerClass`
The object class of the container that contains the objects to be marked.

`result` Your mark token function should return a mark token in this parameter.

DESCRIPTION

To get a mark token, the Apple Event Manager calls your mark token function. Like other tokens, the mark token returned can be a descriptor record of any type; however, unlike other tokens, a mark token identifies the way your application will mark Apple event objects during the current session while resolving a single object specifier record that specifies the key form `formTest`.

A mark token is valid until the Apple Event Manager either disposes of it (by calling `AEDisposeToken`) or returns it as the result of the `AEResolve` function. If the final result of a call to `AEResolve` is a mark token, the Apple event objects currently marked for that mark token are those specified by the object specifier record passed to `AEResolve`, and your application can proceed to do whatever the Apple event has requested. Note that your application is responsible for disposing of a final mark token with a call to `AEDisposeToken`, just as for any other final token.

If your application supports marking, it should also provide a token disposal function modeled after the token disposal function described on page 6-99. When the Apple Event Manager calls `AEDisposeToken` to dispose of a mark token that is not the final result of a call to `AEResolve`, the subsequent call to your token disposal function lets you know that you can unmark the Apple event objects marked with that mark token. A call to `AEDisposeDesc` to dispose of a mark token (which would occur if you did not provide a token disposal function) would go unnoticed.

RESULT CODES

`noErr`	0	No error occurred
`errAEEventNotHandled`	–1708	The mark token function is unable to return a mark token; if the Apple Event Manager gets this result, it attempts to get a mark token by calling the equivalent system marking callback function

SEE ALSO

For more information, see "Writing Marking Callback Functions," which begins on page 6-53.

MyMark

If your application supports marking, you must provide one object-marking function. An object-marking function marks a specific Apple event object.

```
FUNCTION MyMark (theToken: AEDesc; markToken: AEDesc;
                 markCount: LongInt): OSErr;
```

`theToken`	The token for the Apple event object to be marked.
`markToken`	The mark token used to mark the Apple event object.
`markCount`	The number of times `MyMark` has been called for the current mark token (that is, the number of Apple event objects that have so far passed the test, including the element to be marked).

DESCRIPTION

To mark an Apple event object using the current mark token, the Apple Event Manager calls the object-marking function provided by your application. In addition to marking the specified object, your `MyMark` function should record the mark count for each object that it marks. The mark count recorded for each marked object allows your application to determine which of a set of marked tokens pass a test, as described in the next section for the `MyAdjustMarks` function.

RESULT CODES

`noErr`	0	No error occurred
`errAEEventNotHandled`	–1708	The `MyMark` function is unable to mark the specified Apple event object; if the Apple Event Manager gets this result, it attempts to mark the object by calling the equivalent system object-marking function

SEE ALSO

For more information, see "Writing Marking Callback Functions," which begins on page 6-53.

MyAdjustMarks

If your application supports marking, you must provide one mark-adjusting function. A mark-adjusting function adjusts the marks made with the current mark token.

```
FUNCTION MyAdjustMarks (newStart, newStop: LongInt;
                        markToken: AEDesc): OSErr;
```

`newStart`	The mark count value (provided when the `MyMark` callback routine was called to mark the object) for the first object in the new set of marked objects.
`newStop`	The mark count value (provided when the `MyMark` callback routine was called to mark the object) for the last object in the new set of marked objects.
`markToken`	The mark token for the marked objects.

DESCRIPTION

When the Apple Event Manager needs to identify either a range of elements or the absolute position of an element in a group of Apple event objects that pass a test, it can use your application's mark-adjusting function to unmark objects previously marked by a call to your marking function. For example, suppose an object specifier record specifies "any row in the table 'MyCustomers' for which the City column is 'San Francisco.'" The Apple Event Manager first uses the appropriate object accessor function to locate all the rows in the table for which the City column is "San Francisco" and calls the application's marking function repeatedly to mark them. It then generates a random number between 1 and the number of rows it found that passed the test and calls the application's mark-adjusting function to unmark all the rows whose mark count does not match the randomly generated number. If the randomly chosen row has a mark count value of 5, the Apple Event Manager passes the value 5 to the mark-adjusting function in both the `newStart` parameter and the `newStop` parameter, and passes the current mark token in the `markToken` parameter.

When the Apple Event Manager calls your `MyAdjustMarks` function, your application must dispose of any data structures that it created to mark the previously marked objects.

RESULT CODES

`noErr`	0	No error occurred
`errAEEventNotHandled`	–1708	The `MyAdjustMarks` function is unable to adjust the marks as requested; if the Apple Event Manager gets this result, it attempts to adjust the marks by calling the equivalent system mark-adjusting function

SEE ALSO

For more information, see "Writing Marking Callback Functions" on page 6-53.

Summary of Resolving and Creating Object Specifier Records

Pascal Summary

Constants

```
CONST
   gestaltAppleEventsAttr     = 'evnt';   {selector for Apple events}
   gestaltAppleEventsPresent  = 0;        {if this bit is set, Apple }
                                          { Event Manager is available}

   {logical operators for descriptor records with keyword }
   { keyAELogicalOperator}
   kAEAND                  =  'AND ';
   kAEOR                   =  'OR  ';
   kAENOT                  =  'NOT ';

   {absolute ordinals used as key data in an object specifier }
   { record with key form formAbsolutePosition}
   kAEFirst                =  'firs';
   kAELast                 =  'last';
   kAEMiddle               =  'midd';
   kAEAny                  =  'any ';
   kAEAll                  =  'all ';

   {relative ordinals used as key data in an object specifier record }
   { with key form formRelativePosition}
   kAENext                 =  'next';
   kAEPrevious             =  'prev';

   {keywords for object specifier records}
   keyAEDesiredClass       =  'want'; {object class ID}
   keyAEContainer          =  'from'; {description of container}
   keyAEKeyForm            =  'form'; {key form}
   keyAEKeyData            =  'seld'; {key data for specified key form}

   {keywords for range descriptor records}
   keyAERangeStart         =  'star';     {beginning of range}
```

```
keyAERangeStop          =  'stop';     {end of range}

{values for the keyAEKeyForm field of an object specifier record}
formAbsolutePosition    =  'indx';     {for example, 1 = first }
                                       { element in container, -2 = }
                                       { second from end of container}
formRelativePosition    =  'rele';     {key data specifies element }
                                       { before or after container}
formTest                =  'test';     {key data specifies a test}
formRange               =  'rang';     {key data specifies a range}
formPropertyID          =  'prop';     {key data is property ID}
formName                =  'name';     {key data is element's name}

{descriptor types used to identify Apple event objects}
typeObjectSpecifier     =  'obj ';     {object specifier record, often }
                                       { used as keyAEContainer}
typeObjectBeingExamined =  'exmn';     {used as keyAEContainer}
typeCurrentContainer    =  'ccnt';     {used as keyAEContainer}
typeToken               =  'toke';     {substituted for 'ccnt' }
                                       { before accessor called}
typeAbsoluteOrdinal     =  'abso';     {formAbsolutePosition}
typeRangeDescriptor     =  'rang';     {formRange}
typeLogicalDescriptor   =  'logi';     {formTest}
typeCompDescriptor      =  'cmpd';     {formTest}

{various relevant keywords}
keyAECompOperator       =  'relo';     {operator for comparison: }
                                       { '=', '<=', etc.}
keyAELogicalTerms       =  'term';     {an AEList of terms to be }
                                       { related by 'logc' below}
keyAELogicalOperator    =  'logc';     {kAEAND, kAEOR, or kAENOT}
keyAEObject1            =  'obj1';     {first of two objects being }
                                       { compared; must be object }
                                       { specifier record}
keyAEObject2            =  'obj2';     {the other object; may be }
                                       { simple descriptor record }
                                       { or object specifier record}

{special handler selectors used with AESetObjectCallbacks}
keyDisposeTokenProc     =  'xtok';
keyAECompareProc        =  'cmpr';
keyAECountProc          =  'cont';
keyAEMarkTokenProc      =  'mkid';
```

```
   keyAEMarkProc           =  'mark';
   keyAEAdjustMarksProc    =  'adjm';
   keyAEGetErrDescProc     =  'indc';

   {additive values for callbackFlags parameter to AEResolve}
   kAEIDoMinimum           =  $0000;   {server does not support whose }
                                       { descriptor records or marking}
   kAEIDoWhose             =  $0001;   {server supports whose }
                                       { descriptor records}
   kAEIDoMarking           =  $0004;   {server supports marking}

   {constants for whose descriptor records}
   typeWhoseDescriptor     =  'whos';  {whose descriptor record}
   formWhose               =  'whos';  {key form for key data of descriptor }
                                       { type typeWhoseDescriptor}
   typeWhoseRange          =  'wrng';  {whose range descriptor record}
   keyAEWhoseRangeStart    =  'wstr';  {beginning of range}
   keyAEWhoseRangeStop     =  'wstp';  {end of range}
   keyAEIndex              =  'kidx';  {index for whose descriptor record}
   keyAETest               =  'ktst';  {test for whose descriptor record}
```

Data Types

```
TYPE
   ccntTokenRecord =                     {used for rewriting tokens in }
   RECORD                                { place of 'ccnt' descriptor }
      tokenClass:    DescType;           { records; only of interest to }
      token:         AEDesc;             { those who, when they get ranges }
   END;                                  { as key data in their object }
                                         { accessor functions, resolve }
   ccntTokenRecPtr = ^ccntTokenRecord;   { the object specifier records }
   ccntTokenRecHandle = ^ccntTokenRecPtr; { for the end points manually}

   DescPtr = ^AEDesc;
   DescHandle = ^DescPtr;

   AccessorProcPtr = ProcPtr;
```

Routines for Resolving and Creating Object Specifier Records

Initializing the Object Support Library

```
FUNCTION AEObjectInit : OSErr;
```

Setting Object Accessor Functions and Object Callback Functions

```
FUNCTION AEInstallObjectAccessor
                            (desiredClass: DescType;
                             containerType: DescType;
                             theAccessor: AccessorProcPtr;
                             accessorRefcon: LongInt;
                             isSysHandler: Boolean): OSErr;
FUNCTION AESetObjectCallbacks
                            (myCompareProc, myCountProc,
                             myDisposeTokenProc,
                             myGetMarkTokenProc, myMarkProc,
                             myAdjustMarksProc, myGetErrDescProc:
                             ProcPtr): OSErr;
```

Getting, Calling, and Removing Object Accessor Functions

```
FUNCTION AEGetObjectAccessor
                            (desiredClass: DescType;
                             containerType: DescType;
                             VAR theAccessor: AccessorProcPtr;
                             VAR accessorRefcon: LongInt;
                             isSysHandler: Boolean): OSErr;
FUNCTION AECallObjectAccessor
                            (desiredClass: DescType;
                             containerToken: AEDesc;
                             containerClass: DescType;
                             keyForm: DescType;
                             keyData: AEDesc;
                             VAR theToken: AEDesc): OSErr;
FUNCTION AERemoveObjectAccessor
                            (desiredClass: DescType;
                             containerType: DescType;
                             theAccessor: AccessorProcPtr;
                             isSysHandler: Boolean): OSErr;
```

Resolving Object Specifier Records

```
FUNCTION AEResolve          (objectSpecifier: AEDesc;
                             callbackFlags: Integer;
                             VAR theToken: AEDesc): OSErr;
```

Deallocating Memory for Tokens

```
FUNCTION AEDisposeToken     (VAR theToken: AEDesc): OSErr;
```

Creating Object Specifier Records

```
FUNCTION CreateOffsetDescriptor
                            (theOffset: LongInt;
                             VAR theDescriptor: AEDesc):
                             OSErr;
FUNCTION CreateCompDescriptor
                            (comparisonOperator: DescType;
                             VAR operand1: AEDesc;
                             VAR operand2: AEDesc;
                             disposeInputs: Boolean;
                             VAR theDescriptor: AEDesc): OSErr;
FUNCTION CreateLogicalDescriptor
                            (VAR theLogicalTerms: AEDescList;
                             theLogicOperator: DescType;
                             disposeInputs: Boolean;
                             VAR theDescriptor: AEDesc): OSErr;
FUNCTION CreateRangeDescriptor
                            (VAR rangeStart: AEDesc;
                             VAR rangeStop: AEDesc;
                             disposeInputs: Boolean;
                             VAR theDescriptor: AEDesc): OSErr;
FUNCTION CreateObjSpecifier (desiredClass: DescType;
                             VAR theContainer: AEDesc;
                             keyForm: DescType;
                             VAR keyData: AEDesc;
                             disposeInputs: Boolean;
                             VAR objSpecifier: AEDesc): OSErr;
```

Application-Defined Routines

Object Accessor Functions

```
FUNCTION MyObjectAccessor   (desiredClass: DescType;
                             containerToken: AEDesc;
                             containerClass: DescType;
                             keyForm: DescType; keyData: AEDesc;
                             VAR theToken: AEDesc;
                             theRefcon: LongInt): OSErr;
```

Object Callback Functions

```
FUNCTION MyCountObjects     (desiredClass: DescType;
                             containerClass: DescType;
                             theContainer: AEDesc;
                             VAR result: LongInt): OSErr;
```

```
FUNCTION MyCompareObjects   (comparisonOperator: DescType;
                             theobject: AEDesc;
                             objectOrDescToCompare: AEDesc;
                             VAR result: Boolean): OSErr;
FUNCTION MyDisposeToken     (VAR unneededToken: AEDesc): OSErr;
FUNCTION MyGetErrorDesc     (VAR errDescPtr: DescPtr): OSErr;
FUNCTION MyGetMarkToken     (containerToken: AEDesc;
                             containerClass: DescType;
                             VAR result: AEDesc): OSErr;
FUNCTION MyMark             (theToken: AEDesc; markToken: AEDesc;
                             markCount: LongInt): OSErr;
FUNCTION MyAdjustMarks      (newStart, newStop: LongInt;
                             markToken: AEDesc): OSErr;
```

C Summary

Constants

```
enum {
      #define gestaltAppleEventsAttr     'evnt' /*selector for Apple events*/
      gestaltAppleEventsPresent       = 0   /*if this bit is set, then */
                                            /* Apple Event Manager is */
};                                          /* available*/

/*logical operators for descriptor records with keyword */
/* keyAELogicalOperator*/
#define kAEAND                      'AND '
#define kAEOR                       'OR  '
#define kAENOT                      'NOT '

/*absolute ordinals used as key data in an object specifier */
/* record with key form formAbsolutePosition*/
#define kAEFirst                    'firs'
#define kAELast                     'last'
#define kAEMiddle                   'midd'
#define kAEAny                      'any '
#define kAEAll                      'all '

/*relative ordinals used as key data in an object specifier record */
/* with key form formRelativePosition*/
#define kAENext                     'next'
```

```
#define kAEPrevious                'prev'

/*keywords for object specifier records*/
#define keyAEDesiredClass          'want'   /*object class ID*/
#define keyAEContainer             'from'   /*description of container*/
#define keyAEKeyForm               'form'   /*key form*/
#define keyAEKeyData               'seld'   /*key data for specified key */
                                            /* form*/

/*keywords for range descriptor records*/
#define keyAERangeStart            'star'   /*beginning of range*/
#define keyAERangeStop             'stop'   /*end of range*/

/*values for the keyAEKeyForm field of an object specifier record*/
#define formAbsolutePosition       'indx'   /*for example, 1 = first */
                                            /* element in container, -2 = */
                                            /* second from end of */
                                            /* container*/
#define formRelativePosition       'rele'   /*key data specifies element */
                                            /* before or after container*/
#define formTest                   'test'   /*key data specifies a test*/
#define formRange                  'rang'   /*key data specifies a range*/
#define formPropertyID             'prop'   /*key data is property ID*/
#define formName                   'name'   /*key data is element's name*/

/* descriptor types used to identify Apple event objects*/
#define typeObjectSpecifier        'obj '   /*object specifier record, */
                                            /* often used as */
                                            /* keyAEContainer*/
#define typeObjectBeingExamined    'exmn'   /*used as keyAEContainer*/
#define typeCurrentContainer       'ccnt'   /*used as keyAEContainer*/
#define typeToken                  'toke'   /*substituted for 'ccnt' */
                                            /* before accessor called*/
#define typeAbsoluteOrdinal        'abso'   /*formAbsolutePosition*/
#define typeRangeDescriptor        'rang'   /*formRange*/
#define typeLogicalDescriptor      'logi'   /*formTest*/
#define typeCompDescriptor         'cmpd'   /*formTest*/
/*various relevant keywords*/
#define keyAECompOperator          'relo'   /*operator for comparison: */
                                            /* '=', '<=', etc.*/
#define keyAELogicalTerms          'term'   /*an AEList of terms to be */
                                            /* related by 'logc' below*/
#define keyAELogicalOperator       'logc'   /*kAEAND, kAEOR, or kAENOT*/
```

```
#define keyAEObject1                'obj1'   /*first of two objects being */
                                             /* compared; must be object */
                                             /* specifier record*/
#define keyAEObject2                'obj2'   /*the other object; may be */
                                             /* simple descriptor record */
                                             /* or object specifier record*/
/*special handler selectors used with AESetObjectCallbacks*/
#define keyDisposeTokenProc         'xtok'
#define keyAECompareProc            'cmpr'
#define keyAECountProc              'cont'
#define keyAEMarkTokenProc          'mkid'
#define keyAEMarkProc               'mark'
#define keyAEAdjustMarksProc        'adjm'
#define keyAEGetErrDescProc         'indc'

/*additive values for callbackFlags parameter to AEResolve*/
#define kAEIDoMinimum               0x0000   /*server does not support */
                                             /* whose descriptor records */
                                             /* or marking*/
#define kAEIDoWhose                 0x0001   /*server supports whose */
                                             /* descriptor records*/
#define kAEIDoMarking               0x0004   /*server supports marking*/

/*constants for whose descriptor records*/
#define typeWhoseDescriptor         'whos'   /*whose descriptor record*/
#define formWhose                   'whos'   /*key form for key data of */
                                             /* descriptor type */
                                             /* typeWhoseDescriptor*/
#define typeWhoseRange              'wrng'   /*whose range descriptor */
                                             /* record*/
#define keyAEWhoseRangeStart        'wstr'   /*beginning of range*/
#define keyAEWhoseRangeStop         'wstp'   /*end of range*/
#define keyAEIndex                  'kidx'   /*index for whose descriptor */
                                             /* record*/
#define keyAETest                   'ktst'   /*test for whose descriptor */
                                             /* record*/
```

Data Types

```
struct ccntTokenRecord {                     /*used for rewriting tokens */
   DescType tokenClass;                      /* in place of 'ccnt' */
   AEDesc token;                             /* descriptor records; only */
};                                           /* of interest to those who, */
```

```
                                             /* when they get ranges as */
typedef struct ccntTokenRecord ccntTokenRecord, /* key data in their object */
*ccntTokenRecPtr, **ccntTokenRecHandle;      /* accessor functions, */
                                             /* resolve them manually*/

typedef AEDesc *DescPtr, **DescHandle;

/*typedefs providing type checking for procedure pointers*/
typedef pascal OSErr (*accessorProcPtr) (DescType desiredClass,
                                          const AEDesc *container,
                                          DescType containerClass,
                                          DescType form,
                                          const AEDesc *selectionData,
                                          AEDesc *value, long LongInt);
typedef pascal OSErr (*compareProcPtr)(DescType oper, const AEDesc *obj1,
                                       const AEDesc *obj2,
                                       Boolean *result);
typedef pascal OSErr (*countProcPtr)(DescType desiredClass,
                                       DescType containerClass,
                                       const AEDesc *container,
                                       long *result);
typedef pascal OSErr (*disposeTokenProcPtr)(AEDesc *unneededToken);
typedef pascal OSErr (*getMarkTokenProcPtr)(const AEDesc *ContainerToken,
                                             DescType containerClass,
                                             AEDesc *result);

typedef pascal OSErr (*getErrDescProcPtr)(DescPtr *appDescPtr);
```

Routines for Resolving and Creating Object Specifier Records

Initializing the Object Support Library

```
pascal OSErr AEObjectInit   ();
```

Setting Object Accessor Functions and Object Callback Functions

```
pascal OSErr AEInstallObjectAccessor
                            (DescType desiredClass, DescType containerType,
                             accessorProcPtr theAccessor,
                             long accessorRefcon, Boolean isSysHandler);
pascal OSErr AESetObjectCallbacks
                            (compareProcPtr myCompareProc,
                             countProcPtr myCountProc,
                             disposeTokenProcPtr myDisposeTokenProc,
                             getMarkTokenProcPtr myGetMarkTokenProc,
                             markProcPtr myMarkProc,
                             adjustMarksProcPtr myAdjustMarksProc,
                             getErrDescProcPtr myGetErrDescProc);
```

Getting, Calling, and Removing Object Accessor Functions

```
pascal OSErr AEGetObjectAccessor
                            (DescType desiredClass, DescType containerType,
                             accessorProcPtr *theAccessor,
                             long *accessorRefcon, Boolean isSysHandler);
pascal OSErr AECallObjectAccessor
                            (DescType desiredClass,
                             const AEDesc *containerToken,
                             DescType containerClass, DescType keyForm,
                             const AEDesc *keyData, AEDesc *theToken);
pascal OSErr AERemoveObjectAccessor
                            (DescType desiredClass, DescType containerType,
                             accessorProcPtr theAccessor,
                             Boolean isSysHandler);
```

Resolving Object Specifier Records

```
pascal OSErr AEResolve      (const AEDesc *objectSpecifier,
                             short callbackFlags, AEDesc *theToken);
```

Deallocating Memory for Tokens

```
pascal OSErr AEDisposeToken (AEDesc *theToken);
```

Creating Object Specifier Records

```
pascal OSErr CreateOffsetDescriptor
                            (long theOffset, AEDesc *theDescriptor);
pascal OSErr CreateCompDescriptor
                            (DescType comparisonOperator, AEDesc* operand1,
                             AEDesc* operand2, Boolean disposeInputs,
                             AEDesc* theDescriptor);
pascal OSErr CreateLogicalDescriptor
                            (AEDescList *theLogicalTerms,
                             DescType theLogicOperator,
                             Boolean disposeInputs, AEDesc *theDescriptor);
pascal OSErr CreateRangeDescriptor
                            (AEDesc *rangeStart, AEDesc *rangeStop,
                             Boolean disposeInputs, AEDesc *theDescriptor);
```

```
pascal OSErr CreateObjSpecifier
                            (DescType desiredClass, AEDesc *theContainer,
                             DescType keyForm, AEDesc *keyData,
                             Boolean disposeInputs, AEDesc *objSpecifier);
```

Application-Defined Routines

Object Accessor Functions

```
pascal OSErr MyObjectAccessor
                            (DescType desiredClass,
                             const AEDesc *containerToken,
                             DescType containerClass,
                             DescType keyForm, const AEDesc *keyData,
                             AEDesc *theToken, long *theRefcon);
```

Object Callback Functions

```
pascal OSErr MyCountObjects (DescType desiredClass, DescType containerClass,
                             const AEDesc *theContainer, long *result);
pascal OSErr MyCompareObjects
                            (DescType comparisonOperator,
                             const AEDesc *theObject,
                             const AEDesc *objectOrDescToCompare,
                             Boolean *result);
pascal OSErr MyDisposeToken (AEDesc *unneededToken);
pascal OSErr MyGetErrorDesc (DescPtr *errDescPtr);
pascal OSErr MyGetMarkToken (const AEDesc *containerToken,
                             DescType containerClass, AEDesc *result);
pascal OSErr MyMark         (const AEDesc *theToken,
                             const AEDesc *markToken, long markCount);
pascal OSErr MyAdjustMarks  (long newStart, long newStop,
                             const AEDesc *markToken);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

`_Pack8`

Selector	Routine
$023A	`AEDisposeToken`
$0536	`AEResolve`
$0738	`AERemoveObjectAccessor`
$0937	`AEInstallObjectAccessor`
$0939	`AEGetObjectAccessor`
$0C3B	`AECallObjectAccessor`
$0E35	`AESetObjectCallbacks`

Result Codes

`noErr`	0	No error
`paramErr`	–50	Parameter error (for example, value of handler pointer is `NIL` or odd)
`eLenErr`	–92	Buffer too big to send
`memFullErr`	–108	Not enough room in heap zone
`userCanceledErr`	–128	User canceled an operation
`procNotFound`	–600	No eligible process with specified process serial number
`bufferIsSmall`	–607	Buffer is too small
`noOutstandingHLE`	–608	No outstanding high-level event
`connectionInvalid`	–609	Nonexistent signature or session ID
`noUserInteractionAllowed`	–610	Background application sends event requiring authentication
`noPortErr`	–903	Client hasn’t set `'SIZE'` resource to indicate awareness of high-level events
`destPortErr`	–906	Server hasn’t set `'SIZE'` resource to indicate awareness of high-level events, or else is not present
`sessClosedErr`	–917	The `kAEDontReconnect` flag in the `sendMode` parameter was set, and the server quit and then restarted
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAEDescNotFound`	–1701	Descriptor record was not found
`errAECorruptData`	–1702	Data in an Apple event could not be read
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

`errAENewerVersion`	–1706	Need a newer version of the Apple Event Manager
`errAENotAppleEvent`	–1707	Event is not an Apple event
`errAEEventNotHandled`	–1708	Event wasn't handled by an Apple event handler
`errAEReplyNotValid`	–1709	`AEResetTimer` was passed an invalid reply
`errAEUnknownSendMode`	–1710	Invalid sending mode was passed
`errAEWaitCanceled`	–1711	User canceled out of wait loop for reply or receipt
`errAETimeout`	–1712	Apple event timed out
`errAENoUserInteraction`	–1713	No user interaction allowed
`errAENotASpecialFunction`	–1714	The keyword is not valid for a special function
`errAEParamMissed`	–1715	Handler cannot understand a parameter the client considers required
`errAEUnknownAddressType`	–1716	Unknown Apple event address type
`errAEHandlerNotFound`	–1717	No handler found for an Apple event or a coercion, or no object callback function found
`errAEReplyNotArrived`	–1718	Reply has not yet arrived
`errAEIllegalIndex`	–1719	Not a valid list index
`errAEImpossibleRange`	–1720	The range is not valid because it is impossible for a range to include the first and last objects that were specified; an example is a range in which the offset of the first object is greater than the offset of the last object
`errAEWrongNumberArgs`	–1721	The number of operands provided for the `kAENOT` logical operator is not 1
`errAEAccessorNotFound`	–1723	There is no object accessor function for the specified object class and token descriptor type
`errAENoSuchLogical`	–1725	The logical operator in a logical descriptor record is not `kAEAND`, `kAEOR`, or `kAENOT`
`errAEBadTestKey`	–1726	The descriptor record in a test key is neither a comparison descriptor record nor a logical descriptor record
`errAENotAnObjectSpec`	–1727	The `objSpecifier` parameter of `AEResolve` is not an object specifier record
`errAENoSuchObject`	–1728	A run-time resolution error, for example: object specifier record asked for the third element, but there are only 2.
`errAENegativeCount`	–1729	Object-counting function returned negative value
`errAEEmptyListContainer`	–1730	The container for an Apple event object is specified by an empty list
`errAEUnknownObjectType`	–1731	Descriptor type of token returned by `AEResolve` is not known to server application
`errAERecordingIsAlreadyOn`	–1732	Attempt to turn recording on when it is already on